no d'ordre: 2014-07-TH

THÈSE DE DOCTORAT

DOMAINE: STIC

SPECIALITÉ: Informatique

École doctorale " Sciences et Technologies de l'Information des Télécommunications et des Systemes "

Presentée par:

Bassem KHOUZAM

Sujet:

Les Réseaux de Neurones Comme Paradigme de Calcul Cellulaire pour le Traitement de Séquences Temporelles

(Neural Networks as Cellular Computing Models for Temporal Sequence Porcessing)

Soutenue le 13 février 2014 devant les membres du jury:

Frédéric ALEXANDRE	DR Inria	Rapporteur
Bernard GIRAU	Pr Univ. Lorraine	Rapporteur
Jean-Sylvain LIÉNARD	DR CNRS	Examinateur
Arnaud REVEL	Pr Univ. La Rochelle	Examinateur
Yolaine BOURDA	Pr Supélec	Directrice de thèse
Hervé FREZZA-BUET	Pr Supélec	Co-Directeur de thèse

À Liana et Hadi.
À ma famille.
À ma Syrie.

Remerciements

Je voudrais tout d'abord remercier la directrice de cette thèse Mme. Yolaine Bourda qui a généreusement accepté de diriger ma thèse quand mon inscription a été interrompue ailleurs.

Je tiens à remercier M. Hervé Frezza-Buet, mon co-directeur de thèse avec qui je suis honoré d'avoir eu l'opportunité de travailler. Il a toujours été là pour me guider et me soutenir au cours de l'élaboration de cette thèse, non seulement par ses conseils et son appui scientifique, mais aussi par son soutien personnel et moral lors des complications administratives et difficultés psychologiques que j'ai eues. Sans lui, cette personne inoubliable, cette thèse n'aurait jamais été menée à son terme.

Mes remerciements vont aussi à M. Jean-Sylvain Liénard qui m'a accordé l'honneur de présider mon Jury de soutenance. M. Frédéric Alexandre et M. Bernard Girau ont accepté d'être les rapporteurs de ma thèse et m'ont fourni des remarques et des suggestions importantes qui m'ont aidé à améliorer la qualité de ce mémoire, je leur en suis reconnaissant. Je remercie également M. Arnaud Revel qui m'a fait l'honneur d'être examinateur de mon travail. Je remercie le jury dans son ensemble pour la discussion riche que l'on a eue lors de la soutenance.

Mon travail de recherche a été financé par la région Lorraine et Supélec qui tout fait pour m'assurer de bonnes conditions de travail, je les en remercie. Je remercie aussi la direction de Supélec, notamment M. Serge Perrine le directeur du campus de Metz, pour son aide administrative. Je remercie l'équipe de l'administration, notamment Thérèse Pirrone, pour son aide persistante.

Au cours de ma thèse, j'ai fait partie de l'équipe IMS, les discussions que j'ai pu avoir avec Jérémy Fix, Matthieu Geist, Édouard Klein, Lucie Daubigney et Denis Baheux m'ont beaucoup apporté, je remercie donc toutes ces personnes ainsi que tout les membres de l'équipe que je suis ravi d'avoir côtoyés. Je remercie particulièrement les anciens doctorants de Supélec, Lucian Alecu, Costantinos Makkassikis et Vianney Caullet, pour leur amitié, encouragement et soutien incessant.

Enfin, je remercie ma famille pour penser toujours à moi, mon épouse Liana pour tout ce qu'elle a supporté pour moi, pour sa patience et soutien inconditionnel, et mon fils Hadi qui m'a donné la raison pour persévérer quand j'avais perdu toute autre raison.

Contents

CHAPTER 1

Résumé en Français

1.1 Préambule

Ce chapitre reprend l'ensemble des développements présentés dans ce manuscrit, en langue française, puisque le reste du manuscrit est, lui, rédigé en anglais. La structuration des paragraphes au sein de ce chapitre suit la structuration de la thèse.

L'objectif de ce chapitre est de restituer l'argumentation scientifique qui est articulée tout au long du manuscrit en anglais. Bien évidemment, comme il s'agit d'un résumé, il nous a semblé que faire apparaître l'ensemble des références bibliographiques aurait nui à la lecture. Nous renvoyons donc le lecteur aux chapitres suivants, en anglais, pour des références plus complètes.

La thèse est organisée en six chapitres, sans compter celui-ci. Le premier de ces chapitres est une introduction situant notre travail dans une perspective informatique, c'est-à-dire dans la démarche de contribuer à définir ce que peut on ou non calculer une machine. En effet, dans la mesure où nous abordons des systèmes liés à la vie artificielle et aux réseaux de neurones, nous aurions pu vouloir proposer une modélisation du vivant, ce qui nous aurait conduit à d'autres raisonnements ainsi qu'à d'autres justifications. Cette précision du cadre informatique est nécessaire dans le contexte pluridisciplinaire dans lequel notre travail s'insère. Le deuxième chapitre porte sur une étude des différents paradigmes de calcul, pour aboutir à la notion de calcul cellulaire, auquel nous avons proposé une contribution. Une fois que les paradigmes de calcul qui nous concernent sont définis, le troisième chapitre aborde la question du temps dans les réseaux de neurones, en passant en revue les différentes approches que l'on peut trouver dans la littérature. Dans la mesure où nos travaux abordent explicitement la prise en compte du temps, il s'agit dans ce chapitre de donner les éléments de bibliographie qui permettent de positionner notre approche par rapport à l'état de l'art, dans la perspective du calcul temporel. Le quatrième chapitre est un développement du troisième, puisqu'il met en avant les approches par cartes auto-organisatrices des apprentissages temporels. Nous profitons de ce chapitre pour rappeler les bases des cartes auto-organisatrices, puisque nos travaux reposent sur ce paradigme d'auto-organisation pour capturer la dynamique du flux des entrées soumises au système. Le cinquième chapitre présente notre modèle à proprement parler, modèle qui se situe à la frontière des systèmes de calcul cellulaires, des systèmes d'apprentissage temporel, et des systèmes auto-organisés. C'est cette position, à la croisée de ces trois domaines, qui est selon nous originale dans nos travaux. Enfin, le sixième chapitre conclut le manuscrit en faisant le bilan de l'approche et en ouvrant quelques perspectives pour les travaux qui pourront faire suite aux nôtres.

Avant d'entrer dans le résumé de la thèse, nous souhaiterions souligner ici qu'au-delà de la présentation du modèle que nous proposons, nous avons souhaité faire un point le plus complet possible sur l'état de l'art des trois domaines mentionnés ci-dessus, de façon synthétique. Nous

espérons également par cette contribution bibliographique pouvoir clarifier le domaine et aider l'orientation des travaux futurs de l'équipe.

1.2 Introduction

De l'époque sumérienne (2400 avant JC) où les premières formalisations du calcul ont vu le jour à l'époque actuelle, l'Homme a cherché à automatiser les calculs, pour des besoins économiques et scientifiques. Initialement, il s'agissait de développer les mathématiques, les outils permettant d'enchaîner les calculs étant essentiellement le papier et le crayon, ce qui a changé récemment avec l'arrivée des ordinateurs et leur montée en puissance. Actuellement, cette puissance est telle qu'elle génère de plus en plus de besoins en calcul. Pour répondre à ces besoins, les scientifiques envisagent des paradigmes pris à d'autres disciplines, comme la mécanique quantique, l'ADN, mais aussi le cerveau humain.

1.2.1 Émergence de la calculabilité, fonctionnalisme et intelligence

Dès les années 30, en réponse au problème de décision posé par Hilbert, Alan Turing propose un outil formel, la machine de Turing, permettant de définir la calculabilité. Cette machine consiste en une bande infinie mono-dimensionnelle sur laquelle peuvent être inscrits et lus des symboles. À cette bande s'ajoute une machine à états, dont les transitions sont contrôlées par le symbole présent sur la bande à l'endroit pointé par une tête de lecture. Une transition conduit à un changement d'état, mais aussi à l'écriture d'un symbole sur la bande, ainsi qu'au déplacement de la tête de lecture d'une position, vers la gauche ou vers la droite. Les règles décrivant les transitions sont le programme exécuté par la machine, et sont la définition formelle d'un calcul.

La calculablilité ainsi définie, par la thèse de Church-Turing, permet de délimiter le périmètre des méthodes calculables, c'est-à-dire des méthodes impliquant un nombre fini d'étapes. Toutefois, toutes ces méthodes ne sont pas nécessairement efficaces, et il existe des problèmes non calculables. Le calcul de ces problèmes-là est l'hyper-computation, pour lequel d'autres machines formelles ont été définies, mais aucune ne semble physiquement réalisable. La notion de machine a fait l'objet de nombreux débats philosophiques, avec l'idée de savoir si cette notion pouvait s'étendre au cerveau humain, par exemple.

Bien au-delà de la réponse au problème de décision de Hilbert, la formalisation de la calculabilité sous forme de machines pose la question de savoir si une machine peut penser, être dotée d'intelligence. Alan Turing a proposé le fameux test, très controversé, où c'est à un humain de qualifier si oui ou non l'agent avec lequel il interagit via un clavier, est doté d'intelligence. Turing pensait que la physique du système nerveux pouvait être approchée d'aussi près qu'on le souhaitait par un programme informatique, qui par conséquent aurait les même propriétés d'intelligence que le système nerveux qu'il modélise. C'est le type d'arguments repris plus tard par Putman, sous le terme de théorie fonctionnaliste, lorsqu'il introduit l'intelligence comme une fonction entre entrées et sorties, indépendemment du support physique (i.e. il n'est pas nécessaire de posséder un cerveau biologique pour être intelligent, il suffit que le système ait de quoi réaliser les mêmes fonctions).

C'est ainsi qu'est introduite l'idée de l'intelligence, la pensée, comme un processus computationnel. Par conséquent, le cerveau, lui, apparaît selon cette approche comme un calculateur, et

donc comme le siège de processus informatiques. Là encore, cette idée fait l'objet de nombreux débats philosophiques, argumentant que les ordinateurs ne font qu'imiter l'intelligence (i.e. rejet de l'argument fonctionnaliste).

Au niveau informatique, ce qui ressort de ce débat est l'idée qu'une informatique adéquate, différente de l'algorithmique traditionnelle, peut permettre d'aborder l'intelligence. Turing lui-même, alors que l'informatique n'était qu'à son commencement, avait émis l'hypothèse d'une informatique évolutionniste, de systèmes adaptatifs pouvant modifier les règles qui commandent au calcul qu'ils réalisent.

1.2.2 Architectures numériques

Depuis les premières machines à calculer numériques, faites de ressorts, de billes, puis la notion introduite par Alan Turing de machines où les programmes sont stockés, qui a été instanciée physiquement par Von Neumann, les électrons remplaçant les billes, des efforts de technologies colossaux ont été réalisés pour construire des ordinateurs de plus en plus puissants.

L'architecture de Von Neuman, que l'on retrouve encore dans les CPU d'aujourd'hui, souffre d'un goulot d'étranglement, puisque c'est sur un bus commun que circulent les instructions et les données, séquentiellement. L'essentiel du trafic d'information n'est pas dû au calcul lui-même, c'est-à-dire aux instructions et aux données, mais à la circulation des informations (adresses) nécessaire à retrouver ces instructions et données en mémoire. Certaines architectures dédiées (comme les DSP) s'écartent du paradigme de Von Neumann, pour des raisons d'efficacité, mais elles sont en général moins flexibles en terme de programmation (modification du programme en cours de son exécution, etc.).

Aujourd'hui, il y a une saturation des accélérations que l'on peut attendre des architectures de Von Neumann, pour des raisons physiques, aussi bien en raison de la limitation des vitesses d'échange à la vitesse de la lumière que de la résolution spatiale des constituants élémentaires, sans parler de la consommation énergétique qui s'élève significativement avec la miniaturisation et l'augmentation des fréquences d'horloge.

Pour ces raisons, des constructeurs comme Intel ont choisi de s'engager sur la voie de l'augmentation du nombre de cœurs sur un même composant plutôt que l'accélération de la fréquence et la miniaturisation, ce qui implique que l'augmentation de la puissance de calcul réalisable sur une machine passe désormais nécessairement par la capacité des algorithmes que l'on souhaite exécuter à être parallélisés, ce qui n'est pas assuré pour un algorithme séquentiel quelconque.

Il est alors important de distinguer le parallélisme des architectures de celui des programmes. En effet, le premier est lié à la machine, et correspond à une solution physique pour sortir du goulot d'étranglement imposé par l'architecture de Von Neumann. C'est celui que nous avons évoqué ci-dessus. Le parallélisme des programmes, lui, est une propriété de l'algorithme qui autorise le calcul à être exprimé comme l'exécution de plusieurs processus concurrents. Cela dit, un algorithme s'exprimant en termes de processus concurrents ne s'implémente pas forcément naturellement sur les architectures multi-cœur actuelles.

En conclusion, les architectures matérielles parallèles sont indéniablement les machines qui permettront à l'avenir d'augmenter la puissance de calcul des ordinateurs, mais écrire des algo-

rithmes qui aient une nature distribuée compatible avec ces machines-là reste un défi, pour lequel il est nécessaire de pouvoir penser les concepts de calculs adéquats.

1.2.3 Le calcul cellulaire

Le cerveau humain, si l'on accepte l'idée de le considérer comme un calculateur, est indéniablement différent des machines de Von Neumann du fait de son architecture, qui met en relation 10^{11} neurones, avec des poids de connexion évoluant au fur et à mesure du temps et des stimulations reçues. Nous adhérons à l'idée selon laquelle, du fait de son architecture non séquentielle, le cerveau humain peut implémenter des calculs super-Turing.

Pour cette raison, nous nous intéressons ici au domaine du calcul neuronal, puisque c'est selon ce paradigme que le cerveau produit le comportement, les raisonnements, les différents types de mémoires, etc. C'est en cela que notre démarche, du point de vue informatique, est résolument connexionniste.

Parallèlement aux formalisations séquentielles du calcul proposées par Turing, Von Neumann et Ulam ont proposé une autre formalisation du calcul, plus proche dans sa nature de ce que réaliste le cerveau : les automates cellulaires. Il s'agit d'unités réalisant toutes le même calcul en parallèle, avec une connectivité entre elles qui repose sur une notion de voisinage. Le jeu de la vie de Conway en est un exemple célèbre. Différents types de calculs ont été abordés par des approches cellulaires, allant de la création d'un additionneur binaire au traitement d'image, avec des supports variés, comme par exemple des bactéries.

Il convient, afin de préciser la suite de notre argumentation, de reprendre la taxonomie de Sipper à propos des systèmes informatiques pour y situer notre approche. Un calcul cellulaire est définit par l'équation

$$\text{simple} + \text{largement parallèle} + \text{local} = \text{calcul cellulaire} \tag{1.1}$$

Un calcul *simple* signifie que chacun des calculateurs élémentaires de l'architecture réalise un calcul simple, une opération logique par exemple. Le caractère *largement parallèle* signifie que le nombre de calculateurs élémentaires est de l'ordre de quelque milliers, centaines de milliers, voire plus encore. Enfin, la localité concerne la connectivité. Elle suppose une topologie au niveau des calculateurs de l'architecture (par exemple leur localisation dans un espace). La localité de la connectivité signifie alors que les voisinages définis par les connexions sont les mêmes que ceux définis par la topologie, i.e. que les calculateurs connectés entre eux sont ceux qui sont voisins au sens de la topologie.

1.2.4 Capacités du calcul cellulaire

Les deux voies classiques pour aborder le calcul sont la construction de systèmes physiques qui réalisent des calculs d'une part, et des formalisations et outils mathématiques permettant de définir la calculabilité, de dériver des preuves, d'autre part. Wolfram propose une troisième voie, celle du *calcul expérimental*.

Dans le contexte du calcul cellulaire en effet se manifeste la notion d'émergence. Il s'agit de propriétés qui apparaissent au sein du calcul sans que le calcul ait été spécifiquement conçu pour les faire apparaître. Les phénomènes émergents sont en général difficile à formaliser, le lien

entre la propriété qui émerge et le calcul qui la fait émerger étant souvent complexe à établir, du fait de la non-linéarité des calculs par exemple. C'est pourquoi il est intéressant de considérer le système qui calcule comme un objet sur lequel on peut expérimenter, ce qui est assez inhabituel en mathématiques, mais très fréquent dans le domaine du calcul cellulaire.

En fait, nombre de systèmes naturels peuvent se modéliser sous forme d'un calcul cellulaire. Il s'agit des systèmes composés de multiples constituants interagissant localement les uns avec les autres. Dans ces modèles, les calculateurs élémentaires, i.e. les cellules, représentent les constituants et les connexions entre cellules représentent les interactions au sein du système modélisé. Aussi peut-on modéliser sous cette forme la mécanique des fluides, des réseaux sociaux, l'économie, etc.

Wolfram fait le constat que les mathématiques sont plutôt inadéquates pour exprimer la complexité. Or, contrairement à la mécanique de certains fluides qui se décrit analytiquement par une théorie des champs, il existe des systèmes complexes que l'on ne peut réduire à des équations de champ, et dont la formalisation minimale est celle d'un automate cellulaire, dont on ne peut explorer les propriétés autrement qu'en émulant son exécution. C'est ce qu'évite l'analyse à partir de systèmes d'équation différentielles quand le système modélisé le permet.

Wolfram s'est donc attaché, dans le cas ultra-simplifié des automates cellulaires binaires, à explorer exhaustivement les règles de mise à jour possible des cellules, et à documenter le comportement de l'automate cellulaire résultant dans chacun des cas.

Depuis, on a constaté que les règles conduisant à des comportement intéressants étaient relativement rares, et que certaines d'entre elles conférait à l'automate cellulaire qu'elles génèrent une puissance de calcul équivalente à celle des machines de Turing. Toutefois, comme déjà énoncé, un des enjeux de la compréhension de ces systèmes et bien d'aborder la calculabilité super-Turing.

Aujourd'hui, il existe des composant physiques dont la structure peut être décrite comme celle d'un calculateur cellulaire. Il s'agit par exemple des FPGA et des GPU. Rappelons que le calcul intensif de demain repose sur la possibilité de distribuer nos algorithmes sur ce type de structures, ce qui est un argument de poids pour la promotion du calcul cellulaire.

1.2.5 Systèmes neuronaux cellulaires et complexes pour la modélisation de l'état d'un système dynamique

La plupart des processus naturels s'inscrivent dans l'écoulement temporel. Autrement dit, il s'agit de systèmes dynamiques. L'étude des systèmes dynamiques est un domaine entier des mathématiques. Dans ce contexte, on définit un système dynamique comme un système ayant à chaque instant un état, dont l'évolution dépend de l'état au temps précédent et de l'entrée externe[1] reçue au temps précédent[2]. Le système produit une sortie, parfois considérée comme une observation, qui est fonction de l'état courant et de l'entrée courante. La loi d'évolution ne s'appuie que sur le temps précédent, ce qui fait du système dynamique un système markovien. En revanche, la sortie/observation peut être ambiguë, lorsqu'à deux états pourtant disjoints correspond une seule et même observation. On parle dans ce cas de systèmes dynamiques partiellement observés.

[1] Quand une entrée externe est considéré, on parle de système dynamique non autonome.

[2] La notion de pas de temps précédent renvoie à des systèmes dynamiques à temps discret. Il existe bien entendu des formalisation à l'aide d'équations différentielles pour les systèmes dynamiques à temps continu, formalisations pour lesquelles ce que l'on dit dans ce paragraphe se généralise aisément.

L'étude de systèmes dynamiques via la séquence des observations qu'ils produisent est largement abordée dans la littérature informatique, du fait des enjeux de modélisation de systèmes réels (bourse, etc.) qui motivent ces recherches. Cette étude est rendue encore plus complexe lorsque le système est partiellement observé, puisqu'il faut alors inférer l'état réel à partir du flux d'observations pour comprendre "où le système en est" de sa séquence d'exécution.

Nous nous intéresserons dans ce document aux approches neuronales de ces systèmes, car ces approches, qui reposent sur la notion de réseaux récurrents, n'ont eu de cesse d'étudier cette question, sur la base d'une informatique qui justement correspond à celle à laquelle nous nous proposons de contribuer. Toutefois, à l'heure actuelle, le domaine est surtout dominé par les réseaux de neurones, qui sont certes une mise en œuvre de calculs distribués mais ne sont pas pour autant des approches cellulaires à proprement parler[3]. Les approches cellulaires, de leur côté, restent assez focalisées sur les automates cellulaires, qui modélisent des systèmes dynamiques autonomes. Il nous a donc semblé pertinent d'aborder la question de l'identification des états d'un système autonome d'après la séquence d'observations qu'il produit, mais en sortant du cadre des réseaux neuronaux pour s'aventurer dans le territoire plus restreint des systèmes cellulaires où ces questions n'ont pas vraiment été abordées à notre connaissance.

1.2.6 Problématique de la thèse

Les réseaux de neurones ont su faire la preuve de leur capacité à résoudre des problèmes sur la base d'une architecture connexionniste et distribuée. Certains de ces réseaux sont capables également d'auto-organisation, qui est une propriété émergente de première importance. Dans le cadre des réseaux de neurones, l'étude de systèmes dynamiques a été largement abordée dans la littérature, mettant l'accent sur une notion particulièrement puissante d'apprentissage temporel, mais soulignant également les difficultés posées par ces concepts.

Les réseaux de neurones ne sont toutefois pas parfaitement compatibles avec la définition du calcul cellulaire, définition qui, nous l'avons vu, est un garant de la possibilité des algorithmes cellulaires à profiter des évolutions technologiques futures des calculateurs.

La thèse propose par conséquent de faire le point sur ces différentes approches, et de proposer une architecture cellulaire, au sein de laquelle émergent des phénomènes d'auto-organisation, pour la prise en compte de séquences temporelles issues d'un système dynamique. Le modèle proposé sera étudié expérimentalement, à l'instar de ce que prône Wolfram pour les systèmes complexes, et il est conçu comme pouvant constituer lui-même un composant élémentaire d'une architecture cellulaire beaucoup plus vaste.

[3] Il leur manque l'argument de localité.

1.3 Paradigmes de calcul parallèle : du calcul classique au calcul cellulaire

1.3.1 Introduction

Comme énoncé précédemment, les limitations de l'architecture de Von Neumann résident dans le fait que les données comme le programme doivent partager le même bus, ce qui rend les exécutions intrinsèquement séquentielles. Toutefois, la taille des problèmes informatiques ne cessant d'augmenter, du point de vue des calculs comme du point de vue des données, la recherche en architecture des machines n'a eu de cesse d'améliorer les instanciations de l'architecture de Von Neumann.

En ce qui concerne les données, elles sont fréquemment trop grandes pour pouvoir tenir en mémoire, et il existe de nombreuses techniques (mémoire virtuelle, etc.) qui permettent au programmeur de disposer d'énormément de mémoire, comme si celle-ci était physiquement présente sur la machine qui réalise les calculs. On rencontre ce type de problèmes dans le vaste domaine de la fouille de données, où sont en plus développés des algorithmes incrémentaux plutôt que des algorithmes *batch* qui nécessiteraient de disposer de toutes les données en une fois. D'autres techniques comme la sélection de données sont également employées, mais l'ensemble de ces approches reste difficile à mener dans le cas de très grandes quantités de données.

D'autre part, certains algorithmes sont gourmands en calcul bien plus qu'en données, comme par exemple les méthodes de Monte Carlo où les méthodes où l'on doit explorer systématiquement l'ensemble des paramètres pour trouver une solution optimale. Dans ces problèmes-là, il n'y a pas de raccourci algorithmique, de dépendance entre certains aspects, qui puissent accélérer la résolution.

Pour ces raisons, il a été nécessaire d'envisager des architectures parallèles pour sortir du cadre de l'architecture de Von Neumann, même si la technologie permettait de l'accélérer. L'idée la plus immédiate, du fait de l'existence de réalisations physiques de la machine de Von Neumann, a été de construire des architectures parallèles en mettant ce type de machines en réseau. Toutefois, il existe un autre paradigme de parallélisation, qui consiste à considérer une myriade de petites unités de calcul. Ce paradigme, dont nous avons parlé précédemment, a été suggéré dès les années 40, comme l'illustre par exemple les travaux sur les automates cellulaires où les réseaux de neurones. Cette idée touche également à des considérations philosophiques et physiques sur le monde, puisque certains auteurs considèrent l'univers comme un gigantesque calculateur, cellulaire en se sens que les calculs sont localisés sur l'espace physique, au sein duquel ils interagissent de proche en proche.

Finalement, on peut considérer que tout ce qui se produit dans le monde est une forme de calcul, dont on approchera la nature par une étude informatique des systèmes à grains fin. Dans les paragraphes suivants, nous nous efforcerons de bien introduire ce concept, sur la base des fondements plus théoriques de l'informatique.

1.3.2 Modèles de calcul

Depuis les travaux de Church et Turing visant à définir la calculabilité, dont nous avons parlé précédemment, l'informatique théorique a distingué différents types de complexité des calculs,

invoquant pour ce faire des machines abstraites. La capacité d'un calcul à être réalisé ou non par telle ou telle machine permet de classifier les problèmes.

Ces machines théoriques sont toutes dotées d'états, au sein desquels on distingue un état initial et un ou plusieurs états finaux. Les états sont visités suite à des transitions. Les transitions d'état sont consécutives à la présentation d'une succession d'entrées, d'une séquence. Si la réalisation du calcul conduit la machine dans l'un de ses états finaux, la séquence d'entrée est acceptée par la machine, sinon elle est rejetée. La séquence d'entrée est une instance du problème à traiter, et on parle de *problème de décision* quand il s'agit de déterminer si une machine va ou non accepter une séquence donnée.

Depuis Chomsky, ces problèmes de décision se ramènent à l'étude de la théorie des langages formels, les langages étant regroupés en une successions d'ensembles inclus les uns dans les autres, pour lesquels l'appartenance d'une séquence aux ensembles peut être décidé par un type de machine adéquat.

1.3.2.1 Machines à états finis

Il s'agit d'un graphe orienté où les sommets sont des états, et où les arêtes sont des lettres[4]. Quand une machine est dans l'état courant, la présentation de la prochaine lettre en entrée provoque la transition de la machine vers l'état relié à l'état courant via l'arête étiquetée par cette lettre-là.

Si pour chaque état, il n'y a qu'une seule arête pour chaque lettre, on parle de machine à état déterministe (DFA[5]). Les séquences que l'on peut reconnaître par une DFA forment la classe des langages réguliers. Une classe importante de machine à états finis sont les *transducers*. Ces machines génèrent en sortie une séquence de même taille que le mot en entrée. Pour ce faire, dans le cas des machines de Mealy, on définit pour la machine une fonction qui à un couple état-entrée associe une sortie. Cette fonction est appliquée à l'état courant et à l'entrée courante pour former la sortie.

Les machines à états finis peuvent reconnaître le langage 0^n1^m, mais ne peuvent pas n'accepter que les mots pour lesquels $m > n$. Cet exemple illustre le besoin de machines plus complexes pour reconnaître des langages[6] plus difficiles à décrire.

1.3.2.2 Automates à pile

Les automates à pile permettent de reconnaître des langages non contextuels (e.g a^nb^n). Il s'agit d'un automate à états finis auquel on adjoint une pile infinie. Les transitions peuvent empiler ou dépiler des symboles sur la pile. Toutefois, le langage $a^nb^nc^n$ ne peut pas être reconnu par un automate à pile. Ce type de langage, dit langage contextuel, requiert la puissance d'une machine de Turing.

[4] Conformément à la théorie des langages, nous parlerons d'entrées littérales pour les machines, la séquence d'entrée formant ainsi un mot.

[5] Deterministic Finite Automaton

[6] des ensembles de mots.

1.3.2.3 Machines de Turing et ordinateur de Von Neumann

Les machines de Turing sont définies par une machine à états finis à laquelle on ajoute une bande infinie[7]. La séquence d'entrée est écrite sur la bande. La machine dispose d'une tête de lecture qui pointe sur une case de la bande. La transition est déterminée par l'état courant et la valeur de la bande au niveau de la case pointée. La transition consiste à écrire une autre valeur sur la bande, déplacer la tête de lecture d'un cran à gauche ou à droite, et changer d'état.

Les langages reconnus par les machines de Turing sont dits langages récursifs. Les problèmes de décision associés (décision d'appartenance) deviennent assez épineux, car on ne sait pas toujours déterminer si la machine conduira où non à une décision d'appartenance (voir problèmes de décidabilité).

La grande force des machines de Turing est l'existence d'une machine de Turing universelle. En effet, pour une machine donnée, ces règles d'activation peuvent être codées et écrites sur une bande, suivies de la séquence à reconnaître. La machine de Turing universelle est une machine capable de lire ces règles, et d'appliquer le calcul correspondant à la donnée qui les suit sur la bande.

On peut déduire de cette propriété que les règles régissant la machine de Turing universelle peuvent guider la conception d'une machine physique réelle, pouvant réaliser un calcul décrit en mémoire. La mémoire (la bande), devient donc le siège de l'hébergement des données mais aussi d'une description du calcul à réaliser, en d'autres mots, du programme. C'est sur cette base que Von Neumann a pu définir l'architecture qui préside à la construction des ordinateurs, avec les problèmes de séquentialité dont nous avons parlé.

1.3.3 Modèles d'exécution massivement parallèles

1.3.3.1 Définitions et terminologie

Les architectures parallèles peuvent se décomposer en deux catégories :

- les architectures à gros grains, qui sont la majorité des architectures parallèles, et consistent en la mise en réseau de processeurs complexes, i.e. en la mise en réseau de machines de Von Neumann,
- les architectures à grain fin, qui sont la mise en connexion de calculateurs bien plus simples, mais en plus grand nombre. Ces architectures sont bien moins répandues, et sont davantage un paradigme prometteur. Le calcul cellulaire est une sous-classe des architectures à grain fin.

La distinction entre les deux catégories nous semble relever de la logique du calcul élémentaire (simple vs. complexe) plutôt que du nombre de processeurs lui-même, puisque l'on trouve aujourd'hui des architectures à gros grains pouvant compter un millier de processeurs.

1.3.3.2 Synchronisation

La synchronisation des calculs est un point crucial au sein des architectures parallèles.

[7] suivant les deux directions.

On parlera de système synchrone si chaque calcul est sujet à une exécution pas de temps par pas de temps, et que ces pas de temps sont effectués en même temps sur chacun des processeurs, chacun n'ayant connaissance que des informations fournies au pas de temps précédent par les processeurs auxquels il est connecté.

Les systèmes asynchrones sont ceux pour lesquels les pas de temps sont réalisés sans synchronisation au niveau global de l'architecture. On pourra parfois vouloir s'assurer que tous les processeurs aient exécuté un pas de temps avant que ne soit autorisé pour chacun d'eux l'exécution du pas de temps suivant.

1.3.4 Modèles à gros grain

L'idée de ce paragraphe est de rappeler les grands principes de constitution des architectures parallèles classiques, afin de mieux articuler plus tard les différences avec les architectures qui nous concernent. Ces architectures parallèles classiques diffèrent entre elles pas la nature du couplage des processeurs, et les ressources qu'ils partagent. Ainsi, selon les cas, les processeurs peuvent être très fortement couplés, ou au contraire très peu.

1.3.4.1 Multiprocesseurs fortement couplés

Le cas de multiprocesseurs fortement couplés recouvre celui des machines qui partagent la même mémoire, voire le même bus. On peut l'étendre au cas où la mémoire est distribuée. Dans ce cas, seuls des groupes de processeurs partagent entre eux la mémoire, mais ces groupes communiquent via un bus partagé dédié. Ce couplage fort se traduit également au niveau du système d'exploitation, qui a connaissance de l'architecture parallèle et qui lui est, lui aussi, dédié.

La taxonomie de Flynn permet de regrouper ces architectures parallèles en trois grandes classes :

- Single Instruction, Multiple Data (SIMD). Chaque processeur applique la même séquence d'instructions à des données différentes.
- Multiple Instruction, Multiple Data (MIMD). Il s'agit du cas typique d'un réseau de machines.
- Multiple Instruction Single Data (MISD). C'est un modèle plutôt rare, utilisé par exemple dans le cas d'architectures redondantes pour la robustesse aux pannes matérielles.
- Single Instruction, Single Data. Il s'agit d'une machine de Von Neuman classique, non parallèle en fait.

Outre les problèmes matériels de partage de mémoire, la programmation de ces architectures parallèles n'est pas forcément simple.

1.3.4.2 Multiprocesseurs faiblement couplés

Les machines réparties sur internet, ayant des architectures logicielles, des périphériques, voire des systèmes d'exploitation différents, peuvent être vues comme un calculateur parallèle à partir du

moment où elles concourent à l'exécution d'un même calcul. Le lien entre les différentes machines peut se faire via un *middleware*. Là aussi, la programmation de ces systèmes n'est pas toujours facile.

1.3.4.3 Systèmes hybrides

L'arrivée des GPU[8] sous la pression de la demande du domaine des jeux vidéo modifie profondément le paysage des machines parallèles. Ces machines ne sont en effet plus dédiées aux calculs graphiques, car elles sont devenues au cours des évolutions de plus en plus paramétrables, pour être largement programmables aujourd'hui. Il s'agit de machines majoritairement SIMD, qui travaillent de concert avec un CPU.

Aujourd'hui, il existe des super-calculateurs qui sont des grappes de PC dotés de GPU, ce qui en fait des architectures hybrides. Même si des langages de programmation des GPU de plus en plus évolués voient le jour, la programmation de ces architectures reste délicate.

1.3.5 Modèles distribués à grain fin

Même si l'on dispose aujourd'hui de machines parallèles avec de nombreux processeurs, pouvant être connectés via des bus très rapides, l'on est encore loin, en terme de parallélisme, des architectures des systèmes nerveux. En fait, ces derniers reposent sur l'interconnexion d'un nombre de neurones dépassant de plusieurs ordres de grandeur le nombre de processeurs des plus grosses machines parallèles d'aujourd'hui, mais ces neurones réalisent des fonctions relativement simples, en parallèle. Ce type de calcul est très fréquent en biologie.

Nous souhaitons insister sur le fait que les modèles distribués qui nous concernent se réfèrent à des paradigmes de programmation et non à une réalisation physique. Il s'agit de concevoir des algorithmes qui s'expriment comme l'exécution parallèle de calculs simples interconnectés, ce parallélisme pouvant très bien être simulé sur des machines séquentielle dans le but d'étudier la dynamique de l'exécution de ces algorithmes.

1.3.5.1 Automates cellulaires

Un automate cellulaire est une machine composée de cellules disposées suivant une topologie, usuellement en ligne ou en grille 2D. Chaque cellule se comporte comme un automate à états finis, avec uniquement deux états, notés 0 et 1. L'entrée selon laquelle sont déclenchées les transitions est la configuration des états des cellules avoisinantes. Ainsi, même si chaque cellule perçoit des activités qui lui sont externes puisqu'il s'agit de l'état de ses voisines, l'ensemble des cellules n'est pas soumis à une entrée externe, ce qui en fait un système dynamique discret autonome. L'évaluation des automates cellulaires est synchrone.

Le jeu de la vie de Conway est certainement l'automate cellulaire le plus connu. Il s'agit d'un automate dont les cellules sont disposées suivant une grille 2D. Il a été démontré qu'il était suffisamment riche pour que l'on puisse y coder, en déterminant une configuration bien précise d'activations

[8] Graphical Processing Unit.

initiales, une machine de Turing. Cette propriété est également vraie pour des automates cellulaires aux cellules disposées en ligne.

Utiliser les automates cellulaires comme calculateurs universels est très inefficace, mais l'idée est de montrer qu'ils ont une puissance d'expression très large. Toutefois, pour qu'un automate cellulaire exhibe une dynamique riche, et donc expressive, il faut que les règles de transition d'état des cellules qui le régissent autorisent une grande variabilité de motifs dans les distributions d'états, sans être pour autant chaotiques. De tels systèmes sont décrits comme évoluant à la frontière du chaos, et on montre que les règles qui conduisent à cette propriété sont rares, et donc difficiles à trouver.

L'intérêt des automates cellulaires réside dans la promotion qu'ils font d'un calcul massivement parallèle, impliquant des connexions locales. Nous avons déjà évoqué qu'ils servent de ce fait de modèle à une vue de l'Univers comme un processus de calcul, ce qui a des répercussions aussi bien en physique qu'en philosophie. De plus, c'est au sein de ce type de structures que peuvent émerger des phénomènes complexes, comme l'auto-organisation, ce qui montre que l'on peut observer une complexité sans qu'il soit nécessaire que les mécanismes qui lui donne naissance soient, eux, complexes.

1.3.5.2 Réseaux de neurones comme modèles à grain fin

A l'instar des automates cellulaires, les réseaux de neurones sont un paradigme de calcul à grain fin inspiré de la nature. Ils sont constitués d'unités de calcul simples[9], mises en connexion. Les activités des neurones ne sont pas booléennes mais scalaires, et sont obtenues comme une fonction de la somme, pondérée par les poids hébergés par les connexions, des unités voisines.

Une autre caractéristique différencie fondamentalement les réseaux de neurones des automates cellulaires. Il s'agit de la notion d'apprentissage. En effet, les influences d'un neurone sur un autre sont dues au poids de la connexion qui les relie, et ce poids est sujet à des modifications consécutives aux activations des neurones connectés. A l'instar du cerveau, les réseaux de neurones sont des systèmes adaptatifs.

La topologie des réseaux de neurones est moins contrainte que celle des automates cellulaires, les connexions pouvant être quelconques, sans qu'il soit nécessaire de définir une topologie de localisation des unités. Il existe ainsi plusieurs connectivités typiques décrites dans la littérature, comme la connectivité totale, mais surtout comme la connectivité en couches successives[10].

Le perceptron multi-couches est l'archétype du réseau de neurones utilisé en apprentissage automatique supervisé, mais il en existe des variantes où les couches les plus élevées sont reconnectées aux couches inférieures, formant ainsi des réseaux à couches récurrents, sur lesquels nous reviendrons.

Par réseaux de neurones, on entend en général le modèle mathématique du neurone formel introduit par MacCulloch et Pitts, qui est au cœur des perceptrons. Nous nous intéressons principalement à ces modèles dans la mesure où ils offrent un paradigme de programmation à grain fin, laissant de côté leur capacité à effectivement modéliser la biologie des véritables neurones. Mentionnons néanmoins l'existence de modèles plus élaborés, que nous pouvons, eux aussi, considérer

[9] les neurones

[10] Réseaux à couches *feed-forward*

sous l'angle des paradigmes de programmation à grain fin.

Parmi ces modèles plus réalistes, on trouve l'immense littérature sur les neurones impulsionnels. En effet, le neurone formel de MacCulloch et Pitts manipule des scalaires, qui modélisent les fréquences de trains d'impulsions électriques sur les fibres nerveuses. Cette approche fréquentielle n'autorise pas de considérer les phénomènes de synchronisation d'impulsions par exemple. Nous ne développons pas ce sujet dans nos travaux de thèse, car notre travail n'y est pas rattaché. Cela dit, les évolutions futures des travaux sur les neurones impulsionnels devraient certainement fournir de nouveaux paradigmes à l'informatique. Sans rentrer dans une modélisation impulsionnelle, il existe des raffinements du modèle de MacCulloch et Pitts qui rendent compte de la dynamique des neurones en proposant des équations différentielles temporelles pour calculer les activités des neurones.

L'universalité[11] des réseaux de neurones a été établie, à condition de ne pas se limiter aux architectures en couches. Il a même été établi que les réseaux de neurones à activités continues pouvaient être super-Turing. Ce résultat théorique est toutefois discutable, car il repose sur la précision infinie des nombres scalaires, précision qui n'existe pas au sein de dispositifs physiques.

Autant on peut qualifier les réseaux de neurones comme des systèmes à grain fins, autant il est plus délicat d'affirmer leur caractère parallèle. En effet, même si le paradigme neuronal de base, à savoir l'interconnexion de neurones formels, n'exclut pas une évaluation parallèle, à l'instar des automates cellulaires, il s'avère que la plupart de réseaux de neurones exigent l'activation successive de couches. Une part non négligeable des algorithmes à base de réseaux de neurones est donc fondamentalement séquentielle.

Cette restriction est principalement valable pour les réseaux de neurones à couches, mais tous ne sont pas de cette nature, ce qui réconcilie les réseaux de neurones avec la notion de parallélisme. Notons par exemple les réseaux de Hopfield, qui implémentent une mémoire adressable par le contenu, ou les cartes auto-organisatrices sur lesquelles nous reviendrons, qui réalisent un apprentissage non-supervisé.

Déterminer l'architecture neuronale adéquate pour une tâche donnée reste difficile, même si l'universalité des réseaux neuronaux affirme que cette architecture existe. Il existe quelques approches abordant ce problème, dont des approches évolutionnaires comme celles proposées par Randall Beer.

Enfin, mentionnons ici l'existence de travaux de parallélisation des algorithmes de réseaux de neurones, sur clusters de PC ou sur FPGA. Il s'agit fréquemment de parallélisation de perceptrons multi-couches, et la séquentialité que nous avions soulignée pour ces réseaux-là pose effectivement des problèmes.

1.3.5.3 Réseaux de neurones cellulaires

Les réseaux de neurones cellulaires sont un paradigme intermédiaire entre les réseaux de neurones et les automates cellulaires. Leur caractère cellulaire est motivé par la contrainte d'implanter ces réseaux sur la surface de dispositifs physiques. Les neurones de ces réseaux sont des systèmes analogiques réduits, qui s'échangent des signaux via un motif de connexions locales. Les neurones

[11] C'est-à-dire leur capacité à être aussi puissants que les machines de Turing

sont dynamiques, implémentant des équations différentielles temporelles[12], couplées spatialement via le motif de connexion.

Ces systèmes sont appliqués au traitement d'image, à la résolution analogique d'équations différentielles couplées. Ils sont efficaces puisque le parallélisme à grain fin n'est pas simulé, mais effectivement réalisé par le composant.

Ces systèmes ont la puissance de Turing si l'on autorise une adaptation du motif de connexion, ce qui est faisable sur les derniers développements des réseaux de neurones cellulaires.

1.3.5.4 Un nouveau concept de calcul

Les modèles à grain fin que nous avons évoqués constituent bel et bien un concept de calcul, de programmation, radicalement différent de l'algorithmique classique et des architectures matérielles de type Von Neumann. Cette différence se situe à deux niveaux, au niveau structurel et au niveau fonctionnel.

Au niveau structurel, on trouve ce que nous avons déjà mentionné, à savoir la simplicité des calculateurs élémentaires dans le cas du grain fin, qui s'oppose à la puissance des processeurs utilisés dans les architectures parallèles à gros grain. Une autre différence structurelle réside dans la circulation de l'information.

Au niveau fonctionnel, dans les architectures à grain fin, le goulet d'étranglement que constitue le bus où circulent à la fois les instructions et les données n'existe pas. Les calculs et les données se font au même endroit, et les deux sont répartis.

Enfin, les architecture à grain fin peuvent adapter le calcul qu'elle réalisent au flux de donnée qui les traverse, via des capacité d'apprentissage que nous avons citées dans le cas des réseaux de neurones.

1.3.6 Calcul cellulaire

Le calcul cellulaire est un cas particulier de calcul à grain fin. À la caractéristique d'être constitué de nombreux processeurs simples connectés s'ajoute la notion de localité des connexions, par rapport à une topologie qui décrit la position des processeurs locaux dans un espace, et l'exigence d'une évaluation strictement décentralisée.

Afin de préciser cette notion de localité, il convient de distinguer deux formes de localité. La localité fonctionnelle et la localité topographique. La localité fonctionnelle signifie qu'un processeur n'est connecté qu'à un nombre faible d'autre processeurs, faible comparé au nombre total de processeurs. La localité topographique ajoute que ces quelques processeurs soient en plus des voisins, au sens topographique.

La localité topographique, exigée pour que le calcul à grain fin puisse être qualifié de cellulaire, à plusieurs avantages. Premièrement, elle facilite l'implémentation physique du calcul. Deuxièmement, lorsque l'on change la taille de l'automate, il n'y a pas d'explosion combinatoire liée à l'augmentation des connexions.

Le terme cellulaire est emprunté à la biologie, puisque les cellules d'un organisme se comportent comme des agents interagissant localement les uns avec les autres. Même si le terme cellulaire

[12] La présence de condensateurs ajoute des termes dérivés dans la production des signaux électriques.

est parfois utilisé pour des architectures multi-cœurs par IBM, nous le réserverons aux systèmes vérifiant l'équation 1.1 introduite page 4.

Suivant cette restriction, il est notable que les automates cellulaires ainsi que les réseaux de neurones cellulaires entrent naturellement de le cadre du calcul cellulaire, mais que la plupart des réseaux de neurones, eux, en sont exclus, du fait des connexions totales entre couches qui ne respectent pas le principe de localité topographique et de l'évaluation des couches dans un ordre précis.

Notons qu'il existe un domaine en émergence, le calcul amorphe, qui consiste à définir des calculateurs répartis aléatoirement sur un espace physique, espace qu'ils utilisent pour communiquer avec leurs voisins, via des mécanisme de diffusion. Ces approches-là sont elles aussi cellulaires. De plus, contrairement aux automates cellulaires, les calculateurs s'évaluent de façon indépendante, c'est-à-dire de façon asynchrone. Les calculs modélisés sont par exemple des calculs de réparation de structure, de routage d'information, réalisés de façon strictement distribuée et non supervisée. Ces approches, du fait qu'elles sont conçues pour être adaptatives sans aucun pré-requis sur la répartition exacte des processeurs, sont robustes aux dommages de structure. Il existe également des variantes où les particules sont mobiles, ce qui revient à reconfigurer le graphe des connexions, les relation de localité étant modifiées par le mouvement des particules.

1.3.6.1 Décentralisation au sein du calcul cellulaire

Nous avons parlé précédemment des deux types de localités que l'on rencontre dans les systèmes à grain fin, localité fonctionnelle et localité topographique. Cette notion de localité rejoint la notion de décentralisation des calculs.

La décentralisation est la propriété du calcul de se décomposer en plusieurs exécutions sans que celles-ci n'aient à partager une ressource commune, de type mémoire partagée ou variable globale typiquement. Le fait d'exiger qu'une horloge commune synchronise les calculs est contraire à la propriété de décentralisation.

La plupart des approches de type réseaux de neurones se ramènent à du calcul matriciel. Il en résulte que l'algorithme neuronal, une fois ramené à des opérations matricielles, n'est plus décentralisé. Que ce calcul matriciel soit une façon de simuler le parallélisme d'un système décentralisé n'est pas toujours vrai selon les architectures neuronales. De toute façon, le calcul matriciel utilisé pour évaluer les réseaux de neurones correspond à une évaluation synchrone des neurones, synchronicité qui est déjà une entorse au principe strict de décentralisation.

1.3.6.2 Propriétés architecturales

Un modèle cellulaire est nécessairement doté d'une topologie, c'est-à-dire que ses cellules sont liées à une position dans un espace muni d'une notion de voisinage.

La topologie est déterminée à la construction du modèle cellulaire. De même, les connexions entre cellules sont déterminées a priori, suivant la topologie. Dans le cas des automates cellulaires, les connexions sont de simples moyens d'accès à l'activité des unités voisines, alors que dans le cas des réseaux de neurones cellulaires, les connexions peuvent héberger un poids.

Les propriétés temporelles des modèles cellulaires sont également définies à la construction.

Cette définition passe par le choix d'un mécanisme d'évaluation, qui peut être synchrone ou asynchrone, l'asynchronisme pouvant s'implémenter de différentes façons dans le cas où une évaluation asynchrone est retenue pour le modèle.

Les cellules sont usuellement toutes identiques[13], de même que les connexions. Toutefois, il existe des contraintes aux bords des grilles qui peuvent ou non se résoudre par une connexions suivant un tore (les unités sur un bord sont voisines de celles du bord opposé).

Les sorties des cellules peuvent être soit discrètes, comme dans le cas extrême des automates cellulaires à deux états d'activation, soit continues, comme dans le cas de réseaux de neurones cellulaires.

Le comportement dynamique des unités dépend des règles d'évaluation impliquées. Nous avons déjà évoqué la question du syncrhonisme/asynchronisme dans l'évaluation, qui a une influence essentielle sur la dynamique globale du système cellulaire, mais le caractère déterministe ou stochastique de la règle de mise à jour est aussi prépondérant dans la détermination de cette dynamique. À ceci s'ajoute la modélisation du temps, qui est souvent discrète, même s'il existe dans la littérature des modèles à temps continu, comme par exemple les réseaux de neurones cellulaires.

1.3.6.3 Propriétés opérationnelles

Programmer un modèle cellulaire consiste à ajuster ses propriétés architecturales de sorte à ce que l'automate réalise la tâche que l'on souhaite. Du fait de la difficulté de maîtriser les phénomènes complexes d'émergence qui surviennent au sein de ces structures, cet ajustement peut être extrêmement difficile. Des approches évolutionnistes ont été proposées pour résoudre ce problème de spécifications.

Les systèmes cellulaires sont censés être robuste aux dégradations, ou plus précisément à se comporter d'une façon dégradée qui reste élégante. Bien que cette propriété n'ait pas été confirmée dans les modèles actuels[14], le fait que le cerveau soit un calculateur cellulaire et qu'il soit extrêmement robuste est l'un des arguments en faveur de la robustesse aux pannes que l'on peut attendre de ces approches.

Mentionnons enfin les extensions du calcul cellulaire à des systèmes adaptatifs, capables d'apprentissage. Ces approches, que l'on rencontre pour les automates cellulaires comme pour les réseaux adaptatifs, sont une alternative au tâtonnement ou aux approches évolutionnistes lors de la définition de l'architecture.

1.3.6.4 Ce que l'on peut attendre du calcul cellulaire

Il est courant en informatique de constater que, du moment que les modèles de programmation sont tous Turing-équivalents, ce qui préside au choix de l'un d'entre eux pour tel ou tel problème est qu'il peut-être plus aisé de le résoudre avec une architecture plutôt qu'une autre. Dans le cas des approches cellulaires, si l'on s'en réfère au cerveau par comparaison aux machines de Von Neumann, il semblerait bien qu'il y ait des problèmes faciles pour les humains et difficiles pour les machines, comme les problèmes de reconnaissance de visage. C'est pour ces problèmes-là que l'on

[13] Il existe des exceptions, conduisant à des modèles plus riches de systèmes cellulaires.

[14] Rappelons que la programmation cellulaire n'en est qu'à ses débuts.

peut raisonnablement espérer une solution élégante par des approches cellulaires, puisque c'est ce que le cerveau met en œuvre.

L'informatique traditionnelle semble adaptée aux tâches où l'on résout un problème par application successive de règles arithmétiques, mais, contrairement aux systèmes nerveux des animaux, elle est en difficulté dès qu'il s'agit de prendre des décision en étant aux prises avec un flux perceptivo-moteur permanent, incertain, et bruité. C'est ce type de tâche qui nous paraît solliciter une recherche sur les systèmes cellulaires.

Le domaine du calcul cellulaire est encore balbutiant, et peu étudié car la démarche informatique classique vise à décomposer les problèmes en une séquence de tâches imbriquées, avant d'essayer de reconnaître dans les problèmes des parties autonomes qui coopèrent. La science des systèmes cellulaires est donc encore loin de proposer des solutions clé-en-main pour les problèmes informatiques d'aujourd'hui.

1.3.7 Conclusion

Les systèmes à grain fin et les systèmes cellulaires sont issus d'une formalisation de la façon dont la nature calcule, que ce soit au niveau des système nerveux ou au niveau plus fondamental de la physique de l'univers.

La question pourrait être de savoir si l'on peut réaliser par les systèmes à grain fin les mêmes calculs que sur les machines classiques, de type Von Neumann, ou clusters de machines de Von Neumann. Cette question n'est pas la bonne, puisque les deux approches sont Turing-équivalentes. La question est plutôt de cerner les problèmes qui pourraient, à terme, être résolus élégamment par les approches à grain fin, ou, dit autrement, de comprendre quelle informatique découle le plus spontanément, le plus naturellement, de ces modèles.

L'existence d'un calculateur à grain fin, conçu par l'évolution, dans le crâne de chacun de nous, est un réel moteur pour persévérer à comprendre cette informatique-là, tant les tâches résolues par le cerveaux sont difficiles à reproduire par les approches informatiques classiques.

À ceci s'ajoute que ce sont ces architectures-là, qui lorsqu'elle sont cellulaires, peuvent prétendre à bénéficier des évolutions technologiques futures, où le progrès viendra de la capacité des algorithmes à s'étaler sur l'espace physique pour s'y exécuter en parallèle. Il n'est certainement pas fortuit que la nature ait opté pour un système cellulaire, disposant d'une technologie lente[15] et du volume limité du corps des animaux.

1.4 Encodage du temps dans les réseaux de neurones dynamiques

Les automates cellulaires et les réseaux neuronaux cellulaires ont été utiliser pour modéliser des phénomènes temporels. Ils faut toutefois distinguer entre les cas où le modèle cellulaire est en interaction avec un flux d'entrée des cas où il se comporte en système dynamique autonome (par exemple quand un réseau de neurones cellulaire résout une équation différentielle).

Les réseaux de neurones artificiels ont également été utilisés pour modéliser des phénomènes temporels, mais nous avons vu que ces modèles à grain fin ne sont pas cellulaires. En revanche, ils

[15] Les signaux électriques dans le cerveau ont une vitesse de quelques mètres par seconde au maximum.

sont plus fréquemment utilisés que les modèles cellulaires pour modéliser des systèmes dynamiques non autonomes.

Il reste toutefois le problème de l'apprentissage au sein des structures à grain fin, problème surtout décliné dans le cadre des réseaux de neurones, par des méthodes de type rétro-propagation du gradient d'erreur. Ces méthodes ne sont pas compatibles avec un calcul local et distribué, et passent difficilement à l'échelle quand le nombre de neurones augmente, puisque le nombre de poids explose de façon combinatoire.

Nous définissons dans ce qui suit les notions de séquences et série temporelles, puis passons en revue les différentes approches neuronales du traitement de séquence, laissant toutefois pour le paragraphe suivant les approches fondées sur les cartes auto-organisatrices.

1.4.1 Traitement de données temporelles

Les données temporelles sont des données pour lesquelles on dispose d'une information sur la date d'acquisition. Le traitement de cette information peut se faire en ligne, auquel cas l'algorithme reçoit les données une par une, et réalise un calcul à chaque nouvelle donnée reçue. D'autres méthodes, dites *batch*, consistent à collecter les données dans un premier temps pour ensuite effectuer un calcul sur l'ensemble des données collectées.

Il nous paraît nécessaire de faire la distinction entre les séries temporelles et les séquences temporelles. Les séries temporelles sont l'échantillonnage d'un phénomène temporel[16], la période d'échantillonnage étant dépendente du problème étudié. En revanche, les séquences temporelles se réfèrent à des données où le label temporel désigne un ordre, comme par exemple les mots d'une phrase ou les nucléotides d'une séquence d'ADN. Alors que ces données sont d'un point de vue informatique de même nature, la sémantique du label temporel est différente dans les deux cas, il ne s'exprime en secondes que dans le cas des séries temporelles.

1.4.1.1 Traitement de séries temporelles

Le traitement de séries temporelles consiste en général en une classification des séries, ou de parties de séries, mais aussi en une prédiction de l'élément de la série à venir compte tenu des éléments déjà reçus.

La prédiction de l'élément à venir est un problème de régression, le nombre d'éléments passés qu'il est nécessaire d'avoir à disposition dépend de la nature du problème qui génère la séquence. Parfois, on souhaite simplement prédire, dans le cas de séquences scalaires, si le prochain élément correspond à une augmentation du signal ou à une diminution. Le problème de prédiction, dans ce cas, se réduit à un problème de classification.

Si la fonction de prédiction est correcte, on peut l'utiliser pour modéliser le phénomène qui génère la séquence. Il suffit pour cela de prédire successivement les valeurs, et d'utiliser les valeurs prédites jusqu'ici pour nourrir la fonction de prédiction qui ainsi générera les valeurs futures.

[16] à temps continu donc

1.4.1.2 Traitement de séquences temporelles

On retrouve pour les séquences temporelles les problématiques de prédiction, comme pour les séries temporelles, et par conséquent la notion de génération de séquence. Une séquence étant, contrairement aux séries temporelles, finie, on trouve également dans la littérature des problématiques de reconnaissance, qui consistent à dire si une séquence remplit certains critères, appartient à une classe de séquences.

Parmi les techniques impliquant des séquences, citons la prise de décision séquentielle. Ils s'agit d'orienter une séquence d'action vers un but à atteindre. Le cadre du traitement de séquences temporelles est propice à l'application de techniques d'apprentissage supervisé et non-supervisé, et a donc naturellement été abordé par les approches à base de réseaux de neurones.

1.4.1.3 Ambiguïté des éléments d'un séquence

Les éléments d'une séquence, lorsqu'ils se répètent, peuvent être cause d'ambiguïtés, surtout si le but est de s'en servir pour prédire les éléments à venir. Dans la séquence CONFONDRE, la sous-séquence ON est suivie tantôt d'un F, tantôt d'un D. Déterminer ce qui suit le N requiert de considérer les deux pas de temps précédents. On dit que la séquence CONFONDRE est alors de profondeur 3. En d'autres termes, le processus qui génère la séquence à partir de l'élément courant n'est pas markovien, c'est celui qui génère l'élément suivant de la séquence à partir des trois derniers éléments qui l'est, d'où la profondeur 3.

1.4.2 Réseaux de neurones pour le traitement de séquences temporelles

Notre travail porte sur les systèmes cellulaires temporels. Même si les réseaux de neurones ne sont pas cellulaires, ils n'en reste pas moins des systèmes à grain fin, et les modifications qui leur ont été apportées pour prendre en compte les données temporelles sont intéressantes vis-à-vis de nos travaux.

1.4.2.1 Temps interne et externe

Rappelons ici que l'activation d'un réseau de neurones, en elle-même, s'effectue suivant plusieurs pas de temps. C'est le cas pour les perceptrons, qui ont une phase feed-forward puis feed-back de couches en couches, mais aussi pour la relaxation d'un réseau de Hopfield, ou les mécanismes de compétition d'une carte auto-organisatrice. Nous qualifierons ce temps-là de temps interne, lié à l'activation de l'algorithme, même quand il traite des données statiques.

Ce qui nous intéresse ici est d'avantage un temps externe, présent dans les données, qui requiert parfois d'adapter la structure des algorithmes neuronaux.

1.4.2.2 Les différentes représentation du temps dans les réseaux de neurones

Les adaptations des réseaux de neurones aux données temporelles consistent à modifier explicitement les réseaux dans ce but. Parmi les modifications, on trouve l'ajout de tampons circulaires qui gardent en mémoire les n derniers éléments de la séquence. Cette méthode convertit le caractère

temporel des entrées en un vecteur spatial, appliquant alors les algorithmes classiques sur ces données temporelles spatialisées. La taille du tampon doit alors être supérieure à la profondeur de la séquence, profondeur qui ne peut pas être nécessairement déterminée a priori.

Une autre technique consiste à ajouter des propriétés temporelles aux activations des neurones, comme dans le cas de neurones intègre-et-tire, ou aux poids, mais la technique la plus utilisée est celle qui consiste à considérer l'état interne du réseau comme représentatif de l'état courant de la séquence. Il s'agit d'ajouter dans ce cas des connexions récurrentes, avec délai, au réseau, de sorte que l'activation courante intègre les activations aux pas précédents.

1.4.2.3 Composants temporels des réseaux de neurones

Le traitement du temps suppose de pouvoir considérer l'entrée courante dans le contexte de ce qu'il vient de se passer. Le réseau doit donc, d'une façon ou d'une autre, être doté de mécanismes de mémorisation permettant le stockage en mémoire de ce contexte.

Une autre caractéristique des réseaux de neurones de type perceptron est qu'ils apprennent de façon supervisée, en corrigeant la valeur qu'ils prédisent d'après la valeur qui était attendue. Dans le cas des séquences, il s'agit de prédire l'élément courant, sur la base du contexte mémorisé précédemment introduit.

Dans les approches neuronales donc, on retrouve ces deux composants : un système de mémorisation qui construit un contexte temporel, et un système de prédiction qui s'appuie sur ce contexte.

1.4.3 Réseaux feed-forward pour le traitement de séquences temporelles

Les réseaux feed-forward sont typiquement les perceptrons multi-couches, appliqués en ce qui concerne le temps à des données temporelles spatialisées, comme par exemple un mot pris globalement et non comme une séquence de lettres. Leur apprentissage est supervisé, et se base classiquement sur des techniques de rétro-propagation du gradient d'erreur, de couche en couche.

D'un point de vue informatique, ces techniques n'apporte rien de nouveau du fait de la présence de données temporelles. Notons toutefois qu'en tant que perceptron, ces méthodes ne peuvent être considérées comme cellulaires.

1.4.4 Réseaux dynamiques avec lignes à retard

1.4.4.1 Lignes à retard standard et filtrage

Il s'agit d'un tampon de type fille d'attente. À chaque pas de temps, l'entrée courante est insérée dans la file et la perception la plus ancienne de la file en est retirée. Le réseau établit donc sa prédiction sur la base de l'historique récent des entrées. Il existe également des systèmes de file d'attente qui appliquent des opérations aux entrées au fur et à mesure de leur circulation dans la file.

Une autre modification de l'architecture du réseau proposée dans le cadre de réseaux à lignes à retard est le partage des poids. En effet, les lignes à retard ajoutent des dimensions aux entrées, et multiplient ainsi le nombre de poids, au risque de sur-apprendre. Le partage de poids consiste

à imposer que des groupes de poids soient identiques entre eux. Cette technique est toutefois une entorse de plus au paradigme de calcul distribué, puisque le maintien de valeurs égales entre ces groupes de poids est un mécanisme de supervision global des calculs.

Il existe également, pour constituer un contexte temporel sur lequel baser la prédiction, des approches à base de filtrage temporel du flux d'entrée. Mais quoi qu'il en soit, les méthodes à base de lignes à retard ou de filtrage ne sont pas cellulaires, du fait même de l'utilisation d'un perceptron.

1.4.5 Réseaux de neurones récurrents comme modèles à états

Plutôt que d'utiliser une structure de type lignes à retard pour représenter le contexte temporel du flux d'entrée, l'idée développée dans le cadre des réseaux récurrents consiste à considérer que l'activation du réseau elle-même peut stocker le contexte temporel. Pour ce faire, on ajoute à l'architecture classique des connexions qui transmettent l'activité distante avec un pas de temps de retard. Ainsi, l'activation courante du réseau dépend de son activation passée, qui elle même dépendait de l'activation encore précédente, etc. Même si les retards ne sont que d'un pas de temps, l'état instantané du réseau peut potentiellement dépendre d'un passé assez lointain, ce qui permet d'envisager la construction d'un contexte pour des séquences très profondes.

La plupart des réseaux de neurones récurrents sont à temps discret. On peut les diviser en deux classes, d'une part ceux qui ont des connexions symétriques[17], pour lesquels on sait ramener la dynamique à la minimisation d'une fonction d'énergie, et les réseaux aux connexions non symétriques. Parmi les premiers, on trouve les réseaux de Hopfield et les machines de Boltzmann. Les seconds sont très nombreux et l'on trouve parmi eux les versions récurrentes des perceptrons et les approches de type réservoir.

L'idée générale, dans la plupart des cas, est que le réseau de neurone doté de connexions à délai puisse apprendre la dynamique des entrées, et pour certains modèles, il a été montré qu'ils étaient des approximateurs universels des systèmes dynamiques dont ils reçoivent le flux d'entrée[18].

1.4.5.1 Les réseaux basés sur le perceptron multi-couches

Les réseaux de neurones d'Elman et Jordan sont des perceptrons, avec par conséquent une structure en couche feed-forward. À cette structure de perceptron classique, on ajoute des lignes de délai temporel d'un pas de temps. Ces lignes servent à activer des neurones, dont l'activité n'est que la copie différée d'un pas de temps d'un autre neurone du réseau. Conservant la structure de couche pour ces neurones-là également, les réseaux d'Elman et Jordant disposent en fait d'une couche de neurones qui est la copie d'une des couches du perceptron. Il peut s'agir d'une copie d'une couche cachée ou de la copie de la couche de sortie. Cette couche de copies différées est concaténée à la couche d'entrée, si bien que la couche qui usuellement reçoit la couche d'entrée dans un perceptron considère comme entrée non seulement les signaux d'entrées, mais aussi une entrée générée par le réseau lui-même. Ces réseaux peuvent être entraînés pour générer des séquences.

Mentionnons également, toujours sur la base du perceptron, le réseau NARX, qui est un mélange de réseaux récurrents et de réseaux à base de lignes à retards évoqués précédemment.

[17] Le poids entre un neurone A et un neurone B est toujours le même que le poids entre B et A.

[18] Cette propriété d'approximation universelle ne résout pas la question de l'apprentissage.

L'idée est de construire une FIFO pour l'entrée, mais aussi pour la sortie du perceptron. C'est la concaténation de ces deux FIFO qui constitue l'entrée du perceptron.

Ces réseaux, très populaires, exhibent une dynamique temporelle riche, et exploitent un principe de récurrence pour engrammer le contexte temporel dans leur état d'activation. Basés sur le perceptrons, ils n'ont toutefois pas le caractère cellulaire qui nous intéresse, comme nous l'avons déjà dit. Les techniques d'apprentissage sont en effet toujours basées sur la rétro-propagation d'un gradient d'erreur, adaptée à la structure récurrente du réseau.

1.4.5.2 Réservoirs

Les réseaux de type réservoirs reposent eux aussi sur le principe que nous avions énoncé précédemment, à savoir la constitution d'un contexte temporel puis la prédiction à partir de ce contexte, plutôt qu'à partir de l'entrée instantanée. Autant l'apprentissage permet aux réseaux récurrents de construire ce contexte, autant ici, l'idée est de mettre en place une architecture produisant des contextes riches, desquels seront extraits les contextes pertinents.

Un réservoir est constitué d'un ensemble de neurones, connectés aléatoirement par des connexions avec un délai d'un pas de temps. Sous réserve de certaines conditions sur la matrice de connexions, l'activité des neurones est telle qu'elle garde une trace des activités passées. Toutefois, l'influence d'un événement passé s'évanouit au fur et à mesure du temps, ce qui fait que le système n'est pas chaotique, même s'il est dépendent du passé. On retrouve la notion de *frontière du chaos* que nous avions évoquée dans le cadre des automates cellulaires.

Certains neurones du réservoir reçoivent le flux d'entrée externe, et l'ensemble du réservoir exhibe donc un contexte temporel complexe pour ce flux d'entrée. Il existe des méthodes dites de plasticité intrinsèque qui adaptent les connexions au sein du réservoir au flux d'entrée pour maintenir le réseau à la frontière du chaos.

Le réservoir contient un très grand nombre de neurones, dont il faut extraire ceux sur lesquels peut se baser la prédiction. Cette extraction se fait par un perceptron simple, dont la couche d'entrée est l'ensemble des neurones du réservoir. Le réservoir, même si en pratique il est souvent implémenté via un calcul matriciel, et donc une évaluation synchrone, est un système à grain fin qui pourrait de plus être cellulaire, si on y ajoutait un critère de localité topographique sur les connexions. Toutefois, le perceptron monocouche en sortie, du fait de sa connexion à *tous* les neurones du réservoir, est une structure globale, ce qui va à l'encontre de l'approche cellulaire qui nous motive.

1.4.5.3 Réseaux de Hopfield

Les réseaux de neurones de Hopfield sont des réseaux totalement connectés, avec des connexions non orientées puisque le poids de la connexion de A vers B est par construction le même que celui de B vers A. Cette contrainte, peu plausible biologiquement, a toutefois l'avantage de permettre l'expression d'une fonction d'énergie pour le réseau. L'activation des neurones est asynchrone. En effet, une étape d'évaluation consiste à choisir aléatoirement une unité, puis à ré-évaluer son activité en fonction des unités auxquelles elle est connectée.

Cette architecture implémente une mémoire adressable par le contenu. En effet, lors d'une phase d'apprentissage préalable, si l'on présente des motifs d'activation et que l'on adapte les poids

par une règle Hebbienne, on crée des bassins d'attractions vers ces motifs-là dans la dynamique du réseau. Ainsi, si l'on active le réseau par la suite en lui imposant une activité proche de l'un des motifs, la relaxation du réseau, c'est-à-dire la dynamique obtenue en réalisant successivement des mises à jour asynchrones des activités des neurones, conduit à retrouver le motif initialement engrammé qui est le plus proche de l'activation initiale.

Le gros intérêt de ces réseaux est l'existence de la fonction d'énergie qui décrit l'évolution du système[19], qui permet d'établir des propriétés mathématiques et d'envisager des extensions. Ainsi, il existe des versions où l'évaluation des neurones est stochastique, ce qui conduit à la définition de machines de Boltzmann.

Les réseaux de type Hopfield ont une dynamique temporelle basée sur une évaluation asynchrone, et sont indéniablement à grain fin. Il ne leur manque que des propriétés de connectivité topographiquement locale pour être des systèmes strictement cellulaires.

1.4.5.4 Autres réseaux

Nous souhaitons ici mentionner d'autres réseaux temporels, un peu moins classiques. Il s'agit par exemple de réseaux où les activités ou les poids sont modulés par d'autres activités, ce qui fait que l'apprentissage se fait à différents moments à différents endroits du réseau. Il s'agit par exemple des réseaux *Long Short-term Memory*, ou de réseaux de d'ordre 2 impliquant des triades synaptiques.

Il existe d'autre part des approches à bases de réseaux récurrents à temps continus, dont l'activation est définie par une équation différentielle. L'apprentissage pour ces réseaux est très artificiel, impliquant parfois le recours à des approches évolutionnistes, peu compatibles avec les modèles cellulaires qui nous intéressent.

1.4.6 Réseaux de neurones et modèles de calcul

Dans la littérature, on trouve de nombreux rapprochements entre les réseaux de neurones récurrents et la théorie de la calculabilité. En effet, les réseaux récurrents, une fois l'apprentissage réalisé, exhibent des états différents, issus d'une transition déterminée par l'état courant et l'entrée courante. Ils se comportent ainsi comme une machine à états, voire un *transducer* quand on considère la sortie qu'ils produisent.

Même si les problèmes liés à l'apprentissage de ces réseaux sont épineux, mais il s'agit si l'on compare aux machines d'une phase de programmation qui n'est jamais triviale, les réseaux récurrents réalisent des machines théoriques de reconnaissance de langages, ayant potentiellement la puissance d'expression des machines de Turing, ce qui motive par conséquent des recherches plus proches de l'informatique théorique à leur sujet.

1.4.7 Conclusion

Peu de modèles cellulaires ont fait la preuve qu'ils pouvaient implémenter une structure de traitement de l'information temporelle. En revanche, même s'ils ne sont pas cellulaires au sens strict, les réseaux de neurones sont des systèmes à grain fins proches des systèmes cellulaires. Ils ont été

[19] Il s'agit d'une fonction de Lyapunov.

très utilisés pour le traitement de séquences temporelles, et on tracé la voie du traitement séquentiel par les architectures à grain fin, traitement qui rejoint des arguments sur leur puissance de calcul théorique.

Il reste toutefois une classe de réseaux de neurones que nous n'avons pas abordée, les cartes auto-organisatrices. Elles sont au cœur des travaux que nous proposons, aussi leur avons nous réservé le paragraphe suivant.

1.5 Cartes auto-organisatrices pour le traitement de séquences temporelles

Initialement inspirées du cortex cérébral, les cartes auto-organisatrices de Kohonen (SOMs) sont aujourd'hui une technique d'apprentissage automatique non-supervisé utilisée pour de nombreuses applications. Elles ont mis en avant des techniques de quantification vectorielle avec restitution de topologie, techniques qui ont évolué depuis. Dans ce paragraphe, nous nous concentrerons sur le principe des cartes de Kohonen et examinerons la façon dont elle peuvent permettre une auto-organisation temporelle.

1.5.1 Cartes auto-organisatrices de Kohonen

Une carte de Kohonen est un ensemble d'unités organisé suivant une topologie, souvent 2D. En ce sens, leur structure est assez proche de celle d'un automate cellulaire. Cette topologie permet de définir une notion de voisinage entre les unités. Ce voisinage est donné a priori, lors de la définition de la carte.

Comme nous nous situons dans un contexte de quantification vectorielle, chaque unité représente un prototype des configurations que peuvent prendre les entrées fournies à la carte. Ce prototype est une entrée particulière, stockée par l'unité. L'espace des entrées est un espace métrique, et la distance au sein de cet espace est à bien distinguer de la distance induite par la topologie mentionnée plus haut. En effet, deux unités proches sur la carte peuvent avoir des prototypes éloignés dans l'espace des entrées, c'est-à-dire des prototypes très peu ressemblants.

L'apprentissage au niveau de la carte se fait en lui présentant successivement des entrées. L'algorithme consiste à d'abord déterminer quelle unité a le prototype le plus proche de l'entrée. On appelle cette unité *best matching unit* (BMU). Notons, même si ce n'est pas nécessaire ici, que l'on peut définir pour chaque unité une *activation*, qui est une fonction décroissante le la distance entre le prototype de l'unité et l'entrée courante. Au regard du profil d'activation des unités, la BMU est donc celle qui est la plus active. Contrairement aux cartes de Kohonen, d'autres algorithmes requièrent que cette notion d'activation soit explicitée. Une fois la BMU déterminée donc, on définit une zone d'influence au niveau du voisinage sur la carte de la BMU. L'influence est maximale au niveau de la BMU, et décroît progressivement au fur et à mesure que l'on s'en éloigne. Une fois cette influence déterminée, l'apprentissage consiste à rapprocher les prototypes de toutes les unités de la carte de l'entrée courante. Toutefois, ce rapprochement est modulé par l'intensité de l'influence, si bien que seuls les prototypes des unités proches de la BMU se rapprochent significativement de l'entrée courante.

Au fur et à mesure de la présentation des entrées, on obtient une répartition des prototypes sur la carte qui est telle que deux prototypes d'unités voisines sur la carte sont également proches (ressemblants) au regard de la métrique de l'espace des entrées. La réciproque, elle, peut être fausse.

1.5.2 Traitement de séquences avec les cartes auto-organisatrices classiques

Le premier cas que nous allons considérer est celui des SOMs classiques appliquées à une transformation temporelles des entrées. On retrouve ici la même idée que pour les perceptrons auxquels on soumet comme entrée les éléments d'une FIFO temporelle. L'idée ici est de construire également une FIFO de longueur donnée, et de considérer ce vecteur comme l'entrée courante d'une SOM classique. Il existe d'autres techniques de pré-traitement, comme l'application d'ondelettes ou autres filtrages, mais l'idée reste de finalement n'appliquer ni plus ni moins qu'une SOM classique à ces données pré-traitées.

Le second cas que nous allons considérer est celui où le calcul temporel est un post-traitement. Considérons une SOM classique, une fois l'apprentissage réalisé. Au fur et mesure que l'on présente la suite des entrées, on obtient une suite de BMU qui y répondent, donc une suite de positions d'unités sur la carte. Ces trajectoires, sur l'espace topologique de la carte, sont significative de la séquence qui a été soumise en entrée. Il existe donc plusieurs techniques visant à comparer ces trajectoires, à les classifier, afin d'effectuer un traitement temporel.

1.5.3 Traitement de séquences avec des cartes auto-organisatrices modifiées

Dans le paragraphe précédent, nous avons mentionné des méthodes où l'algorithme SOM est utilisé tel quel, le traitement séquentiel intervenant en pré- ou post-traitement. Il existe bien entendu des adaptation de SOM pour tenir compte du caractère temporel des données.

Le modèle Hypermap introduit par Kohonen en est un exemple. Il consiste à utiliser deux fenêtres temporelles glissantes centrées sur l'élément courant de la séquence. L'une est étroite et l'autre est plus large, servant de contexte. Les prototypes de la carte sont alors doubles, puisqu'ils représentent ces deux motifs. Le processus de détermination de la BMU s'effectue en deux temps. Premièrement, une BMU est choisie, en fonction d'une distance n'impliquant que les prototypes larges (contexte). On délimite dans la carte, autour de cette BMU une région au sein de laquelle on cherche une BMU, mais cette fois-ci, en fonction des prototypes étroits, liés à fenêtre temporelle étroite. C'est cette BMU-là qui sera retenue pour l'apprentissage. On voit ici que la spatialité de la carte est mise en correspondance avec la temporalité de la séquence, qui s'exprime par les prototypes contextuels. D'autres variations sur ce principe existent.

Une autre approche consiste à présenter les entrées une à une, comme on le ferait pour une SOM classique, mais à restreindre la recherche d'une BMU dans un voisinage de la BMU élue par l'entrée précédente. Ce mécanisme est une modification assez légère de SOM, qui là aussi crée une dépendance entre la disposition des prototypes sur la carte et la succession des entrées de la séquence.

1.5.4 Modèles contextuels basés sur les cartes auto-organisatrices

À l'instar des réseaux récurrents où la ré-entrance permet de construire un état du réseau qui tienne compte de l'historique, il existe des modèles à base de SOMs qui implémentent une ré-entrance, de sorte à conférer au processus d'auto-organisation la capacité à représenter le contexte temporel d'une séquence.

Dans le modèle *Temporal Kohonen Map*, la ré-entrance est limitée à un filtre récursif au niveau de l'activation de chaque neurone. De façon assez proche, l'algorithme *Recurrent SOM* se base d'avantage sur la variation de l'activation, limittant la récurrence à un calcul temporel sur l'activation de chaque unité.

Nous nous intéressons plus avant aux modèles où une véritable récurrence est implémentée. C'est le cas de l'algorithme *Recursive SOM*. Le profil d'activation des unités de la carte est un vecteur, qui sert de contexte. Lorsqu'une entrée est présente, elle est comparée à un prototype de même nature, mais les unités disposent également d'un prototype de contexte, auquel est comparé le profil d'activation de la carte du temps précédent. L'activation de l'unité se fait d'après les comparaisons aux deux prototypes, si bien que l'unité choisie reflète l'entrée courante mise en contexte de la séquence passée. Cette forme de récurrente est similaire à celle implémentée sur la base du perceptron dans le cas des réseaux d'Elman et Jordan que nous avons mentionnés.

Le vecteur de contexte dans *Recursive SOM* est de haute dimension, au regard de la dimension souvent faible des entrées. L'idée qui sous-tend *SOMSD* et de n'utiliser l'ensemble des activations comme contexte dans la ré-entrance, mais simplement la position de la BMU sur la carte. De façon analogue, *Merge SOM* utilise le prototype de la BMU comme contexte.

Sans discuter plus avant ici des différences entre *Recursive SOM*, *SOMSD* et *MergSOM*, soulignons que ces réseaux, à l'instar des réseaux récurrents, implémentent une forme de machine à état, voire d'automate à pile.

1.5.5 Modèles de cartes auto-organisatrices hiérarchiques

On peut avoir une vue hiérarchique des séquences, considérant qu'une séquence se compose de sous-séquences, elles-mêmes formées par des séquences plus élementaires, etc. Il existe des approches où cette hiérarchie se reflète par des structures multi-cartes hiérarchisées. Ces approches partagent avec le modèle que nous proposons le fait d'impliquer plusieurs cartes auto-organisatrices simultanément, même si notre modèle se rapproche davantage dans son fonctionnement des cartes récurrentes mentionnées précédemment.

1.5.6 Du caractère cellulaire des cartes auto-organisatrices

Les cartes auto-organisatrices mettent en œuvre des unités disposées suivant une topologie, usuellement une grille, qui sert à définir un voisinage entre les unités. Cette caractéristique en fait un modèle proche des modèles cellulaires. Toutefois, la sélection de la BMU reste un processus centralisé incompatible avec une approche cellulaire.

Cette incompatibilité nous a amené à considérer un autre mécanisme de sélection sur la surface de la carte, d'après les travaux passés dans l'équipe. Ces mécanismes se basent sur la notion de champs neuronaux dynamiques, qui sont de bons candidats pour ramener les processus

d'auto-organisation mis en avant par les cartes auto-organisatrice dans le domaine des algorithmes cellulaires.

1.5.7 Calcul cellulaire à base de cartes auto-organisatrices et de champs neuronaux dynamiques

Les champs neuronaux, comme le terme "champ" l'indique, sont des réseaux de neurones organisés suivant une topologie, usuellement bidimensionnelle. Chaque neurone du champ est relié à ses voisins au sens de cette topologie via deux types de connexions. Les connexions excitatrices le relie à un voisinage proche, et amènent le neurone à être excité si ses voisins proches le sont. Les connexions inhibitrices, elles, s'étendent plus largement autour du neurone, et tendent à désactiver celui-ci si un voisin est actif. Le neurone, enfin, est soumis à une entrée scalaire, dont l'intensité favorise l'excitation du neurone.

Le modèle classique définit l'activation des neurones par une équation différentielle, les influences via les connexions de voisinage étant calculées par convolution. On montre que l'on peut obtenir dans ce cas un profil d'activation à l'équilibre qui correspond à quelques zones d'activité compactes sur la carte, voire à une seule. La population de neurone réalise ainsi une prise de décision collective, ne retenant sur sa surface que les neurones pour lesquels l'entrée est localement la plus forte.

L'idée des travaux récent de l'équipe est de se servir de ces champs neuronaux comme processus de sélection d'une carte auto-organisatrice, laissant le champ neuronal, par sa dynamique, déterminer la zone d'influence qui guide l'apprentissage des prototypes, et s'affranchissant ainsi du processus global de sélection de la BMU.

Les modèles de champs neuronaux classiques ne permettent toutefois pas de piloter convenablement les processus d'auto-organisation, ils sont en général utilisés pour modéliser des processus de sélection attentionnels. Il a donc fallu que nous nous basions sur des résultats développés dans notre équipe pour mettre en place d'autres champs neuronaux afin de disposer d'un modèles de cartes auto-organisatrices dont le mécanisme de compétition est distribué. Aujourd'hui, ce modèle est en place, mais il repose sur une connectivité aléatoire des connexions inhibitrices, qui n'a pas la localité topographique requise pour que le modèle soit strictement cellulaire. Nous reviendrons sur ce point en conclusion.

1.6 Un modèle cellulaire auto-organisant pour le traitement de séquences temporelles

1.6.1 Introduction

Le modèle que nous proposons est un algorithme de traitement de séquence qui soit cellulaire. Ce modèle est comparable aux réseaux récurrents dans sa capacité à se comporter comme une machine à états qui restitue la dynamique du processus qui génère la séquence, séquence qui peut-être ambiguë. Cela dit, du fait de notre volonté de contribuer à l'informatique cellulaire, nous avons exclu, parmi les modèles récurrents, les modèles basés sur le perceptron. Nous leur avons

préféré une approche à base de cartes auto-organisatrices couplées, cartes qui sont contrôlées par un processus de compétition distribué inspiré des champs de neurones dynamiques.

L'architecture sur laquelle nous nous sommes basé est le modèle bijama[20] développé dans notre équipe. Il s'agit d'une architecture à grain fin, distribuée, qui permet de concevoir des algorithmes comme une population d'unités de calcul connectées entre elles. Chaque unité héberge plusieurs activités scalaires, qui sont mise à jour à chaque pas de temps. Cette mise à jour dépend des activités d'autres unités, le rôle des connexions étant de fournir un droit de lecture pour une unité aux activités de l'unité distante.

L'évaluation des unités dans le modèle bijama est asynchrone. À chaque pas de temps, toutes les unités sont évaluées une seule fois, dans un ordre aléatoire qui change d'un pas de temps à l'autre. L'environnement bijama induit également une méthodologie dans la définition de mise à jour. Les différentes activités appartiennent à des couches, prenant chacune plusieurs entrées et fournissant plusieurs sorties. Les entrées d'une couche peuvent être les activités de sorties de couches inférieures, où les informations collectées via les liens. Les sorties sont des activités de l'unité, à disposition des couches supérieurs et pouvant être lues via les liens.

L'architecture bijama est implémentée sous forme d'une bibliothèque C++, qui permet, sans effort supplémentaire de programmation, d'instancier le système à grain fin que l'on définit sur un cluster de PC, le cluster Intercell. Cette implémentation est possible du fait du caractère distribué et décentralisé imposé par bijama, et elle sera d'autant plus efficace que le réseau implémenté est cellulaire.

Même si l'idée de bijama est de permettre la programmation de couches que l'on peut empiler pour définir le fonctionnement d'une unité, il existe des couches déjà réalisées, sur la base de travaux précédents de l'équipe. C'est le cas de la couche de compétition que nous avons utilisée, qui implémente les modifications des champs neuronaux requises pour construire des cartes auto-organisatrices dont la compétition n'est pas centralisée sous la forme d'un *winner-take-all*, comme dans les cartes de Kohonen.

Une des intérêts de bijama est de faciliter la construction de cartes, et par conséquent de permettre d'exprimer des architectures impliquant plusieurs cartes. La connectivité entre deux cartes est une connectivité en bandes, qui est partielle. Les connexions inter-cartes, dites connexions corticales, hébergent un poids qui est en permanence ajusté par apprentissage. Lorsque plusieurs cartes sont interconnectées, elles s'influencent l'une l'autre, par un mécanisme de résonance, de sorte que les activités en sortie des champs neuronaux produisent au sein de chaque carte une seule petite région compacte active, que nous appellerons une bulle d'activité. La résonance assure, suite à quelques étapes de relaxation, que les bulles au sein de chacune des cartes se trouvent situées à des endroits connectés entre eux d'une carte à l'autre, ce qui n'est pas aisé du fait de la connectivité en bande qui est partielle. Ainsi, par ces mécanismes, la connectivité partielle évite l'explosion combinatoire du nombre de connexion avec l'accroissement de la taille des cartes, et la résonance assure des activations qui restent à des endroits connectés, et donc des apprentissages au sein des cartes de représentations qui, si elles sont concomitantes, sont liées par des connexions après auto-organisation.

Le problème que nous nous proposons de résoudre est celui d'une machine à états autonome,

[20] Biologically-Inspired Joint Associtive Maps.

périodique, qui génère des observations. L'observation courante est obtenue en fonction de l'état courant, mais cette fonction n'est pas bijective. Il existe donc des états différents qui produisent la même observation. Dans notre modèle, l'observation est un scalaire, compris entre 0 et 1, qui prend des valeurs notées A,B,C,D,E,F, avec 0 pour A, 1 pour F, et une répartition linéaire entre ces deux valeurs pour les autres lettres. Ainsi, une machine à 5 états, qu'elle visite tour à tour, pourra donner la séquence AFFFF d'observations, si la fonction d'observation associe A au premier état et F aux autres. Alors que la séquence ne présente que deux types d'observation, A et F, nous souhaitons que notre architecture soit capable, sur la surface d'une carte organisatrice, de correctement retrouver que le système dynamique qui génère les observation est bien un système à 5 états, et non 2, même dans le cas d'une séquence aussi ambiguë.

1.6.2 Notre architecture

Notre architecture se comporte de trois cartes. Nous ne le détaillons pas dans ce résumé, mais les mécanismes implémentés sont très homogènes d'une carte à l'autre, ce qui rationalise la procédure d'ajustement des paramètres.

La carte principale est la carte d'entrée. Une bulle d'activité sur cette carte a vocation à désigner une position sur la carte qui représente l'état du système dynamique qui fournit les entrées. Cette carte gère des unités dont les prototypes combinent l'entrée[21] courante et un contexte. Ce contexte provient justement de la seconde carte, dite carte associative, qui est connectée à la carte d'entrée via des bandes de connexions adaptatives, comme le suggère le modèle bijama. La troisième carte est une carte de délai, qui est une copie, différée d'un pas de temps de séquence, des activités de la carte d'entrée. La carte associative associe donc la carte d'entrée avec sa copie différée, via deux séries de bandes de connexion, provenant chacune de l'une des deux cartes. La carte d'entrée reçoit donc des bandes de connexion de la carte associative, qui comme nous l'avons dit déterminent le contexte temporel, mais la carte associative reçoit en retour des bandes de connexions de la carte d'entrée. Cette boucle dans l'architecture et le siège de mécanismes d'auto-organisation complexes, discutés dans le manuscrit, qui font l'un des intérêts de nos travaux.

1.6.2.1 Résultats expérimentaux et discussion

Nous avons soumis plusieurs séquences à cette architecture, afin d'évaluer sa capacité à reproduire fidèlement les états du système dynamique qui génère les entrées ambiguës. Nous avons d'ailleurs comparé nos résultats avec *Recursive SOM*, qui est le modèle non-cellulaire existant le plus proche de nos travaux.

Nous avons montré que l'architecture était capable de créer différents états pour les mêmes entrées en cas de séquences ambiguës, c'est-à-dire de donner naissance à des bulles d'activités à des endroits distincts de la carte d'entrée. L'architecture est donc bien capable de s'organiser pour représenter sur le substrat cellulaire la dynamique du processus qui génère les entrées.

Nous avons également montré que l'architecture reste adaptative. En effet, aucune initialisation particulière n'est requise, et les paramètres du système restent constants au cours de son évolu-

[21] A, B, C, D, E ou F.

tion[22]. Ainsi, si l'on change le système dynamique, l'ensemble du système cellulaire se réorganise spontanément pour représenter le nouveau système sur son substrat.

Toutefois, même si nous nous réjouissons de ces résultats, nous avons observé que, parfois, la représentation d'état du système dynamique qui fournit les entrées, même si elle reste correcte, dérive sur la surface de la carte. Ce type d'instabilité n'a pas été décrit sur des systèmes auto-organisant plus simples, et il est difficile de déterminer ses causes, et de savoir s'il sera un obstacle à la constitution d'architectures impliquant plusieurs cartes temporelles.

1.7 Conclusion

Cette thèse s'est résolument inscrite dans la perspective de contribuer à l'essor du calcul cellulaire, en proposant d'aborder par ce type d'approche le problème complexe de l'auto-organisation de la représentation de séquences ambiguës. Les résultats sont prometteurs, puisque les effets de résonance, qui sont des effets émergents comme ceux que l'on attend des systèmes cellulaires, concourent effectivement à organiser l'ensemble de l'architecture sans supervision, en allouant le substrat de calcul disponible à une représentation non ambiguë, i.e. markovienne, du système dynamique autonome qui fournit les observations.

Nous avons, du fait de la réalisation de nos recherche, mis en lumière plusieurs difficultés, qui sont autant de perspectives à ce travail. La première est l'apparition d'instabilités de représentations sur la surface, qu'il conviendrait de mieux analyser. Cela dit, se phénomène apparaît justement car, fort des outils de simulation que nous avons utilisés, nous avons pu manipuler un système suffisamment complexe pour que ces effets se manifestent. La seconde difficulté est certainement le fait de n'avoir pas pu pousser jusqu'au bout la logique cellulaire. En effet, les processus de compétition distribués au sein des cartes supposent une connectivité aléatoire, qui n'est que la détérioration d'une connectivité totale, et n'a donc pas la localité topographique attendue des systèmes cellulaires. Il reste donc à concevoir des mécanismes de compétition distribués qui s'affranchissent de l'exigence d'une connectivité totale, même dégradée.

Enfin, la restitution d'états à partir d'observations non-markovienne n'est pas sans rappeler la problématique de contrôle de systèmes dynamiques partiellement observés, et par conséquent les processus décisionnels de Markov partiellement observés (POMDPs). Une des motivations de cette thèse était de pouvoir proposer des solutions cellulaires à cette question-là, et nous n'avons pu y répondre que partiellement puisque le système dynamique dont nous extrayons les états est autonome, et non soumis aux actions de contrôle d'un agent. La poursuite de nos travaux dans cette direction nous semble essentielle.

Enfin, nous sommes fiers d'avoir pu contribuer à l'aventure humaine qu'est la maîtrise du calcul, son automatisation, et à l'exploration de paradigmes un peu en marge de la tendance majoritaire à ce domaine, nous inscrivant ainsi dans la lignée des travaux initiés par les sumériens, en 2400 avant JC.

[22] Contrairement par exemple aux cartes de Khonen pour lesquelles le rayon d'influence autour de la BMU diminue au cours du temps.

CHAPTER 2

General Introduction

Contents

Sumer, 2400 BC, is where the first computation machine appeared. By 3000 BC, Sumerians (who lived in modern Iraq) were representing objects by signs, each object was given a different sign. The development in economy and trade led to an increased variety of goods and a growing in goods quantity that made it difficult to draw hundreds of signs to represent a single number.

Sumerians then started to give a symbol for each group of objects, the same symbol regardless of the counted object, and introduced thereby, for the first time the *abstract concept of a number*, which will be primordial for developing computation methods and machines in later times of human history. They represented 10 objects by a cone, 6 cones by a circle, 10 circles by another shape and so on. This was the very origin of their sexagesimal numeral system (base 60) which is still used in one of its forms up to our days for measuring time and angles.

By the end of the third millennium BC, Sumerians developed their numerical system to a *place value system*, where the value of the number is determined by its place. A number was represented using simple grooves in the sand with stones in the grooves, or using a board of wood with stones for calculation and dust for drawing geometric shapes. Delimited grooves or columns were used to signify the different orders of magnitude of numbers in the sexagesimal system. This was the very first tool for representing numbers: *the Sumerian Abacus*. While Sumerians used abacus only for number *representation*, their successors, the Babylonians, are thought [Carruccio 2006] to have used abacus for arithmetic such as addition and subtraction, making of it the first devised apparatus by human to carry out **computation**, i.e. the first computer.

Archaeological evidences show that Greeks used abacus for the first time about the 5th century BC. They called it αβαξ (abax) to signify the dust used for drawing, the Greek word itself is thought to be a borrowing from the Phoenicians (the sailors).

Aside from developing a complex numeral system, Sumerians who were traders of oil and grains calculated the area of the triangle and the volume of the cube, motivated by their needs. This short story in solving existing problems is not only the story of Sumerians; it is a repetitive story in

human history: “need motivates innovation”.

Since the era of Sumer in the early bronze age until our days of infant information age (born 1970 with the first personal computer), the need for improving computation methods and devices never ceased to increase. Over the time, people developed computation in parallel with mathematics, physics, and astronomy. Computation passed through several steps of evolution, the most important, the human computer (using the terms of the 18th century) using pens and papers, and mechanical computers. The most important step was the invention of digital computer that revolutionized computation and empowered the acceleration of scientific research, industry and economy, ending up with the knowledge-based society that depends on a high-tech global economy. This course of evolution seems to be more requiring for computation power each new day.

The ever increasing need for computation has led people to investigate the potential of whatever useful basis in other fields of science to devise more powerful machines in order to meet the increasing needs. Quantum computing from physics, living cells and DNA molecules from biology are examples. However, the most important source of inspiration and the ultimate goal of computational models is the “machine” that devised all other computation machines since the very start: the human brain.

2.1 Emergence of computability, functionalism and intelligence

Humanity cumulatively integrates a legacy of knowledge through generations, until when it is ripe it fires some strides forward through some pioneers. The wide stride in computation was made by Alan Turing in mid 30’s and John Von Neumann in mid 40’s of the past century.

Alan Turing, was looking to find an answer to the decision problem posed by Hilbert which is called “Entscheidungsproblem”. The problem can be reduced to the following question “was there a method by which it could be decided, for any given mathematical proposition, whether or not it was provable?”, or alternatively “is there a mechanical procedure for separating mathematical truths from mathematical falsehoods?”. To give the answer, Turing proposed in 1936 his famous machine and used one of its properties “the halting problem” to prove that a general solution to this problem is impossible.

His proposal was a description for a hypothetical machine that is able to compute any computable sequence that a human calculator can do using unlimited time, energy, paper and a pencil. This machine is known as the *Turing machine (TM)* [Turing 1936]. Turing describes the machine as follows:

> *“...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings”.* A.Turing, Intelligent Machinery. 1948.

A Turing machine consists of a control unit (a finite state machine) with an open ended tape (representing memory) on which is written a *table of behavior* (a program, in today terms). Each table of behavior corresponds to a different Turing Machine, thus there is an infinite number of TMs. The program, or the table of behavior of a TM implements an *algorithm* that consists of a set of *sequential atomic operations*.

Using the definition of the TM, Turing and Church defined *computability* through defining *computable functions*. In their *Church-Turing thesis* [Kleene 1967] they define computable functions to be the functions whose values are *effectively calculable*, i.e the functions that could be computed by a finitely specifiable *algorithm*. The original thesis defines computable functions as: "Every effectively calculable function is a computable function", or equivalently " If and only if it is computable by a Turing machine". In modern terms computable functions are those for which there exists some algorithm to compute, in fact, a computable function is the formalized analogue of the notion of an algorithm. If computable functions are defined for all possible input arguments then they are called *recursive* functions, otherwise, if they are defined for certain inputs only then they are called *partially recursive* functions.

The expression "effectively calculable" designates "effective methods" in modern mathematical terms, which are the methods that give a correct answer in finite time steps for all problems of the same class. Effective calculability is now used by mathematicians to classify functions, into Turing-computable functions or *computable functions*, and non-Turing-computable ones or *uncomputable functions"*. However, effectively computable functions are not always efficiently computable: even if there exists some algorithm to compute them, it is not always possible to compute them in a reasonable time (with some effectively computable functions the computation time increases exponentially or super-exponentially with the length of the input). Waiting a very long time is not efficient even if the function is calculable. Thus one can argue the *feasible computability* of these functions. The latter term became a stand-alone field that studies efficient function computation.

Examples on computable functions are the addition of natural numbers, and all functions defined on a finite domain (like a finite sequence of natural numbers). An example on non-computable functions is the set of finitary functions (which take a finite number of inputs) on natural numbers (the whole set of naturals). These functions are not computable because they are uncountable, although one can find some computable functions between them (like 7-input addition function on natural numbers). Not only a function could be non-computable, but also could be a number like the infinite group of natural numbers, or fractions in real numbers because there is no machine that can compute them to an infinite precision.

Hypercomputation [Copeland 1999] refers to the computation of the uncomputables mentioned above including all non-recursive functions, i.e. the models of computation that compute functions beyond Turing machine capabilities. Typically, they also go beyond the human capabilities when not equipped by a machine [Copeland 2004]. There were many proposals for hypercomputing machines [Stannett 2006], but none of them had yet been proved to be physically realizable. The reason is that most of those theoretical models require computing infinite precision real numbers in a finite time. The term "super-Turing computation" refers to systems that outperform Turing machines in terms of complexity or other measures, and don't necessarily compute recursive functions [Stannett 2006]. However, super-Turing computation and hypercomputation tend to be used

in literature interchangeably.

In his writings, it was clear that Turing was very cautious in giving details about the terms he used, giving only very sophisticated precision, and left the definition of the word “machine” as general as possible. This later led to a whole literature of debate on many aspects related to Turing machine, most of which was purely philosophical. This was the case of the concept of “algorithm” and “machine”. Some scientists and philosophers tend to extend the basic definition of a Turing Machine to a more general concept which is “a computing machine, occupying a finite space and with working parts of finite size” based on a later explanation of Church in 1937. For instance Hodges [Hodges 2002] built a criticism on Church explanation of the Turing machine by using the term “machine” instead of “computing machine”. The latter is claimed by [Copeland 2004] to be different from “machine” in that the computing machine is working in accordance with systematic methods or algorithms, while the “machine” was not simply defined as such. So, was Turing trying to extend the definition of a machine to contain the brain?

When he put his model, Turing wanted to go beyond the “Entscheidungsproblem” problem and to capture what a human mind can do when it carries out a procedure [Hodges 2011], i.e. a sequential procedure of solving a problem. Moreover, his interest went beyond the framework of algorithmic functions. He was interested in the theoretical limits of computation, and focused on the comparison of the computational power of a computer with that of the brain, and thought that it is possible that a well programmed computer could rival the brain. He argued against the proposal that a machine can think in his article "Computing Machinery and Intelligence - 1950” and proposed the *Turing Test* that checks if a judge can distinguish a human from a machine in a chat session without prior knowledge of their positions. If he can’t reliably determine on which side is the machine, the machine passes the test, and then it is considered as intelligent. Passing the test is not related on how much its answers were correct, but rather on the distinguishability from human behavior. This reveals that Turing view to intelligence is behavioral by nature. Actual artificial intelligence is following Turing’s view, and is concerned by building intelligent systems that behave like the human regarding some task. If given an input, an intelligent system should give the same output (behavior) as a human. However, other people like Jeff Hawkins [1], claim that intelligence should not be defined by behavior, but should rather be defined by predictions and should be tested by the correctness of these predictions. Hawkins claims that the mechanism of intelligence in the human brain resides in the neo-cortex that stores (memorizes) sequences of patterns and makes constant predictions all the time. For example we are predicting what is the next song on a CD after listening to it several times, and if we hear another song then we will immediately feel that there is something wrong.

Back to Turing,

> *“The Argument from Continuity in the Nervous System, in particular, simply asserts that the physical system of the brain can be approximated as closely as is desired by a computer program”*. A.Turing, in Mind, 1950.

It is obvious that Turing was an early supporter of the *Functionalism* theory proposed later in 1961 by H.Putnam. Functionalism claims that every mental state can be identified by functional

[1] TED talk by Jeff Hawkins: How brain science will change computing, (February 2003).

roles, and that brains are physical devices with a neural substrate that perform computation on inputs and internal states to give an output. Earlier contributions in the direction of this theory came from McCulloch and Pitts, the pioneers of artificial neural networks. In their influential paper "A logical calculus of the ideas immanent in nervous activity. 1943", they considered that every neural activity is computation in the sense of Church-Turing-Thesis and that every neural activity is explained by some neural computation, thus it is "metaphorically" possible to implement by a finite number of interconnected logical devices. They thought that their proposed neural network was less powerful than Turing machines but if provided by a tape (memory) they will be Turing-equivalents.

Functionalism was later developed by Fodor to the *Computational Theory of Mind* [Fodor 1978] which looks at the brain as an information processing system and thinking as computation. However, many people argued against both theories. Objections have been raised by people like Penrose, Searle, and later from Putnam himself. Penrose for instance, argued that there should be uncomputable physical operations in the brain as it can see formally unprovable truth. An illustration of formally unprovable truth is the Gödel's sentences. A Gödel sentence contains a strange loop which occurs due to self reference. Gödel modified the liar paradox which is expressed by the sentence: "this sentence is false" by replacing "false" by "not provable". So, Imagine a theory T which contains a sentence G that says "G is true, but not provable in the theory T". The brain can formally see this sentence while it can't be proven, and this is the main claim against functionalism and the computational theory of mind.

One of the most discussed topics in Turing time was "intuition" as a form of intelligent behavior, and whether there is a mathematical framework to describe it, and if a machine can exhibit intuition. Human intelligence, including intuition, requires the ability to evaluate functions that are beyond the classical boundaries of computability [Kugel 2002]. For Kugel, computers -limited to computing- can only fake intelligence when performing artificial intelligence tasks. Hodges [Hodges 1997] claimed that Turing changed his mind after the war influenced by the great extent that machines helped in breaking the U-Boat enigma during the second world war. But in fact, rather than being a belief, Turing had some clear insights about intuition and intelligence, that came partially true concerning intelligence.

> *"This – raises the question 'Can a machine play chess?' It could fairly easily be made to play a rather bad game. It would be bad because chess requires intelligence. We stated—that the machine should be treated as entirely without intelligence. There are indications however that it is possible to make the machine display intelligence at the risk of its making occasional serious mistakes. By following up this aspect the machine could probably be made to play very good chess."* A. M. Turing's ACE report of 1946.

> *"We may hope that machines will eventually compete with men in all purely intellectual fields"*. A.Turing, Computing machinery and intelligence, Mind, 1950.

In his writings, not only he talked about machines that can perform intelligent computation to develop intuition, but also talked about systems that can *modify their own programs* including nets of logical components whose properties could be trained to implement a desired function. This was long before the rise of artificial neural networks. He also talked about "genetical or evolutionary

search" of a machine to modify the program and exhibit intelligence. This also was before genetic algorithms. In all this, Turing was clear about looking at these methods as non-algorithmic and that the implementation of such structures is only possible on what he suggested to call *Universal Practical Computing Machine*.

Computation emerged as an indispensable need for Human and was developed cumulatively in several steps throughout history. The development of computation allowed to explore the possibilities to build intelligent computing machines that do some functions of the human brain. People have looked to the brain as a computing machine, and efforts in computerized computation tried since the very start to mimic its functions.

2.2 Digital computer architectures

The idea of building machines that carry out computation goes way back. Thomas Hobbes, explained in 1651 how arithmetic and logic are the same thing, and how artificial thinking and artificial logic can be done using arithmetic operations. Leibniz said in 1679 that all can be done with addition only, and was the first to talk about building a machine having gates that can be shifted (same as shift registers) with holes which can be opened for 1 and closed for 0, then marbles can flow in open holes [2]. Digital computers do the same thing but using electrons instead of marbles.

In his model, Turing put the basic idea of *stored program computer*. Later, Von Neumann introduced in June 1945 a detailed architecture of serial computation based on Turing model. Von Neumann's architecture consists of an arithmetic processing unit, a control unit (both make the central processing unit-CPU), a memory for both data and programs, and input-output devices. On the basis of this architecture was built the first digital computer.

Basically, Von Neumann built a physical machine in order to do bomb calculations. The demanding industry in wartime had affected the work of scientists and led them to focus on industry rather than pure scientific research. People efforts were focused on the vertical development of the basic structure, rather than thinking of better alternative models. This led to accumulate efforts on the investment of Von Neumann architecture rather than developing it, and forced the adoption of the basic computer structure instead of developing it and losing the accumulated efforts. Although many later modifications and improvements were made on the basic Von Neumann architecture over time, manufacturing computers continued in adopting the same principle: serial computation of instructions, one instruction by time. Indeed, even nowadays programming languages are sequential in order to fit with the hardware.

A well known drawback of the Von Neumann architecture is what is called *Von Neumann bottleneck*. Von Neumann architecture employs a shared bus for the program memory and data memory that limits the data transfer rate (throughput) between the CPU and memory. The fact that there is a shared bus makes it impossible to access data and instructions at the same time. This means that the processor has to wait for data to be transferred to and from the memory and most of the processing time goes for instruction fetching and data loading. The problem became more

[2] TED talk by George Dyson: At the birth of the computer, (Mars 2003).

serious with the increasing speed of processors and memories (though memories are becoming increasingly slow relative to processors).

> *"Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the Von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the Von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."* John Backus [Backus 1978].

There had been many efforts to reduce the effect of Von Neumann bottleneck [Markgraf 2007], for example by using the cache memory, a small but fast memory between the processor and the memory to partition instructions and data, creating some type of separate paths (though non-physical), or providing processors with stacks to reduce data transfer with the memory. An alternative solution was the Harvard Architecture and the Modified Harvard Architecture, which physically separates paths for data and instructions and separates the address spaces for data and programs allowing for faster processing. This architecture, being more robust and fast is more expensive and limits programming flexibility compared to Von Neumann architecture that allows a program, for example, to change its own code, resulting in self-modifying code, impossible on Harvard architecture. Harvard architecture is mainly used in digital signal processors (DSPs) and microcontrollers such as AVR and PIC families in which the focus is on the processing speed.

However, both Von Neumann and Harvard architectures are processing one instruction at a time, thus they are serial. The ongoing development and manufacturing of these serial architectures will unavoidably reach the limit.

By now, the clock speed is almost saturated due to technology dependent issues. Physics imposes an upper bound to the clock speed, because of the fundamental physical law that imposes that no signal propagation speed can exceed that of the light. Not only clock speed is bounded by physical laws, but also the power-law of increasing density and the decreasing size of microscopic elements in chip fabrication is expected to reach the limit within one or two decades due to physical laws of quantum mechanics. Shrinking to the scale of nano-technology makes it impossible to maintain the controllability of nano-elements due to the ruling laws at the nano-scale such as Heisenberg's uncertainty principle. Chip manufacturers are still able to increase transistors density, while they are bound by now with the clock speed. Increasing the clock speed increases heat and power consumption, the thing that requires a significant power source and heat dissipation technology.

Indeed, it may be noticed that processors clock cycle rate saturated since a decade. Intel, The world's largest and highest valued semiconductor chip maker, released in 2004 a 4GHz processor. The processor suffered from high power consumption and current leakage. Since that time, Intel engaged with a careful clock speed race with AMD, the other processor manufacturing giant. The highest clock speed reached thereafter by Intel was 3.8 GHz in its Pentium Family. To this date,

Intel's most powerful processor according to PassMark CPU benchmarking [3], was released in 2012: The cream of the crop, Xeon Processor family, with its top-of-the-line E5-4650 processor with a clock speed 2.7GHz. Instead of the clock speed, Intel turned to improve the processing performance using multi-threading and multi-core technology.

It is clear from this realistic example that manufacturers resorted to increase the number of transistors on chip components in such a way that implements concurrency, while saturating (even reducing) the clock speed.

The laws of physics ruling processors industry, being fundemental in nature, means that they will apply on future design and manufacture of computer processors. Moors's law will sooner or later be no longer valid, but at the same time, Human will continue to double the size of produced information every 18 months or even in a shorter time interval. Therefore, other solutions for more computation power are to be found.

One solution, emerged since 1990s, also derived by the need for higher performance, particularly computational speed is *parallelism* and *distributed computing*.

Parallel computing emerged motivated by application fields like engineering, real time processing, weather forecasting, and other fields. Calculations in parallel computing are carried out simultaneously. Architectures allowed for parallelism in different levels: bit-level, instruction level, data, and task level. And as mentioned before, Intel uses some parallelism techniques using multi-thread and multi-core processors. This is a modification of the basic Von Neumann architecture in the form of parallelism.

Parallelism is rather architectural and hardware-related, and should be distinguished from *Concurrent computing*. The latter is rather a programming paradigm: programs are written so that they can be divided into separate processes (or threads), then they can be executed either on the same processor by interleaving the execution steps, or on parallel processors simultaneously. Concurrent computing is oriented to a specific nature of applications, for instance, database applications which need to allow several users to access the database at the same time, same can be said on web servers.

Distributed computing (in the mainstream sense), consists of multiple software components that run on multiple computers (here we are no longer talking about processors), but they are all seen as a whole system, and they cooperate with each other to achieve a common goal. Although the overlap in terms and use, in parallel computing processors usually share the same memory and exchange information within this shared memory, while in distributed computing ones, each computer has its own memory. Even though some parallel computers like supercomputers started to use separate processor memories, using distributed systems is not an inherent property of parallel computing. In brief, parallel computing is related to computing simultaneously, while distributed computing is related with distributed parts carrying out computation.

But still, these computing approaches, being parallel or distributed, are using modifications or combinations of the basic Von Neumann architecture to get better performance or to fit with the task requirements, and we are still stuck at the same bottleneck. Solutions like parallelism and distribution (in the mainstream sense) seem to be sufficient for some application fields but for sure this is temporary in other highly expanding fields. Hence the need for another approach of computation.

[3] CPU Benchmarks: High End CPU's, (read on 4 June 2013).

Actual processors in today computers are Von Neumann architectures that are based on the theoretical Turing machine which is serial in nature. Several structural enhancements were added to the core of the Von Neumann architecture. Other major enhancements were added through the revolutionary parallel and distributed computing paradigms which duplicate the processing power and allow for extensible usability. Although, these enhancements are either modifications or aggregations of the serial Von Neumann processors.
The increase of processors speed will reach its limit in the near future due to physical laws. Therefore, the serial processing nature of Von Neumann processors became an intellectual bottleneck that should be overcome in order to stay in line with the ever increasing required computation power and the demanding applications. This requires the thinking of radically different processing concepts.

2.3 Cellular computing

Knowledge about the human brain inspired both computation and intelligence domains. Von Neumann architecture processes one instruction at a time, thus it is way far from resembling to "computation" in the human brain. Same thing can be said about Harvard architecture. After having put his architecture, and after the wartime, Von Neumann reconsidered a brain-like computation mechanism. Just before his death, Von Neumann left an unfinished book "The Computer and the Brain" which was published later. In his book, Von Neumann viewed the brain as a computing machine, and discussed the differences between the brain and the computers of his days, especially concerning speed and parallelism.

Indeed, unlike the sequential Turing Machine, computation in the brain is parallel, carried out by populations of relatively slow entities (neurons) that fire around 10 times per second. The human brain consists of $10^{10} - 10^{11}$ interconnected neurons with synaptic connections strength changing in an adaptive and continuous manner (synaptic plasticity). With this structure the brain is able to exhibit intelligence and, therefore, what we can be sure of, at least, is that it can't be perceived as a Turing Machine, because it doesn't implement a "finitely specifiable algorithm". Instead, we stand for the opinion that says that the brain activity can't be seen as computation in the Turing sense, but rather can be seen to implement Super-Turing computation, i.e. for us, what happens in the brain is a form of computation, thus it is possible to simulate in the future, but not by Turing machines.

The field of *neural computing* is based on perceiving the brain as an information system. In this perspective, senses are viewed as feeding inputs to the system, and the system encodes data (knowledge) in some way like membrane potentials and neurons firing rates and stores information via different kinds of memory (short-term, long-term and associative memories). The neural activities that the brain performs are considered as computation. Decisions, motor commands, feelings, and thoughts are perceived as the system output. Knowledge in the brain is encoded in the organization of neurons, their activities, and the connections weights (synaptic efficiency). The neural computing field is usually referred to as *connectionism*. Knowledge representation in the brain is distributed, a concept is stored by a net of nodes (neurons) and corresponds to some pattern of activity over all nodes. Moreover, each node could be involved in representing more than one concept.

It was paradoxical that although Turing was aiming to approach the brain computation, it was Von Neumann with Stanislaw Ulam who proposed a mathematical model that performs parallel computation which is similar to some degree to what happens in the brain: the *cellular automata*. Cellular automata consists of a population of locally interconnected units, in which each unit is connected to a limited number of neighbors. This is somehow similar to neurons connectivity in the brain, with the difference in the number of connections.

Based on cellular automata, Von Neumann proposed a universal constructor, a machine for self-replication that copies itself with an open ended complexity of growth allowing for mutations, as observed in biological organisms. This constructor incarnates the phenomena of natural selection adopted by modern Darwinism. In 1970, Conway proposed his famous two dimensional automata "game of life" and launched the true interest in cellular automata.

Later on, structures of cellular populations originating in different scientific fields showed the ability to perform computation. Cellular automata was used to implement binary addition [Benjamin 1996], cellular neural networks have been used in image processing applications like contour extraction [Xiao-hua 2009], DNA computing was used to solve the directed Hamiltonian path problem [Adleman 1994]. Very recently, bacterial cells were transformed to living calculators that are able to divide and compute logarithms and square roots. Existing and engineered genes of the bacteria were used to create synthetic computation circuits inspired by the analog electronic circuits [Daniel 2013].

Structures that carry out computation coming from different fields form the domain of *cellular computing*. Populations of cellular elements that carry out computation are sometimes called *cellular machines*. The first concrete work about cellular computing was [Sipper 1998b], in which the term of cellular computing was coined.

While parallel computing deals with a *small number* (tens up to tens of thousands in supercomputers) of powerful processors able to perform a single complex task in a sequential manner, cellular computing is based on another philosophy: simplicity of basic processing cells, their vast parallelism, and their locality. These properties were expressed by [Sipper 1999] to be an equation: $simple + vastlyparallel + local = cellularcomputing$.

Simplicity refers to the processing capabilities of the basic unit in a cellular processor which is the cell. Unlike parallel computing processors that can perform complicated tasks, the cell can do very little computation. An illustrative simplicity example is the comparison between what can do a NAND logic gate with what can do a computer processor. *Vast parallelism* in cellular computing refers to involving a completely different scale of cell numbers. It could be expressed in an exponential notation 10^x. *Locality* refers to local connectivity patterns between processor cells: a cell can communicate with a few other cells, usually those who are physically close. Connections hold a small amount of information. One direct implication of locality is that there is no central controlling unit in cellular processors.

Figure 2.1 shows a representation of computing paradigms respecting three features: the simplicity of the basic computing unit (on the x-axis: varying from complex to simple), the locality of connections (on the y-axis: varying from global to local), and the scale of parallelism (on the z-axis: varying from serial to parallel). The figure illustrates the relative positioning of cellular computing compared to other computing paradigms.

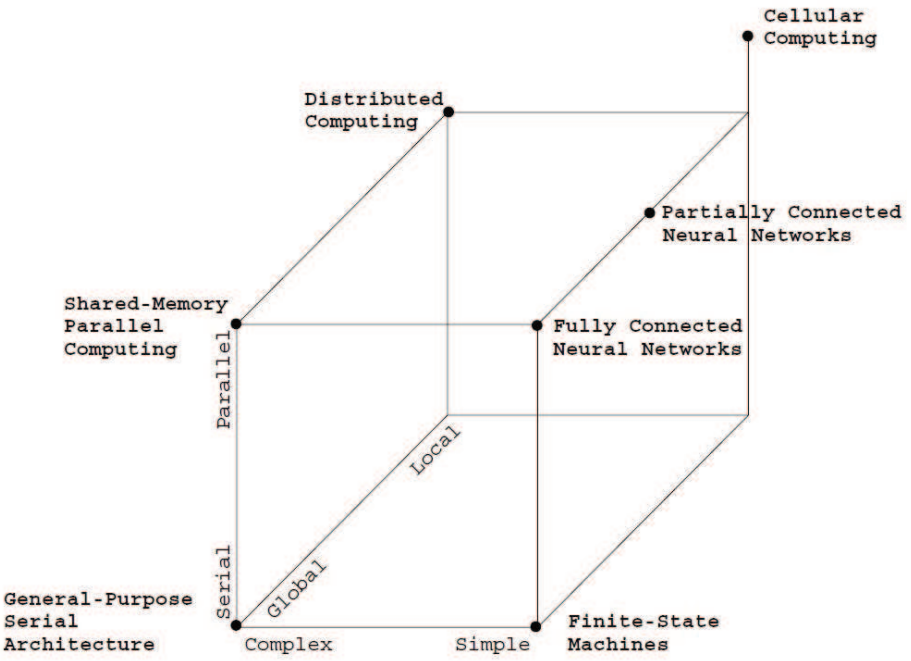


FIGURE 2.1: The computing cube; a graphical representation of the equation: *simple* + *vastly parallel* + *local* = *cellular computing* [Sipper 1999]. Distinction between some existing computing paradigms can be made respecting complexity and parallelism and locality of connections. Cellular computing is carried out by populations of locally connected simple cells, distinguished by their vast parallelism due to the large number of cells involved in the population. Extracted from [Sipper 1999].

The *general purpose serial architectures* like Von Neumann processors are complex processors performing serial processing, and in the case of a single processor there is no connectivity to other processors. A group of such serial processors results in the *parallel computing paradigm* that uses a shared memory as discussed above. More simple serial processing units result in *finite state machines*. *Fully connected neural networks* like Hopfield networks have each cell globally connected to all other cells within the network and each computing cell is a simple processing unit, and units are computing in parallel. The main stream *distributed computing* discussed above is built on combining in parallel a set of complex serial processors with local connections, where a processor needs not necessarily to be connected to all processors in the system, while this task is entrusted to network protocols.

Inspired from physics and biology, especially from the human brain, cellular computing is a promising new computation paradigm which is based on distributed units with small processing capabilities. It offers a completely different scale of computational power than the Von Neumann machines.

2.4 Capabilities of cellular computing

Since the crystallization of computation in Turing days, computation was approached by two ways: either by engineering methods by building practical systems performing computation, or mathematically by proposing and proving theorems about computation. A third approach was proposed by Stephen Wolfram, the founder of Mathematica, and the computational knowledge engine Wolfram Alpha, this approach is the *experimental computation*. Wolfram [Wolfram 2002] empirically studied cellular automata, besides to cellular structures originating from different domains, and claimed that they are good candidates to build processors. Wolfram studied systems which are composed of populations of more or less identical elements, in which each element can be in a finite number of states. The states are functions of the current element state, the states of adjacent elements, and the rules determining state changes. Although cells are running simple programs, the population of elements can show interesting range of complex behaviors. Getting complex behavior from the interaction of a population of elements carrying out simple programs is called *emergence*. It is getting new phenomena starting from basic elements. Nobel winner, the physician Murray Gell-Mann describes emergence by "*Emergence means that you don't need something more in order to get something more*" [4].

Emergence is a property related to *complex systems*. Physical world phenomena are usually studied by translating them into mathematical formulae which are then used in control and prediction. Despite the witnessed success of this approach, it has its limits, especially when it comes to understand complexity that surrounds us. Physical complexity couldn't be explained by conventional ways of mathematics, and thus, complex systems emerged as new branch of mathematics in order to emulate physical complexity.

Complex systems are made of many interconnected or interacting parts, like swarms of birds or fish, ant colonies, financial markets, ecosystems, and of course, the brain. These systems are hard to map into mathematical equations, but fortunately, what looks as complex behavior from the outside is actually the result of few simple rules of interaction, therefore one can forget about mathematical equations, and try to understand the system by looking at the interactions.

The emergent behavior of a complex system means that the whole is literally more than the sum of its parts (in term of behavior). A straight-forward example is the self-organization phenomena in which a global coordination spontaneously arises out of local interactions between the parts of an initially disordered system. Self-organization can be seen in physical systems like convection patterns in liquids heated from below, chemical systems like chemical oscillators, biological systems as in the cerebral cortex, and social systems like swarms of birds. This behavior of a complex system can't be understood or predicted by looking at the parts of the system. However, one can forget about the individual parts of the system, whether they are cells or ants or birds or other, and focus on the rules of interactions between them in order to understand or predict the emergent complex behavior.

In this view, network representation (nodes and links) turns out to be the ideal representation that facilitates the study of such systems, where the nodes represent the system parts, and links represent interactions. Such networks enable the study of complex systems behavior. These networks

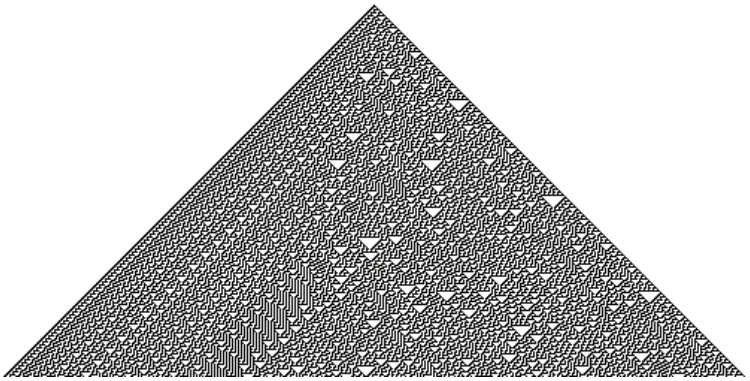
[4] TED talk by Murray Gell-Mann: On beauty and truth in physics, (Mars 2007).

are for the study of complex systems where equations are for the study of physics. Following this approach, many complex systems were studied in physics, biology, computer and social science. A work that attracted the attention of international media was applied in economics. Such network was applied on studying global corporate control [Vitali 2011] in order to find "who rules the world". The authors built an ownership network of Transnational Corporations (TNCs) based on financial data of 2007. Network nodes represent shareholders (which could be persons, companies or governments), and links represent the shareholding relations (for a directed connection from a node A to a node B, node A holds some percentage of node B). The 43.000 studied TNCs resulted in a network of 600.000 shareholders and 1 million links. Regarding the structure, the network was organized in a periphery and a center that contains 47% of all shareholders. Besides, the authors found in the center a small dominant core region which represents 36% of highly connected TNCs that makes 95% of the total revenue of all TNCs. As ownership gives a voting right for shareholders, one can trace the capability of shareholders in controlling the economy. It has been found that 737 shareholders (about 0.1% of all shareholders) has the potential to control 80% of the whole TNCs value, and 146 of them have the potential to control 40% of TNCs values. The authors argue that this result could probably be due to the self-organization phenomena found in complex systems (in economics it is called the invisible hand of the market), rather than a top-down approach or conspiracy (from our side, we think that what could have emerged as a natural self-organizing phenomena is an opportunity that would be hardly left unseized). Whatever, the authors claim that high connectivity between players, can cause a systemic risk to the global economy, because high connectivity makes instability highly spread across the network.

In line with the precedent claim related to complex systems, Wolfram claimed that traditional mathematics was failing to describe complexity in systems, and that these systems are *computationally irreducible*, i.e. it is not possible to describe the complex behavior of a system in a simple way, alternatively, there is no easy theory of almost all behaviors that seem complex. In such systems, the best way to describe the system is to simulate it. He claimed, however, that *any* complex behavior, can be captured by populations of elements running simple programs (like cellular automata). Indeed, complex patterns that can be found in nature are seemingly the result of sampling what is out there in the computational universe. Astonishing patterns can be obtained from running simple rule cellular automata. While some rules give simple behaviors, others give very complex ones. For example Rule 30 (figure 2.2(a)) is particularly interesting, it is a one-dimensional cellular automaton that shows a chaotic behavior which can be used like a pseudo-random bit generator. Interestingly, it generates patterns similar to a sea snail called the Textile Cone(figure 2.2(b)). Other two-dimensional rules can give patterns similar to zebra stripes.

Wolfram [Wolfram 2002] recommended to *experimentally explore the nature of cellular computation, and document what they do*, as the results would have great importance in understanding the natural world which he assumes to be digital, and that could result in a new scientific field just like physics or chemistry.

Actually, we know that when programmed with the appropriate rule, cellular automata are shown to be universal computers. For instance, The rule 110 that Wolfram conjectured in 1985 was proven in 2000 by Matthew Cook to be Turing-complete, i.e. it can be used to simulate a single-taped Turing machine and is thus theoretically able to perform as powerful as actual serial proces-



(a) Rule 30

(b) Textile Cone

FIGURE 2.2: Rule 30 cellular automaton shows aperiodic chaotic behavior. It shows how simple rules produce complex structures and behavior that can be found in nature

sors. Cook's paper wasn't published until 2004 due to rights conflict with Wolfram [Cook 2004]. Even earlier, Von Neumann proved that an automaton consisting of cells with four orthogonal neighbors and 29 possible states would be capable of simulating Turing machine [Gardner 1983]. Implementing some rule on cellular machines in order to prove their universality basically aims to prove that they are at least as powerful as existing machine. However, this degenerates the main aspect of cellular models: parallelism.

Indeed, recalling the limitations of actual computation paradigms, cellular models are promising approaches for implementing distributed and parallel computation, with which computation could restart *ab initio*. Let's also recall that some cellular populations, like artificial neural networks, are proven to be good candidates to perform super-Turing computation [Siegelmann 1995a].

Cellular architectures were used in different application domains, showing the ability to generate patterns, cluster input data, solve differential equations. Except for some image processing applications, cellular architectures were rarely used in temporal problems. In particular, there is no work, to our knowledge, in which cellular architectures deal with temporal sequences.

Some cellular structures were implemented in hardware, like FPGA (field-programmable gate array) chips and GPU (graphics processing unit) processors. Although, these efforts are still the first steps, and cellular computing is still a young domain. The computational capabilities of today cellular structures are still to be investigated before being able to build powerful general-purpose programmable cellular processors.

Cellular structures have the ability to capture complex behaviors. They are thought to have the ability to regenerate the emergent behavior of complex systems, i.e. they themselves can behave as complex systems. However, the investigation of their capability in temporal problems is far from being sufficient, and need more exploration.

2.5 Complex and cellular neural systems for modeling the state of a dynamical system

As mentioned earlier, the behavior of a complex system is determined by the interaction between its elements. Complex systems are in the first place the biological systems of nature, but they also involve systems originating from other fields like mathematics, physics and computer science.

Theoretically, a system is a set of interacting elements that form an integrated whole and could have interactions with the outer world via inputs and outputs. Most systems are dynamical by nature, their state of internal components, input and output change with time. Although most systems are dynamical, *dynamical systems* is a mathematical notion and a field of study with specific methodologies that are used in some scientific domains like Newtonian mechanics, fluid dynamics, mathematical economics and others. Examples of dynamical systems are everywhere, the stock market, the weather, the motion of an object, sugar dissolving in a cup of coffee, traffic, etc. Dynamical systems could miss either their inputs or outputs or even both, one example of the latter case is the theoretical model of the universe. A dynamical system that has no input is referred to as *autonomous*. In the dynamical systems study, the major focus is on how outputs are issued from the system state which is a compound of the internal variables. The state itself is a function of the previous state and the system inputs. In the case where the system is autonomous the state depends only on the previous state. In this sense, autonomous dynamical systems are conform with the Markov property. The latter, basically issued from probability theory is extended to describe any working environment. If an environment has the Markov property then its state depends only on the previous state.

Dynamical systems need sometimes to be modeled for the purpose of study, prediction or control. Some other times it may be needed to build a representation of their state for the purpose of learning like in some reinforcement learning algorithms. However, it is not always possible to measure the internal state variable; it could be impractical to measure like in the study of weather where it is not possible to know the temperature and humidity for every point in the system, or it could be hard to measure like the case of a metal oven where the variable to be measured is the degree of crystallization of the metal, or even inaccessible like in the study of the solar system. In such cases, people try to model the dynamical system starting from the available observations like the temperature in some points, or partial measurements like sensory information at the start and the end of the melting process. Such environments are partially observable; the measured observations are not sufficient to precisely determine the actual state of the system, thus they are called ambiguous hereafter. In such cases, the construction of the state or a representation of the state behavior should be done starting from the available observations, dealing with them as a *temporal sequence*.

In computer science domain, artificial neural networks are dynamical systems in which neurons values change over time and the network state is determined by the previous state and the current inputs (thus they are non-autonomous). In order to deal with temporal data, scientists developed a family of neural networks that was proved to be efficient to comply with the job, this family is the *recurrent neural networks*. Also, some kinds of neural networks exhibit the phenomena of self-organization, which is also a property of complex systems. Indeed, the dynamics allowing the

organization is not yet fully understood. Besides to being dynamical systems, neural networks are fine-grain models of computation, that comply partially to the cellular computing paradigm.

Our work subscribes to the experimental approach of cellular computation as proposed by Wolfram. At the same time, it is motivated by the knowledge about the dynamical properties of neural networks and their witnessed ability to encode temporal data. In the current work, we are proposing a cellular and complex neural system with a self-organizing emergent behavior in order to build a representation of the state of dynamical system starting from observations on the system. We would like to do this in the case of an autonomous dynamical system where the temporal sequence of its measured observations violates the Markov property, however, we want the built representation by the cellular neural processor to maintain it.

Neural networks are distributed models. Adopting distribution in this work is not imposed by the need to solve some specific problem that couldn't be solved by other means. It is rather a choice of research that aims to explore the capabilities of neural networks as cellular structures.

2.6 Problem definition

Neural networks are distributed models that were used in various applications, the one of major interest in the context of this manuscript is sequence processing. Some applications require accounting for the temporal dimension in the processed sequence, recurrent neural networks appeared as a reply for this requirement. Self-organizing neural networks have interesting emergent properties, they were used in several works for spatio-temporal problems, besides, by their 2-dimensional topology they offer a natural representation surface.

Neural networks differ from cellular computing models like cellular automata by their high and non-local connectivity, and thereby, their non-local computation. Another difference is their non-decentralized way of processing in learning and run. Due to their high and non-local connectivity and the control of their operation by central processors, neural models are typically implemented on classical digital computers, in particular, large networks are run on parallel computers. Because of these properties, their simulation on classical computers even parallel ones turns out to be complicated and lacks scalability. Besides, unlike other cellular models, implementing neural networks on hardware happens to be difficult.

Regarding temporal sequence processing, networks designed for accounting for the temporal dimension in input sequences have recurrent connectivity, global in most cases, and the problem is: local connectivity contradicts with connectivity recurrence found in most neural architectures in literature. Not to mention that in most cases such networks are computed by some central processing mechanism. Indeed, using neural networks as cellular computers for temporal sequence processing is not a straightforward task.

In this work, we explore the possibility of building distributed neural network structures following the cellular computing paradigm and investigate their potential in processing temporal data. We will see if networks with local connectivity patterns and decentralized learning and run could process temporal sequences. In order to account for the temporal dimension of inputs, the principle of recurrence is implemented in a different way than traditional models, in such a way that doesn't

contradict with the cellular nature of the network. One direct outcome from the combination of these two approaches is to be able to process temporal sequences, all in facilitating the network implementation on both software and hardware, and getting their scalability enhanced.

We propose such a model that integrates the temporal dimension of input sequences, while borrowing from the principle of recurrence as found in recurrent neural networks. Being hybrid between cellular automata and neural networks, this approach is based on a population of neural cells with small processing capability, which are locally but stochastically interconnected.

We investigate the capability of the proposed model in temporal sequence processing to construct a representation of an autonomous dynamical system states, by profiting from the representational power of self-organizing maps. The idea is to process temporal sequences of observations on the dynamical system that violate the Markov property, using a recurrent scheme of competitive self-organizing populations of distributed neural cells, in order to build a state representation that maps one-to-one to the dynamical system states. Being distributed and local, this approach allows for asynchronous update regime and decentralized processing and scales well on parallel computers. Moreover, this approach is online, adaptive and unsupervised. These later properties play an important role when the dynamical system is non-stationary.

2.7 Plan of the thesis

Cellular computing is a recent trend in computation. Cellular structures originating from different scientific fields were used to carry out some more or less complicated computations. These structures emerged from physics like cellular automata and cellular neural networks, bioinformatics like artificial neural networks, and molecular biology like DNA processors. Cellular processors originating from biology are out of the scope of this manuscript.

This English introduction constitutes the second chapter of this manuscript. The third chapter starts by discussing the abstract models of computation and their related formal languages in order to facilitate the discussion of the computational power of the cellular models. Then we remind with the classical models used for parallel computation that we refer to as coarse-grain models. This is intended to help contrasting them against the cellular paradigm of computation that will be introduced after presenting its parent family, the fine-grain computation paradigm. Then we remind with the existing fine-grain models in computer science; we briefly introduce the theoretical basics of cellular automata, and move on to present the related cellular neural networks, then the prevailing models of artificial neural networks. We discuss the computational capabilities for each of these models and show that they are all capable of universal computation. At the end of this chapter, we present a definition of cellular computing as a parallel computation paradigm, and discuss the main properties of cellular populations and the main differences between a cellular processor and today's classical parallel models. Different properties on both the cell and the population levels are discussed, besides to the operational and behavioral aspects of cellular populations.

The fourth chapter discusses the problem of processing temporal data in artificial neural networks. We first talk about the temporal dimension of input data, and discuss time series processing and temporal sequence processing and focus on the conceptual differences between them and their domains of application and the most important required learning tasks for each of them. Then we

review the main approaches of dynamical neural networks that encode the temporal dimension of input sequences. In this review, we show how static neural networks are used in some tasks for temporal sequence processing, and show some network architectures that employ tapped delay lines with a multi-layer perceptron to implement a memory mechanism. We also present the prevailing neural architectures used for temporal sequence processing; the recurrent neural networks. During this review, we show how apt these architectures are to build a cellular processor. At the end we discuss the relation between artificial neural networks and the previously presented models of computation.

The fifth chapter discusses a special type of neural networks applied to spatial and spatio-temporal problems: the self-organizing maps. We first present the basic self-organizing map of Kohonen and talk about its properties. We then review the existing self-organizing architectures for sequence temporal processing, starting from using the algorithm in its basic form in temporal sequence processing using pre-processing or post-processing, then we present models that modify the algorithm in order to incorporate the temporal context in the inputs, and the models that use recurrent connections to feedback the map activity into its dynamics so that it encodes the temporal dimension of input sequences. We show how all these models based on the self-organizing map and used in temporal tasks are not cellular. At the end of the fifth chapter we present the neural field paradigm as a distributed competitive mechanism that is able compute the activity of self-organizing maps, and show how it enables to change the way that self-organizing maps are viewed, from being a neural network to being a population of neurons that fulfills the cellular computing requirements.

The sixth chapter presents a recurrent neural/cellular architecture that is able to process temporal sequences. The architecture is based on a recurrent scheme of self-organizing maps with an auxiliary delay structure. In this chapter we show how this architecture is applied to the problem of disambiguation of ambiguous observations on a dynamical system, and how it is able to construct a state representation that maps one-to-one to the dynamical system states. We also show how the architecture behaves in the case where the dynamical system is non-stationary. We compare the obtained results to a reference non-cellular recursive self-organizing algorithm, and discuss the differences in their capability to learn sequences and compare the stability and extensibility of both methods.

Chapter seven concludes this manuscript by discussing the applicability of the proposed cellular architecture in some problems like POMDP and dynamical modeling.

CHAPTER 3

Parallel Computing Paradigms: From Classical to Cellular Computing

Contents

3.1 Introduction

As mentioned in the previous chapter, no physical nor logical distinction could be made in the Von Neumann machines between where data is stored and where it is processed, and therefore, data

should travel from the memory to the processor and the way back at a speed inferior to the speed of light. Besides, the processor -even equipped with some parallelism techniques such as pipelining- still handles one basic instruction at a time, making the processing sequential in nature. For this reason, and given the physical limitations explained in the previous chapter, the processing speed will stay limited in Von Neumann based computers.

Improving the performance of computers, especially the processing speed, was a subject of ultimate interest for researchers in computer science through the past decades. Scientists and engineers from different domains deal with growing sizes of data and more complicated tasks every new day.

On the one hand, there are applications that need to process large volumes of data. This is challenging because data could be much larger than the available memory. The problem is called *data bottleneck*. The traditional solution is to resort to the " virtual memory " in which the computer *memory management unit* visualizes the available storage space (secondary memory) as a contiguous global address space in order to process some or all the tasks of the system using the virtual memory. The use of this technique is well known to be slow as the storage is not expected to perform as fast as the memory (RAM). Besides, the use of virtual memory usually requires high level expertise from the user. People generally tend to avoid the use of virtual memory and prefer, instead, to deal with reduced sizes of datasets while using less accurate operations to process their data. This also requires more time and effort in developing algorithms that work on partial data, not to mention data reduction techniques.

One good example of such tasks come from data mining domain. The classification of commercial data (e.g. clients or transactions data) deals usually with large datasets. Building a classifier (like a decision tree) based on a large volume of data needs the use of virtual memory, nevertheless, this could take a considerably long time. This problem is usually turned around by using incremental algorithms, except that the latter result in less classification quality than batch algorithms. This is also the main reason behind the existence of the "data selection" phase in the field of data mining and knowledge discovery in database (KDD) in general. As data grows, the problem becomes more and more serious.

On the other hand, some processing tasks take a long time to run. Two examples are Monte Carlo simulations and parameter sweep, to mention a few. Such algorithms need to deal with large number of combinations that should be processed separately, while there is no data interdependencies that could reduce the processing. This is why such tasks are called "embarrassingly parallel". One illustrative example on parameter sweep is the research for primary numbers in a large range. Here, no information about a sub-range of numbers could give an information about the whole range or other sub-ranges, therefore, there is no solution but to process the whole range of numbers one-by-one. When tasks of this nature are run on a serial Von Neumann processor, they scale linearly with time and the wait time becomes impractical, thus, people tend to run fewer simulations or coarser parameter sweep. The other solution is to distribute the task on several computing machines. In the example of primary numbers, this means affecting to each machine the computation of a sub-range of numbers.

Problems like the aforementioned ones, besides to the limitations of the general purpose computers due to the Von Neumann bottleneck discussed in section 2.2 motivated the research in parallel models.

Coupling multiple Von Neumann machines to work in parallel in order to multiply the computational power of these machines is an intuitive idea that was realized as a parallelization paradigm.

The other paradigm was inspired from biology and our physical universe in general. It relies on myriads of small computing elements (fine-grain), that work in unison to perform computation. This paradigm found all the necessary reasons to motivate it. First, it became clear that the foreseen limit of computational power of Von Neumann machines is inevitable, this is imposed by the physical laws conditioning their functioning at the very basic level. Second, there were already some mature fields that perform computation using agglomerations of small computing elements like artificial neural networks and cellular automata known since the 1940's and the 1960's respectively. Third, the new advancements in biology especially the information about microbiological cells and their DNA mechanism and the possibility to control them, even going further and engineering such cells. And fourth, and more importantly, the scientific ambition to better understand the physical nature of the universe, and the philosophical conjectures relating physics to computation. This new paradigm is tightly related to the philosophical points of view that consider our universe as a huge computer, this claim was recently proposed by Wolfram [Wolfram 2002], but Schmidhuber [1] pays our attention that the idea was first proposed by Konrad Zuse in 1969 and published in 1969 [Zuse 1969] (in German, [Zuse 1970] is the English version).

Indeed, all the processes that happen around us imply some form of computation, this intrinsic property of our physical world needs to be studied and simulated. Such understanding of our physical world allows us to hold the ambition that one day we will get more powerful and natural computing machines that need farther less energy and exhibit more scalability and fault-tolerance.

All these reasons motivated the distinction of a new computing paradigms which is intrinsically parallel, thus recently allowed us to hear about terms like "fine-grain computing" and "cellular computing". However, those terms were used in different contexts, leading to terminology ambiguity, that will be clarified in the coming sections.

In the following sections we dive more in parallel computing and remind of the existing models, their major families, and discuss their main properties and their pros and cons, then we present the fine-grain distributed models for parallel computing and present the main models pertaining to computer science. Finally, we present the cellular computing paradigm and show how it is a special case of the fine-grain paradigm, and discuss its main properties on the structural and operational levels. But before all, we start by reminding of some theoretical models of computation that will be referred to in later sections and chapters.

3.2 Models of computation

This section is intended to remind with the theoretical basis of the models of computation. As mentioned in the previous chapter, Turing and Church have put the Church-Turing thesis and defined computability that allowed for reinforcing research in the field of computation.

During the work on computing different functions, the need to classify these functions into groups of equally difficult functions had arose, and led to the appearance of a whole field in com-

[1] Schmidhuber home's page: Origin of the main ideas in Wolfram's book "A New Kind of Science", (Online, visited 03 August 2013).

puter science that is concerned in classifying problems according to their difficulty and is called *computation complexity theory*. Each class of equally difficult functions has been associated with a set of rules that describes the class. Each set determines the allowable operations in order to compute the class of functions. This resulted in different *theoretical machines* (or abstract machines) that compute exactly a related class. These machines became reference *models of computation*. The field that studies how efficiently problems are solved by the models of computations is called *the theory of computation*.

Theoretical machines start from a *start state*, and change their internal state during computation. The machine is in one state at a time, called the *current state*. The change from the current state to another occurs after processing an event or input, so the machine makes a *transition*. Each machine has a set of terminal states called *final states*, when reached, computation stops. When the machine changes its state, it could output some value. The transition between distinct states implies that the functions recognized by these machines are the discrete functions. If an input string of symbols causes the machine to reach a final state, it is said that the machine *accepted* the string, if not, it is *rejected*. In the language of the theory of computation, the problem of deciding whether a string will make the machine reach a final state is called a *decision problem*, and the input string is called *problem instance*.

Each machine is related to some type of strings, normally generated by another set of rules that generates strings accepted by the machine, called *formal grammar*. The set of strings generated by such a grammar and accepted by the related machine is called *formal language*. The field that studies these languages is the *formal language theory*. Languages are of particular importance in computation, and have been classified into different classes, the reference and most important one is the Chomsky hierarchy [Chomsky 1956]. There exists two standard ways to characterize a formal language, either by generating it by a grammar, or by specifying the abstract machine that recognizes it.

There are several models of computation, the difference between them is related to the set of allowable operations or rules. The most famous models are finite-state automata with their different forms, and the Turing machine, each one of those machines accepts a language that belongs to the Chomsky hierarchy.

3.2.1 Finite-state machines

A Finite-state automaton (FSA), also known as finite-state machine (FSM), is a machine that has a fixed number of different internal states. Each machine changes its states after processing an action. Some real life applications implement such behavior like the vending machine, the elevator, the light switch, to name a few. Traffic lights are another example that differs from the previous examples in that it makes state transitions autonomously.

Finite state machines can be of different types depending on their defined behavior, namely, transducers, classifiers, sequencers and acceptors. The acceptor FSM is of particular interest because it can be used to test a condition, and if an input string of symbols is a member of some language.

Figure 3.1 shows a simple acceptor FSM that decides whether a binary input string of any length contains an even number of 0's or not. The computation begins with the FSM in the start

state. When the test is successful it reaches one state of a set of final states, also called *acceptor states*. The sequence of states from the start state to the acceptor state is called the *accepting path*. In Figure 3.1 the start state and the acceptor state are $S1$.

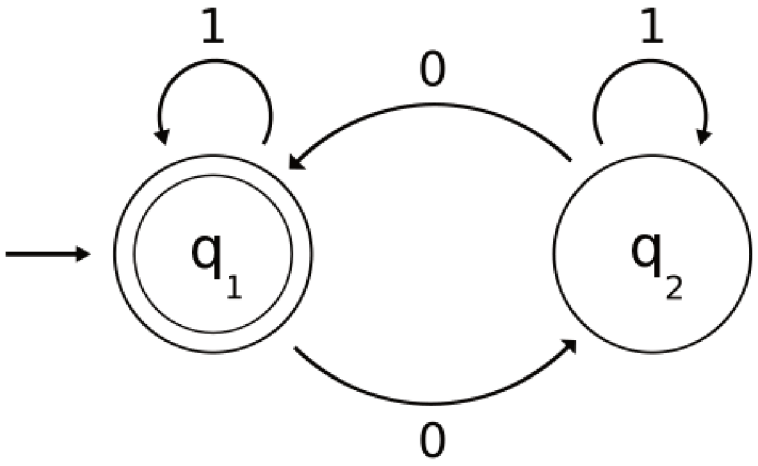


FIGURE 3.1: A finite state machine of type "acceptor". The machine tests whether an input string of bits contains an even number of 0's. The start state is distinguished by receiving an arrow coming from nowhere. The end state is distinguished by a double circuit. q_1 is the start and end state in this example.

The class of functions computed by FSMs are called *regular* and the languages accepted by FSMs are called **regular languages** according to Chomsky hierarchy [Chomsky 1956]. Conversely, a language is regular if there is some FSM that accepts it. A simple example of a language from this class is a^n with $n > 1$, which can be accepted by a two state automaton in which the reading of the first a values causes the transition from the start state to the acceptor state, where it remains whatever a's came after.

If each state of the FSM has exactly one transition for each possible input event, then the FSM is called **deterministic finite automaton** (DFA). If more than one transition is possible, then an input string may possibly have more than one accepting paths. In the case then it is a **non-deterministic automaton** (NFA). The FSM shown in figure 3.1 is a DFA.

Formally, a DFA can be modeled as a quintuple $M = (Q, \Gamma, q_0, F, \delta)$, where Q is a finite non-empty set of states, Γ is a finite non-empty set of symbols called the *alphabet*, q_0 is the initial state, F is a set of final states and is a subset of Q, and δ is the state transition function that gives the next states defined as follows: $\delta : Q \times \Gamma \mapsto Q$. When the machine is in the state $q_i \in Q$ after reading a symbol $\gamma_j \in \Gamma$ it moves to the state $\delta(q_i, \gamma_j) \in Q$. In this case the FSM is deterministic, hence it is a DFA.

Let $B = \gamma_1\gamma_2 \cdots \gamma_n$ be a string over the alphabet Γ, the automaton accepts the string B if there exists a sequence of states $q_0q_1 \cdots q_n$ (starting with q_0) such that $q_{i+1} = \delta(q_i, \gamma_{i+1})$ for $i = 0, \cdots, n-1$ and $q_n \in F$.

When $\delta : Q \times \Gamma \mapsto \mathcal{P}(Q)$ with $\mathcal{P}(Q)$ is the power set of Q, i.e. the set of all subsets of Q, we have $\delta(q_i, \gamma_j) \in \mathcal{P}(Q)$, and the FSM is a non-deterministic finite automaton. The NFA accepts the string B when $q_{i+1} \in \delta(q_i, \gamma_{i+1})$ for $i = 0, \cdots, n-1$ and $q_n \in F$.

Another important class of FSMs is the transducer machines, also called translators, because they are useful in computing functions. They take an input string and generate an output string of the same length. The most important transducers are Mealy machine [Mealy 1955] and Moore machine [E.F. Moore 1956]. Formally, a **Mealy machine** is a 6-tuple $(Q, \Gamma, \Sigma, q_0, \delta, \lambda)$, defined like a FSM with Σ the finite output alphabet, : $\lambda : Q \times \Gamma \mapsto \Sigma$. When the machine is in the state q_i after reading a symbol γ_j it write a symbol $\lambda(q_i, \gamma_j) \in \Sigma$.

Moore machine is similar to Mealy machine, the difference is that the output symbols are written after the next state has been reached. The output depends only on the new state rather than on the previous state and the input, therefore in Moore machine λ is defined as follows: $\lambda : Q \mapsto \Sigma$.

It is possible also that a FSM generates no output, it can only process inputs and perform state transitions. A FSM of this type is called a **semiautomaton** or **transition system**. Semiautomata are Moore-type automata that can be expressed as a quadruple (Q, Γ, q_0, δ). Another Moore-type automata are **autonomous automata**, which are also without output, but their input set Γ contains only one element.

FSMs are pretty weak, mainly because they don't have a dynamic memory. Their memory is determined by a finite number of states. For example there is no FSM that can decide whether the number of 1's is greater than the number of 0's in a string of the form 0^n1^m in *any* binary input string using a pre-fixed memory only. It is impossible to build a FSM that remembers the number of 1's for every value of n because this requires to have a different state for each value of n, while the number of states in a FSM is finite by definition (Q is finite).

3.2.2 Pushdown automata

Languages like $L = a^n b^n$ that contain strings like $ab, aabb, aaabbb, \cdots$ for all $n > 0$ belong to a class called **context-free languages**. They are recognized by FSMs provided with infinite stacks, called **Pushdown automaton** (PDA), which enable to push and pop symbols. In the case of $a^n b^n$, the PDA reads an a and pushes some symbol onto the stack as it counts the number of a's, and pops the pushed symbols while checking if there are the same number of b's. However, PDA can't process languages like $L = a^n b^n c^n$, because after pushing some symbols for the a's count and popping then to check if the count of b's is the same, it will have nothing left in the stack that helps to check if the number of c's is the same. The language $L = a^n b^n c^n$ belongs to a class of languages in the Chomsky hierarchy called **context-sensitive languages**.

3.2.3 Turing machine and the general-purpose computer of Von Neumann

A Turing machine consists of a control device and an infinite tape in one or two directions. At the beginning of computation a binary input string is written on the tape surrounded by infinite *blank* symbols on both sides (this implies a finite number of symbols on the tape). The tape is accessed by a read/write head that is located at the leftmost of the input string at the beginning of computation. At each step during computation, the head reads a symbol from the tape, checks the current state of the control device, and performs three operations: it writes a binary symbol on the tape under the head, moves the head one step to the left or to the right, and changes the current state of the control device. The computation stops when the control device reaches one of the final states called the

halting states or when it reaches a configuration where there is no transition defined. The result of computation is the string left on the tape after halting. Thus, the input/output of the computation is the string written on the tape before and after the computation. The previous assumption of TM dealing with binary symbols doesn't affect TM generality, as every other set of symbols could be encoded in binary ones. Figure 3.2 illustrates a Turing Machine.

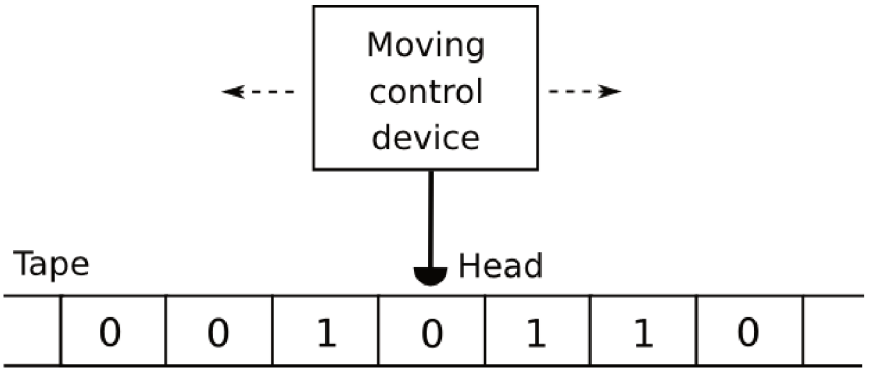


FIGURE 3.2: A Turing machine.

Formally, a Turing machine is a 7-tuple $TM = (Q, \Gamma, b, \Sigma, q_0, F, \delta)$ where Q is a finite non-empty set of states, Γ is a finite non-empty set of symbols of the tape symbols (alphabet), b is the *blank* symbol and the only symbol allowed to occur infinitely on the tape, $\Sigma \subseteq \Gamma \setminus \{b\}$ is a set of input symbols or the alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final accepting states in which the machine halts, and δ is the transition function defined as: $\delta : Q \setminus F \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ where L is the left shift of the head and R is the right shift.

The control device in a TM is an FSM that could be implemented using an action table with a state register. Conversely, an FSM is a restricted Turing Machine where the head can only perform read operations, and movement is restricted from left to right. The control device in a TM implements a specific FSM. That is similar to a computer with a ROM memory that can't execute but a single program.

A universal Turing machine (UTM) is a theoretical machine that can simulate an arbitrary TM on arbitrary input, i.e. it is programmable: it is possible to change the FSM of the control device. It is *universal* because any *symbolic* computation system can be emulated using a Turing machine [Kolen 2001]. Also, a two-stack PDA is equivalent to Turing machine, so that each stack plays the role of one direction of the infinite two-directional tape. Note that, a Turing machine without the tape, is only as powerful as an FSM.

If the Turing machine was restricted to only write on a portion of the tape which length is bounded by some linear function of the input length, then we get the **linear bounded automata** (LBA) which can accept context-sensitive languages in the Chomsky hierarchy like $a^n b^n c^n$.

The TM can theoretically compute any function that our today computers are able to compute. TMs are more powerful than other theoretical machines, as at least, today computers can compute (decide) using a simple program whether the number of 1's is greater than the number of 0's in *any* binary input string.

Recursive languages are those languages for which Turing machines halts and accepts every string of the language, and halts and rejects every string not in the language. An example of

such language is the *True quantified Boolean formula* which are propositional logic formulae like $\forall n \in \mathbb{N}, 2.n > 2 + n$ that could be evaluated by Turing machine.

There are languages for which the Turing machine also accepts all their strings, but if a string not in the language is presented, the machine response is unpredictable: it either halts and rejects or returns no answer (loop forever: no halt). These languages are called **recursively enumerable languages** or partially **decidable languages**. An example of such language string is the Halting problem and Entscheidungsproblem decision problems discussed in the previous chapter. Computing the languages that Turing machine can't compute are referred to as *super-Turing computation*.

Today computer architectures are based on the Von Neumann architecture. They are general-purpose computers that work as a UTM: the program can be changed, hence the name "stored program computer". Figure 3.3 shows an illustration of Von Neumann general-purpose computer.

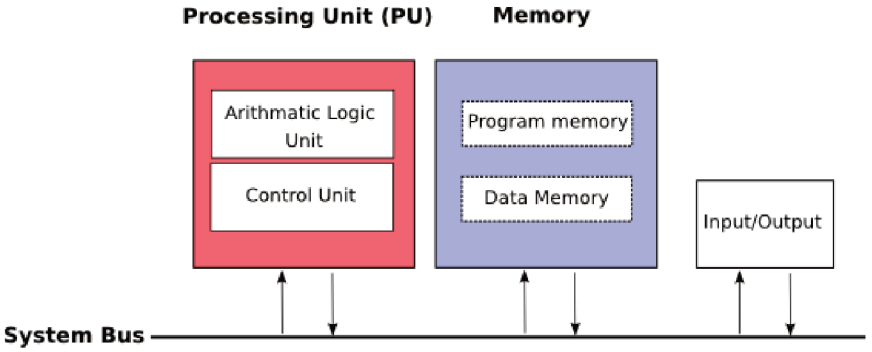


FIGURE 3.3: Von Neumann stored program architecture

However, in [Kolen 2001], Wiles et al. argue that Turing machines stay a theoretical model. Strictly speaking, our today computers with their finite memories are really finite-state automata, with large number of states ($2^{2^{35}}$ possible states for a 4-gigabyte computer) that enable us to think of them as Turing machines. The perfect computation is not possible all times, especially when the input size exceeds some critical limit, which result with catastrophic failure of the type "stack overflow", or "out of memory". Maybe, someone has been fed up by such error messages, and thought of computing in parallel.

3.3 Massively parallel computing models

3.3.1 Definitions and terminology

The idea of using parallel models that contain multiple processors aims essentially at processing more than one instruction at a time. At this point, one should ask multiple questions, the first of which, naturally, is the nature of basic processing entities. This criterion guides to distinguish two broad families of parallel computing models depending on the degree of simplicity of the basic processor:

- *Coarse-grain models* that are based on complicated sequential processors and are the main stream paradigm in parallel computing.

- And the newly defined and promising paradigm in computation: the *fine-grain models* that are based on a way far simpler processors relying on the *connectionist* paradigm. Cellular computing paradigm is a subclass of this one as explained later.

Although it was newly used to signify computing models, the term "fine-grain" (or "fine-grained") is not new. It was used on both software and hardware levels. On the software level, it is usually used to describe code-related elements, for example, in object-oriented programming, replacing an object that holds many variables with many objects with less variables and simpler behavior is considered as fine-grain programming. This is usually used in flexible environments for the purpose of data hiding, or in order to favor low coupling and high cohesion, and make objects simpler to develop, test, modify, reuse, and deploy. However, this technique increases network calls. Fine- and coarse-grain techniques are also familiar terms in parallel programming domain like fine-grain adaptation of program behavior (one example is [Kang 2009]).

It should be emphasized that the terms "fine-grain" and "coarse-grain" in this work, concern the conceptual architectures of parallel computing. In this context, coarse-grain models are intended to refer to massively parallel computer architectures built of the traditional powerful sequential processors and deal with the symbolic paradigm of information representation, while fine-grain architectures refer to massively parallel models that deal with the connectionist paradigm of information representation.

It is noteworthy to mention that even coarse-grain architectures in our defined sense, could be found in literature as a subject of classification between coarse-grain and fine-grain depending on the number of processors. When some parallel computing structure contain a large number of processors, say 10000, or uses processors which contain many cores on the same die, it could be referred to as fine-grain parallel architecture. However, we restrict ourselves to the previously defined terminology.

3.3.2 Synchronization

When talking about coordination between multiple processors, a general issue related to distributed and parallel systems arises: how computation carried out by different processors is coordinated? Which processor computes which operation or task, and at which time relative to other processors? This leads to distinguish two major update regimes: the *synchronous* and *asynchronous* ones. In the synchronous update regime, there exists a global device called "phase clock" [Boulinier 2005] that defines a discrete time framework allowing to distinguish successive "phases of computation". The phase clock maintains the current phase number ($\tau = 0, 1, 2, \cdots$) so that all processors perform their operation of tasks (we can say: updated) in the phase τ, then this is repeated for $\tau + 1$, and so on. The notion "phase clock" is not to be confused with the notion of *timestep*. The latter is also a time discretization into successive instants ($t = 0, 1, 2, \cdots$), but it is reserved to control the computation carried out by processors within each phase. In the asynchronous update regime, there is no need for a global signal, instead, it is sufficient to specify an order, so that processors perform their operations sequentially, one after the another in a cyclic form. However, it should be ensured that no processor continues to compute in a new cycle, before all other processors do in the previous

cycle. There are several scenarios to implement asynchronous update, either by specifying a fixed order, or a random order for each cycle [Schönfisch 1999].

3.4 Coarse-grain models

This is the trend in parallel computing and the *classical paradigm* of building parallel computing architectures. This section is a reminder of these models, presented here in order to contrast its specifications with those of fine-grain and cellular models presented later in this chapter. A model from such a class is an agglomeration of *powerful sequential processors* (of the Von Neumann type) that can be organized in multiple ways. A single Von Neumann processor is illustrated in Figure 3.3.

Different design criteria arise when thinking of coupling processors together: the way these processors are organized and interconnected and the way that such processors communicate and coordinate to split the processing tasks and share resources like memory between them, specifically, whether they share memory or not, and last, how tight are those processors coupled together (geographically, and depending on connection speed whether it is a high speed bus or merely some type of area network connections) .

Some design criteria are interdependent, for example, sharing the storage medium between processors makes the communication between them more likely to occur through the shared storage, this in turn makes it more likely that they are communicating with the storage through buses rather than network connections. Of course, there are configurations that violate this rule, where, for example, processors share the storage while communicating through the network, or, tightly coupled processors like in some supercomputer configurations could have each its own memory. But in general, criteria tend to cluster together, leaving room to a possible coarse classification of parallel coarse-grain architectures into *tightly-coupled* and *loosely-coupled* ones.

Processors in coarse-grain models are complex and capable of universal computation, and the number of processors in such parallel models varies and increases with time, for example Blue Gene supercomputer is one hardware implementation of coarse-grain models, it runs in its Blue Gene/P generation 250000 processors that has a computation power that exceeds 1 peta flops (10^{15} floating point-operating per second) , and the Chinese Tianhe-2 that consists of 3.12 million cores in 16000 processors exceeds 33 peta flops, while it is estimated that there are some $10^8 - 10^9$ machines connected to the world wide web, which could also be thought of as a parallel model. The model proposed in this thesis is run on a parallel coarse-grain architecture called `InterCell` .

It is important to distinguish between the theoretical basis of the models of parallel computation paradigm from their hardware implementations. The latter are listed in this manuscript for the purpose of illustration of the underlying model. In the previous example, Blue Gene is a hardware implementation that illustrates coarse-grain models, but it is not the main concern per se.

3.4.1 Tightly-coupled multiprocessors

Multiprocessors are said to be tightly-coupled when a group of sequential powerful processors communicate either through a shared memory or over a shared bus. Example of such class are

supercomputers. Two main configurations could be distinguished. The first is when processors share the same memory thus they are called *shared memory machines*. In this case, the communication between processors occurs through writing to and reading from the global shared memory. Figure 3.4 illustrates this configuration.

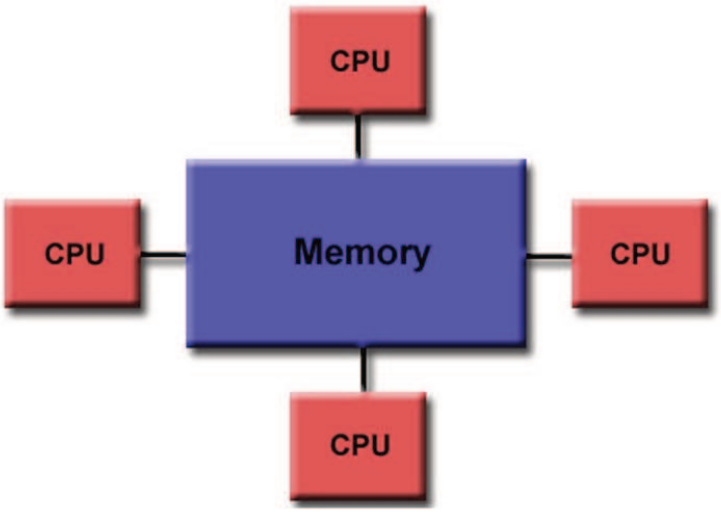


FIGURE 3.4: Tightly-coupled shared memory multiprocessor models. Extracted from [Barney 2013]

The second configuration could be found when processors don't share memory but instead, each processor or small agglomeration of processors has its own memory thus they are called *distributed memory machines*. Communication in such configuration occurs through a shared bus (typically, of type backbone). Figure 3.5 illustrates this configuration.

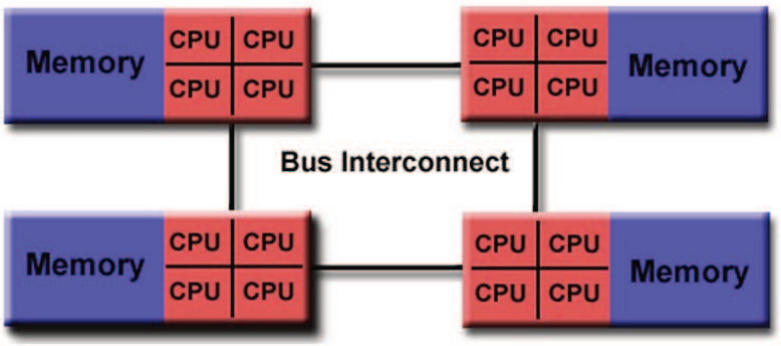


FIGURE 3.5: Tightly-coupled distributed memory multiprocessor models. Extracted from [Barney 2013]

Some models rely on bus-connected processors that have each its own memory but share the address space. In some other models, processors could be interconnected to use both shared and distributed memories, i.e. all the processors share a global memory, but at the same time each processor or small agglomeration of processors share a small memory or a cache. Both models are tightly-coupled but could be seen as hybrid between shared and distributed memory machines.

This is somehow similar to what is found in our today laptops and PCs, except that they contain one processor with multiple cores packed in the processor socket, and cores often share the cache memory.

In both tightly-coupled configurations, the multiprocessor system has its own input/output peripheral devices, and a shared operating system that is the same on all processors.

Apart from architecture details, tightly-coupled systems could be classified depending on other criteria. Another important classification for such systems is depending on how instruction streams and data streams are processed. This classification is called *Flynn's taxonomy* (figure 3.6) for computer architectures:

- When all processors select the same instruction from the instruction pool, but each processor executes it on a different piece of data, then the obtained model is called *Single Instruction, Multiple Data (SIMD)*, figure 3.6(c) shows the schematic of this model.
- Similarly, *Multiple instruction, Multiple Data (MIMD)* models are obtained when processors execute different instructions on different data as in figure 3.6(d).
- *Multiple Instruction, Single Data (MISD)*, as depicted in figure 3.6(b), is a rarely materialized model. Its main use is in task replication where redundant instruction execution is needed in fault-tolerant computers.
- While the *Single Instruction, Single Data (SISD)* model refers to the Von Neumann sequential processor, it is depicted in figure 3.6(a).

A difficult task in shared memory machines is how to ensure scalability. The problem arises when processors are added or the shared memory is increased in size. Distributed memory machines became more common as they are more scalable. However, they raise another problem which is managing the message passing between them, which is used for processors communication and coordination. In general, programming both types of machine is not straightforward and requires advanced skills and considerable programming time.

3.4.2 Loosely-coupled multiprocessors

This design approach is the same as the main stream sense of the word as discussed in 2.2. In this approach, the system is compound of different machines, each of them is an autonomous machine that contains its own processor (or multiprocessors) and input/output peripherals. Different machines in the system could have different operating systems and different hardware manufacturers and settings. Such systems are loosely-coupled and distributed, because the system machines don't communicate neither through shared memory nor through a shared bus, instead, they communicate using a network that could span a wide geographical area like in Wide Area Networks (WAN). Figure 3.7 shows an illustration of loosely-coupled systems.

The World Wide Web that spans the globe, is itself a valid example of loosely-coupled distributed systems. One could claim that distributed systems are expected to carry out a common goal, which is not the case of the Web. To answer this claim, one could recall the example of large computation tasks that are partitioned into smaller ones and sent over the web to users around the

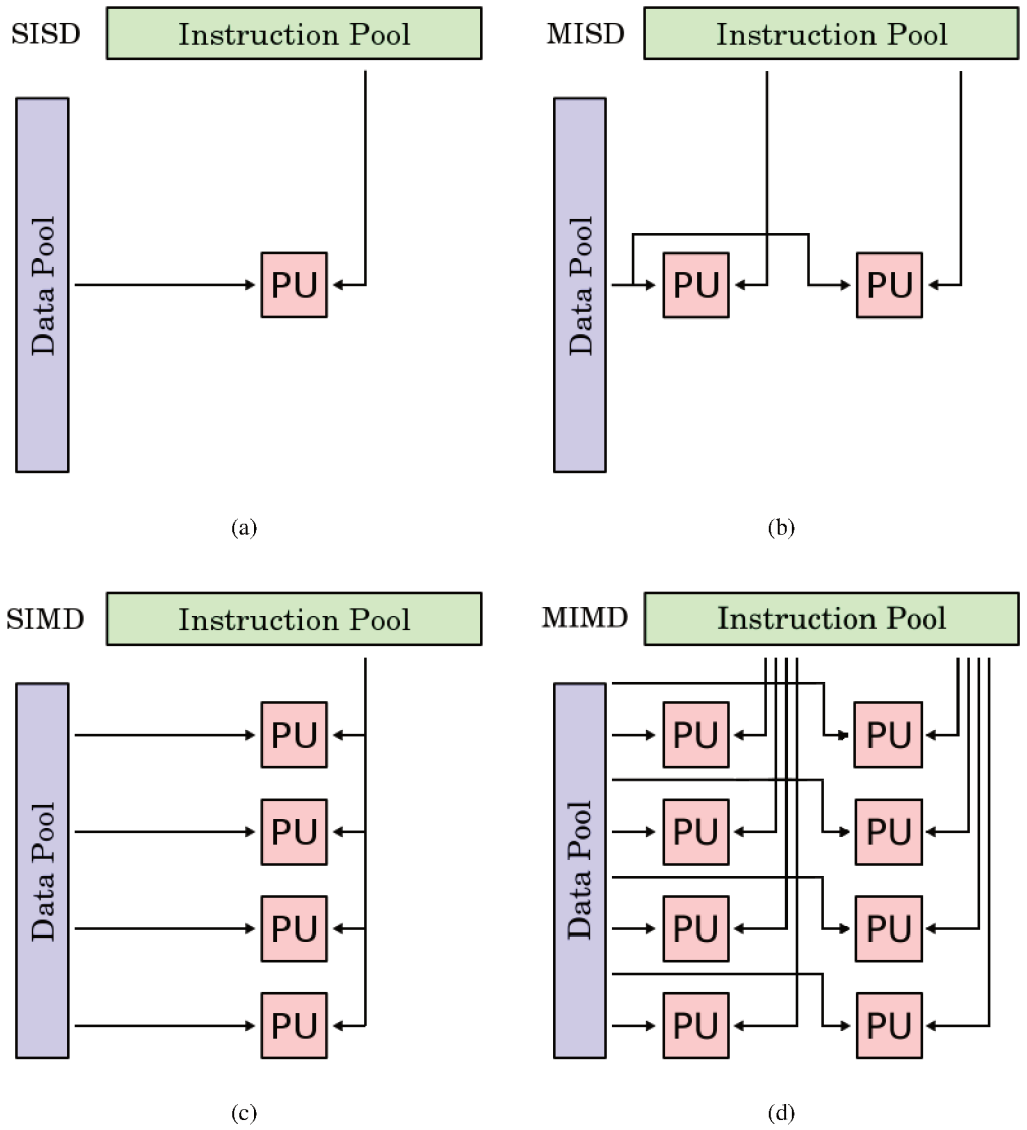


FIGURE 3.6: Flynn's taxonomy for computer architectures (extracted from Wikipedia [Wikipedia 2013]).

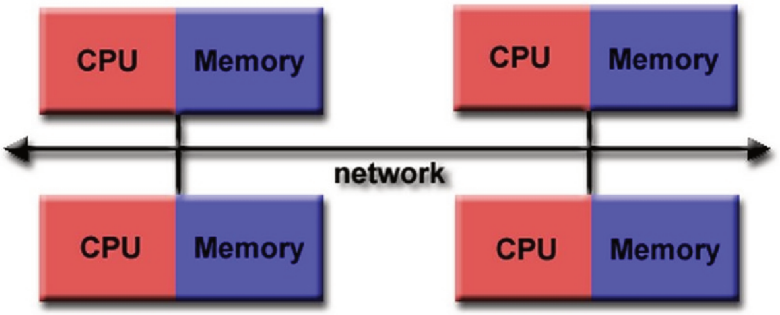


FIGURE 3.7: Loosely-coupled multiprocessor models. Extracted from [Barney 2013]

globe who accept to run a small part of the computation on their machines (Example, SETI experiment that uses internet-connected computers in the search for "extraterrestrial intelligence" [2]).

In order to facilitate communication and coordination between machines, a middle-ware is normally installed on each machine. The middle-ware aims to set a common communication language between different machines, so they can use it to send and receive messages over the network.

Loosely-coupled systems scale better than their tightly-coupled counterparts; they could be extended, modified, and reconfigured more easily. They also allow for more flexible energy dissipation management, leaving place for integrating large number of machines, and thus, for more computational power.

Their disadvantages compared to tightly-coupled systems is that they are harder to program, control, monitor and predict due to the possible different operating systems and access right issues over the network. Information security is another problem to solve. Their scalability implies an additional task of keeping track of topology change. Setting up machines to work synchronously in such multiprocessor systems is harder than in tightly-coupled ones, so they are mostly run in asynchronous configuration (see 3.3.2).

3.4.3 Hybrid computing paradigms

The classification of coarse-grain multiprocessor models between tightly-coupled and loosely-coupled is somehow a rigid one, as there exists some hybrid models that combine not only mixed architectural aspects but different processor types.

Some newly developed computation hardware implementations like the GPU (Graphics processing unit) allow for parallel computation and are not that "coarse". GPUs are processors that contain a big number of cores (say, thousands) compared to Von Neumann processors. GPU processors are often SIMD implementations.

GPUs were engineered by Nvidia in 1999. It was basically a response to the demanding computation in image processing tasks in graphics cards. In the First GPUs, each core performed a fixed function. Research and development in GPU resulted in programmable GPUs with familiar programming languages like C/C++, and FORTRAN, besides to special development platforms like OpenCL. The program needs to run on a framework such as the dominant CUDA one. As it is possible now to program and use in various application domains, GPUs allow for GPGPU (general purpose computing on graphics processing units).

The use of hybrid architectures consisting of GPUs and a CPUs is referred to "GPU computing". GPUs with their up to thousands cores able to process up to one tera flops (like in Tesla GPUs) are assigned parallel computations, while CPUs with their few cores are optimized for serial computations. Such architectures are now being used in some personal computers like Apple MacBook Pro to attain higher computation speeds.

With the appearance of processors like GPU which allow for integrating a high number of processing cores, people now talk about the "new" Moore's Law: the number of processor cores on a chip will double roughly every 18 month [Furber 2009].

Finally, it is suitable to mention that recent supercomputer architectures benefit from the power

[2] SETI home page, (Online, visited 24 October 2013).

of GPU computing. Manufacturers tend to produce hybrid but tightly-coupled architectures that contain both CPUs and GPUs as illustrated in Figure 3.8.

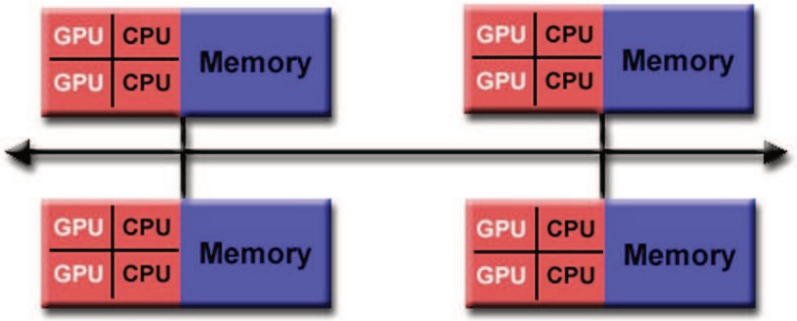


FIGURE 3.8: A hybrid multiprocessor model used in some supercomputers. It combines CPUs and GPUs in one architecture. Extracted from [Barney 2013]

3.5 Fine-grain distributed models

Massively parallel coarse-grain computers developed by computer scientists consist of a relatively small number of processors, each of them can be described to be of high complexity. Their application in various research and engineering fields confirmed their high computation power. These systems allowed to realize complex and demanding computation tasks, although, they are not available to all people because of their high cost and need for special engineering and programming skills. Besides, some computational tasks seem to be more demanding like in statistical physics and biology where tasks could be computational-greedy. Even Blue Gene that has been used (in the context of the Blue Brain project) to simulate only 1% of the human cortex (1.6 million neurons with 9 trillion connections), seems to be limited and far from fulfilling scientists ambition in building a complete model of the human brain.

Natural systems like the human brain (and that of other species), implement an altogether different concept for parallel computation. Rather than depending on a small number of costly, complex, energy demanding processors, they depend on a very large number of highly interconnected low-energy simple processing entities.

Although each neuron in the brain computes a simple function of its inputs and synapses store a small amount of data, undeniably, the brain is a robust, fault-tolerant computation system. Those properties stem from the high number of neurons (processors) working in parallel, and from their high connectivity reflected by the high number of synapses (links) between them.

Inspired from the brain and its computational properties, artificial neural networks have been proposed and used in computer science and found a wide success as witnesses their intensive use in artificial intelligence. The high connectivity and synergical cooperation between their processing units announced for the connectionist paradigm in computer science. Some of the neural networks models have their processing units working in parallel, thus they are considered as parallel distributed processing structures (PDPs).

Cellular automata (CA) is another connectionist model that was originally studied in physics (growth of crystals studied by Stanislaw Ulam) and biology (self-replicating systems studied by Von Neumann) in early 1940's, and was later studied in computer science. They are populations of interconnected cells that run in parallel and carry out computation. By inspiring from both ANNs and CAs, Cellular Neural Networks (CNNs) have emerged as a new connectionist paradigm for computation.

These paradigms rely on a different approach in building massively parallel computing systems. The number of basic processors and the granularity of processing units in such models is much higher than in coarse-grain parallel models, this is why they are called **fine-grain models** that define the fine-grain paradigm in computation. With few exceptions, fine-grain models are parallel computation models, one exception is related to artificial neural networks and is discussed in a later section. While the term "massive parallelism" is frequently used to signify parallel models containing large number of processors including coarse-grain models with thousands of processors, Sipper [Sipper 1998b] proposes the use of the term "vast parallelism" with fine-grain models in order to distinguish the different order of magnitude of processors number that could be expressed by the exponential notation 10^x.

Fine-grain models are theoretical models of computation that shouldn't be confused with how they are implemented. Often, fine-grain models implementation are simulations on coarse-grain parallel machines. However, this is not the only possibility, consider cellular automata for example, they can be implemented (and are implemented) on parallel computers, and using CMOS technology, they are also implemented using nano-crystal semiconductor materials (quantum dots). What is important to retain here, is that although most fine-grain models are implemented as coarse-grain parallel machines programs, they shouldn't be perceived merely as programs, but also as different theoretical models that have different computation capabilities. Coarse-grain implementations limit the parallel computation, and thus, they limit the speed of computation of fine-grain models by their own capability of parallelism, but they don't affect the other aspects of their computational power, such as their emergent behavior.

In the following sections, we present the major fine-grain computing models in computer science, CA, ANNs, and CNNs and discuss their special mechanisms of processing and memory.

3.5.1 Cellular automata

Originally introduced by Von Neumann [Neumann 1966], cellular automata (CA) are decentralized systems that have the potential to perform complex computation with high robustness and efficiency and model the behavior of complex systems in nature [Mitchell 1996].

A *classical* cellular automaton consists of a large number N of cells spatially extended over a lattice called the *cellular space*, topologically distributed on the nodes of regular grid (1D,2D,..). Each cell on the lattice is connected to its neighbors following the same connectivity pattern as all other cells. The whole system is controlled by a global clock that provides an update signal for all cells working in parallel so that they are updated synchronously.

The CA can be seen as a population of identical FSMs interconnected in a regular way. Each cell

in CA implements a semiautomaton FSM (see 3.2.1). The cell's FSM reads the states of neighbor cells and computes a value as its own state, the state value can also be considered as the cell output. The cell state is one of a set Q of the FSM states with $k = |Q|$ possible states. A cell with index i computes a state q_i^t at time t with $q_i^t \in Q$. The cell neighborhood is the topographical neighbors connected to it, it may also include the cell itself, thus in the one-dimensional case, at a time t, the cell has $\mu = 2r + 1$ neighbor states, where r is called the *radius* of the CA. A cell's FSM takes as input the states of neighbor cells at time t and computes its next state $q_i^{(t+1)}$ at time $t + 1$ following a transition rule δ, also called *CA rule*, which is a mapping from the set Q^μ of neighbor states to the set of cell states Q.

When $k = 2$ then the state is binary and its value is either 0 (called the *quiescent state*), or 1 (called the *active state*). In case of binary states, the CA rule is often displayed as a lookup table which lists all possible combinations of neighbor states together with their corresponding output bits.

Cellular automata are closed systems: cells have no connectivity to any external input source, and the cell's only input is the states of the neighbor cells whereas its output is its state itself (the semiautomata generate no outputs). The cells initial states are loaded before the run and their states are read during or after the run. Thus CA behave as autonomous dynamical systems, they are also discrete and deterministic.

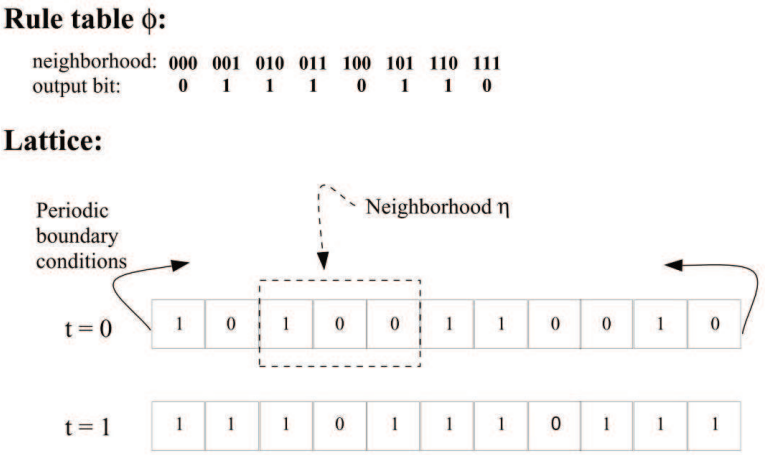


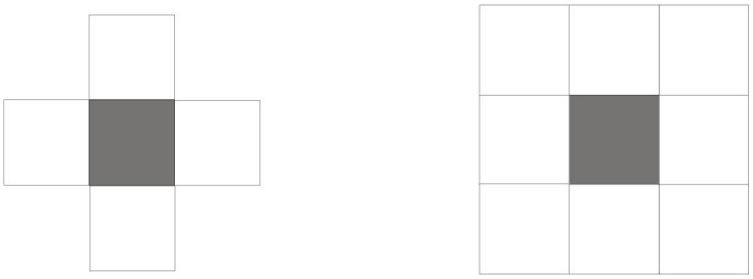
FIGURE 3.9: One dimensional CA shown in two successive timesteps. Each cell has $k = 2$ possible states 0 or 1. $r = 1$ thus the neighborhood of a cell is itself and the cells to the left and the right. The boundary condition is handled by wrapping as show the arrows on the borders. The cell next states are computed following the transition rule ϕ expressed as a rule table in this figure. Extracted from [Mitchell 1996].

Figure 3.9 shows a 1-dimensional cellular automaton, with binary states. The cell neighborhood consists of the cell and its two neighbors to the left and right. As it is a finite automata, the CA is wrapped like a ring for boundary cells update: The leftmost cell is considered as connected to the rightmost cell and vice versa. The CA rule ϕ in figure 3.9 is Wolfram's rule 110 [3].

In a 2D CA cells are distributed on the nodes of a grid where the lattice is the Euclidean space.

[3]The rule's numbering scheme is proposed by Wolfram, the output bits are ordered lexicographically and are read from right to left, so the binary "0111 0110" is read "0110 1110" and thus it equal to the decimal 110

The cell neighborhood could be defined in different ways, the most common ones are Von Neumann and Moore neighborhoods, they are illustrated in figure 3.10.



(a) Von Neumann (b) Moore

FIGURE 3.10: Famous neighborhood functions in 2D CA: (a) The Von Neumann Neighborhood. (b) The Moore neighborhood. The shaded cell is the one to update.

One of the first CA to be studied and the most famous example of 2D cellular automata is John H.Conway's Game of Life, or "Life" for short. The grid could be infinite or wrapped in both directions giving the shape of a torus. In Life, $k = 2$, the state could be 0 (or "dead"), or 1 (or "alive"). The cell neighborhood μ_i is the Moore's neighborhood 3.10(b). The Life rule is *totalistic*: the next state of the cell depends only on the number of alive neighbor cells, regardless of their relative positions. This rule is detailed in table 3.1. Life is normally initialized to a limited number of cells in alive states.

Number of alive neighbors	Current state	Next state
2-3	live	live
0-1, 4-8	live	dead
3	dead	live
0-2,4-8	dead	dead

TABLE 3.1: CA rule ϕ for Conway's game of life

Regarding initial conditions, Life is known to have initial conditions that yield a complex behavior, besides to a number of initial states that yield interesting stable patterns. Some patterns oscillate in various periods, some others yield gliding localized structures that resemble to spaceships (an overview of some structure can be found in [Berlekamp 1982]).

Conway conjectured that there is no initial condition that allows for unlimited growth of the number of the living cells and offered $50 to anyone who can prove or disprove it [Downey 2012]. He paid the money to Bill Gosper who invented in 1970 the "glider gun" which is a structure that emits every 30 timesteps a new propagating structure of living cells (like only 5 alive cells) called "gliders".

3.5.1.1 Universality

The importance of what is now known as Gosper's Guns is that they led to prove that the Game of life is Turing complete. For this purpose, and instead of emulating a Turing machine, basic

logic functions were built up from interactions between glider streams. Some properties of gliders and glider guns were used to build these functions. For example, a stream of gliders shot off from a glider gun is interpreted as a stream of bits, where the presence of a glider means "1" and the absence (a hole in the stream) means "0". Some types of gliders change direction when they collide, others annihilate, the latter property was used to build the NOT gate. The same technique was used to build the Or gate with the help of a glider gun that performs delay. Adding a structure that destroys other ones (called "eaters") are used to build the AND gate. Other amusing techniques were invented to play with glider streams for information copy, delay, and storage in the form of circulating gliders. These techniques in building basic logic functions could be used to compute any recursive function, and thus proving at least a universal computer could be embedded in Life.

As mentioned in 2.4, Rule 110, one of the simplest CA, was proved by Matthew Cook to be a Turing machine in the one dimensional case [Cook 2004]. In the proof, Cook used the same principle of gliders to simulate the tape and functioning of Turing machine. The glider in the one dimensional case is a point or a structure of neighbor points moving with time.

3.5.1.2 Life on the edge of chaos

Rich computation capabilities arise when a CA rule makes the CA work in a maximal complexity, complexity in the context of CA means that information exchange between the CA cells is maximal. Some CA rules endow the CA with the sufficient complex behavior that allows the CA to exhibit the capability of universal computation.

Some CA rules lead to a trivial behavior and poor computational capability, whereas others lead to a useless chaotic behavior of the CA. Stephen Wolfram [Wolfram 2002] claimed that Rule 110 and Life are neither completely stable neither completely chaotic, this is called *the edge of chaos*. The latter is a metaphor that means that a biological, physical, economic or social system has its maximal complexity when it operates in a region between order and randomness. The first researcher that clearly pointed out to this phenomenon is Christopher Langton. A good coincidence is that he was also working on cellular automata.

Langton [Langton 1990] parametrized the space of all possible CA rules by a variable λ, increasing λ gives different rules that lead to increasing the CA behavior complexity. He observed that there exists an upper and lower limit on the complexity of a CA system (in term of λ values), where, in between, the complexity of the system is non-degenerative, constructive, and open-ended. These limits are close to each other and exist on the vicinity of the *phase transition* of the system (the edge of chaos), especially second order (or critical) transitions. Within this range the system dynamics exhibit the capacity of storage, modification and transmission of information, thus well reflecting the emergence of computation. Using the terms of dynamical systems with CA is correct although these terms are basically defined in the case of continuous time continuous space dynamical systems. Wolfram put an analogy between CA classes of behavior and dynamical system ones [Wolfram 1984], which allows for such term borrowing.

The interesting behavior exhibited by systems allowed Berlekamp, Conway, and Guy [Berlekamp 1982] to speculate that: "It's probable, given a large enough Life space, initially in a random state, that after a long time, intelligent self-reproducing animals will emerge and populate some parts of the space."

That speculation was about Life, however, from the realm of computation and cellular automata Langton has put a beautiful and inspiring "text" about the other "life":

> *"Computation may emerge spontaneously and come to dominate the dynamics of physical systems when those systems are at or near a transition between their solid and fluid phases, especially in the vicinity of a second-order or "critical" transition. This hypothesis, if borne out, has many implications for understanding the role of information in nature. Perhaps the most exciting implication is the possibility that life had its origin in the vicinity of a phase transition, and that evolution reflects the process by which life has gained local control over a successively greater number of environmental parameters affecting its ability to maintain itself at a critical balance point between order and chaos".* C.Langton, Computation at the edge of chaos: phase transition and emergent computation. 1990.

This very idea of critical balance of the universe is recently argued by the physicist Gian Giudice [4] , member of CERN's Group for theoretical physics, but this time depending on physical calculations after the discovery of the Higgs boson and the Higgs field. Giudice explains that the Higgs field (the substance that fills all space-time) can also exist in another state, called the ultra-dense Higgs field which is billions times more dense than the actual field state. The mere existence of another state of the Higgs field poses a potential problem, because according to the laws of quantum mechanics, it is possible to have a phase transition between the two field states, a phenomenon called *quantum tunneling*. The intensity of the Higgs field is critical for the structure of matter, if it is only a few times more intense, we would see atoms shrinking, neutrons decaying, nuclei disintegrating, and hydrogen will be the only possible chemical element in the universe. If it is billion times more intense, no molecular structure is possible, and all matter will collapse. Giudice claims that the fate of the Higgs field in our universe is related to the Higgs boson mass, which is about 126 GeV (about 2×10^{-22} grams), and that recent calculations revealed that the Higgs boson mass is very special, as it has just the right value to keep the universe hanging in unstable situation, at the edge of a phase transition (a knife edge), and that it will eventually collapse.

3.5.1.3 Cellular automata as a possible computer of the Universe

Langton has made an allusion to the idea that the universe is describable by information and therefore it is computable. This very idea is the basic premise for the theoretical perspective of *digital physics* field of science that suggests that the universe is a digital computer. It was originally speculated by the German civil engineer Konrad Zuse in his book "Rechnender Raum" ("The Computing Space" in English) in which he suggested that the universe is being computed on a discrete computer, possibly a deterministic cellular automata [Zuse 1969]. The famous movie "Matrix" owes a great deal to Zuse's theories [5].

[4] TED talk by Gian Giudice: Why our universe might exist on a knife-edge, (May 2013).
[5] American Scientist: The Computational Universe, (Online, 28 June 2013).

Along with Langton hypothesis and Zuse speculation, and in the search of the computational model for the physical universe, Wolfram speculated that the research in the computational space that contains all possible rules, also called computational universe, could result in finding the model of our physical universe. This target model could result from simple rules generating an irreducible rich and complex behavior. For Wolfram, the universe is a basic computational structure that resembles to a mesh or a network, and behaves like a continuous space when regrouped in the same way that lots of molecules behave like a continuous fluid. When some rule is applied to these computational structures they result in a universe, different from the universe that results from another rule. He presented some examples of rules with the resulting universe generated by each of them and found some candidate universes that "are not obviously not our universe" . The problem is that such candidates are full of computational irreducibility, therefore, it is irreducibly difficult to verify if they match our physical universe.

The author of this manuscript subscribes himself to this thinking current represented here by the pioneering works and speculations of Zuse, Langton, and Wolfram. We believe that our universe is the outcome of a physical substrate subject to apparently complex, but in essence simple interactions between the units of the physical substrate. The complex phenomena in the universe are thought to be the result of cellular interactions, implying computation, between the basic units of the physical substrate. Even chaos in the universe, is in essence a deterministic phenomena, that the human studies using the black box approach, because he has not yet the sufficient understanding or tools that allow him to model them in the precise way.

The complex phenomena is the outcome of "emergence" of cellular interaction between different levels of agglomerations of the physical substrate. Self-organization is the ultimate form of emergent behavior, different in that it implies an intrinsic power of self-maintenance and stability. Self-organization can be disturbed by external perturbations, possibly emerging from other self-organizing systems, or systems that seek to be organized. We see the universe as a system that permanently seeks self-organization , although, self-organization in the universe might be abstracted to hierarchical levels: objects and creatures are self-organized systems of basic physical units, that have the power of self-maintenance in their actual environments, all within another level of self-organization that includes other objects, or creatures, included finally in a system of large number of self-organizing systems that form our continually self-organizing-while-changing universe. Stillness is impossible, the change is ruling, at least while life is there.

We believe that the universe is a cellular computer, which behavior could be thought of as implying an intrinsic "conscious". The human finally reached an era where he started to carry out complex computations that enable him to willingly participate in controlling the order of the universe. His scientific knowledge can either favor or disturb the existing self-organizing systems, the human is responsible ever after. The human consciousness should keep in harmony with that of the universe as it is embedded in it, this is what his responsibility implies.

3.5.1.4 Cellular automata as a massively parallel computation model

Cellular automata was proved to be universal either by simulating the Turing machine like in the case of Rule 110, or by constructing a serial processor able to perform recursive functions like

the case of Game of Life. Both constructions are not practical for building a general purpose computing machine. One reason is that simulating a serial machine in order to perform some computation results in a very slow computation. Rule 110 as proposed by Cook is an exponentially slow simulator of Turing machines, another solution was proposed by [Neary 2006] to reduce it to a polynomial time. This also degenerates the parallelism aspect of CA as mentioned in 2.4. The other reason is that setting up the appropriate initial configuration that allows for the desired computation is extremely difficult and should be found for every different computation.

That is why efforts to build a vastly parallel universal computer with CA should pass by investing its parallel computation capabilities. Instead of using the CA-built or CA-simulated Turing machines to perform some operation, researchers are working on carrying out computation, like performing some arithmetic operation, by the direct investing of vast parallelism of cellular automata (some examples are [Squier 1994, Weston 2007, Choudhury 2008]).

The presented model of CA in this section is the basic and more used one, nevertheless, CA has many possible architectural and behavioral variations, like stochastic [Fatès 2011] and asynchronous CA [Schönfisch 1999, Bouré 2012, Fatès 2008]. Infinite-size cellular automata are also studied in computation, however, if infinite size is to be allowed, the CA would be a non-realistic computation models. First, perfect synchrony is hard to justify in the case of infinite CA. And second, in order to CA to be a legitimate massively parallel model of computation, it should be defined on finite configurations only [Tosic 2004]. A realistic CA computation machine should allow for a *countably many starting configuration* (inputs) and a *countably many reachable configurations* (outputs), reached after a finite number of steps, while maintaining a finite number of active states at the start and at the end of computation. The latter condition is conform with the constraint imposed on the work of Turing machine where the only allowed symbol to occur infinitely is the blank symbol as explained in 3.2.3.

Today's computer science started to show interest in the way in which computation occurs in the universe, by relying on models that carry out computation using a population of similar computational cells. Computation in the universe is supposedly the result of miniature computations that occur on a very small level, as a result of the interaction between the basic units of the physical substrate with their neighbors, hence, computation in the universe is basically local. Cellular automata, as studied in this section, are fine-grain models that consist of large number of cells computing in parallel. The functional neighborhood of a cell is the same as its topographic neighborhood: each cell computes its state on the basis of the states of its direct topographic neighbors, thus, computation in cellular automata is also local. Computation propagates in the population of cellular cells, allowing complex computations to emerge on the population level. Cellular automata constitute the simplest and more expressive model of computation as occurs in the universe, they are the best to illustrate complex computation that emerges from local interactions between basic computational units. This idea, is in the core of interest of recent computer science, as well as the interest of this work as will be detailed later in this chapter.

3.5.2 Artificial neural networks as fine-grain models

In the previous section, it was shown that cellular automata are fine-grain and parallel computing systems. Another computational model that consists of simple processing units is artificial neural

networks (ANNs), in this section we introduce them and examine their fine-grain and parallelism properties.

There are a large number of different artificial neural networks (ANNs), they vary in their architectures and their paradigm of computation, hence, it is difficult to encompass them all in one formal definition. A possible general definition in the discrete time is introduced, although there are ANNs that work in continuous time.

ANNs are populations of processing units a_i also called neurons, with $i = 1, \cdots, N$ and N is the number of units in the network. Each unit a_i has an output or an activation level $y_i(t)$ at time t that can be rational or binary number. Units are interconnected by connections with weights called *synaptic weights* or *weights* for short, so that, if a unit a_i is connected to another unit a_j, then the *weight* w_{ij} determines how much the activation $y_i(t)$ participates in the input of a_j. The inputs to a unit can be either other units activations, hence inputs internal to the network, or alternatively, they can be external inputs, both input types are presented to the processing unit via weighted connections. Inputs to the unit are referred to as $x(t)$. Units compute their activations starting from their inputs, generally, this is a two step computation, the cumulative effect of inputs to the unit a_j is first computed using a function g, then the activation of a_j at time $t + 1$ is computed using an *activation function* f, so that:

$$y_i(t+1) = f(g(x_1(t), \cdots, x_k(t), w_{1j}, \cdots, w_{kj})) \tag{3.1}$$

where k is the number of inputs to the unit a_j. Depending on the connectivity scheme, different units can have different number of inputs. The function g is often the weighted sum of inputs, that is:

$$g(x_1(t), \cdots, x_k(t), w_{1j}, \cdots, w_{kj}) = \sum_{l=1}^{k} w_{lj} x_l(t) \tag{3.2}$$

The network units are generally organized in layers, namely, input, hidden and output layers. The units in the input layer are set by the environment or the user, the units in the hidden layer are intermediate units that help in the network computation. The units in the output layer are computed using the other units and connection weights, it delivers the computed output to the environment or the user.

Neural networks can optionally update their weights through an adaptive process called *learning* or *training* in order to fit some set of input/output pairs, or to reflect the similarity in input data, depending on whether learning is supervised on unsupervised respectively. There exists several learning methods, called *learning algorithms*.

ANNs are neural in the sense that they are inspired from biological neurons in the brain, but they are not necessarily faithful models of biological neurons. They are rather mathematical models that can work as input classifiers, regression models and clustering algorithms. Information about the environment is presented to the network in the form of input patterns. ANNs acquire knowledge about the environment through learning. Synaptic weights are dynamically modified along the iterative learning process in order to accumulate knowledge about the environment or enhance the accumulated one. The modifiability of synaptic weights is called *synaptic plasticity*.

In his report "intelligent machinery"[Turing 1948], Alan Turing proposed the idea of making

processing from simple units like neurons in the brain, and talked about "unorganized machines" for computation. But before, based on information available about the biological neuron, McCulloch and Pitts developed in 1943 the first conceptual model of the biological neuron and called it the *formal neuron*. The model receives binary inputs, processes them and generates an output. The neuron takes binary values 0 and 1 as values of its output y. Inputs to the neuron $(x_1, \cdots, x_N)$ are values in (0,1) bounded by fixed synaptic weights $(w_1, w_2, \cdots, w_N)$ also normalized in (0,1). The neuron computes the weighted sum of the inputs as the function g and the output is computed as a linear step function of the weighted sum with a threshold T as the function f. The linear threshold activation function is called *hard limiter*. This model is useful in classifying the set of inputs into two different classes when those are linearly separable by a decision surface, or alternatively, by a hyperplan in the uni- or multi-dimensional space representing the distribution of input patterns. The hyperplan is obtained by resolving the equation $\sum w_i x_i = 0$.

In 1949, the psychologist Donald Hebb postulated a rule for self-organized learning that explains the adaptation of neurons in the brain during the learning process, and showing the mechanism of synaptic plasticity [Hebb 2002]. Hebb's postulate could be expressed as "neurons that fire together, wire together". Hebb's rule suggests that when an input to the neuron triggers an output, then the input weight is reinforced, otherwise it is attenuated. This is expressed by $\Delta w_{ij} = \mu x_i y_j$ with μ is a learning rate.

In 1958, Frank Rosenblatt has put the first application of the formal neuron and called it the *perceptron*. It is a modified model that has the ability to learn "on the fly" by adaptive modification of the synaptic weights using the Hebb's rule. The perceptron was able to process real values as inputs, typically bounded in [0,1], and it has more rich activation functions. Depending on the activation function, whether it is a linear function like the boolean-valued linear threshold function or a non-linear one like the sigmoid or Gaussian functions, the perceptron can compute a linear or non-linear combination of the inputs. However, the formal neuron of McCulloch and Pitts used only a step function with $T = 0$. The perceptron found its wide interest because it introduced the notion of learning from examples in such a way that simulates the human intelligence.

Based on the perceptron, the multi-layer perceptron (MLP) was built by arranging neurons in layers: the input, output and some hidden layers, the signal always propagates forward and no backward signal is allowed. MLP is computationally more powerful than the single perceptron. [Hornik 1989] showed that the MLP with n neurons in the input layer and m neurons in the output layer is capable of approximating any continuous function $\mathbb{R}^n \mapsto \mathbb{R}^m$ to any given accuracy, provided that sufficient hidden neurons are available.

The MLP belongs to a class of networks called feed-forward neural networks [Haykin 1998a]. Several other neural models were introduced and found a wide use motivated by several factors, the most important one is that they enable learning from the data related to a problem without necessarily formalizing it.

When loops are allowed in MLPs so that the signal could propagate backwards, in the form of feedback from a given layer to one or more of the previous layers, then one obtains a *recurrent neural networks* (RNNs). The latter can account for the temporal dimension in the inputs as will be discussed in the next chapter.

ANNs models can be classified in different ways. It has been shown that depending on the

direction of signal propagation, they could be classified into feed-forward networks that contain no loops, and recurrent neural networks that contain feedback signals. ANNs may also be classified depending on the learning paradigm into supervised learning like in the MLP and unsupervised learning like in self-organizing maps. In the supervised learning a set of input patterns with their associated desired outputs is available and is used as a training set. In classification tasks, the output could be seen as "labels" of the input patterns . The training set is used for learning network in such a way that it adapts its weights to minimize some error function. After learning is finished, the network becomes ready to compute outputs for the new "unlabeled" inputs that could be unseen before. In the unsupervised case there is no such labeling: no output is associated with the input, and the network is expected to cluster input patterns into groups depending on some intrinsic similarity of input patterns.

There are other adaptive networks that change their weights by reinforcement learning [Sutton 1998a]. Both feed-forward and recurrent neural networks could be either supervised on unsupervised. The various models and classes of ANNs with their different architectures share the same property of involving multiple neurons in one system that exhibits a complex adaptive behavior, this is why there is no single formal definition for ANNs.

The previous neural network models can rely on two formal models of the biological neuron. Basically, the action potentials of biological neurons are brief pulses of about 1ms [Dayan 2005] that are handled as spikes (Dirac pulses in the mathematical models). The *mean firing rate* model of the neuron computes the average spike count within an interval T. In this model, the information about the neuron inputs is assumed to be encoded in the neuron firing rate. The *leaky integrator* or *integrate-and-fire* formal model of the neuron is the classical and most used model and it is based on the aforementioned neural coding. The leaky neuron j computes its output y_j in continuous time as a function of its inputs x_i and synaptic weights w_{ij} as follows:

$$\tau \frac{\mathrm{d}y}{\mathrm{d}t} = -y_j + f\left(\sum_i w_{ij} x_i\right) \tag{3.3}$$

In equilibrium, one obtains the classical model used in the discrete time case:

$$y_j = f\left(\sum_i w_{ij} x_i\right) \tag{3.4}$$

In most cases the function f is the logistic function (or sigmoid) of the form:

$$f(p) = 1/(1 + e^{-p}) \tag{3.5}$$

The other model of neurons in computer science is the *spiking neuron*. The neuron fires at certain points in time and the neuron output consists of discrete pulses called *spikes*. The spiking neuron model describes the transformation of an input spike train into an output spike train. The neuron model comprises some internal state variables that evolve governed by a set of differential equations. The coming input spikes induce discrete changes in the state variables handled by the differential equations, and the output spikes are triggered by threshold conditions.

In these models, it is the precise timing of spikes that carries information. The size and the

shape of the spike is independent from the neuron input, but the firing times depend on this input. Formally the neuron activity in the range $[0, T]$, also called the neuron response, can be expressed as:

$$\rho(t) = \sum_{i=1}^{n} \delta(t - t_i) \tag{3.6}$$

where $\delta(t)$ is the Dirac function at time t, and n is the count of pulses in the interval $[0, T]$ so that $0 \leq t_i \leq T$.

Static neural networks that use the classical neuron model can be emulated to an infinite precision by spiking neurons [Maass 1996], where the activity of a mean firing rate neuron is encoded in the timing of a spiking neural network. This tells that spiking neurons are at least as computationally powerful as classical neurons. Spiking neurons are more realistic approximations to what really happens in biological neurons, however, they are more difficult to simulate. Spiking neurons are out of the scope of this manuscript.

3.5.2.1 Universality

ANNs are at least capable of universal computation. Even in the basic form of ANNs, it is possible by combining linear perceptrons to build the logic gates (NOT, OR, AND), therefore it is possible to build logical or mathematical functions of nowadays digital computers, hence, ANNs have at least the capability of a Turing machine. Also, Artificial neural networks can be used to build a cellular automata as in [Mahajan 2009, Kim 2006, Li 2002], therefore, ANNs inherit the universality of CA. ANNs are also showed to be universal in works like [Gerstner 1992, Kilian 1996].

However, ANNs are proved to surpass the universal computation to super-Turing computation, although under impractical conditions, related to implementation or computation time restrictions.

[Krap 1982] introduced a nonuniform computation model that is computationally stronger than Turing machine. They are nonuniform in the sense that inputs pertaining to a specific formal language are processed using different hardware. In these models, the response time increases in a polynomial manner with the input length, this is handled by allowing the available hardware to grow: for an input of length n, there exists a polynomial $p(n)$ such that the output is calculated by $p(n)$ digital components, like the McCulloch and Pitts neurons. The problem here is that inputs of different lengths, even if they belong to the same formal language, should be computed by different hardware, thus nonuniform implementation, which turns out to be impractical.

Hava T.Siegelmann [Siegelmann 1999] showed that *analog recurrent neural networks* (ARNNs) composed of a finite number of continuous-valued neurons connected in a general fashion, not necessarily in a layered or symmetrical fashion, can perform universal computation and even surpass it to super-Turing computation. In [Kolen 2001], Siegelmann shows that all recursive languages (that Turing machines can compute) can be computed by an ARNN, and require a polynomial computation time as a function of the input length. However, if the computation time is permitted to be exponential, one can specify an analog network for each binary language, including the non computable recursively enumerable languages that surpass the capabilities of Turing machines.

Theoretically, such networks require infinite bit description of real values for both network weights and activations, such infinite precision can't be practically attained. However, Siegelmann claims that the infinite precision of real values is not necessary if computation is carried out in a finite-time interval, whereas they are necessary in long-term infinite-time ones. This is justified by the "linear precision suffices" feature of real-valued neural networks [Siegelmann 2003]: the required precision of real values is a function of the number of computation steps, hence, the required precision is a function of the computation time. This feature is expressed as: For q steps of computation, only the first $O(q)$ bits of weight and activation values influence the result, whereas the less significant bits do not affect it [Siegelmann 1994]. Hence, for time-bounded computations only a finite precision is required.

3.5.2.2 ANNs as a massively parallel computation model

The main concern in this work is not the capability of ANN in artificial intelligence as reflects their adaptability, although it is an important property compared to CA, but rather, it is concerned by what ANNs can offer for massively parallel fine-grain computation.

ANNs are claimed in [Tosic 2004] to be fine-grain models that compute in parallel, however, we don't completely agree with that, and it merits the discussion of some special cases. In essence, a fine-grain model consists of large number of simple processing units, in most cases computation occurs in parallel, except that it is not the case of all neural networks. Let's consider the case of the MLP which is a feedforward ANN, it is a fine-grain model that consists of interconnected simple computational units, that are typically limited in number. However, they are not computing in parallel. Computation in MLPs can be perceived as a cooperative work of multiple units, but is not parallel. The well functioning of an MLPs requires synchronous processing in which all the units activations are computed in each timestep following some order, because computing the activation of some units like those in the output layer is not possible without computing the activation of units in previous layers. Computing units activation in some layer is typically carried out using matrix calculations. Hence, units are cooperating to compute the output of the MLP, but there is no concurrent parallel computation. Parallel computation means that every unit in the model can compute its activation in once cycle of the phase clock without waiting other units to compute, it is the case in CA as mentioned before.

A minimum condition to refer to a model as parallel is that, at least in some phase of the computation, the units compute their activations without waiting other units to compute. This means that, during this phase, computation is possible to be carried out by asynchronous update. However, the asynchronous computation is not possible in the case of MLPs. This holds true for feedforward networks, including recurrent ones based on feedforward architectures.

Although, there are some ANNs models that accept the asynchronous update in some phase during computation, without breaking the functioning of the network, these networks include Hopfield networks and Self-organizing maps. Such networks typically allow for integrating a larger number of units than in feedforward networks, and thus they are fine-grain and parallel models. Hence, unlike CA, with their large number of existing models, ANNs encapsulate different computation paradigms.

However, the computational properties of fine-grain networks remain interesting even if they

are not parallel, their simple processing units and the large number of processor-to-processor connections allow the synergical cooperation between units to yield an important computational power and speed of computation. Those models are the prevailing ANN models, they have been used in real time applications involving pattern recognition like geographic information systems (GIS) and remote sensing applications [Fischer 1997]. The nonlinearity of ANNs, combined with their computational adaptability as used in machine learning, allows them to perform well in tasks like function approximation [Li 2008] and prediction tasks [Bishop 1995b, Qin 2005, Zhang 1998]. In all these tasks, research has already made a wide step, which means that some computations tasks are already well understood.

In their turn, fine-grain and parallel ANNs are used in clustering [Du 2010] and vector quantization [Somervuo 1999], while Hopfield networks are mostly used as content addressable memories [Lopez-Rodriguez 2005].

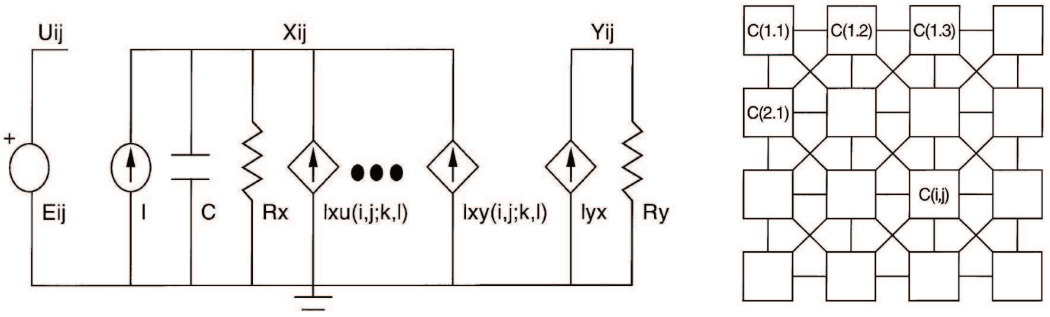
Neural networks are universal and even super-Turing, thus they are able to compute any computable function. However, changing the program that a neural network computes is difficult because it is constrained by its topology. A single "program" that corresponds to some task could be built using a neural network that would be specialized to this very program, and maybe modified to execute other programs in the limit of what the topology allows. Networks with programmable topology are still used for special purpose applications, programmable topologies can be implemented using evolutionary algorithms like the work of Randal Beer [Beer 1996, Gallagher 1999]. ANNs are powerful computational devices, although, finding the suitable ANN for performing a specific computation is not a straightforward task. Indeed, getting ANNs to perform some desired computation is known to be very difficult [Bengio 1994]. Moreover, with sufficiently complicated ANNs like the ones with feedback, the computational capability of the ANN could not be fully discovered, i.e. its computational capability could go beyond the purpose it is designed for [Jacobsson 2005].

We found nothing in the literature about building a general-purpose neural network computer in the common sense of the expression. At the inverse, most neural networks are implemented via computer programs on general-purpose computers whereas few of them are implemented on a specialized hardware. Moreover, parallel processing systems, in the fine-grain sense, are used to implement large networks for faster training and working in real time [Furber 2009, Long 2005, Seiffert 2001, Girau 2000].

3.5.3 Cellular neural networks

Inspired from both cellular automata and neural networks, Chua and Yang came out in 1988 with cellular neural networks (CNNs) [Chua 1988a]. Chua's idea was to use a large array of simple coupled nonlinear dynamic circuits (cells) built of linear and nonlinear analog components in order to process large amounts of information in real time applications. The resulting arrays work in continuous time, they are able to perform time consuming tasks such as image processing and partial differential equation solutions (PDEs), while being suitable for prototyping and implementation on VLSI. The basic cell in Chua's CNNs were made of analog electronic circuits.

Chua's CNN resides in 2-dimensional space with a rectangular topology of size $N \times M$ as shown in Figure 3.11(b). However, higher dimensional CNNs can also be defined. The cell intro-



(a) (b)

FIGURE 3.11: Chua's Cellular neural network: (a) The network cell scheme. (b) A 2-dimensional cellular neural network . Extracted from [Arena 1997].

duced by Chua is a nonlinear circuit as shown in Figure 3.11(a). The cell located at the row i and the column j in the array is denoted $C(i,j)$. u_{ij}, y_{ij} and x_{ij} are voltage values that respectively represent the input, the output and the state of cell $C(i,j)$. The input value u_{ij} is a fixed value determined by the value of the voltage source E_{ij} (cf. figure 3.11(a)) which is set to be less than 1.

The cell $C(i,j)$ interacts locally with its neighbors within a radius r, the cell neighborhood is expressed as $N_{r(i,j)} = \{C(k,l)|max(|k-i|,|l-j|) \leq r\}$ with $1 \leq k \leq M$, $1 \leq l \leq M$. The cell $C(i,j)$ interacts with each cell $C(k,l)$ in $N_{r(i,j)}$ using two voltage controlled current sources for each neighbor cell. $C(i,j)$ is coupled to $C(k,l)$ via the *controlling input* voltage u_{kl}, and the *feedback* from the output voltage y_{kl}.

The participation of the output y_{kl} of $C(k,l)$ in the state x_{ij} of $C(i,j)$ is represented in the value of one current source denoted $I_{xy}(i,j;k,l)$. Similarly, the participation of the input u_{kl} of $C(k,l)$ in the state x_{ij} of $C(i,j)$ is represented in the value of another current source denoted $I_{xu}(i,j;k,l)$. The values of both current sources are given by $I_{xy}(i,j;k,l) = A(i,j;k,l)y_{kl}$ and $I_{xu}(i,j;k,l) = B(i,j;k,l)u_{kl}$, with $A(i,j;k,l)$ and $B(i,j;k,l)$ parameters called the *cloning templates*. These templates define the interaction between neighbor cells, thus they define the dynamics of the CNN.

The interaction with all neighbor cells in $N_{r(i,j)}$ is accumulated by the addition of current sources corresponding to all neighbor cells within $N_{r(i,j)}$, and the state equation of the cell is linear as follows:

$$C.\dot{x}_{ij} = -\frac{1}{R_x}x_{ij}(t) + \sum_{C(k,l)\in N_{r(i,j)}} A(i,j;k,l)y_{kl} + \sum_{C(k,l)\in N_{r(i,j)}} B(i,j;k,l)u_{kl} + I \qquad (3.7)$$

with $1 \leq i \leq M$, $1 \leq j \leq M$, the capacitor and resistance values $C > 0, R_x > 0$ respectively, the initial condition $x_{ij}(0) \leq 1$ and $|x_{ij}| \leq 1$.

The output function is nonlinear and is computed as follows:

$$y_{ij} = f(x_{ij}) = 0.5(|x_{ij}+1| - |x_{ij}-1|) \qquad (3.8)$$

The model is known as the Chua-Yang model or the linear CNN model due to linearity in state computing. However, later generalizations of the basic model included nonlinear interactions with neighbors by using nonlinear cloning templates. Further generalizations were later introduced: different topologies with different cell types or varying neighborhood sizes in the same network, introducing functional dependency using delay templates [Roska 1990, Gilli 1994] that allow for handling past values of neighbor cells inputs/outputs, and introducing discrete-time CNNs [Arena 1997, Fortuna 2001].

After these generalizations, Chua proposed a general definition to CNNs as: "The CNN is a n-dimensional array of mainly identical dynamic systems, called cells, which satisfies two properties: (a) most interactions are local within a finite radius r, and (b) all state variables are continuous valued signals" [Chua 1993].

The bottleneck of CNNs stays that there exists no efficient strategy to learn the suitable templates values in order to perform a given task [Fortuna 2001]. For a new task, the design and the cloning templates are set in most cases by trial and error technique, and is also a question of design. A large number of templates and template algorithms were found to perform a wide range of tasks, almost all of them were found by simulations and trial and error [Arena 1997, Fortuna 2001].

CNN Applications vary from simple image processing tasks [Caponetto 1998] to complex systems modeling [Gollas 2005]. The locally distributed way of analog signal exchange of CNNs allow them to simulate and reproduce in low cost some complex dynamics found in living systems, such as autonomous wave formation and propagation [Chua 1995] and morphological pattern formation [Setti 1996], PDEs solution [Gobovic 1994], modeling physical systems [Slavova 2012], generation of nonlinear and chaotic dynamics, for instance, a simple CNN compound of two cells could result in complex chaotic dynamics [Cafagna 2003].

3.5.3.1 Universality

Universality in CNNs is inherited from that of CA and ANNs. Discrete time CNNs (DTCNNs) were developed and led to building a complex CNN architecture called CNN universal machine (CNNUM) [Roska 1993]. The latter consists of analog circuits completed by digital logic ones. In this structure the templates determine the working of the machine thus they play the role of the program instructions in a general purpose computer. The machine is universal because templates could be changed in order to simulate the program change. It is even possible to change templates during the run: the CNN with a given template is allowed to run on a given time window corresponding to a first phase of computation, then the next phase of computation starts with another template for the CNN but working on the produced data from the first phase, and so on. [Roska 1999] introduced an extended CNNUM that allows for learning and adaptability and was implemented on VLSI, [Paasio 1997] is one example.

3.5.4 A new concept of computation

The fine-grain models introduced in the previous sections implement a different concept of computation from that of coarse-grain models. The differences are structural and functional.

From the **structural** point of view, the first difference between fine-grain and coarse-grain

processing models is the composition and organization of their architectures. Fine-grain models contain a different scale of processors number that exceeds coarse-grain ones in several orders of magnitude, and the processors are much more simple than those powerful ones used in coarse-grain models. A node in coarse-grain models can perform complex computations, while in fine-grain models, the node computes a quite simple function.

Another structural difference between both models resides in the relation between processing and storage. Coarse-grain models are collections of Von Neumann machines that make a clear separation between program execution and storage, whether it is program or data storage. Fine-grain models implement an entirely different concept for processing and for program and data memory, in such models the memory and the processors are hard to separate from each other. Unlike the actual computational systems, in which tasks are written as procedures that are executed starting from the first line of code and moves to the next line in a linear path, fine-grain models are connectionist models in which cells work collectively in parallel.

Consider the case of CA where the processors are the automaton cells. All the processors compute the same function (that the FSM of the cell computes). This function is the program that CA runs, thus, cells are also the program memory. The only input to a CA model, which is a closed system, is its global configuration, i.e. the initial cells values set at the start. During the run, the data processed by a cell is its inputs which in turn, are the states of the neighbor cells, therefore, the data is stored in cells states making of the cells the inputs and data memory as well. The output seen by the outer world is the cells outputs. Indeed, the cell in a CA substitutes the processors, the memory, and the input/output in classical computers.

In ANNs, the neurons are the processors. The program is distributed across neurons: it is the way inputs are combined and the activation function of neurons that determine the function that the ANN computes. The processed data is the previously accumulated knowledge stored in the modifiable synaptic weights (processor-to-processor connections) of the network, which change their values as the network processes new inputs through learning, i.e, the new knowledge is conserved in the weights, thus, ANNs weights play the role of the data memory. Theoretically, ANN are analog models, where values are real numbers of infinite precision, this gives them an advantage over digital computers that can't do that. But in practice, ANNs are simulated on digital computers, so they are bound to inherit the finite-precision decimal numbers representations on the host machine, thus, in practice they can be thought of as digital models.

Concerning CNNs, the processors are the network cells. The cell output function and the cloning templates determine what the CNN computes, therefore they are the program memory. The processed data is the neighbor nodes states and the input patterns coming from the outside world if there are any, while the output is the node states. Here, the cells play the role of the data memory and the output as well.

From the **functional** point of view, data fetching and storage in fine-grain models is a one-shot operation that takes a very small amount of time. It could take no more than one clock cycle to read the neighbor state value in CA or CNNs and compute the output. Similarly, ANNs could take one clock cycle to compute the neuron output, while it could take one or more cycles to compute the new weights depending on the used learning processes. Whatever, the speed of this process is not to be compared with the data fetching and storage in coarse-grain models that suffer from a storage

bottleneck: the Von Neumann bottleneck.

Fine-grain models are complex systems that can exhibit rich and complex behavior that allow for the emergence of computation. Some of these models like ANNs and CNNs can learn, therefore they are complex adaptive systems that change their internal structure based on information flowing through them [Shiffman 2012]. In particular, ANNs offer an excellent adaptive behavior that makes them suitable for many application domains, unfortunately, their implementation on hardware like VLSI is limited due to their global connectivity as explained in the next section.

3.6 Cellular computing

Fine-grain models have the main features of relying on a vast number of simple processors working in parallel. Regarding connectivity, a cell in a fine-grain model could be connected to a few cells, typically the neighbor ones, or could be connected to more cells beyond their neighborhood that could be all the cells in the model in a full connectivity scheme. An example is associative memory ANN models like Hopfield networks that will be explained in the next chapter. *Cellular computing* inspires from computation in our physical universe. It can be perceived as a subfield of fine-grain parallel computing that deals with models that rely on a vast number of simple processors (the cells) with *local connectivity patterns*. Another condition is that cellular models compute in a decentralized way, i.e no central processor is used.

In the context of cellular computing, we distinguish two senses of locality, the first is the **topographic locality**, which means that cells interconnect is bounded in a limited topographic area in the model space. **Functional Locality**, means that the cell computes whatever value on the basis of other cells values to which it has access to by its connections. Locality in this manuscript refers to both senses, unless it is otherwise explicitly specified.

But does this minor difference merit the distinction of a new computation paradigm? The answer is *yes*. It is justified by the consequences of connection locality and decentralization, that distinguish cellular computing from fine-grain computing. There are three major consequence for these conditions: The first is that they facilitate the hardware implementation such as VLSI implementations. The second, is the scalability of the model; the number of cells in the model could be changed easily without the need for an entire review of the model to check if it still complies with the task, normally, minor modifications are required. The third consequence is that the computational power could be measured by the size of the model, the more cells there are, the more computational power is obtained.

Cellular computing has its terminological origins in Biology. The cell is the basic structural, functional and biological unit of all known living organisms and is often called the "building block of life" [6]. Borrowed from the Latin word "cella" which designates a small chamber, the English natural philosopher Robert Hooke coined the term "cell" in his book "Micrographia" (published in 1665) after his research in the science of microscopy.

The biological cell could interact with its neighbor cells, but not with the far ones, at least not directly. Neurons in the brain are special instances of biological cells making of the brain a cellular

[6] Wikipedia: cell, (Online, 23 June 2013).

wetware structure. The term "cellular computing" was imported to computer science from biology by the computer scientist Moshe Sipper [Sipper 1998b] to refer to a similar concept. Sipper described the cellular computing paradigm by the "equation": *simple+vastly parallel+local=cellular computing*. Decentralization is referred to in the paper of Sipper, but we think that it merits to be added to the definition of cellular computing as a model defined as in Sipper's equation can be computed by a centralized processor, that should be prohibited. The term was also used recently to describe some fine-grain models like the IBM multi-core Cell microprocessor. However, we adopt Sipper definition of the term.

The locality in cells interconnect is private to cellular computing and distinguishes it from fine-grain computing; each cell in the cellular model is connected to a few number of cells, typically to the physical neighbors. Sipper indicates that a full connectivity scheme puts the model outside the realm of cellular computing. Locality, being private to cellular computing, in addition to the inherent properties of fine-grain computation; namely, cells simplicity and vast parallelism, leave room for special properties of cellular computing. Different architectural and behavioral properties allow for multiple choices. Each set of choices result in a different model. These properties are the topic of the following subsections.

By contrasting the three properties proposed by Sipper to the properties of fine-grain models , it could be found that the CA and CNNs are cellular by nature, while ANNs *are not*. The difference between neural networks and cellular neural networks is that the latter are "cellular" as the name indicates. Indeed, CNNs share the property of local connectivity with cellular automata, which is not the case in most of the neural networks. It is because of the local interaction between cells that CA and CNNs are more suitable for the physical implementation than ANNs [Fortuna 2001].

A related field to cellular computing that can be also seen as massively parallel computing model is *amorphous computing*. It appeared in mid-1990s inspired from both microbiology and cellular automata. An amorphous computer is a collection of computational particles dispersed irregularly on a surface or throughout a volume, where individual particles have no a priori knowledge of their positions or orientations [Abelson 2000].

One way to realize these particles has emerged through the two past decades; such particles were manufactured as cheap microelectronic mechanical devices combining logic circuits, micro-sensors, actuators, and communication devices on the same chip. One kind of these particles, millimeter-scale particles for environment monitoring applications are commercially available nowadays under the name of *dust networks* [Technology 2013].

Another way for engineering particles comes from biology which motivated several computing metaphors in the past including neural networks, but started recently to show that, by itself, it offers a suitable substrate for computing. As mentioned in the past chapter, engineered cells based on DNA are a technology directly based on microbiology, DNA protein concentrations were used to represent voltage and logic levels, enabling the construction of DNA-based logical circuits [Weiss 2003, Weiss 2001].

All particles in an amorphous computer are fabricated in the same way, a particle has a small computing power and a small memory, they can process at the same speed, and they are all loaded with the same program, although, it is not required that they work in synchrony. Also, particles don't need to work all reliably, besides, their is no need to manufacture a precise geometrical

arrangement or precise interconnection among them. Amorphous computing search for programming aggregations of such particles without requiring the precise control over their interaction and arrangement.

In an amorphous computer, a myriad of particles ($10^{10} - 10^{12}$) are mixed with bulk materials like paint, gel and concrete, so that they act as a host medium. Mixing them with paint result in *smart paint*. The latter could be used to coat walls or bridges in order to cancel noise or to sense vibration and wind loads.

Programming an amorphous computer is done by diffusion, each particle communicates with its few neighbors by message diffusion either by short-distance radio in microelectronic particles, or by chemical signals in the biological particles. Amorphous computing is intended to resemble to the computation in nature, where neighbor cells in an organism could be dead, thus, programming algorithms should be independent from the number of particles, and the performance should degrade gracefully as the number of the particles decreases. An example of message diffusion is finding a chain connecting two particles A and B: A particle A can diffuse a message containing a count to its neighbors, which in turn stores an increased version of the count, re-diffuses the message and ignores future messages, this is repeated until reaching a particle B. B then diffuses a backward message to A asking all the pre-visited particles to register themselves as a part of the chain. The number of particles in the chain between A and B establishes a rough distance measure between the two. There exists several messages diffusion algorithms that detect chain cut and perform re-routing operations. Particles mixed with some host medium, could behave as self-repairing material and self-healing systems, resembling to the mechanism of wound repair [Clement L. 2003]. This is possible when the particles are equipped with actuators so that they can be used to heal small cracks by moving the material to cover them. Such mobile particles are the basis of the new field of *swarm robotics* [Werfel 2005].

The major differences between amorphous computing and cellular computing are that they don't require to ensure reliability, interaction between particles is not required to strictly follow some update regime whether it is synchronous or asynchronous, instead update is arbitrary, the topology of particles need not to be regular, moreover, particles can be mobile.

The state of the art of cellular computing is still in its infancy, the preliminary works (there are very few) on the concepts and properties of cellular computing in computer science and fine-grain parallel models in general, are [Sipper 1999] and [Tosic 2004] respectively. The next subsections discuss the properties of cellular models, some of these properties are expected ones and still to be tested.

3.6.1 Decentralization in cellular computing

One defining property of cellular models is that they compute in a decentralized way. Decentralization is a property that means that there is no global controller that controls the computation of the model, except, of course, for the clock which is some systemic property that offers only a working basis and does not intervene in cell functioning.

This distinction between decentralization and locality is somehow ambiguous, although, they are partially related.

Locality in cellular models is two-fold, topographic and functional. The functional locality

means that each cell in the model is authorized to compute its output and connection weights on the only basis of the values that it has access to by its connections (whether they are neighbor cells outputs or synaptic weights or other). The cell is not allowed to compute a function based on a value that it doesn't have access to by its connections, for example, the values seen by its neighbors connections and not seen by the cell itself, say, the synaptic weights of a neighbor cell connections to other cells.

However, it will be shown in the next chapter that an ANN model called reservoir networks, is functionally local, but cells values are normally computed using matrix computations on the population level, using a centralized processor. But in cellular computing, cells can compute in parallel, independent from other cells, based only on the values they read via their connections, hence there is no need for a central processor. This what decentralization means.

Another example, a model in which each cell is connected to all other cells is not topographically local, but can be functionally local, so that each cell computes its activity using values accessed by its connections. Although, such model can compute in a decentralized way. Decentralization implies that the model doesn't have a shared memory (or global storage variables) between the cells; no global variable can be stored and accessed later. In cellular models, this is not possible due to topographic locality. Decentralization also implies, that it should be possible that the model computes in the asynchronous regime.

Even the brain works in a decentralized way, neurons compute in parallel, and no global controller exists, although some regions in the brain seem to play this role, but they themselves are decentralized structures and the way they control other regions is decentralized as well. Decentralization in cellular models is an important property that facilitates their hardware implementation.

3.6.2 Architectural and design properties

A cellular model consists of a population of distributed cells following some topology. Cells are interconnected with their topographic neighbors using connection links that hold a small amount of information. In cellular automata, links hold no information, but are simply used to read neighbors values.

The properties of cellular models are either related to architecture specifications set at the design time, or to the behavior of the model during the run. Cells could be uniform or from different types, they compute their outputs following some rule, and could follow some temporal regime. These architectural properties, beside to the decentralized behavior described above, are properties set at the design time.

An architectural property of cellular models is their *topology*. The cell population could be arranged on a regular grid of n-dimensional array (often, $n = 1, 2, 3$) in such a way that fills some geometrical shape (a rectangle, a circle, etc..). Regular grids imply that all cells connections follow the same connectivity pattern like in the case of 2-dimensional cellular neural networks with rectangular geometry. Finite grids require determining the boundary condition, as in this case, connectivity pattern for boundary cells will not be the same for other cells. When the effect of this condition is critical for the model behavior, the grid is usually wrapped around itself (giving a ring in case of 1D and a torus in the case of 2D grids). It is also possible to arrange cells on a non-regular grid or a graph. Moreover, cells within the grid, could be uniform or nonuniform in

two ways, either respecting some connectivity pattern, or respecting whether all the cells compute the same function or if they compute different ones. Sipper [Sipper 1998a] showed that using nonuniform cell functions could reflect in definite computational advantages.

Other properties of cellular models that are set at the outset include the nature of the computed value by the cells whether it is discrete or continuous, how the output is computed, and when they are computed.

The *cell output* values can be discrete (like in CA), so that it selects its output from a finite range of values, and thus, could be considered as enumerated states. Alternatively, output values can be continuous (like in most ANNs) and take their values within some range.

The *dynamic behavior* of the cell depends on the way the output is computed starting from neighbor values: when the cell state is discrete, an exhaustive enumeration list could be used to compute the cell output. In this case, the mapping between neighbor cells states and the cell output is established. The cell output could also be computed using a linear or nonlinear function of the neighbor output values. Output functions could also be non-deterministic, like probabilistic functions. In this case, the cell response for a specific input may vary so that different outputs could be obtained for the same input, and the whole population of cells behaves as a non-deterministic dynamical system.

There also exist more sophisticated scenarios for computing cells outputs, such that using a small program or behavioral rules that specify the cell behavior in different situations. The latter case is preferred when simulating the behavior of a biological cell.

Depending on what the cellular population is used to compute and how, different *temporal dynamics* could be obtained. The temporal dynamics depend on the way the cell output is computed respecting time: time could be continuous or discrete. In continuous time dynamics, the output is possible to compute at any time instance, usually, it is computed using a differential equation (regardless of the input, should it be continuous or discrete in time). In discrete time dynamics, outputs are computed at each timestep.

In the case of discrete time, it is possible to compute outputs simultaneously in each timestep so that the cells outputs change follow the synchronous regime (for the definition of update regimes cf 3.4). Alternatively, asynchronous temporal dynamics is possible by computing cells outputs in some order. Event-driven models, like a subclass of spiking neurons-based neural networks [Brette 2007], which use the spiking neuron model presented in 3.5.2, don't rely on any clock signal, but update their values when an event occurs, like when a new input comes. The absence of any clock signal reinforces the biological plausibility of these models. The update regime in event driven models can be classed under the asynchronous update regime [Lin 2009], although there is no clock to control the update phases on the population level.

However, it is possible to use variations of mixed regimes, like dividing the population into groups of cells and update cells within each group in a synchronous fashion, while updating groups following the asynchronous regime. It is noteworthy to mention here that by its locality of cells interconnect, cellular computing naturally allows for asynchronous update regime, which is not always evident in non-cellular fine-grain models like MLPs for instance.

3.6.3 Operational properties

Cellular models operational properties are those related to the run of the model, the important ones discussed here are the model programming, and its robustess and scalablity properties.

Programming in the actual context of cellular computing refers to adjusting the cellular model to behave in such a way that fulfills a target task. This is done either directly by completely specifying the cellular system characteristics at the outset, like specifying the system topology and the cell types, connectivity, or by specifying a part of the system characteristics like the topology and cell type and determining other characteristics by adaptive methods that include learning, evolutionary methods or self-organization. For example, the transition function of a cellular automata could be adjusted by a evolutionary algorithm like in [Sipper 1997]. In general, the dynamics of cellular models as described in the previous section could be considered as first-order dynamics of the model, while adaptive programming could be perceived as higher-order dynamics.

The resilience of some system in the presence of faults defines its *robustness*. When a cell in the cellular model functions incorrectly, or a connection between two cells fails, then the robustness of the model requires that it continues to function or at least show a graceful degradation. One good reason to think that cellular models are fault tolerant is the brain; it is a highly effective neural network that degrades gracefully as there are neurons and synapses that die everyday in our brains but they keep functioning well [Tosic 2004]. The properties of cellular computing are thought to assure fault-tolerance in most cases; when a cell or a connection fails, the local connectivity is thought to prevents the error from spreading on the population level so that it remains bound in a specific region. The large number of population cells and interconnection links are thought to assure that the model stays operational.

Cellular computing models also scale well, much better than classical coarse-grain models. Better *scalability* stems from the fact that the delay in memory access needs not to grow proportionally to the processors number as there is no physically separated memory that cellular processors need to access during computation. Instead, cellular models adopt another concept of memory as discussed in section 3.5.4.

Cellular computing models can adapt to comply the task at hand through learning. So far, it has been pointed out that ANNs can learn, and the choice of CNN characteristics are done by trial and error [Arena 1997, Fortuna 2001]. However, recent works introduced some learning techniques to CNNs [Xavier-de Souza 2005, Luitel 2012, Vilasis-Cardona 2005] and CA [Qian 1996, Nakamura 2010, Beigy 2010, Jr. 2006]. Therefore, they could also be adaptive models. Whether it is the adaptive learning or the run, cellular computing should respect the constraint of decentralization. In the next chapter, decentralization will be verified in the case of some ANN models.

3.6.4 What to expect from the cellular computing paradigm

Respecting application domains, coarse-grain and fine-grain models (including cellular models) could outperform one the other according to the specific domains. There are problems that fit better the architecture characteristics and the computing style of one computing paradigm than the other. We can infer that from the brain that outperforms the Von Neumann machines in super-

computation tasks like intuition and other intelligence tasks, but loses the competition when it comes to computing a fraction to some precision or searching in a database of unordered data. Take the example of artificial neural networks, they are capable -due to their learning mechanism- of performing tasks of the type "easy-for-human, difficult-for-a-machine". Pattern recognition like face recognition is a clear example of such tasks.

Indeed, a computing paradigm in some task type could outperform the other in several orders or magnitude. Tosic [Tosic 2004] compares fine-grain models with the human brain and gives some partial justifications for this difference in computational power: multiplying two 100-digit numbers need no interaction with the environment apart from reading the inputs, digital computers are capable of efficiently solving this task by following a set of simple and fixed-size arithmetic rules. But in tasks that need an ongoing dynamic interaction with a complex and structured environment in which computation should deal with uncertainty, such as in the face recognition task, then computers fail and human brains excel. The brain successfully interacts with the environment in uncertain and fuzzy conditions due to its plasticity, a criteria that enables neurons to learn and adapt to the task while interacting with the environment.

Cellular models outperform classical coarse-grain models when it comes to robustness as the latter could be subject to total dysfunction if a part gets out of service especially that they could be geographically distant, or not very compatible with one another or under control of different users like in loosely-coupled parallel models. Scalability issues are discussed previously and it has been shown that cellular models and fine-grain models in general outperform classical parallel models.

Cellular computing is still a young field of research. Cellular systems like CA are well known since decades, while CNNs are recent cellular models, neither of them is used to build a general purpose computer comparable to sequential machines. Michael Flynn, the pioneer of parallel computing attributes this to the difficulty of partitioning actual serial programs in order to find large and consistent degrees of parallelism within them. Whereas there are some algorithms that are easily parallelized, most existing algorithms are difficult to decompose into concurrent tasks in order to be allocated on parallel processors in such a way that their parallel joint operation be efficient. Flynn states that in order to make better use of cellular models, computational problems should be represented in a cellular form [Flynn 1996].

Nevertheless, computer science is far from developing a general purpose computer architecture based on cellular models that could be programmed and employed for everyday computation. Although, there exists some preliminary attempts like using CA to design a programmable special-purpose computer architecture [Margolus 1993], however, they are far from being mature.

3.7 Conclusion

Computation in nature occurs in physical systems that consist of spatially extended physical substrates, which in turn are populations of basic units that interact with each other in parallel, and their interaction incorporates computation. Computation in nature is local; each unit interacts with its physical neighbors only, it is also decentralized as there is no central processor that computes for the system, instead, computation occurs on the level of basic units. Although, interesting behavioral patterns emerge on the population level, hence, decentralized computing in spatially extended

natural systems gives rise to emergent computation.

Cellular computing is a new paradigm of computation that emerged inspired from the scientific information about computation in the physical universe. This paradigm borrows the same concept of computation as in nature, and brings it to computer science. Cellular models are built of fine-grain populations of simple processors, with local connectivity between the processors. Cellular automata and cellular neural networks are fine-grain models that already exist and fit this concept of cellular computation paradigm. In these fine-grain models, computation is spatially distributed across the population of processors.

The important question here is: is it possible to use fine-grain models in order to compute what the existing coarse-grain parallel models can compute? There is at least one possible way to do that: coarse-grain parallel models are agglomerations of Von Neumann machines that are, in essence, fine-grain populations of transistors that cooperate to compute sequentially. But sequential computation degenerates the parallelism aspect of fine-grain models, hence, there would be no benefit of investigating this new paradigm of computation. However, fine-grain models are Turing-equivalent that can compute, in parallel, what coarse-grain models compute, but the problem is that finding the suitable model for computing a specific function is a difficult task.

The parallel and cooperative nature of fine-grain models complicates function computations, not because they are not able to do that; we already know that they are at least Turing-equivalent, but because all the existing algorithms for computation are sequential, developed to fit the Von Neumann sequential way of computation. Even parallelizing a task on coarse-grain parallel models is carried out by parallelizing the steps of the existing sequential algorithms of computation. Indeed, no pure parallel algorithm exists yet, and new algorithms for functions computation that fit the fine-grain concept of computation should be found.

A standalone fine-grain computer already accomplished is the human brain, with nearly 10^{11} simple processors and 10^{15} links (synapses). Our information about the computational power of the human brain suggests that building fine-grain parallel models that are able to carry out powerful computations with less energy, less heat dissipation and less cost than the coarse-grain ones could be possible.

The question to ask now is: how to organize the population of processors and their interconnectivity in order to obtain the universal computation? Alternatively, how to get one fine-grain model that can compute all computable functions like Von Neumann machines? The answer to this question is not clear yet, but we believe that the right answer should rely on fine-grain models with emergent computation, promoted by the adaptability of their models.

The implementation of fine-grain systems without the conditions of locality and decentralization is difficult, their physical implementation is difficult even with populations of small numbers of processors as will be discussed in the introduction of the coming chapter. Cellular Computing fulfills these two conditions, and maintains the same computational capabilities of the fine-grain paradigm. Most importantly, cellular models are spatially extended systems that exhibit emergent computation.

CHAPTER 4

Encoding Time in Dynamical Neural Networks

Contents

Through the past chapter, several applications and application fields of the cellular computing paradigm in computer science have been mentioned. More detailed works that cover the general applications of CA can be found in [Wolfram 1986, Preston 1985], ANNs general applications are reviewed in [Suzuki 2013, Chi Leung 2011], CNNs general applications are reviewed

in [Chua 1988b, Tetzlaff 2002]. Some examples in both computer science and biological cellular computers can be found in [Sipper 1999].

In this manuscript, we are interested in using the cellular computing paradigm in temporal domains, those tasks require processing spatiotemporal data in which inputs are dependent on past ones, thus the computation of outputs should take into account the input history.

In fact, both CA and CNNs have been also applied to spatiotemporal problems. However, it is noteworthy here to distinguish between two types of spatiotemporal problems, the first is modeling the temporal dynamics of complex systems or some physical phenomena like wave simulations and PDEs solutions, this case corresponds to an autonomous dynamics. In such cases, the input to the model is the cells initial states, and the system runs autonomously without interaction with the outer world. The second type involves interaction with the outside environment; during the run, the system receives inputs from the environment, computes outputs, and passes them back to the environment. The latter type, which corresponds to a non-autonomous dynamics, could be involved in a closed loop in order to control some real system, an example can be found in [Nicolosi 2010].

The cellular nature of CA and CNN allow to use them as software and hardware parallel computing structures with the possibility of large-scale implementations. CA has been applied in spatiotemporal tasks like mining imagery data flow [Fu 2006], partial differential equations (PDEs) [Strader 2008], modeling artificial financial markets [Ding 2011]. CNNs were used in active wave computing [Roska 2002] which involves the generation, expanding, and collision of waves. They were also applied in motion detection [Cimagalli 1993]. CNN Silicon-based chips were implemented for robot navigation tasks [Balsi 2001], artificial retina [Werblin 1997] that process spatiotemporal sensory data. Examples of FPGA hardware implementations of CA and CNNs are [Chuanwu 2008] for CA and [Cheung 2006] for CNNs.

In their turn, ANNs which are fine-grain models (in the cellular computing sense) were the prevailing paradigm used for spatiotemporal tasks. They were used for trajectory planning [Zegers 2003], gesture recognition [Su 1998], speech recognition [Kurogi 1991], spatiotemporal memory [Ramanathan 2009], to mention a few. However, ANNs are not cellular.

The current situation is as follows: CA and CNNs are already cellular models with local connectivity between cells and it is possible that each cellular processor computes without the help of a centralized processor. Hence, these models can be used for large-scale parallel computation. However, they have poor adaptability and interaction with the environment, which limit their usage in temporal tasks. Whereas, ANNs are more adaptable and more interactive with the environment, but they are not cellular (global connectivity and require a centralized processor) which limit their usage in temporal tasks, but for a different reason than CA and CNN. So the goal is: incorporating the interesting properties of ANNs in a cellular model for processing temporal tasks, endowing the cellular computing paradigm with the powerful temporal properties of ANNs. This is the core of discussion in the coming paragraphs.

Indeed, ANNs compete CA and CNNs in temporal tasks. Although CA and CNNs can process spatiotemporal data, they have some drawbacks that limit their potential, especially when compared to ANNs. The first major difference is the interactivity with the environment. CA are closed systems that lack the interaction with the environment; their input and output are the initial and final cells states. Although CNNs partially inherit the adaptability of ANNs, they also inherit the

functionality and applicability from CA; although they have been used in some tasks that require the interaction with the environment, CNNs are more suitable in simulating some autonomous dynamics with no interaction with the environment: inputs are loaded as the initial cells state and the output is read as their final state. This difference between CNNs and ANNs is reflected in a difference in the roles of weights between both models: in ANNs, weights are used to hold an information about the system feedback or the system previous state, while in CNNs, weights are used to determine the dynamics of the system.

The other major difference is related to the system adaptability. On the one hand, learning in CA is very limited compared to ANNs, especially because connections in CA have no weights. On the other hand, although CNNs are more adaptive than CA, their adaptability is limited to training the cloning templates that define the cell interconnect with neighbor cells. This is where ANNs excel, they offer more freedom of design and interconnect than CNNs, besides, there exist an arsenal of effective learning algorithms for ANNs, a task which is not well controlled in CNNs, especially that most CNNs characteristics that fit some task are determined by trial and error [Arena 1997, Fortuna 2001].

However, unlike CA and CNNs, connections in ANNs are global (thus, not local), for example in the multi-layer perceptron a neuron in one layer is often connected to all neurons in the next layer, and in Hopfield networks the network is fully-connected, i.e. each neuron is connected to all other neurons in the network. Global connectivity puts ANNs outside the realm of cellular computing. This lack of cellular representation of ANNs is an obstacle for implementing ANNs on hardware. ANN implementation are often software ones that are run on general-purpose sequential computers, and ANNs with large number of cell and interconnect are implemented on classical parallel computing systems. The few hardware implementations of neural networks show the difficulty of the study and programming of globally connected non-cellular ANNs. An example is [Sahin 2006] in which a three layer MLP with a total of 6 units is implemented on FPGA. Such implementation is far from being scalable, as adding one extra neuron requires an entirely new study of the network and the implementation. A review of ANNs hardware implementations could be found in [Misra 2010].

Connection globality in ANNs is not only an obstacle in hardware implementation, but is also an obstacle in software implementation which is resorted to in the case of large-scale ANNs. One can see the difficulty of this task in works like [Valafar 1993] in which training an ANN by backpropagation [Hecht-Nielsen 1989] is parallelized on massively parallel computers. In such work, the authors profit from ANNs parallelism to find *partial* independent calculations in order to process them on a parallel computer that supports SIMD operations. Similarly, [Scanzio 2010] parallelizes backpropagation on a GPU SIMD architecture, using CUDA framework and uses the learned ANN for speech recognition. Other examples of parallelizing ANNs are [Jiang 1997, Calonge 1997, Blas 2005, Forrest 1987, Przytula 1993].

It is now clear that if attained, locality and decentralization in ANNs facilitates both their hardware and software implementations. In particular, scaling the ANN implementation on parallel computers will not be an issue; if the ANN is a cellular model, one can get more computation power by the mere changing of the variables determining the model dimensions. This research work aims at reconciling the adaptability and interactivity of ANNs with the cellular properties like

in CA and CNNs, namely, connection locality and processing decentralization. For this purpose, we choose to build such cellular model starting from ANNs. Hereafter, this manuscript will focus on studying ANN models, covering their temporal properties and their adequacy with the cellular computing properties, and ends by proposing a cellular model based on neural networks.

In the following sections we start by defining what we mean by temporal data processing and the encountered tasks in such case. Then, we introduce the state of the art of artificial neural networks models for processing temporal data. We start by introducing the unified features of neural networks used in temporal tasks, and focus on their temporal parts. Then we introduce the major existing architectures for temporal sequence processing, with their different approaches for time integration and adaptation. At the end, we show how neural networks with time representation ability are capable of simulating the formal models of computation. During the presentation of different ANN architectures, we explain why all of the existing fine-grain temporal neural networks architectures are not cellular models.

4.1 Temporal data processing

Most data processed by information systems is collected in different points in time. This is the case in processing physical data where measurements like temperature or pressure are collected in successive time instants or in financial systems where prices or exchange rates are watched in several points in time.

Batch learning is the method normally used in data processing; data is collected over a time period and presented to an information processing model as a batch of *offline static patterns* in order to perform some processing. Data is organized in an array or similar structures, the values of data points can belong to some interval or some set of numerical or symbolic values, the data point related to some point in time can also consist of one or more values, therefore they can be *univariate* or *multivariate*. In such situation, input patterns are said to have a spatial dimension.

However, in some applications like real time applications, one is confronted by the case where data is not collected beforehand and should be handled as they arrive. In this latter case, data points are virtually arranged in a sequence of values (or a series) that become available in the course of time. Thus, in addition to the spatial dimension, data is said to have a temporal dimension, and is therefore referred to as spatiotemporal data [Ray 1996]. Often, the time interval separating the collection of two consecutive data points is fixed.

Typically, data points in the sequence of values are presented one by one to the processing model, as soon as they become available. When the model is adaptive, i.e. it changes its computing parameters during data processing like in most neural network models, the model parameters are updated after processing each newly coming value, in an online manner then one may talk about *incremental learning*. It should be noticed that sometimes data is collected and presented to the model offline but in a sequential way, because in some cases, like in machine learning systems, this reduces the computational complexity as the system doesn't need to learn all the previously learned data [Oohira 2003].

Depending on the nature of the temporal data and the related tasks, literature differentiates between two closely-related terms referring to the temporal nature of sequential data: **time series**

and **temporal sequence**.

When the data represent the values of a continuous observable variable like in physical systems, or a variable studied on the long term while taking a big number of measurements like the price of the stock exchange or the number of births in a city, then we are talking about a time series. The variable is measured in different time intervals that can vary depending on the studied phenomena. The time interval may vary from milliseconds like when the variable is the electricity load, to years like when the variable is related to some cosmic phenomenon. The process of taking measurements in time intervals and adding it to the sequence is called *sampling*. Due to the nature of the sampling process, time representation in the resulting sequences is discrete.

Sometimes, sequential data can be related to serially ordered elements by their intrinsic nature, like letters in language processing or symbols in DNA chains. In such cases there is no time in the physical meaning, however, the elements are also obtained by a sampling operation, thus they can still be seen as values separated by time distances, typically equal. In this case, time is implicit, and has the meaning of coordinates of serially ordered elements, and the result is a temporal sequence [Wang 1998]. Temporal sequences are not limited to ordered elements, but can also be related to common physical meaning of time where time is explicit.

From the example of language processing, several differences can be derived between a time series and a temporal sequence. The values of a time series are mostly real values sampled from some natural or physical phenomena to be studied. The values of a sequence can also be real, but sometimes they can be symbolic like in language processing. From the same example, it can be noticed that temporal sequence elements are not necessarily related to a true temporal dimension, but rather to an order relation between them. Besides, in most cases sequences contain a limited number of samples, while the time series can be seen as a continuous function with time, discretized for practical reasons. The nature of the variable, being in principle unbounded in time, leads to time series dealing with considerably larger number of data points. This also leads to consider the statistical properties of the time series like the average and standard deviation, and the study of the series properties like stationarity. One also could be interested in some statistical operations on the time series like regression.

From the processing point of view, there are some differences and similarities between sequence processing and time series processing, depending on the nature of temporal data and the application domain. The latter, besides to the processing tasks in both cases are the subject of the forthcoming subsections.

4.1.1 Time series processing

A time series as defined in [Dorffner 1996] is a sequence of *vectors* that we refer to as *elements*:

$$X = \{x(t)\}, t = 0, 1, \cdots, \infty \tag{4.1}$$

The typical tasks in time series processing include forecasting the future values, classification of the time series or a part of it into one of several classes, modeling the time series by describing the model that generated it, and mapping one time series onto another.

Forecasting the future values of the series is the prevailing task in time series processing. It

is concerned in predicting the value of the series element after some steps in time on the basis of its previous elements values. Formally, forecasting is concerned in finding the function $F : R^{k\times(n+1)} \mapsto R^k$, with k the dimension of x and n the number of previous elements, that gives the estimate $\hat{x}(t+d)$ of x at time $t+d$ given the element values of x up to time t. d is called the *lag of prediction*.

$$\hat{x}(t+d) = F(x(t), x(t-1), \cdots, x(t-n), \pi_1, \cdots, \pi_l) \tag{4.2}$$

With $\pi_1, \cdots, \pi_l$ are some time independent variables related to the model. Written this way, forecasting can be viewed as a *function approximation problem* carried out by regression, where the function F should be approximated as closely as possible.

Sometimes, the exact value of $\hat{x}(t+d)$ is not required, instead, the desired information is to know whether it increases or decreases or remain the same. In this case the problem can be seen as a *classification* problem where the goal is to map the series or a part of it (could be one value) to one of few classes, in this case into rising, falling, or constant classes. Thus, classification can be seen as a special case of function approximation, where the function to be approximated maps the values of inputs to a set of classes $F : R^{k\times(n+1)} \mapsto C$ with:

$$F_c : (x(t), x(t-1), \cdots, x(t-n), \pi_1, \cdots, \pi_l) \rightarrow c_i \in C \tag{4.3}$$

Another addressed task is *Modeling* the time series, which means finding the parameters of the model describing it, this is equivalent to finding the description of the function F in equation (4.2). When the description of F is found, it can be used to generate the time series by successively substituting future inputs by previous estimates.

Mapping a time series means finding the functional mapping between two series, for example, finding the mapping between one series like electricity consuming with another series like the population of a city where the model takes one series as its input and computes the elements values of the second as its outputs.

4.1.2 Temporal sequence processing

A sequence can be expressed as:

$$X = \{x(t)\}, t \in [i, j] \tag{4.4}$$

With $1 \leq i \leq j < \infty$. A part $x(k), x(k+1), \cdots, x(l)$ of the sequence, with $i \leq k \leq l \leq j$ is called a *subsequence*.

Sequence processing problems, or *sequence learning* can be categorized into four categories. Sequence *prediction*, similar to forecasting in times series, attempts to predict the future elements of the sequence on the basis of previous elements: given the sequence elements $x(i), x(i+1), \cdots, x(j)$ with $1 \leq i \leq j < \infty$, $x(j+1)$ is to be determined. When $i = 1$, prediction is made on all previously seen elements of the sequence. When $i = j$ the next element is predicted on the basis of the previous element only.

Sequence *generation* is essentially the same as prediction, it attempts to generate the elements of the sequence one by one in their original order: given $x(i), x(i+1), \cdots, x(j)$, generate $x(j+1)$.

Sequence *recognition* attempts to determine if the sequence fulfills some criterion, like in time series classification, the sequence is to be mapped into one of several classes, for example: $x(i), x(i+1), \cdots, x(j) \rightarrow yes, no$. This definition is conform to the incremental learning approach when values are introduced to the recognition model one by one, nevertheless, sequence elements can also be buffered and presented to the model following batch processing approach.

Sequential decision making is applied in interactive environments, where the sequence processing is related to the state of the environment (the system), and actions are to be taken to change the state. Sequential decision making can be one of two variations: *goal-oriented tasks* in which, given a sequence of states $x(i), x(i+1), \cdots, x(j)$ and a goal state $x(G)$, an action a_j at timestep j that leads to the state $x(G)$ is to be determined. This can be used to generate a trajectory; the task can be seen as trajectory-oriented in which at each timestep, the goal state $x(G)$ is $x(j+1)$. The other variation is *reinforcement learning* tasks [Sutton 1998b], they are tasks in which, given a sequence of state-action pairs $(x(i), a_i), (x(i+1), a_{i+1}), \cdots, (x(j-1), a_{j-1})$ and a state $x(j)$, it is required to choose an action a_j at timestep j that leads to receiving a maximum total reinforcement (or reward) in the future. If the action selection depends only on the immediately preceding state, then the action selection policy is Markovian. If it involves other preceding elements then it is non-Markovian.

The aforementioned sequence learning tasks are tasks that are rather related to machine learning. However, there are other approaches related to statistics like autoregression for forecasting time series [Sandholm 2007], autoregressive moving average for time series [Ben 1999], frequency-based (support) methods [Srikant 1995] and statistical dependence methods [A.E. 1995].

From the point of view of machine learning, sequence learning tasks can be supervised or unsupervised, the previously mentioned learning tasks except for reinforcement learning are often done by supervised methods. Although, unsupervised methods exist for some tasks like the unsupervised forecasting of financial time series [Pavlidis 2003] and the unsupervised face recognition from image sequences [Raytchev 2001]. In supervised methods, sequential data used for training are labeled with the corresponding output, when unlabeled, sequential data is handled by unsupervised learning methods. Learning in this latter case, is expected to discover patterns in the sequence that characterize the data the most. In machine learning, most sequence processing is done by artificial neural networks. Neural networks, by their different paradigms and arsenal of supervised and unsupervised models offer the suitable approach for processing temporal data.

4.1.3 Ambiguity of sequence elements

A sequence of the form $O - N - E$ is called *simple*. If the sequence contains repetitions like $C - O - N - F - O - N - D - R - E$ then it is called *complex* [Araujo 2002, Arbib 2003].

Complex sequences may cause ambiguity in sequence processing, because some sequence elements might occur at different points in the sequence. Let's consider the task of sequence prediction, suppose that we want that a model, say a neural network, predicts the next element in the sequence $C - O - N - F - O - N - D - R - E$. If the network last prediction was the element N, then what will be the next predicted element, F or D? Obviously the network can't determine. This ambiguity in the sequence leaves the network unable to predict which is the following element. However, the network could correctly predict the coming element if it is able to associate the ele-

ment N with its two preceding elements at each occurrence, so that it memorizes the subsequence $C-O-N$ and $F-O-N$. The subsequence that unambiguously determines an occurrence of the element N in the sequence is called the *context* of that occurrence of N. Its length (3 in the example) is called the *degree* of the element N [Wang 1995]. The *degree of a sequence* is defined as the maximum degree of all its elements. The degree is a concept that measures the *temporal dependency* in a sequence. As the network does not know which elements are the ambiguous ones, it should hold for each element in the sequence the subsequence of elements coming before with length equal to the sequence degree. A corollary is that the degree of simple sequences is 1.

Knowing the previous element in a complex sequence is not sufficient to correctly determine the next element, alternatively, the current element is not completely predictable by the past element alone. This is why complex sequences, also called ambiguous, are sometimes called non-Markovian [Sun 2001]: an ambiguous sequence is any sequence which is not Markovian of order 1. In this manuscript, complex, ambiguous, and non-Markovian sequences refer to the same concept.

Literature makes no clear distinction between temporal sequences and time series, although that they are drawn from sources that are different in nature. Their processing contains much similarity, although, it contains some differences that is sometimes reflected in different terminology. In particular, time series processing considers the statistical properties of the series that the processing models should consider. The previous discussion aimed to clarify their differences and similarities. However, regardless from the existing differences, time series are essentially temporal sequences that could have an infinite length. This maintains true at least when discussing processing models that deal with the temporal properties of both of them. Assessing the temporal dimension of data by a processing model whose main interest is the integration of the temporal dimension of the data makes that model blind to whether it is a time series or a temporal sequence. Thus, the use of "temporal sequence" is privileged henceforth, and what is said about temporal sequences is valid for time series. Although, we explicitly mention time series when talking about processing models that consider the statistical properties of temporal data.

4.2 Neural networks in temporal sequence processing

As seen in the previous section, there are various sequence learning tasks. When considering the various real temporal processing problems, one encounters different sequence properties, namely different element types, various sequence lengths, different sequence complexities. One then obtains a combination of different sequence properties and processing tasks that are difficult to handle by a single approach. There exist several approaches for sequence processing tasks, like hidden Markov Models [Bengio 1996], reinforcement learning [Sutton 1998b], evolutionary algorithms [Moon 1998], fuzzy systems [Juang 2004], rule-based systems [Park 2008], besides to neural networks.

In the context of this manuscript, cellular computing models in the temporal domain, we are particularly interested in neural networks as processing models for their fine-grain properties and their ability of time integration. Indeed, neural networks are able to cope with temporal sequences

and represent the temporal dimension of their inputs, and to this end, they implement different techniques than static networks. Unlike static networks, networks that deal with the temporal dimension of data implement an internal dynamics in order to integrate time, we refer to these networks as *dynamical neural networks*.

In the rest of this section, we discuss the main issues of neural networks related to temporal sequence processing. We differentiate between the internal and external time that the neural networks deal with, and talk about the existing approaches for representing time in neural networks, and their main temporal components.

4.2.1 Internal and external time

Neural networks are the dominant model in sequence processing, and was used since the mid 1980s to process temporal sequences. Prototypical neural networks like MLPs, Hopfield networks, and self-organizing maps deal with the static processing of data. Although, even in this case where the data doesn't contain a temporal dimension, networks use an *internal* representation of time. Internal time in neural networks could be continuous like in the continuous-time networks that use neuron models like the one expressed by equation (3.3) or discrete time networks like the one expressed by equation (3.4). Internal time in neural networks is necessary for the well-functioning of the network dynamics, like the computation of activations and weights update in MLP and self-organizing maps, and the relaxation of neurons activities dynamics as in Hopfield networks. When the input data have a temporal dimension, imposed by the nature of the problem at hand, say, speech recognition, then in addition to the use of internal time, a different time is handled by the network, which is the time characterizing the inputs and related to the problem at hand, this usage of time in neural networks is called the *external* time.

4.2.2 Time representation approaches in neural networks

The internal time in neural networks is a part of their internal functioning, it is there even in the case of static inputs, thus the focus is put on external time. Neural networks integrate time in the input data, i.e. the external time, following several approaches. Time is represented in the network either explicitly or implicitly.

The *explicit* time representation means that the integration of time is reflected either by an explicit change in network architectures or as an explicit variable intervening in the functioning of some network component like neurons and connection weights.

The representation of time can be explicitly introduced by adding additional components to the network architecture, typically a buffer implemented by tapped delay lines at the input of feed-forward networks like MLPs. In this case, the sequence elements at the input are pushed down one-by-one in the buffer. Elements are discarded when they arrive to the end of the buffer (FIFO behavior). At each specific timestep, the resulting vector in the buffer is a spatial vector for which an output is computed. In this method, the temporal dimension of the input sequence is converted to a spatial dimension in a time-to-space transformation. The retention of sequence elements besides to the limited number of past elements in the buffer allows the network to be sensitive to the temporal context of sequence elements, nevertheless, for the well-functioning of the neural network, the

degree of the sequence should not be higher than the buffer length. Using tapped delay lines, networks used for static processing could be converted to cope with temporal processing. The buffer can be viewed as a *window in time* of fixed size at the input level [Bengio 1989, Gorman 1988]. However, this windowing in time can be used also at other layer levels like the hidden layers in MLP. An example of the latter case is the TDNN [Lang 1990] networks explained later.

There exists another way to explicitly represent time in neural networks such as adding time at the connection level or at the neuron level. Time integration at the connection level concerns the connection weights, and results in *temporal weights*, which means that the connection performs a delay [Jacquemin 1994, Beroule 1987], or a convolution of the inputs with a temporal kernel [Natarajan 2008, Sutskever 2010]. For example, in [Sutskever 2010], the temporal kernel is exponential, a parameter λ with $0 \leq \lambda \leq 1$ parametrizes the connection established between the network input $x(t-k)$ at $t-k$ time steps in the past, and the network output $y(t)$ at time t so that the connection weight is $w_{xy}\lambda^{k-1}$, with w_{xy} a time independent weight. However, the use of temporal weights is not common in neural networks.

At the neuron level, neuron models presented in 3.5.2 integrate the temporal information in the neuron activity in two ways. The integrate-and-fire model of the neuron, integrates time by means of a differential equation, and the information is encoded in the time-course of the neuron activity. The other neuron models, the spiking neurons, also inspire directly from biology, the activity of the neuron is a train of discrete events, the spikes, and information is held in the timing of these spikes.

The implicit representation of time means that time doesn't appear explicitly in the network, instead, the temporal dimension of input data is integrated in the *internal state* of the network. The state is determined by the activations of some hidden units, these activations depend on the current input and the past activations that represent information about input history. The network state acts as a memory that stores information about past inputs, enabling to store the context of inputs. Hence, time could be thought of as an index of the "sequence of network internal states". This type of memory results from adding recurrent connections to the network giving the *recurrent neural networks* (RNNs). Depending on where the recurrent connections are added in the network, different temporal capabilities arise. The memory implemented by the state units activations is called the *short-term memory*, it is "short" because activations vary rapidly, compared to the network weights that vary based on the experience but in a much slower rhythm than the activations, thus they are also considered as a sort of memory, called the *long-term memory*. RNNs reduce the weight number compared to the case of delay lines because they are able to deal with arbitrary long temporal dependencies [Sutskever 2010]. However, the short-term memory is efficient with "short" temporal dependencies, but when computing the output for the current input depends on far inputs in the past, many existing models have difficulty in handling such situation, while some models (like NARX, explained later) show better results. This problem is known as the problem of *long-term dependency*.

4.2.3 Temporal Components of neural networks

To be able to cope with the temporal dimension of sequences, the neural network should be able to hold information about the context of sequence elements as defined in 4.1. The context of an element depends on its past inputs, and the network should be able to retain an encoding of

information related to some input for many timesteps after its presentation. Not only should the network have some way to store the current context, but it should update it in such a way that reflects the new inputs. In addition, the network should keep the context information until it is needed. For the retention of the context information the network should implement a memory.

Besides to having a memory, it has been shown in 4.1 that most temporal tasks can be seen as performing prediction (or forecasting). On this basis, neural networks for temporal sequence processing are seen to functionally have two main parts (subnetworks): a short-term memory and a *predictor* [Mozer 1994, Vries 1992, Chappelier 2001]. This is illustrated in figure 4.1.

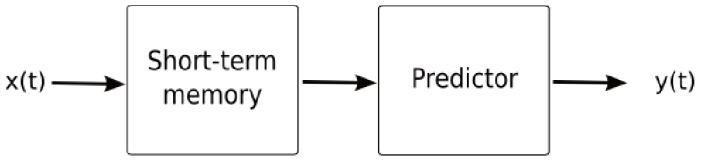


FIGURE 4.1: Parts of dynamical neural networks: two functional parts, a short-term memory that retains information about the input context, and a predictor that computes the output based on the memory content.

The short-term memory is the part of the network responsible for holding the context, it is expected to retain the aspects of the input sequence that are related to the problem. In particular, for each input (could be one sequence element or a set of elements depending on how inputs are introduced to the network), the short-term memory should hold a specific representation (in the form of units activity) that associates the input with the relevant past inputs, i.e the context of the element. The current context depends on the past content of the short-term memory and the current input. The content of the short-term memory is referred to as the *state* of the network.

The predictor part of the network depends on the content of the short-term memory, i.e. network state, to compute the output of the current input. The network can thus be perceived as computing an output sequence as a response to the input sequence, performing some kind of sequence mapping as defined in 4.1. The two parts of the network can be separate from each other, in this case, the predictor part of the network is typically a static network like an MLP as in [Elman 1991, Catfolis 1994] in which the model consists of a recurrent network followed by an MLP, or like in reservoir computing architectures [Embrechts 2009]. However, the two parts can be intermingled so that no distinct separation between them could be made, NARX networks are an example [Diaconescu 2008] .

The short term memory is the important part to study, as it endows the network with its ability to account for temporal information in inputs. The multitude of neural networks architectures for temporal problems processing allowed for the appearance of different implementations of short-term memories. These memories can be classified depending on three axes: the memory *form*, *content*, and *plasticity* [Mozer 1994]. Formally, the short-term memory can be defined as a function of time and the previous inputs [Chappelier 2001]: $f(t, x(t-1), \cdots, x(t-k))$. The memory form is related to the function f itself. In the simplest case, the tapped delay lines implement a spatial representation of the context, and the function f returns $x(t-1), \cdots, x(t-k)$ where k is the length of the delay line. This means that at timestep t, the memory retains a representation of the input back to $t-k$ timesteps in the past. Generally, the memory doesn't contain raw data of the input as in the case of delay lines, instead it could contain a transformed version of the delayed inputs,

which is the output of the function f. The transformation is obtained by the convolution of the input sequence with some *memory kernel* as will be explained later.

The memory content is related to the number of arguments that f takes and the nature of these arguments. The content at time t may depend on the inputs at previous timesteps, or may depend on both the inputs and the previous outputs of the function f itself. In other words, a state unit could compute its activation (or state) based on the current input and the previous activations of neighbor units possibly including itself. The memory content can depend on a limited number b of memory values, thus it can be seen as Markovian of order b. However, sometimes it may depend on all the past history of the memory.

The content of a short term-memory is characterized by two measures; *depth* and *resolution* [Koskela 1996, Mozer 1994]. The depth of the memory refers to how far in the past the memory can retain information, and the resolution refers to how accurate are the retained information concerning the elements of the input sequence. A low depth memory holds only information about the recent sequence elements, while a high depth memory holds information about sequence elements distantly presented in the past. A memory with a high resolution allows to reconstruct the recently stored sequence elements, while a low resolution memory holds coarser information about the stored sequence elements.

The memory plasticity is related to the derivation $\frac{\partial f}{\partial t}$ which indicates how the memory evolves through time. This process is attained by adapting the connection weights between the memory units. Weight adaptation in neural networks is generally known as *learning* or *training*. As will be shown in the following sections when presenting some example neural networks for temporal sequence processing, there are some *static* short term memories in which weights are fixed in advance, those mainly include networks that use delay lines. Adaptive memories on the other side are those in which some connection weights are learned.

4.3 Feedforward networks for temporal sequence processing

Feedforward networks are static networks that perform static processing of data, although, these networks can be used in some special cases of temporal problems. The principal feedforward networks are multi-layer perceptron (MLP) and radial basis functions networks (RBFs).

An MLP with n, k, m units in the input, hidden and output layers respectively, is a function $F^{MLP} : R^n \mapsto R^m$ that takes as input the vector X of length n:

$$F^{MLP}(X) = \sum_{j=1}^{k} v_{jl} f(\sum_{i=1}^{n} w_{ij} x_i) \tag{4.5}$$

With $l = 1, \cdots, m$, and w_{ij}, v_{jl} are the weights of hidden and output layers respectively. The hidden units activities are nonlinear functions f of the inputs, all hidden units activities are computed using the same function which is nonlinear and non-polynomial. The output of the MLP is a linear combination of the hidden units activities.

Radial basis function network (RBF), on the other side are given by the equation:

$$F^{RBF}(X) = \sum_{j=1}^{k} v_{jl}\Gamma(\sum_{i=1}^{n}(w_{ij} - x_i)^2) \tag{4.6}$$

Where Γ indicates the Gaussian function. Like MLPs, RBFs are proven to be universal function approximators given a sufficient number of hidden units [Park 1991]. Both MLPs and RBFs approximate nonlinearity by the superposition of several instances of the nonlinear functions f and Γ.

These static feedforward networks can be used in temporal sequence processing, even in the online mode, in the special case when the stream of sequence elements are temporally independent, i.e. when the sequence is simple. In this case, when the multivariate (i.e. vector) sequence element $x(t)$ is presented as network input, the network computes an output function $F(x(t))$ of a single input.

Another example of static feedforward networks in temporal sequence processing is NetTalk [Sejnowski 1987] which is used for English text pronunciation, it takes a sequence of letters that are spatially presented to the input layer in a serial-parallel transformation, thus, inputs can be seen as a window in time without explicitly having delay lines. However, the number of units in the input layer is large (203 in [Sejnowski 1987]).

Some other tricky methods were used to process temporal data by static networks, whether they are feedforward networks or other static ones like self-organizing maps. Spatiotemporal data can sometimes be pre-processed to fit static models, one approach is to transform the time-domain data to the frequency domain through Fourier transform, this was used in [Amari 1998]. Another approach proposed in [Mozayyani 1995] consists in using complex numbers to encode time, where one quadrant of the number is used to encode past values, and the another quadrant is used to encode present and future values. The authors in [Mozayyani 1995] applied this on an MLP and a self-organizing map.

Learning of feedforward networks is supervised. In supervised learning a set of inputs and the related target outputs is provided, this forms a training set $\left\{x(t), \hat{y(t)}\right\}$ for $t = 1, \cdots, T$, where T is the length of the training set. The goal of learning is to adapt the synaptic weights w of the network in order to fit the best its computed output to the target output, the difference is considered as the network error. In general, it is better to separate the available set of labeled data into a training set and a test set, so one can test the accuracy of the model on data not used in learning. Other training scenarios, like cross-validation, are also possible.

The used algorithm for learning in feedforward networks is *backpropagation*, it consists of two steps: first, the inputs are propagated forward through the network to compute the output, and second, the gradient of the errors between the computed and the target outputs propagated back in the network and used to update the weights. The error to minimize is often the mean square error (MSE) given by:

$$E = \sum_{i=1}^{T} E(t) = \frac{1}{2}\sum_{i=1}^{T}(\hat{y(t)} - y(t))^2 \tag{4.7}$$

In batch learning mode, the weights of the network are updated after the presentation of a

training set that consists of a part or all the available inputs. In the incremental learning mode, the weights are updated after the presentation of each single input, and thus, the error is computed based on $E(t)$ instead of E, i.e. for each input element. There exist different learning algorithms, most of which are gradient descent algorithms [Bishop 1995a] .

Let's now discuss whether these feedforward networks are cellular models, to this end, let's consider a feedforward network with three or more layers. These networks are fine-grain ones with simple processing units working in parallel. However, the computation of the network output requires the propagation of information forward, so that, in some time step, computation consists of ordered subsequent actions: first the output of the input layer units is computed on the basis of the network inputs, then the output of hidden layer units is computed on the basis of the output of the input layer units, and third, the output layer units compute their outputs based on the output of the hidden layer units. Thus, it is clear that in order for the output layer units to compute they require an information related to the input layer to which it has no connections, which contradicts with the functional connectivity condition. Also, the update of the weights at the input layer depends on the error E which is a global value computed on the network level by a central processor, this value is propagated back in the network, so that for example the weights in the input layer are updated according to values computed on the basis of the globally computed error E to which units at the input layer has no direct access by their connections. This also contradicts with the functional locality condition, besides, computing E contradicts with the decentralized processing condition. In addition, as mentioned earlier in this manuscript, the full connectivity between layers, reflected in equations (4.5) and (4.6), is contradictory with the topological locality of cellular models.

As a result, although feedforward networks fill the simplicity and parallelism conditions of cellular computing paradigm, the weights update process is not decentralized, the connectivity is not local, and the computation is not functionally local as well. Hence, **feedforward networks are not cellular models**.

4.4 Dynamical networks with delay lines

Among the arsenal of dynamical neural networks capable of temporal sequence processing, there are architectures that implement short-term memory by holding past input values in the network or a transformation of these inputs. For this purpose delay lines are typically added at the input layer of the network so that input elements are fed to the buffer obtained by the delay lines, and values transformation may or may not be applied. Models that rely on delay lines make a clear separation between the short-term memory and the predictor parts. The short term memory consists of the buffer formed by delay lines. The predictor part is normally a static feedforward network such as MLP or RBF which consists of no, one, or multiple hidden layers. In the rest of the section, the most important architectures that rely on delay lines are presented and their main capabilities relative to temporal sequence processing are explained.

4.4.1 Standard tapped delay-line networks

In these networks, tapped delay lines are added at the input layer of feedforward networks like those presented in the previous section, so that they implement a buffer; the sequence elements are buffered so that the network input at time t is the sequence $x_1(t), x_2(t), \cdots, x_n(t)$. Figure 4.2 illustrates a tapped delay line. When the content of the tapped delay line consists of the n past sequence elements $x(t), x(t-1), \cdots, x(t-n-1)$ then we refer to it as "standard" tapped delay line in order to differentiate it from the other type explained later. At each timestep, sequence elements are shifted one by one into the buffer at the input layer. The value of $x_i(t)$ is equal to $x_{i-1}(t-1)$. The vector of inputs held in the buffer at each timestep is processed in a static way by the predictor part to compute the output.

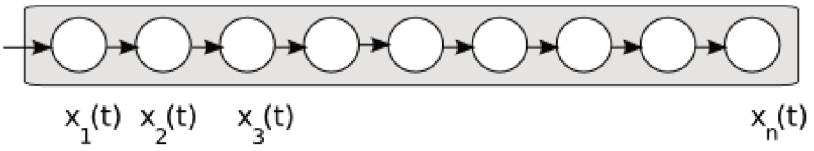


FIGURE 4.2: Tapped delay line: it implements a buffer of length n at the input layer. Sequence elements are shifted one by one at each timestep, when an element arrives to the end of the buffer, it is discarded in the next timestep.

Let's consider that the output layer of the feedforward network consists of a single unit. Then the network, which is a universal function approximator, is capable of approximating the function F in equation (4.2). Let's also put the lag of prediction d in equation (4.2) to 1. Then the network can compute the prediction function:

$$x(t+1) = F^{NN}(x(t), \cdots, x(t-n-1)) \tag{4.8}$$

Where F^{NN} is a nonlinear function that could be F^{MLP} or F^{RBF}. The operation that the network performs is called nonlinear autoregression of order n (the number of past elements used to predict the next element) and is denoted AR[n]. Autoregressive models are used in time series processing, but most existing non-neural models assume a linear relationship between the sequence elements [Dorffner 1996] of the form:

$$x(t) = \sum_{i=1}^{n} \alpha_i x(t-i) \tag{4.9}$$

These linear models assume the stationarity of the time series, which means that the statistical properties of the sequence elements such as the mean and standard deviation are fixed on all sequence parts. The nonlinear autoregression that the neural networks are capable of can approximate the linear function F in the past equation that linear models suppose, but it is more powerful than traditional linear autoregressive models. Examples of such networks are used in one step prediction of time series line in [Koskela 1996].

One practical problem in these architectures is that learning the network weights is computationally demanding due to the big number of the to-be-adapted weights resulting from adding delay lines . A variation of these models that intends to reduce the number of learned weights is the Time

Delay Neural Network (TDNN) [Lang 1990]. In this architecture, the full connectivity between layers is avoided. The delay line at the input layer is functionally separated to multiple time slices. Each time slice contains a few units so that units in a time slice are used to compute the activity of one unit in the next layer. Adjacent time slices have unified connection weights. As an illustrative example, the weight of the first connection in the first time slice connecting the first unit to the unit in the next layer is the same as the weight of the first connection in each other time slice connecting the first unit in that time slice to a unit in the second layer, this is illustrated in figure 4.3.

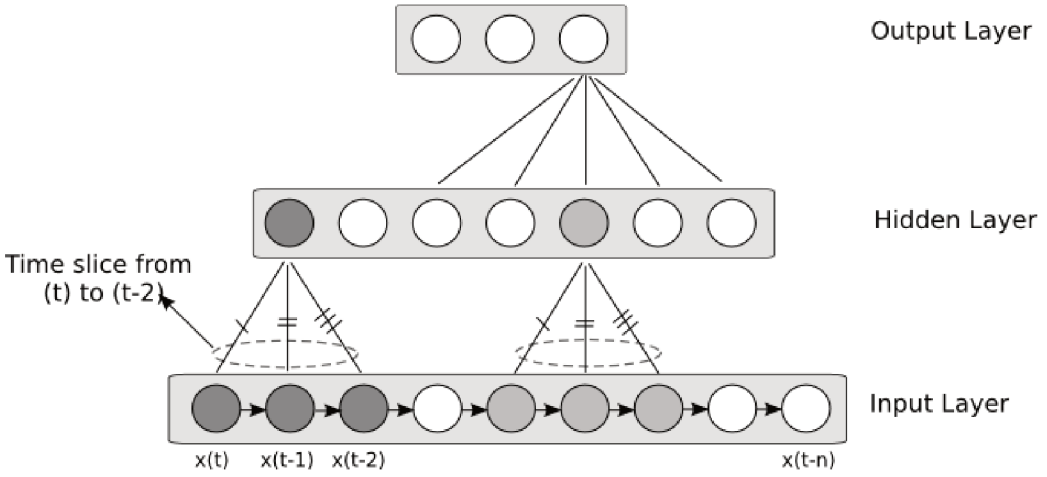


FIGURE 4.3: TDNN: the network used tapped delay line at the input and reduces the number of weights by implements partial connectivity; each limited number of units in each layer forms a time slice and units in a time slice are connected to one unit in the next layer. The number of units in the time slices is unified in each layer. Weights are shared between connections in the time slices of a specific layer as indicate the signs of equality on time slices connections.

Unlike the previous architectures in which the predictor part implements the common functionality of feedforward networks, in TDNN the same principle of time slices used in the input layer is also used in hidden layers, with potentially different sizes of time slices, the only difference with the input layer is that there is no shift of values in hidden layers. Weights unification between time slices in one layer is called *weight sharing* or *weight tying*. This mechanism aims to reduce the number of weights that need to be learned, thus to reduce the computation required for learning, however, the number of weights and the required computation remain considerable. Besides, This engineered mechanism requires a centralized processor to maintain the weight identical between connections sharing the same weight, making the network not cellular. Other architectures that use the standard tapped delay lines as short-term memory are [Elman 1988, Waibel 1989] used for speech recognition and [Lapedes 1987] used for signal processing, among others.

4.4.2 Memory kernel networks

In TDNNs, the buffering of the sequence elements is done by a "standard" tapped delay line, in which a unit in the line simply contains the content of the previous unit in the previous timestep. However, there are other forms of delay lines, that implement more complicated relations between

the delay line elements, hence, implementing another memory form.

In the standard tapped delay line seen before, the relation between the buffer content and the input values is given by the following: $x_i(t) = x(t - i + 1)$. It can also be written as $x_i(t) = x(t - w_i)$ for arbitrary long delay. However, there exist other buffer types in which when shifted in the delay line, the value of the retained element changes. This is obtained by the convolution with a kernel c_i called the *memory kernel*. The general formulation for this convolution is given by:

$$x_i(t) = \sum_{\tau=1}^{t} c_i(t - \tau)x(\tau) \tag{4.10}$$

Delay lines with an arbitrary delay w_i can be obtained by the kernel:

$$c_i(t) = \begin{cases} 1 & \text{if } t = w_i \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

Where standard delay lines can be obtained by setting $w_i = i - 1$ for each $x_i(t)$. By using different kernels, one obtains different memory forms. For example using the following kernel with μ_i in the interval $[0, 1]$, the *exponential trace memory* [Mozer 1995, Jordan 1990] is obtained:

$$c_i(t) = (1 - \mu_i)\mu_i^t \tag{4.12}$$

By comparing equations (4.11) and (4.12), a straightforward difference between standard delay lines and the exponential trace memory can be drawn. The latter doesn't sharply drop off at a fixed point in time like standard delay lines, instead, a unit $x_i(t)$ in the line depends on more than one past value; it keeps trace of the values that pass through it, and this trace decays exponentially with time. This way, the recent inputs to the delay line are more relevant to the actual output computation, while information related to past inputs become more difficult to take into account by the network with the time course. As a result, the standard delay line has a lower depth compared with the exponential trace memory, but has a higher resolution because information about past sequence elements does not decay like in the exponential trace memory.

One important property of the exponential trace memory is that the values of $x_i(t)$ can be computed incrementally:

$$x_i(t) = (1 - \mu_i)x_{i-1}(t) + \mu_i x_i(t - 1) \tag{4.13}$$

The value $x_i(t)$ is a linear combination of its past output $x_i(t-1)$ and the current output of the past unit $x_{i-1}(t)$. This can be perceived as a moving average filter of order 1 (MA[1]) parametrized by μi. If μ_i is close to 1 then the unit output $x_i(t)$ is mainly determined by the past value of the same unit $x_i(t - 1)$, if μ_i is close to 0 then the $x_i(t)$ is mainly determined by the output of the past unit $x_{i-1}(t)$. In the latter case the behavior of the exponential memory resembles to that of the standard tapped delay line. It can be noticed from equation (4.13) that a unit with index i in the delay line performs a value decay and can be seen as leaky integrator unit. For this reason, exponential trace memories are also called *feedforward exponential decay* (FED) memories [Barreto 2003].

Let us define the shift operator q such that $q^{-1}x_i(t) = x_i(t-1)$. By rearranging the terms of equation (4.13) and using the shift operator, it can be rewritten in the form of a filter:

$$x_i(t) = \frac{1-\mu_i}{1-q^{-1}\mu_i}x_{i-1}(t) \tag{4.14}$$

The gamma memory obtained by the gamma filter presented in [Vries 1992] is intended to generalize across delay lines and exponential trace memory, in such a way that allows to parametrize a continuum of memory forms ranging from high resolution and low depth, to low resolution and high depth. Given γ_i in $[0, 2]$ the gamma filter is given by:

$$x_i(t) = \frac{\gamma_i q^{-1}}{1-(1-\gamma_i)q^{-1}}x_{i-1}(t) \tag{4.15}$$

There exist other memory forms, that can be expressed as memory kernels or alternatively, as filters. Each different filter results in a different memory form. Other important filters implemented by tapped delay lines to implement memory kernels are Laguerre filters [Wahlberg 1991], and IIR/FIR filters.

There are some drawbacks related to delay networks for time representation. The buffer length that determines the length of the window in time is fixed in advance, therefore, it is necessary to adapt the buffer length with the maximum temporal dependency length in the input sequence, i.e the degree of the sequence (as defined in 4.1). This implies that the degree of the processed sequence should be known beforehand, which is not always possible, especially when sequences arrive online. The network may either encounter sequences with longer temporal dependency than the size of the buffer, or inversely, the buffer length may be larger than the degree of the processed sequence, the latter case means allocating extra computational power that exceeds the task needs. Even though, typically not all sequence elements are of the same degree, and thus, irrelevant and redundant temporal information are presumed and computed for elements with lower orders. Memory kernel networks have better memorizing capability as they have higher depth than standard delay lines, thus they can retain information related to sequence elements past in time that exceeds the buffer length, however, the resolution drops exponentially and the information becomes sooner useless. What is common to both standard delay lines and memory kernels is their lack of "selectivity" to the different degrees of sequence elements; the context of all elements is memorized in the same way.

Another drawback is that adding delay lines causes delay networks to contain a large number of weights that causes over-training [Chappelier 2001]. Weight sharing in TDNNs tries to reduce the number of weights to be learned, although, their computation remains expensive, especially when dealing with high degree sequences requiring large delay lines. In this latter case, delay networks doesn't hold practical because of the large number of added weights and the high computation time.

Learning in the delay networks is supervised, it consists in adapting the synaptic weights of the adaptive part which is the predictor, the latter being a feedforward network. No learning occurs on the level of short-term memory whether it is a standard delay line or a memory kernel.

The benefit of the delay networks is that they are easy to build and tune the depth or the resolution, and most importantly, they can be trained with classical backpropagation without any modification. Training is carried out in a similar way as in static feedforward networks. As explained in the previous section, a centralized processor is used for this training, and the weight update depends on computed values that are not accessed by units connections, thus, **delay networks are inherently not cellular models**.

4.5 Recurrent neural networks as state models

The delay approach for short-term memory presented in the previous section uses delay lines to hold the temporal context of inputs. Delay networks, suppose that all input elements have the same degree, and remember their history equivalently. In fact, these networks could be thought of as dealing with a spatial representation of time that is maintained in the delay line. The consecutive outputs are then computed starting from inputs following the same functional operations, this computation could be referred to as *static*. As has been explained, when they should retain information about inputs deep in the past, these networks employ a large number of units in the delay line, and the number of weights explodes, leading to a large computation time.

Another approach for accounting for the temporal context of inputs relies on providing the network with a dynamical *internal state*, obtained by adding recurrent connections to static networks, leading to the distinction of the major family of dynamical neural networks dedicated for temporal sequence processing, the *recurrent neural networks* (RNNs).

Recurrent connections are looped connections that implement a delay, typically with one timestep. Units in some layer are connected to the other units in the same or previous layers. These connections implement a form of positive feedback that aims to maintain the previous network state information in the network by feeding them back to the processing units, creating a recurrent path. Recurrent connections reinject the past units activities in the network and allow the network to develop a self-sustained temporal activation dynamics along the recurrent path; i.e. the network state that changes with time, making of the RNN a dynamical system. Hence, the computation of the output is no longer static as in delay networks. This dynamical computation is the main functional difference between RNNs and both static feedforward networks that behave as functions and dynamical delay networks.

The internal state of the obtained dynamical system is called the *network state*, it is a vector of activations whose elements are maintained in *state units*. The work of state units is to latch information about the past activations of some units in the network, that vary depending on the architecture. The stored information are nonlinear transformations of the temporal inputs presented to the network.

The network state serves as a short-term memory that maintains information about past inputs for an arbitrary period of time. This allows the network to process the context of past inputs, in a different way than the window in time used in delay networks.

Representing the arbitrary context of different inputs in the dynamical state of the network, replaces the need for delay lines that impose large number of connections and weights. Besides, learning in RNNs allows the short-term memory to adapt to the degree of sequence elements, so that the network maintains no more than the sufficient context information for each input.

RNNs are proved to be universal approximators of dynamical systems [Funahashi 1993], and are known to be capable of representing high complex functions on input sequences [Sutskever 2010] and thus suitable for nonlinear temporal sequence processing problems [Lukoševičius 2009] .

With very few exceptions, almost all recurrent networks work in discrete time. RNNs implement an implicit representation of external time characterizing the inputs sequence. Time can be seen as an index of network internal states where the state of the network changes with each new processed input, this is why RNNs are sometimes called "adaptive state networks" [Koskela 1996].

From this perspective of dynamical systems, two classes of RNNs can be distinguished, the first class is characterized by symmetric connectivity between the network units and an energy minimizing dynamics. The most famous example is the Hopfield networks [Sulehria 2007], and their stochastic variant Boltzmann machines [Ackley 1985]. These networks have functionally local connections that make their asynchronous update possible (see 3.3.2). However, with the asynchronous update regime, supervised learning is not possible, thus they are trained using unsupervised learning rules. The second class is characterized by directed connections and deterministic update dynamics, these networks can be perceived as nonlinear filters that associate an input sequence with an output one (sequence mapping). RNNs of this class are mostly trained in a supervised way.

There is yet no general theoretical framework that describes the properties of recurrent networks [Cruse 2006], however, almost any recurrent neural network architecture can be expressed in the form:

$$S(t) = f(S(t-1), X(t), \pi_t^f) \tag{4.16}$$

$$Y(t) = g(S(t-1), X(t), \pi_t^g) \tag{4.17}$$

where $S \in \mathbb{R}^{N_S}$ represents the network state which is a vector of activations, $X \in \mathbb{R}^{N_X}$, $Y \in \mathbb{R}^{N_Y}$ represent the input and the output vectors of the network respectively, π_t^f, π_t^g represent the model parameters which are typically the weights, and f, g are functions that depend on the specific architectures, and they are applied element-wise. f can be seen as the short-term memory function, and g is the function implemented by the predictor part. Here, the dimension N_S of the state vector is usually higher than the dimension N_X of the input. This implies that the state results from an *expansion* of the input space to a higher dimensional space. One can think of X as included in S as the latter implements a transformation of X.

The functional dependency in RNNs between the past state, the current state, the current input and the current output is illustrated in figure 4.4(a).

This general representation for RNNs functional dependency fits equations (4.16) and 4.17. However, some networks compute their output from the current state, not the past one, without involving the current input. Thus equation (4.17) can be simplified to:

$$Y(t) = g(S(t), \pi_t^g) \tag{4.18}$$

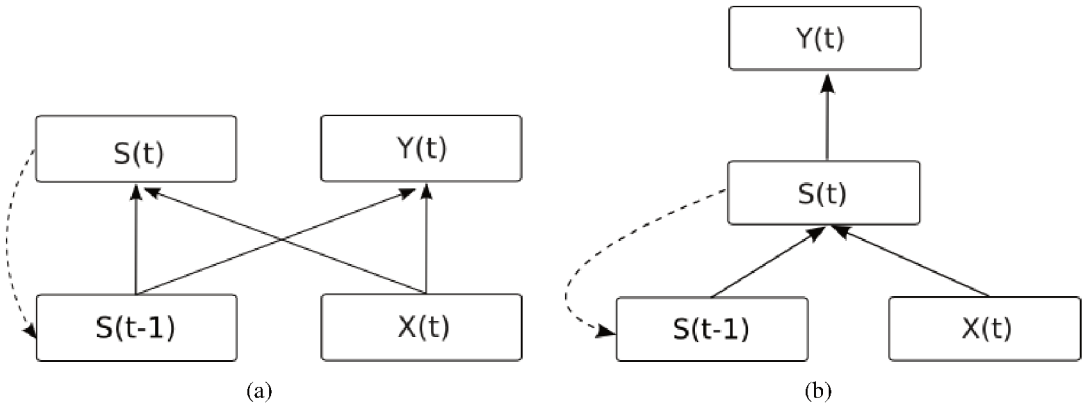


FIGURE 4.4: Two forms of the functional dependency of the state, input, and output of recurrent networks. (a). The output depends on the past state and the current input as in equation (4.17). (b). The output depends on the current state as in equation (4.18). The two forms are equivalent.

The functional dependency here can be seen as equivalent to the former one, as $S(t)$ depends on $S(t-1)$ and $X(t)$. The functional dependency that fits this case is illustrated in figure 4.4(b).

In order to illustrate what is different in recurrent network from static networks, feedforward networks given by equations (4.5) for MLPs and 4.6 for RBFs, can be each expressed as a set of two equations:

$$S(t) = f(X(t), \pi_t^f) \tag{4.19}$$

$$Y(t) = g(S(t), \pi_t^g) \tag{4.20}$$

It is obvious that the main difference, is that the computation of the state $S(t)$ in recurrent network takes into account past state information, however, it can be said that it takes into account the input history as well. Nevertheless, the notion of state is not defined in the case of feedforward networks as they are not dynamical systems, instead, $S(t)$ represents the hidden layers activity. Delay networks can also be expressed similarly as they are based on feedforward networks.

Recurrent connections can be added to different networks at different parts of them, leading to a large number of implemented recurrent networks. Recurrent connections can be added to existent models like feedforward networks or to delay networks. Hence, recurrent networks are a more general model of neural networks than feedforward ones, because any feedforward model can be obtained by setting recurrent connection weights to 0. Besides to adding recurrent connections to existing models, some new recurrent architectures relying on new concepts were also developed such as reservoir computing networks.

This multitude of recurrent neural network architectures is used for temporal sequence processing. Architectures resemble and differ depending on several criteria, such as where the recurrent connections are added in the network, the form of short-term memory, how output is computed. There are many attempts to find unifying themes of RNNs and dynamical networks in general, leading to the appearance of different unifying views and taxonomies, the most important ones

are [Nerrand 1993, Tsoi 1997, Tsoi 1998, Kremer 2001, Barreto 2003].

In the following subsections, the most important architectures for time integration are presented, while briefly discussing the learning algorithms and contrasting these models against the requirements of cellular computing paradigm.

4.5.1 Elman networks

Elman architecture [Elman 1990] is an MLP with one hidden layer with recurrent connections. One-to-one recurrent connections with weights fixed to 1.0 are added to the hidden layer of the MLP, so that they copy the activation of hidden units to some context units in the input layer as called by Elman, that is:

$$a_i^C(t) = a_i^H(t-1) \tag{4.21}$$

where C indicates a context unit in the input layer, H a state unit in the hidden layer, and i the index of the unit in the layer, this implies that the number of context units is equal to the number of hidden units. At time t, the context units maintain a copy of the past state $S(t-1)$. The state units in the hidden layer then compute their activation on the basis of input units and context units activations in the input layer. This way, the past state $S(t-1)$ participates in computing the new state $S(t)$. An Elman network is depicted in figure 4.5.

The units in the hidden layer that maintain the dynamical state $S(t)$ of the network, besides to context units in the input layer, implement both the short-term memory mechanism. This short-term memory part of the network performs three operations: input, computed as in MLPs, and then comes the copy operation: an input element $x(t)$ is first loaded into the input layer, then the current input with the previous state $S(t-1)$ maintained by context units are used to compute the current state $S(t)$, which is in turn used to compute the output $Y(t)$ following 4.18. The final operation is copying the computed state back to the context units.

The state computation given in equation (4.16) is given as follows for Elman networks:

$$S(t) = f(A.S(t-1) + B.X(t)) \tag{4.22}$$

where A, B are matrices in which is hidden the term π_t^f referring to weights in equation (4.16), and f is the nonlinear sigmoid function.

Concerning learning, the recurrent weights values are fixed, and only feedforward connections are adapted. The original algorithm used by Elman [Elman 1990] was learned using a truncated version of gradient descent, that assumes the gradient of context units activations with respect to the network weights is zero, allowing to use the standard backpropagation algorithm for learning. With this assumption, learning in Elman networks can be carried out using any algorithm used for MLPs. Elman networks were also trained by Kalman filter algorithm as in [Williams 1992b].

4.5.2 Jordan networks

The original model of Jordan [Jordan 1986] is an MLP with one hidden layer that contains one-to-one recurrent connections from the output layer to some context units in the input layer as in Elman

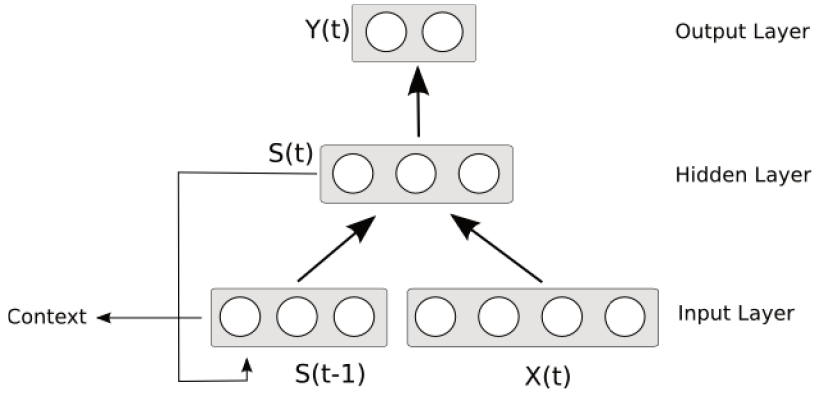


FIGURE 4.5: Elman network: backwards recurrent connections copy state units activity to the context units at each iteration. Bold arrows represent full connectivity between layers, while other arrows represent one-to-one connections.

networks. The weights of these connections are fixed to 1.0. Hence, the state of Jordan network is the activity of the output layer units.

Besides to recurrent connections from the output to the input layers, Jordan networks also include recurrent connections that connect context units to themselves. This can be thought of as making a copy of the context units, this copy maintains the past context.

However, depending on the application, the connectivity pattern of context units in Jordan networks varies from one source to another.

Some sources use fully recurrent connections between the past output copy and the past context copy on the one side, and the current context units on the other side. The weights of these connections are inferior to 1.0. This architecture is illustrated in figure 4.6.

Context units (not their delayed copy) in the input layer are computed on the basis of the past output and the past context. However, some other sources use self-recurrent connections in the context layers, i.e. one-to one connections between the context units and themselves, in this case the activation of a context unit is computed as follows:

$$a_i^C(t) = y(t-1) + a_i^C(t-1)d_i \tag{4.23}$$

where d_i is a decay weight of the context unit i. The weights d_i of recurrent connections are fixed and set to be less than 1.0. Self-recurrent connections decay the context units activations and feed them back to the original context units, so that context units behave as integration units and are sometimes called *decay units* or *capacitive units*. Units connected either by fully-recurrent or self-recurrent connections are intended to serve as a memory for previous context information, but at the same time, having weights connections inferior to 1.0 allows the context units to forget past values and account for new ones.

The short-term memory in Jordan networks implements a more complex dynamics than in Elman networks. The context information in Jordan networks circulates infinitely in the network, while in Elman it is copied one time then changed. This can be noticed by comparing equa-

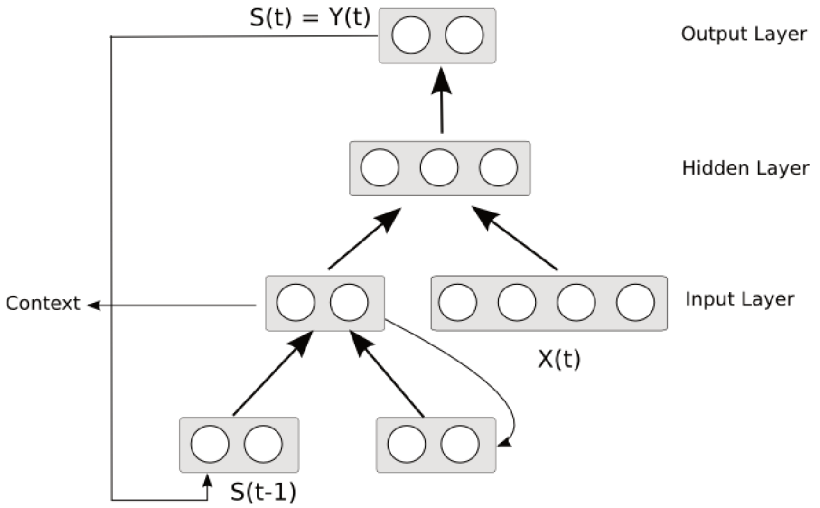


FIGURE 4.6: Jordan Network: The state is maintained in the output layer and is copied using backwards one-to-one recurrent connections to the context units. A copy of context units activity is also maintained for one timestep by one-to-one recurrent connections. Both the copied output and context activities participate in the computation of the next context.

tions (4.21) and (4.23).

Because all recurrent connections have fixed weights, learning in Jordan networks can be done as in standard backpropagation. Moreover, as output activations are recurrently transmitted to the input, the backpropagation algorithm in Jordan nets could be simplified to teacher forcing [Williams 1988], which is a special case of real time recurrent learning algorithm (RTRL) in which target outputs, instead of the computed ones, are fed back at each time step to context units. This method is proved to converge faster to the desired network dynamics used in the exploitation phase. Other conventional supervised methods like backpropagation through time (BPTT, detailed further) are using the target values to compute and accumulate the error then use it in one shot at the end of a time cycle of some length, while RTRL and teacher forcing RTRL apply the learning at each timestep. To illustrate this difference, the authors in [Toomarian 1991] give the example of a parent teaching his child to ride a bicycle by watching him from the window and giving remarks at each trial, this is similar to conventional supervised methods, while teacher forcing resembles to going out with him and giving him a teacher information concerning each move.

Jordan and Elman networks can be trained to generate a set of output sequences with fixed inputs, so that each different input triggers a different output sequence. With sequences as inputs, they can be trained to recognize different sequences, or to predict next sequence as in [Cheng 2007, Koskela 1996].

The capability of the recurrent network is related to its architecture and connectivity pattern. On the one side, besides to Elman and Jordan networks architectures, there exist other similar recurrent architectures for sequence learning, some of them are given in [Rolf Pfeifer 2010]. One example is a feedforward network that consists of input, hidden and output layers, with self-recurrent con-

nections added to the input layer, here, input units are also the context units. Another architecture consists of input, context, hidden and output layers, with self-recurrent connections only in the context layer. These latter architectures are well suited for sequence recognition but less well suited for sequence generation like Elman or Jordan networks [Rolf Pfeifer 2010].

On the other side, in Elman architecture, and in some instantiations of Jordan architecture, each context unit receives connections from all other copied context units, these architectures and similar ones [Williams 1988] are called *fully recurrent networks*. In other network architectures like the ones used in [Bengio 1992, Back 1991] the context units are connected to themselves via self-recurrent connections, but not to other context units, this type of networks is called *locally recurrent neural networks*. Fully recurrent networks provide more complex processing of activation values and have greater representational power [Kremer 1999].

4.5.3 NARX networks

An important class of discrete time nonlinear systems is the *Nonlinear AutoRegressive with eXogeneuos inputs* (NARX) [Leontaritis 1985, Chen 1990] . The model computes its output not only on the basis of the inputs but also on the basis of past outputs, that is:

$$y(t) = f(y(t-1), \cdots, y(t-d_y), x(t), \cdots, x(t-d_x)) \tag{4.24}$$

where f is a nonlinear function, d_x, d_y are the lengths of the input and output history involved in computing the current output.

It is possible to implement this model by neural networks; the nonlinear function f can be approximated by an MLP, and tapped delay lines of lengths d_x, d_y can be used to buffer the input and output respectively. If implemented this way, this network combines Jordan networks with delay networks. It uses a first tapped delay line at the input layer and feedback connections from the output to the input, this feedback is connected to the input layer by recurrent connections with adaptive weights, the past output activations are buffered by a second tapped delay line that makes part of the input layer. Although a delay network, the feedback connections from the output to the input layer make these networks recurrent ones. NARX nonlinear systems implemented by a neural network give *NARX recurrent neural networks*. Figure 4.7 shows a possible schematic of such networks.

NARX networks with their nonlinear function f are able to simulate a linear discrete system that extends equation (4.9) to account for the outputs:

$$F(x(t)) = y(t) = \sum_{i=1}^{d_x} \alpha_i x(t-i) + \sum_{j=1}^{d_y} \beta_i y(t-j) \tag{4.25}$$

where α_i, β_i are constant coefficients. In time series modeling, these models are referred to as autoregressive moving average model (ARMA), and NARX networks are capable of computing such linear models as a special case. [Dorffner 1996] presents a variation of the NARX architecture that works as an ARMA model. The models that nonlinear NARX networks compute are sometimes referred to as NARMA models.

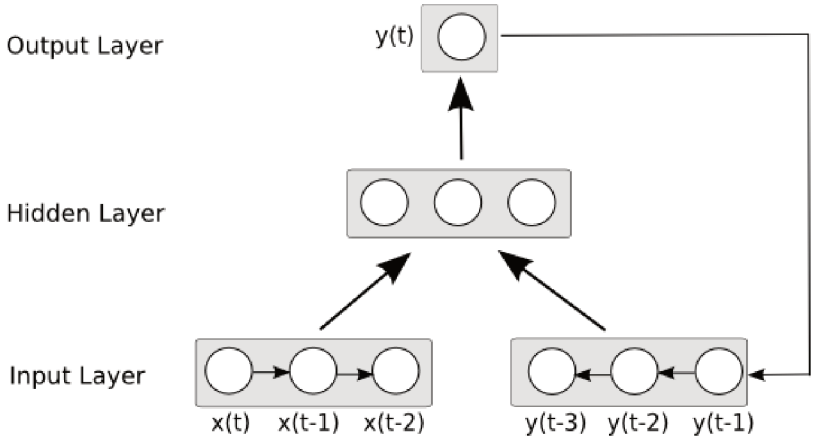


FIGURE 4.7: NARX network: the input layer contains two delay lines, one to buffer input elements, and the other to buffer past output activations. The network computes the output as a an MLP, and the feedforward weights besides to the weight of the recurrent connections are to be adapted by the learning algorithm.

4.5.4 Gradient descent in recurrent neural networks is hard

The already mentioned *backpropagation through time* (BPTT)[Jaeger 2002b] is an extension of the standard backpropagation algorithm and the most popular training algorithm for recurrent networks [Kolen 2001]. Standard backpropagation presumes no cycles in the network, but when recurrent connections with adaptive weights should be considered, it can't be directly applied. BPTT [Rumelhart 1985] algorithm is based on unfolding the RNN in time for a limited number k of timesteps. For each of the k timesteps, a copy of the RNN is created, each copy having the same weights than the original RNN. Recurrent connections in copies are then redirected between subsequent copies, this results in a many-layered feedforward network, then, it can be trained by standard backpropagation, with the main difference that the input to the unfolded network consists of k successive temporal inputs. After training, the appropriate weight change is the sum of all the weights changes for all connections in the unfolded network that share the same weight. Let us stress here on the idea that standard backpropagation and BPTT rely both on fixing the network weights for a specific number of time steps, this means that they use an accumulated error in the gradient descent. Thus, learning does not occur at each time step, but at the end of time cycles that could be the length of the processed input sequence.

Generally, RNNs learning by gradient descent methods (including BPTT) is hard [Sutskever 2010] because when the computation of the current output depends on inputs far in the past, the problem of long-term dependency appears. The reason is that the current timestep is separated from far past timesteps by a large number of nonlinearities [Bengio 1994] causing the error gradient information to vanish as it flows back through time (equivalent to flowing back in RNN temporal copies). This gradient information vanishes exponentially with the time lag of the input to be remembered [Hochreiter 2001]. In practice most RNNs trained by gradient descent methods are no more useful than delay networks, and the main gain remains in

the computation time.

However, NARX recurrent networks are known to be better than other RNNs concerning the problem of vanishing gradient, because unfolding the network in time makes the output delays appear as "jump ahead" connections. For example unfolding the network in figure 4.7 will result in connections between each copy of the network with all the $d_y = 3$ preceding copies in time (this doesn't hold true for the last three copies $k-2, k-1, k$ as there are no unfolded copies left). These connections offer a shorter path for propagating gradient information back through network copies corresponding to previous timesteps, thus avoiding network nonlinearities and reducing the sensitivity of the network for long-term dependencies. One can roughly conclude that NARX are d_y times more powerful than RNNs with no output delays. But in fact, this is not perfectly correct, as the benefit of output connections is related to the portion of the gradient that flows in the jump ahead connections that should be sufficiently large relative to the portion that flows in the network, this relation should maintain in order to have NARX network able to deal better than other RNNs with long-term dependencies. In practice, BPTT is not that easy to use and requires an important effort to choose the initial weights. Indeed, using BPTT is not straightforward, because unlike backpropagation which deals with networks implementing functions, BPTT deals with networks approximating dynamical systems, thus they are not guaranteed to reach a local minimum of the error while this is guaranteed in backpropagation. Networks with internal dynamics can also encounter bifurcations when the initial weights induce a different behavior than what the task requires. Near bifurcations, the gradient information may become useless or the error may dramatically increase and the learning fails to converge [Jaeger 2002b].

NARX networks were used in the prediction of chaotic time series [Diaconescu 2008]. They are also proved to be as computationally powerful as fully connected networks thus at least as powerful as Turing machines [Horne 1995].

Elman, Jordan, and NARX networks are based on MLP feedforward networks that are trained by backpropagation, BPTT or RTRL, all are gradient descent methods. For the same reasons of learning and architecture as discussed in section 4.3, **Elman, Jordan, and NARX networks are not cellular models**.

Another implementation of equation (4.24) that uses self-organizing maps was given in [Barreto 2001]. Learning in such neural implementation of NARX is unsupervised, thus avoids learning algorithms based on gradient descent. However, NARX implementation that relies on self-organizing maps does not fit the requirements of cellular computing paradigm, that will be discussed in the coming chapter.

4.5.5 Random networks maintaining states: Reservoir Computing

Although RNNs are suitable for nonlinear modeling and processing of temporal sequences, RNNs trained by gradient descent are limited, mainly because they suffer from the vanishing gradient problem. Besides, the gradual change of network parameters during training may drive the network dynamics through bifurcations so that the gradient information degenerates and the convergence to error minima can't be guaranteed [Doya 1993].

A recent neural paradigm for nonlinear modeling appeared to overcome these problems and

was independently and concurrently developed by different researchers. It is based on the idea first pointed out by Peter F. Dominey, that it is possible to build recurrent networks able to carry out temporal processing, without adaptation in the recurrent part of the network that implements the short-term memory, and that a simple adaptation is required in the predictor part [Dominey 2000]. This paradigm consists of networks that also make a clear separation between the short-term memory and the predictor parts. The short term memory is the recurrent part and is the one that is called *the reservoir*. The reservoir is a recurrent network with a large number of units (neurons) (50 to 1000) with *random* connectivity that remains unchanged during training. This random network expands the input vector of the network to a higher dimensional vector which is the state of the network. This expansion often implies that the input vector $X(t)$ forms a part of the state vector $S(t)$ maintained in the reservoir. So when a new input arrives the reservoir is excited by the inputs and maintains in its state a nonlinear transformation of the input history.

The predictor part is a feedforward network that computes a linear combination of the reservoir units activities, this part is called the *readout* in the terminology of reservoir computing paradigm. The state should maintain a representation rich enough so that the output could be linearly computed from the state, and the main challenge in these networks is to design a reservoir that makes this possible.

Reservoir units could be thought of as computing some "basic functions" of both the input and its past activity. The readout learns to combine these basic functions in the best way to compute the target output. Thus, unlike other RNN models, the reservoir doesn't need to be trained, the only part to train is the predictor part of the network; the readout. Hence, state representation is realized in a different way, that doesn't require learning, and the problems of gradient vanishing or degeneration are avoided.

Two main classes of reservoir networks were independently developed, they are encapsulated under the name *reservoir computing* paradigm. The first class relies on spiking neurons and dynamic synaptic connections, their design is biologically motivated. Using spiking neurons with their ability to represent real values makes these networks computationally powerful, but hard to train and use, and require more computational power. When inputs are presented to the network, the internal state change resembles to the ripples on the surface of a pool of water, hence the name of this class of networks *liquid state machines* (LSMs) [Maass 2002]. Reservoirs of this type are proved to be computationally universal and can model by their bounded time and value resolution any continuous time, continuous-valued system [Maass 2006]. However, we choose not to discuss this class of reservoirs because by its low-level modeling it is mainly suitable to simulate the brain and because it is harder to control for the same reason.

The second class is called *echo state networks* (ESNs) [Lukoševičius 2009]. The neural structure that holds the state in ESNs, i.e. the reservoir, is different from those in other recurrent networks, a reservoir contains large number of mean firing rate or leaky integrator neurons (units) with sparse and random interconnectivity, thus the dimension of the state vector is considerably larger than this of the input vector, i.e. $N_S \gg N_X$. Sparseness is attained by less than 1% of cells interconnectivity. This property results in loosely coupled subsystems within the reservoir all along with a dynamics rich enough that allows the readout to compute the output linearly.

Although the design of the ESNs reservoirs is different from feedforward networks, the state of

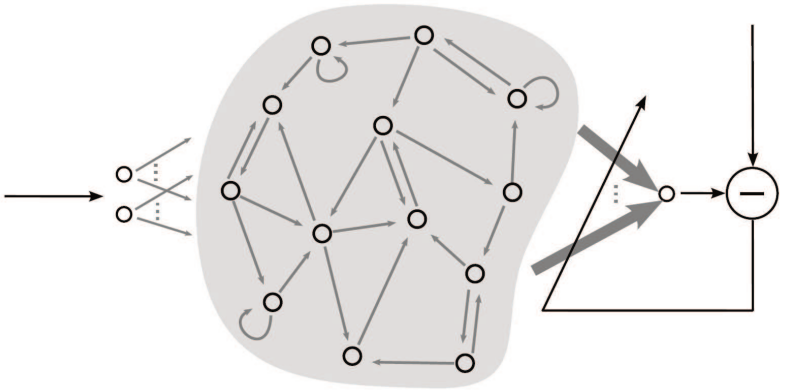


FIGURE 4.8: Reservoir. Extracted from [Lukoševičius 2009]

an ESN reservoir is formally given by a similar equation:

$$S(t) = f(A.S(t-1) + B.X(t) + C.Y(t-1)) \tag{4.26}$$

where f is usually the sigmoid function applied piece-wise. A, B, C are weight matrices similar to those seen in Elman networks. A difference from the Elman networks is the possibility to involve the feedback from the output Y in the computation of the current state. However, in many applications it is not used, thus, equation (4.26) turns out to be formally similar to (4.22) for Elman networks.

ESNs should have the *echo state* property, that requires that the effect of the current input $X(t)$ and the current state $S(t)$ on future state $S(t+k)$ should behave like an echo, i.e. it should vanish gradually when $k \to \infty$. This is why the states of the reservoir are termed "echos", hence the name of echo state machines. For this property to be attained, the spectral radius of the weight matrix $\rho(A)$ should be inferior to 1, irrespective of the input. For such value the reservoir is on the edge of chaos, thus exhibits interesting memory and computation capabilities, all while guaranteeing that the effect of the input dies out with time. For a value of $\rho(A)$ slightly larger than 1, the reservoir starts to exhibit oscillations, while for larger values it exhibits chaotic behavior. Generally, the random creation of reservoirs is not always the best choice, instead, changing some characteristics at the design could result in reservoirs that are better adapted for specific tasks [Lukoševičius 2009]. Literature talks about some "goodness measures", some of them aim to make possible the separation of inputs by a linear readout, this is called the *linear separation property*. Some other goodness measures address the computational power and memory capabilities of the reservoir like the echo state property discussed before, and the short-term memory measure established in [Jaeger 2002a], or the entropy of $S(t)$ that measures the local information transmission in the reservoir [Jaeger 2005].

After the state is computed, the readout computes the output of the network. The readout part is typically a single-layer feedforward network, it computes the output of the reservoir following equation (4.18) where g is usually the sigmoid function. Let's notice that both the reservoir and the

readout parts are, in principle, allowed to compute nonlinear functions of their inputs. However, in many practical applications, it is often sufficient to use a simple linear readout of the ESNs in which g is a linear function, given D is a weight matrix:

$$Y(t) = D.S(t) \tag{4.27}$$

Training the readout is mostly non-temporal supervised task, often carried out by linear regression when the readout is a single layer perceptron like in [Jaeger 2001]. MLPs can also be used when single layer perceptron gives rough mappings between X and Y. Because learning in ESNs occurs only in the readout, learning is extremely fast compared to other temporal networks [Sutskever 2010].

However, ESNs are also learned by unsupervised learning, here, the goal is not to associate inputs with target outputs, but to set out the reservoir in such a way that it fulfills some goodness measure like the memory power, by setting it to have the echo state property, and to have a rich signal pool that allows for the linear separation of whatever inputs by the readout. To obtain this property, the reservoir is either properly set out the design phase, or alternatively by learning the reservoir part. In this later case, the reservoir part is adaptive, and learning occurs either by local methods in the functional sense of locality like the Hebbian learning [Hebb 1949], or by global methods that consider the whole reservoir and thus require matrix calculations. A somehow similar model to unsupervised reservoirs that seeks to reach the edge of chaos by some plasticity techniques is the one called SORN [Lazar 2009], it is discussed in the coming chapter.

What remains a special advantage of reservoir computing and distinguishes it from other RNNs approaches is that the two parts of the network, the reservoir and the readout, can be created and trained separately, so that for a specific task, the network can be built by selecting reservoir and predictor modules with known suitable characteristics for this type of the task.

Because reservoir computing networks do not suffer from vanishing gradient problems as there is no gradient descent based learning, these networks have outperformed previous recurrent methods in many domains like classification and pattern generation, forecasting of time series, and nonlinear system identification [Lukoševičius 2009]. Besides, they perform very well in learning long-term dependencies [Sutskever 2010].

However, reservoirs use a large number of neural units in order to create a sufficiently rich signal pool from which the readout learns to select the appropriate ones through linear combinations. This means that there will be many unused signals that have already been computed by many units. Thus, reservoirs can be said to be computationally inefficient.

Reservoir networks are not cellular models because, first, although reservoir units activity does not depend on units that they are not connected to, the next state of the reservoir is computed by matrix computations on the population level, i.e. using a centralized processor, thus they are not computed in a decentralized way. Although, it is possible to overcome this condition like in adaptive reservoirs learned by Hebbian learning. However, it is the readout part that can not be cellular, even if it is a one layer perceptron (a feedforward network), because of the global nature of matrix calculations necessary for linear regression, that requires a centralized processor which holds true for all variations of ESNs.

Thus, **the reservoir part can be cellular, but the reservoir network with readout is not cellular**.

4.5.6 Hopfield networks

Units update in the presented models so far is synchronous; all model units are updated at each timestep. They are hierarchical models arranged in layers, even reservoir networks contain two hierarchical levels, the reservoir and the readout, that latter could be hierarchical as well.

Hopfield networks [Hopfield 1982] represent a different model in which units are not hierarchically organized and they are updated asynchronously. This model is considered as a milestone in neural networks domain, because of its interesting properties, especially biological plausibility, also, because the Hopfield model drew the attention of the scientific community on neural networks as valid models for modeling and solving problems. These networks are later proven to be Turing equivalent [Sima 2003, Sima 1999].

A Hopfield network consists of a fully connected neural network with N units; each unit is connected to all other units but not to itself, this implies that the connectivity between every two units i, j are bidirectional. The weights of connection between units i and j are symmetric, i.e. $w_{ij} = w_{ji}$, so the connectivity matrix is symmetric with zeros on the diagonal. Hence, these networks are recurrent as the bidirectional connections make cycles and deliver a form of feedback to the units.

There exist binary and continuous versions of Hopfield networks, the latter has continuous valued units activations. For simplicity, the binary one is presented. The units in the binary version of Hopfield networks have their activations in $\{-1, 1\}$, or $\{0, 1\}$. In what follows units activations $\{-1, 1\}$ are adopted. The network dynamics in the absence of inputs is determined by the way the units activity is computed, it is sometimes called the *Hopfield dynamics*, for a unit i:

$$a_i(t) = sgn(\sum_i w_{ij} a_j(t-1) + \theta_i) \tag{4.28}$$

$$\text{where } sgn(x) = \begin{cases} 1 & \text{if } x \geqslant 1 \\ -1 & \text{if } x < 0 \end{cases} \tag{4.29}$$

and θ_i is an activation threshold of the unit i, it is considered to be 0 in what follows.

The inputs to the network are vector patterns of the form $X(t) = (x_1(t), x_2(t), \cdots, x_N(t))$. The length of each input pattern is N, the same as the network units, so that the element i in the vector is presented to the unit i in the network. When an input pattern $X(t)$ at time t is considered, then the activity of unit i is computed as follows:

$$a_i(t) = sgn(\sum_i w_{ij} a_j(t-1) + x_i(t)) \tag{4.30}$$

The asynchronous update in Hopfield networks can be carried out in two ways, either by the sequential update of single units according to equation (4.28) in a random order, or by letting network units update themselves with certain probability.

Hopfield networks are mainly used to work as *associative memories* so that they learn static patterns and reproduce them upon the presentation of inputs. The basic idea of associative memories can be formulated as follows:
"Store a set of p patterns $\{\xi^1, \xi^2, \cdots, \xi^p\}$ in such a way that when presented with a new pattern X, the network responds by producing whichever one of the stored patterns most closely resembles to X."([Polk 2002]).

Hopfield networks as associative memories store static patterns. Each stored pattern ξ^k with $1 \leq k \leq p$ is a vector with the same number of elements N as the network nodes. That is, when recalled, elements of a stored pattern ξ^k are distributed over all the network units: $\xi^k = (\xi_1^k, \xi_2^k, \cdots, \xi_N^k)$, where ξ_l^k represents the bit l of the pattern ξ^k.

The activity vector of the N units in a Hopfield network defines the state of the dynamical system that the network constitutes. In the binary case, this vector takes its values in a finite set. The temporal change of the activity vector determines the dynamics of the network.

Hopfield networks are dynamical systems with discrete states, Hopfield showed that the network can learn to store patterns as stable states, in such a way that patterns become *attractors* in the state space (RNNs terminology) or *phase space* (dynamical systems terminology) of the dynamical system that the network represents. This means that when an input $X = \xi^k + \Delta$ is presented to the network, with Δ a sufficiently small deviation from the stored pattern ξ^k, then the dynamics of the network spontaneously moves to ξ^k after several timesteps necessary for the relaxation of the network dynamics. The states in which the dynamics passes from X to ξ^k are called the *trajectory* of the system. Possible trajectories that lead to ξ^k in the state space define a region called *basin of attraction* illustrated in 4.9. This capability of the network to converge to an attractor although the deviation in the input pattern from the stored pattern means that it is able to recall stored patterns even if the input pattern contains only a part of the stored pattern, or even if the input pattern is a distorted version of the stored pattern, due to noise for example. Thus, Hopfield networks as associative memories are robust to noise. Recalling a stored pattern starting from a distorted or non-complete one is often called *pattern completion*. The stored patterns are recalled by a content related to them, i.e. to the memory content, this is why associative memories are also called *content-addressable memories*.

It is convenient here to briefly remind with the attractor types of dynamical systems. When the state dwells at a single state for a considerable period of time as in Hopfield networks, then it is called *fixed point* attractor. When the systems exhibits a cyclic behavior in which the trajectory passes by the same states then one talks about *periodic attractors* (or *limit cycles*). However, when the state is continuous (like Hopfield networks with continuous activations), then although the cyclic behavior of state, it never returns to the same exact point, thus this type of attractors is called *quasi-periodic attractor*. Finally, if the trajectory remains within a bounded region in the state space but it is unpredictable, then this region is called *chaotic attractor*.

Hopfield has another contribution in neural networks, which is the introduction of the concept of energy function, for his network he defined an energy function $E(t)$ for the network state at time t as follows:

$$E(t) = -1/2 \sum_{i,j} w_{ij} u_i(t) u_j(t) \tag{4.31}$$

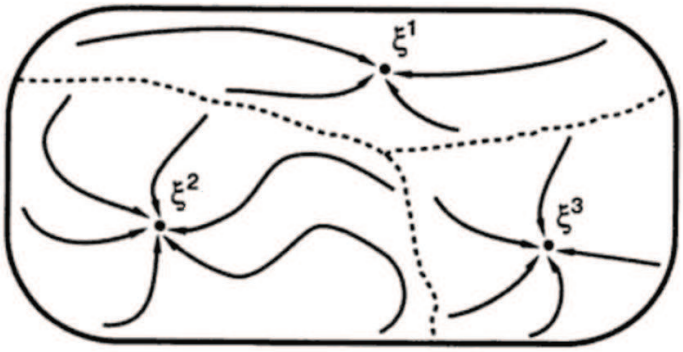


FIGURE 4.9: Attractors and basins of attraction of associative memory: the delimited regions represent the basins of attraction for each attractor, when input patterns fall within any of these regions the network dynamics moves the state toward the attractor. After [Polk 2002]

He proved that in an asynchronous update regime, E always decreases whatever was the initial state, and that the network dynamics converges toward local minima of the energy function, that correspond to the stored patterns in the associative memory. It should be also mentioned that the synchronous update of Hopfield networks is possible, but it is stated in [Polk 2002, Cheung 1987] that it may result in instability of the network dynamics. [Cheung 1987] gives a more detailed study of stability conditions, asynchronous update regime is guaranteed to converge, but it may require involving units thresholds θ_i, while stability of synchronous update regime requires more strict conditions. The same work further considers networks with additional self-recurrent connections, hence another variation of Hopfield networks.

Although these networks consist of N binary units, one shouldn't expect that they are able to store 2^N different patterns, nor even N patterns, in fact, the capacity of the network is limited to about $p = 0.14N$ [Hertz 1991]. The reason is that the network state space (consisting of vectors) is not entirely allocated for storage, instead, only a few states can be memories, those are the attractors, while the rest of states form the basins of attraction.

The update rule 4.28 of Hopfield network is deterministic, but a stochastic counterpart also exits, it is called Boltzmann machines [Ackley 1985]. In these networks, units are updated following a probability dependent on the energy of the network at each timestep. Being stochastic, the main usage of these networks is to build an internal representation that allows the network to generate patterns that have the same statistical distribution as the input patterns.

Learning in Hopfield networks is unsupervised. Learning the network to store patterns $\{\xi^1, \xi^2, \cdots, \xi^p\}$ means that if the network is in some state ξ^k it remains in it, moreover, it should converge to the attractor of ξ^k whenever it is in its basin of attraction. In Hopfield network, learning stable states for the to-be-stored patterns leads to weights w_{ij} proportional to the elements of these patterns that correspond to units i and j, which is an interesting property. A corollary is that Hopfield networks can be learned with Hebb rules seen in 3.5.2. A learning rule for storing p patterns

is:

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{p} \xi_i^k \xi_j^k \tag{4.32}$$

Clearly, Hebb learning rule is local in the functional sense as the synaptic weight w_{ij} takes into account only the activations of units i and j at its sides. However, this rule needs knowing all patterns in advance, and doesn't work incrementally, thus it fits static pattern learning and not temporal patterns presented as temporal sequences. For the latter case, another functionally local rule exists, it takes into account a local field [Storkey 1999] around each unit; a network trained by Storkey rule allows for incremental learning, moreover, they offer more capacity than the same network trained by Hebb rule. Hopfield networks are used to learn static patterns, but they are also used to learn temporal sequences of patterns. For example, [Gas 1993] extends Hopfield networks to achieve unsupervised learning of temporal sequences. [Miyoshi 2004] presents an associative memory network with delayed synapses, able to encode a temporal sequence. This ability was extended in [Maurer 2005] to encode several temporal sequences. [Chang 2004] uses a two layer Hopfield network to process 3D magnetic resonance imaging (MRI) spatiotemporal data.

Interestingly, Hopfield networks are decentralized models. From the one hand, when trained with functionally local rules like Hebb or Storkey, synaptic weighs depend only on the activations of units at their sides, on the other hand, the activation of a network unit during learning or functioning depends only on the units to which it is connected, in addition to their synaptic weights as shows equation (4.28). However, it is the fact that each unit is connected to all other network units that violates the condition of topographic locality required in cellular models as defined in 3.6. The mitigation of the strict conditions put by Sipper, especially the condition of topographic locality, makes Hopfield networks cellular models.

Indeed, although their full connectivity, Hopfield networks are straightforward to implement, they scale easily, they allow for the asynchronous update, and the major drawback of their full connectivity is the larger computation time. As a result, depending on one's perspective, **Hopfield networks might or might not be seen as cellular morels**.

4.5.7 Long Short-term Memory Networks

These networks were developed to overcome the problem of vanishing gradient that happens usually when training networks by BPTT or real-time recurrent learning (RTRL) [Williams 1988] which is an online gradient descent algorithm that, contrasted to standard backpropagation and BPTT, applies the weight updates at each time step. The *long short-term memory* networks (LSTM) [Polk 2002] are "engineered" networks with explicit memory cells that intend to extend the time of information storage in the short-term memory. The memory cell is called a *block*, it consists of several units, and it is intended to replace hidden units in state models (RNNs).

The idea of the block in LSTM networks is to "isolate" the state unit from the network events that aren't related to the state information, and allowing it to handle only salient information. The block has input and output gates that learn to open and close at appropriate times, either to allow block inputs to affect the stored state, or to let the state activation affect other units in the network. The block may contain more than one state unit, but they should all be controlled by the same gates.

A block dented $c_j$with one memory unit is illustrated in figure 4.10.

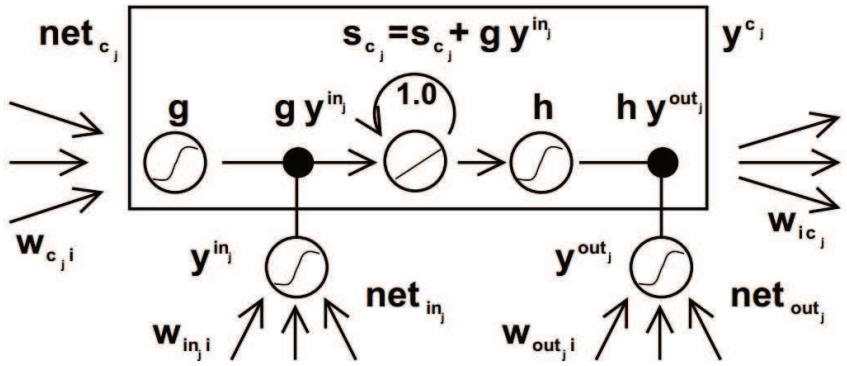


FIGURE 4.10: A block in LSTM networks: it contains a state unit with a recurrent connection. The access to the state unit activity $s_{c_j}(t)$ is controlled by the outputs y^{in_j},y^{out_j} of an input and output gate units respectively. After the network has learned, gate units learn when to allow access and when to prohibit it. For more details, see the text. Extracted from [Polk 2002]

The state unit inside the block is a simple linear unit with a self-recurrent connection with weight 1.0. This unit maintains a state activation $s_{c_j}(t)$. The access of input information to the block is controlled by an *input gate unit*. Also, the reading from the state unit and the backpropagation of the error to the memory cell are controlled by an *output gate unit*. The input gate and the output gate units compute their outputs y^{in_j}, y^{out_j} on the basis of their inputs, denoted here net_{in_j}, net_{out_j} respectively. These outputs are weighted sums of the gate units inputs passed to an activation function, typically a sigmoid.

The input to the memory unit is the weighted sum of the block inputs which is denoted net_{c_j}, it is passed to an activation function g, typically a sigmoid. The input to the memory unit is controlled by y^{in_j} such that:

$$s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j} g(net_{c_j}), \text{ and } s_{c_j}(0) = 0 \tag{4.33}$$

When the output y^{in_j} of the input gate is close to 0 then the block input net_{c_j} doesn't reach the state unit and it maintains its past state.

The output of the block is the output of the state unit, it is controlled by y^{out_j}, such that:

$$y^{c_j}(t) = y^{out_j} h(s_{c_j}(t)) \tag{4.34}$$

where h is an activation function, typically a sigmoid. When the output y^{out_j} of the output gate is close to 0 then the state value doesn't affect other units connected to c_j, whether they are the output layer units or other blocks or gates.

Inputs to the block, come from the input layer, gates, and from other blocks in the same layer via recurrent connections. Inputs to the gates come from all non output units, i.e. from input layer units, from their blocks or their neighbor blocks, or even from their own outputs. Another variation of LSTM blocks contains an additional *forget gate* which resets the value in the state unit.

LSTM networks are trained by a combination of BPTT and RTRL. During learning, the gates

learn to control the access to their blocks. Besides to their explained functioning, output gates also control the backpropagation of error gradient, if an output gate allows the gradient to pass back from an output layer unit to the state unit in the block, it is "trapped" in the state unit and flows freely through the self-recurrent connection probably indefinitely, this is why LSTM networks doesn't have the problem of vanishing gradient. Similarly, the gradient information don't affect the weights of input layer connections unless the input gate allows the error to pass back to them.

LSTM networks can solve problems where important information are separated by thousands of timesteps while state RNNs like Elman and Jordan may fail to learn in the presence of 10 timesteps lag. An example is using LSTM RNNs in robotic control (knotting in heart surgery) that learn from training sequences of more than one thousand elements [Mayer 2006]. These networks resemble to Elman networks, when they are trained by gradient descent, therefore they are not cellular models. LSTM networks can also be trained by unsupervised learning in order to maximize some objective function. An example is the work of [Klapper-Rybicka 2001] in which groups of temporal sequences were discriminated by unsupervised learning of an LSTM network according to features emphasized by the binary information gain optimization objective function [Schraudolph 1993], and in another experience by the non-parametric entropy optimization [Viola 1996]. The outputs of the used logistic activation functions in all units were interpreted as binary values beforehand. In this work, the authors used one block network with two state units with the first objective function, and a one-block network with one state unit for the second. Clearly, it is not possible to discuss cellular properties in a one cell model.

A larger LSTM network trained by supervised gradient descent methods is obviously not cellular. If the network is trained by unsupervised learning such as the presented ones, global computation of objective functions require centralized processors, thus **LSTMs are not cellular models**.

4.5.8 Second order recurrent networks

These networks have the structure of MLPs, they implement the short-term memory using a single-layer but with second-order units to compute the state based on the previous context and the current input. Connections in such networks are called *second order connections*. They connect three nodes instead of two, so that the activation of a node is modulated by the activation of the other, and transmitted to the third. The latter then computes a weighted sum of all the products and passes the result to an activation function that computes the state activations. An example is the network presented by [Giles 1991]. In these networks the activation of the state unit is computed as follows:

$$a_i^S(t) = f(\sum_{j,k} w_{ijk} a_j^C(t-1) x_k(t-1)) \tag{4.35}$$

where f is a sigmoid activation function, a^S, a^C, x are the state, context, and input activations respectively.

Understanding the operation of second order connections, reveals that these networks allow the inputs to gate the previous state information. This makes them well suited for modeling finite state automata [Goudreau 1994]. These networks are also trained by a gradient descent method [Omlin 1992], hence, **they are not cellular models**.

4.5.9 Continuous time recurrent neural networks

These networks inspire from neurons in the brain where neurons work in continuous time. Unlike all the previously presented discrete time networks, units in such networks are continuous leaky integrator neurons as given by equation (3.3). These neurons partially decay and partially recycle their activations, and serve by themselves as a short-term memory. The activity of leaky integrator unit j in continuous time recurrent neural networks (CTRNNs) [Tani 2008] is given by:

$$\tau \frac{\mathrm{d}y_j(t)}{\mathrm{d}t} = -y_j(t) + \sum_j w_{ij} f(y_j(t) - \theta_j) + \sum_k w_{ik} x_i(t) \tag{4.36}$$

Here, $y(t)$ is the state of the unit, τ is its time constant, f is a sigmoid, θ is a bias term proper to each unit.

Because of their complex dynamics, CTRNNs are used in evolutionary robotics where the dynamics can be exploited by the robot. Weights are determined by evolutionary algorithms as in [Campo 2010], hence, obviously **CTRNNs not cellular models**.

4.5.10 Other networks

There exist many other recurrent neural models and learning algorithms that are not explained here, we just mentioned the most important ones that cover the maximum underlying concepts of implementation. Other networks include recurrent cascade correlation networks (RCC) that learn to map an input sequence to an output one and are used in learning finite state grammar [Fahlman 1990], and recurrent neural trees that extend recurrent networks to acyclic graphs and used in the processing of structured data [Frasconi 1998]. These two models use constructive algorithms that build up the network architecture based on the gradient information during learning. Evolutionary algorithms in their turn can be viewed as constructive algorithms because more complex networks can evolve within the population of networks.

Besides to the mentioned algorithms so far, other learning algorithms include extended Kalman filter (EKF) [Williams 1992a] which can also be used for online training and include changing the units activations in addition to the weights, temporal difference (TD) [Sutton 1988] used for interrelated predictions [Sutton 2005]. Both algorithms are gradient descent ones. Stochastic algorithms are also used to train recurrent networks like expectation maximization (EM) algorithm that decomposes the learning of complicated recurrent networks to the training of several feedforward networks as in [Ma 1998].

Most of the presented neural network models so far are typically trained in a supervised way, with the exception of Hopfield network. However, some a priori supervised models like ESN and LSTM networks can also be trained in unsupervised way as mentioned before in 4.5.3 and 4.5.5.

Generally, there are two approaches for building unsupervised networks for temporal sequence processing. The first is to start from supervised temporal networks and change the objective function. In supervised learning, this function is mostly the mean square error between the computed and the target outputs. The new objective function of the network should be one which is suitable for unsupervised learning, i.e. it should be related to the processed data itself such as entropy functions as seen for LSTM and ESNs, or some distance function between the probability distribu-

tion of the network output and some target probability distribution using some information divergence measure like Kullback-Leibler divergence as in [Sum 2007], or Jensen-Shannon divergence as in [Mishtal 2012].

The second approach is to start from an unsupervised network and provide it with short-term memory so that the network gains dynamic properties that allow to integrate time. The short-term memory mechanism can be implemented by delay lines, or feedback loops, or even by leaky integrator neurons as in exponential trace memory or other memory kernels. Normally, the activation and learning rules of the network should be revised in order to take into account the temporal aspect of the input sequence [Barreto 2003]. Most temporal unsupervised networks respecting this second approach are built up starting from self-organization maps (SOMs) which is an important unsupervised model in the field of neural networks. SOMs and their temporal extensions are the subject of the next chapter.

It has been shown that the presented neural models trained with supervised learning can't be cellular models. Unsupervised learning methods may or may not be cellular models depending on the used methods for computation and update. On the one hand, when unsupervised learning is carried out by global methods of computing on the population level as in reservoir computing, it requires centralized processing. In this case the network can't be a cellular model. On the other hand, unsupervised local methods can be cellular, as weights are computed locally, under the condition that the network doesn't violate other cellular computing requirements such as the topological locality that Hopfield networks violate. When weights are possible to update locally, the asynchronous update regime of the network becomes possible. Indeed, having neural models to be cellular is closely tied to the possibility of the asynchronous update regime, which is in turn closely tied to local computation.

4.6 Neural networks and models of computations

The formal models of computation presented in 3.2 fit the symbolic computation type, while the fine grain computation in neural networks is numeric by nature. However, neural networks with their parallel paradigm of computation, can embed such abstract models. The relation between computational models and RNNs was initiated since the very first RNN of Elman proposed in 1990. Indeed, there exist a correspondence between formal languages, their accepting automata and neural networks [Crutchfield 1988]. This correspondence was studied in literature motivated by the idea that using neural networks in specific applications doesn't offer the convenient framework to study their computational capabilities. Instead, the formal models of computation offer that convenient framework that allows for better understanding of the power of neural networks, and then selecting the suitable one for a specific task.

Dynamical networks, both recurrent and delay ones can be used to implement models of computation. However, delay networks have poor dynamical properties that limit their potential in simulating formal models. Example networks are feedforward networks provided with a delay line at the input layer, or with delay line associated with each unit. Such networks can simulate DFA (deterministic finite automaton) whose state depends on a limited history only, determined by the length of delay lines. The formal models that this type of networks can simulate are called *definite*

memory machines (DMMs).

Recurrent networks are more powerful in simulating formal models because they are dynamical systems with internal state that theoretically allow for infinite memory, thus one can assume that they are non-finite memory machines.

Recurrent networks deal with temporal sequences as inputs, but it is also the case of models of computation that deal with sequences of actions as their inputs. In the models of computation, the next state is determined by the past state and the current input to the model, and some of them possibly issues an output like in transducer FSMs (finite state machine) like Mealy and Moore machines presented in 3.2. In their turn, RNNs compute their next state on the basis of their past state in addition to the currently available input in each input presentation cycle. The output of the network is computed starting from the current state and possibly the current input. This analogy motivates the relation between RNNs and the models of computation and makes it possible to view RNNs as state machines. It also suggests that it is possible to use RNNs to implement the models of computation if they can exhibit the sufficient complexity as the target model of computation. Many works (like [Haykin 1998b, Kremer 1996, Tsoi 1997]) proposed to view RNNs as *neural state machines* (NSMs), parallel to Mealy and Moore machines, NSMs are defined as follows:

A neural state machine is a 6-tuple $NSM = (S, X, Y, s_0, f, g)$, where $S = [R_0, R_1]^{n_S}$ is the state space of the NSM with R_0 and R_1 are the values defining the range of the output of the activation function. n_S is the number of state units. $X = \mathbb{R}^{n_X}$ is the set of possible input vectors and n_X is the number of input units. $Y = [R_0, R_1]^{n_Y}$ is the set of outputs of the NSM with n_Y is the number of output units. $f : S \times X \mapsto S$ is the next state function that computes the state $s(t)$ from the past state $s(t-1)$ and the current input $x(t)$. s_0 is the initial state of the NSM, i.e $s(0)$. g is the output function, it differs between Mealy and Moore machines. In NSMs simulating Mealy machines $g : S \times X \mapsto Y$, the output $y(t)$ is computed from the previous state $s(t-1)$ and the current input $x(t)$. This fits equation (4.17) and figure 4.4(a). In NSMs simulating Moore machines $g : S \mapsto Y$ and the output $y(t)$ is computed on the basis of the current state $s(t)$ only. This fits equation (4.18) and figure 4.4(b).

This relation between RNNs and formal models is bidirectional: Neural networks can be used to implement models of computation, and conversely, it is possible to determine the model of computation that mimics the RNNs to a satisfactory degree, this is referred to as *rule extraction* (RE) [Jacobsson 2005].

Theoretically, RNNs are powerful computational tools that are Turing equivalent [Siegelmann 1995b], thus can compute any function a digital computer can do. Although, even if RNNs have the capability to simulate some formal model, getting a RNN to perform a specified computation is very difficult because it is hard to find the suitable instantiations of the network to do the computation [Bengio 1992]. These instantiations include, the initial state, the weights, and learning rates.

The capability of the RNN to simulate a formal model depends on several factors, it depends first of all on the architecture that defines the dynamics, and on the nature of the activation function of the network units whether it is a hard limiter (a threshold) that computes binary values or if it is a continuous function that allows for a richer dynamics. It also depends on the number of units in the RNN, especially those dealing with finite precision real numbers in weights and activa-

tions. The correspondence between RNNs and formal models requires finding a mapping between a subset of the network states and transitions, and those of the formal models. Hence, the practical implementation of models of computations using RNNs is not that straightforward.

An example on the effect of the architecture on the computational power of RNNs and their ability to implement some formal model is the work of [Lin 1996]. In this work, Elman and NARX networks are used to simulate a finite state acceptor automata with 5 states, and with various length sequences of booleans as inputs. The two networks used have the same number of units and delay elements and weights. Varying the input sequence length from 10 to 30 gives that NARX largely outperforms Elman when it come to long-term dependencies in the inputs. Here, varying the network architecture for the same number of units and connections changes the computational capability of the network and the input history that the FSM can consider.

As has been mentioned, the network capability to simulate a formal model is also related to the nature of the activation function of the network units. If the activation function of the state units is a hard limiter like in McCulloch and Pitts neurons, then the resulting set of internal activations (state vectors) is finite. Such networks are candidates to exhibit the regular abilities associated with FSMs. It is sufficient to find a mapping between some states and transitions of the network, and those of the formal model. The bottom line, one can build a RNN that accepts the same language as a DFA $M = (Q, \Gamma, q_0, F, \delta)$ with n states and m inputs (defined in 3.2). A straightforward RNN architecture can be obtained by using nm hard limiter neurons of McCulloch and Pitts as the network units, so that each unit corresponds to one pair (state $q \in Q$, input $\gamma \in \Gamma$) of the DFA, and the weight of the connections between units i and j is set to $w_{ij} = 1$ if there is an input symbol γ_k such that $\delta(q_i, \gamma_k) = q_j$, otherwise, the weight $w_{ij} = 0$. Hence, at each timestep there exists only one unit that has its activity set to 1, it is the unit that corresponds to the current DFA state. The network has a single output unit in its output layer, each state neuron is connected to the output neuron with a connection weight $w_{oi} = 1$ if the state q_i is an accepting state of the DFA, and 0 otherwise. With hard limiter activation functions, Elman networks can simulate any DFA as proved in [Kremer 1995], also can do other fully recurrent networks. Same can be said about NARX recurrent networks as proved in [Siegelmann 1996]. Turing machines are also constructed by RNNs with hard limiter units [Siegelmann 1995b], in this work the RNN simulates a DFA and the network is provided with two external binary stacks.

RNNs with smooth activation functions (like sigmoids), are preferred on hard limiter ones because they allow the construction of gradient descent-based learning algorithms that need the activation function to have a derivation. These networks can simulate the same models as hard limiter activation functions if the networks weights are extremely large so the units activities saturate, so that they can be interpreted as binary. In this case, the range of network units activations is divided to several intervals: high, low and forbidden, networks are trained to have their units activations outside the forbidden interval. Thus, the state vector of these networks can also be seen as a finite set, so they can exhibit the same possibility of simulating FSMs.

Network with increasing and bounded continuous functions (true for sigmoids) and with small weights, can also exhibit a regular behavior if their internal activation vectors start to cluster in a finite set of clusters or regions in the state space S *and* if they are stable. Thus, in order to be able to simulate an FSM, the RNN must *split* its states into distinct regions that fit the FSM states. Pre-

senting a new input changes the network state from within one region to another. These transitions enable RNNs to recognize regular languages that FSMs recognize. For an RNN (S, X, Y, s_0, f, g) to be able to simulate a state machine $M = (Q, \Gamma, \Sigma, q_0, \delta, \lambda)$ (as defined in 3.2), it should fulfill several conditions related to the state representation and the network dynamics. The representation conditions, include that each state $q_i \in Q$ is assigned to one region $S_i \subseteq S$ in the RNN state space. These regions must be disjoint: $S_i \cap S_j = \emptyset$ if $q_i \neq q_j$. The initial state q_0 is assigned to the initial RNN state s_0. Concerning the interpretation of inputs, each possible input symbol $\gamma_i \in \Gamma$ should be assigned to a different input vector $x_i \in X$ of the RNN, or alternatively, a region $X_i \subseteq X$. Similarly, each possible output symbol $\sigma_k \in \Sigma$ is assigned to a non empty region $Y_k \subset Y$, these regions should be disjoint as well.

The dynamic of the RNN should fulfill these two conditions related to the correctness of the next state and output functions:

$$f(S_j) \subseteq S_i \; \forall q_j \in Q, \gamma_k \in \Gamma : \delta(q_j, \gamma_k) = q_i \tag{4.37}$$

$$g_k(S_j) \subseteq Y_m \; \forall q_j \in Q, \gamma_k \in \Gamma : \lambda(q_j, \gamma_k) = y_m \tag{4.38}$$

with $g_k(A) = \{g(s, x_k) : s \in A\}$.

RNNs with continuous activation functions are used to learn FSMs from input samples representing the grammar that recognizes the FSM, example works are [Gori 1998, Manolios 1994, Tino 1995]. Examples of simulating Mealy machines using a second-order RNN can be found in [Forcada 1994, Giles 1992] . Examples of simulating Mealy machines using Elman networks can be found in [Carrasco 1996, Blair 1996]. There exist other RNNs that split their states into a number of regions that fits the number of every possible pair of states and inputs (q_i, γ_k), one example is the RNN proposed in [Alquezar 1995]. In all these works the RNNs learned to recognize grammar strings of the same length of those used in training. Presenting longer input strings from the same grammar leads the network internal state to blur and merge, thus gives wrong outputs. This phenomenon is referred to as *instability*, which limits the generalization of RNNs ([Horne 1998]). Another problem that limits the infinite state representational power of RNNs is the problem of long-term dependency as discussed before concerning the work of [Lin 1996].

The state of RNN with continuous activation functions doesn't always cluster; the input sequence could lead the network to move between infinite internal states. Although this behavior could be undesired for some applications, it holds the potential for more computational complexity that can give rise to the behavior of pushdown automata or Turing machines. They could even be Super Turing, when real values of activations and weights are possible, as explained in 3.5.2.

Any network that can simulate a DFA provided with an infinite stack can simulate a PDA and the network should learn to manipulate the stack (pop, push, no-operation). Neural network pushdown automata (NNPDA) were implemented in [Das 1992, Sun 1998]. However, the stack can theoretically be implemented as a part of the network state so that the RNN can recognize context-free languages, but in practice, this is not possible, because an infinite memory stack requires a network with an infinite precision, so most RNNs simulating PDA try to implement a finite stack. Like PDA, RNN networks simulating Turing machines are PDA networks augmented by a second external stacks, an example of implementation using RNNs is in [Williams 1989].

4.7 Conclusion

Between the existing cellular computing models, few of them are well suited for processing temporal data. Depending on the nature of the temporal problem, the temporal data can have different characteristics. They can either be seen as temporal sequences or time series, they are slightly different in some properties and the required processing tasks.

The existing cellular models dealing with temporal data, have poor adaptive properties, and their training is not yet well established. Neural networks, at the contrary, have witnessed powerful and adaptive tools for temporal data processing.

The large number of neural networks offers different architectures that can deal with sequential data. Static feedforward networks like multi-layer perceptrons implement nonlinear functions that can be used to process temporal data in some special cases. Feedforward networks provided with traditional tapped delay lines exhibit dynamical properties that fit the processing of temporal data. They can be used to buffer portions of the sequence so that the temporal context of sequence elements is explicitly preserved in the network and intervenes in output computation. Memory kernels implement more complicated behaviors than the shift operations in simple delay lines; a unit in the delay line keeps trace of the past values shifted through it, so that it can maintain an information about the past inputs, which is forgotten gradually as time passes.

Recurrent neural networks, the major family of dynamical networks for temporal data processing, maintain the context of inputs differently. They are networks that contain feedback recurrent connections, this endows them with dynamical properties and transforms them into dynamical systems. Such dynamical networks integrate and represent time in their internal states. The context of input sequences is maintained with fewer number of connections and less computation time compared to delay networks. Learning endows dynamical networks with their ability to exhibit the desired behavior that fits the task requirement. The dynamical properties of recurrent networks models resemble those of formal models of computation, and many works established the bidirectional mapping between both models.

In the previous chapter, it was briefly mentioned that neural networks are fine-grain models that are not cellular. In this chapter, we surveyed the most important dynamical networks in order to figure out if any of them is cellular. It was found that only Hopfield networks might be considered cellular, only if the topographic connection locality condition as proposed by Sipper is mitigated. Reservoir computing networks can have their memory part, i.e. the reservoir, cellular when trained by local unsupervised learning methods, but the network as a whole can't be a cellular model.

One remaining important model of neural networks used in temporal sequence processing, the self-organizing maps, is still to be discussed. Their use in temporal tasks and their compliance with the conditions of the cellular computing paradigm is the subject of the next chapter.

CHAPTER 5

Self-Organization Maps for Temporal Sequence Processing

Contents

The previous chapter reviewed the most important artificial neural networks models used in processing temporal data. As has been shown, almost all of those models turn out to be not conform with the requirements of the cellular computing paradigm.

One last important model that has been used for the purpose of temporal data processing is the Self-organizing maps (SOMs), also called topographic maps. They are two-dimensional grids of computing units, that perform as static models used for processing sets of static patterns. SOMs are nonlinear projection methods that map a higher, finite-dimensional continuous vector space (the data space) onto two-dimensional discrete space of the map in a topology-preserving fashion. The original static SOM was developed by Kohonen [Kohonen 1982] for processing data where the sequential order is not significant, i.e context-independent. Thus its application in temporal problems was limited.

SOM was intensively used in many static tasks, and some modified and enhanced versions of the basic model were proposed for processing static data. Although, the representational capabilities and the internal representation of SOMs are not well understood [Tino 2006].

Recently, SOM was extended from processing vectorial data to more general data structures like temporal sequences and trees. SOM-based models are being used more and more as standalone sequence processors. However, some modifications are introduced to the original SOM in order to be able to process such recursive data types. In this chapter, we are interested in SOM-based models for sequential data, while SOMs for tree structures are not discussed.

Many different SOM-based models for sequence processing were proposed with different underlying mechanisms, however, there is yet no general consensus on the better SOM-based mechanism for addressing temporal problems, and the subject is an active concern of the research community in neuroscience.

SOM, and SOM-based models are fine-grain ones that compute in parallel at least in some phase of their computation. Besides to the temporal properties of these fine-grain models, we are interested in the cellular nature of the computation.

In this chapter we first remind with the original SOM model as proposed by Kohonen. We then review the most important SOM-based models for temporal sequence processing, with their different underlying mechanisms and focus on those using feedback connections. We then show how SOM-based models are not conform with the cellular computing paradigm. Then we briefly introduce the concept of neural fields, and show how we use them to guide learning in SOMs, and how this combination of methods allows SOMs to be not only fine-grain parallel models, but also implementing cellular computation.

5.1 The self-organizing map by Kohonen

The original self-organizing map by Kohonen [Kohonen 1982] is a neural network that consists of a set $\{1 \cdots N\}$ of N units arranged in a grid $\mathcal{A}$ which in turn, lies in two-dimensional space $\mathbb{R}^2$. We refer to $\mathcal{A}$ as the *map space*. Each unit is defined by its index $i \in \{1 \cdots N\}$ and topographic coordinates r_i in $\mathbb{R}^2$. SOM is also a projection method that associates a weight vector $\omega \in \mathbb{R}^d$, also called *codebook vector* or *prototype*, to each unit in a map space $\mathcal{A}$. The prototype vector is of the same dimensionality as the input vectors $\{x\}$ sampled from the *data space* $\mathcal{X} \subseteq \mathbb{R}^d$. Usually, the number of units N in the map is determined after some experiments to find the suitable mapping, but a rule of thumb, is to start with $N = \sqrt{M}$, where M is the number of available input vectors [Peres 2006] .

A mapping is established between the prototypes of the map units with a subset of vectors in the data space. This mapping associates multiple vectors in the data space to one prototype in the maps space, hence it can also be perceived as performing *vector quantization* (VQ), in which a larger number of vectors is mapped to a smaller number of vectors. This mapping is topology preserving, which means that neighbor units in the map correspond to neighbor regions in the data space, so that the distribution of the prototypes over the map space has the form of clusters.

The association of each unit to a prototype is attained by an adaptation process using an unsupervised learning method. The learning process implies both competitive and cooperative mechanisms. Competition occurs between the prototypes associated to the map units for the right to represent some data point in the data space. The unit with the winning prototype is called the winner unit or the *best matching unit* (BMU). When a prototype acquires the right to represent a data point, it adapts itself to that point. Cooperation means that not only this winning prototype updates itself, but also the prototypes of neighbor units, although to a lesser degree, depending on how topologically close their units are from the BMU; farther units prototypes are updated less than close ones. Updating the prototypes of the BMU neighbors is sometimes referred to as *radial adaptation*. This policy of prototype update of the BMU as well as its neighbors prototypes is referred to as *winner-take-most* (WTM). The other possible update policy is to only update the prototype of the BMU, this is referred to as *winner-take-all* (WTA). However, the WTM update is the prevailing policy in SOM models in literature. The static SOM is illustrated in figure 5.1.

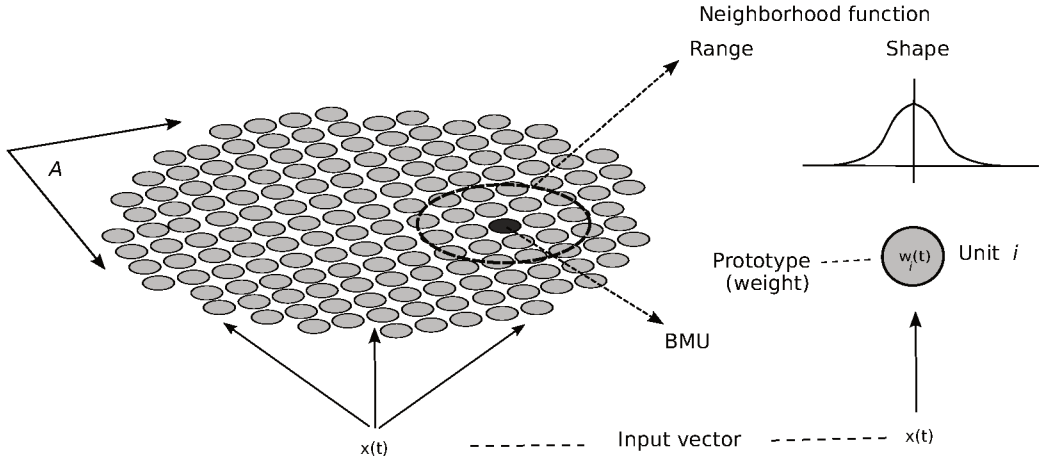


FIGURE 5.1: A Self-Organizing map: at each timestep, the input to the map is a vector, it is introduced to all map units that compare the input vector to the stored prototype in each unit (a vector of the same dimension). The map computes the best matching unit (BMU) which is the unit with the closest prototype to the input vector in Euclidean distance in most cases (Other distance measures are also possible, see for example [Demartines 1992, Horio 2008, Hajjar 2013]). Learning then occurs in a bounded region around the BMU, depending on the neighborhood function which is a Gaussian. See text for more explanation.

A SOM of N units, is defined by its prototype matrix $\Omega = (\omega_1, \omega_2, \cdots, \omega_N)$, with $\omega_i \in \mathcal{X} \subset \mathbb{R}^d$. Prototype values are randomly initialized at the outset.

At each timestep t, one input vector $x(t)$ is sampled from the data space $\mathcal{X}$ and processed by

all the map units. The map implements a function $i_{bmu}(x(t)) : \mathcal{X} \mapsto \mathcal{A}$ that assigns to the actual input vector $x(t)$ at time t a unit index $i_{bmu}(t) \in \mathcal{A}$ through a competitive process:

$$i_{bmu}(t) = \text{argmin}_{i \in \mathcal{A}} E_i(t) \tag{5.1}$$

where $E_i(t)$ is given by:

$$E_i(t) = d_{\mathcal{X}}(x(t), \omega_i(t)) = \|x(t) - \omega_i(t)\| \tag{5.2}$$

with $d_{\mathcal{X}}(x(t), \omega_i(t))$ is a distance defined in the data space $\mathcal{X}$.

The distance $E_i(t)$ is a value that measures how much $\omega_i(t)$ matches $x(t)$. It can also be regarded as an error between them, where the prototype $\omega_i(t)$ is regarded as the expected value by the unit i, and $x(t)$ is the actual input it receives. The algorithm tries through learning to minimize this error. Computing $E_i(t)$ for each unit in the map is referred to as the *matching* process, and $E_i(t)$ itself is referred to as the matching value as well.

After computing the BMU, the prototypes of the BMU and its neighbor units are adapted via a cooperative learning rule:

$$\omega_i(t+1) = \omega_i(t) + \alpha(t) h(d_{\mathcal{A}}(i, i_{bmu}(t)))(x(t) - \omega_i(t)) \tag{5.3}$$

with $0 \leq \alpha(t) \leq 1$ is the learning rate, and $d_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}$ is a decreasing distance function defined in the map space $\mathcal{A}$, usually taken as the Euclidean distance:

$$d_{\mathcal{A}}(i, j) = \|r_i - r_j\| \tag{5.4}$$

with r_i and r_j are, as defined before, the coordinates of the units i and j in the map space $\mathcal{A}$, and h is a radial weighting function, called the *neighborhood function* that delimits the adapted neighborhood of the BMU at time t :

$$h(\theta) = exp\left(-\frac{\theta^2}{2\sigma^2(t)}\right) \tag{5.5}$$

with $\sigma(t)$ the radius of the neighborhood function at time t. This function is typically a Gaussian. As can be noticed, there is no computation of the units activations in the basic SOMs, as they play no role in its functioning. However, this neighborhood function used for learning purpose can be regarded as the output that the map computes at each timestep, it is associated with the actual output: the BMU.

The elements of the set $\{x\}$ of inputs are presented iteratively to the map, one at each timestep, and learning continues to adjust the prototypes of the map units until the steady state of the prototypes is arrived, the map is then said to be converged. The change of the prototypes during learning in order to reach a global steady state is referred to as self-organization.

A common practice during learning is to decay the learning rate $\alpha(t)$ and the radius $\sigma(t)$ with the time in order to guarantee the convergence of prototype vectors to a stable steady state. This can be thought of as conducting learning in two phases, a coarse- and a fine-tuning ones.

The SOM works in two phases, the learning phase as explained above, and the exploitation phase in which it is used to compute outputs for its inputs without prototype update.

What the SOM performs, is sometimes expressed as approximating the input space, because it can map a large set of input vectors $\{x\}$ to a smaller set Ω of prototypes. Self-organization in the SOM results in a topology preserving distribution of prototypes, so that adjacent units prototypes in the map space $\mathcal{A}$ correspond to adjacent regions in the data space $\mathcal{X}$. The topology preservation property of the SOM combined with the quantization of input vectors results in clusters in the map space that correspond to their distribution in the data space.

In addition to the input space approximation and the topology preservation properties, the SOM has another interesting property which is density matching which means that the SOM reflects the statistical distribution of inputs. Normally, higher density regions in the data space $\mathcal{X}$ are sampled with higher probability, and the SOM maps these regions into larger regions on its space $\mathcal{A}$. Hence, higher density regions in $\mathcal{X}$ are assigned higher resolution regions in $\mathcal{A}$ than lower density ones.

Some modification of the original SOM have been proposed in literature, for example [Yin 2002] proposes a visualization-induced SOM called ViSOM, it is a data projection method that reflects better on its lattice the distance between data points in the input space, and offers a parametrized method to changing the resolution of the resulting clusters. Another variant of SOM is proposed in [Lee 2002] with a new learning rule that updates neighbor units to the BMU based on the topographic relation between each unit and the BMU without necessarily taking into account the BMU prototype, thus allowing for possible non-radial adaptation of the neighborhood.

SOMs were applied in various applications, like vector quantization [Heskes 2001], dimensionality reduction [Campoy 2009], data visualization [dong Jin 2002, Heskes 2001], clustering [Vesanto 2000, Xiao 2003], and classification [Bogdan 1996].

5.2 SOM-based temporal sequence processing

Although Neural networks were used for Temporal sequence processing tasks, SOMs were not used for such tasks until the past fifteen years. The main reason is that sequence processing tasks were thought of as supervised learning ones, in which the input is the current and the past values and the output is the one-step ahead prediction. On the other side, SOMs were always thought of as unsupervised methods for vector quantization [Barreto 2007]. For this reason, supervised models like the MLPs and RBFs and their temporal variations discussed in the previous chapter, were used for the supervised learning of temporal sequences, while SOM was thought of as not suitable for sequence processing tasks. However, several SOM-based models were proposed to carry out sequential data processing. The investigation of SOMs in sequence processing was first motivated by scientific curiosity, but there are other reasons as discussed in [Barreto 2007].

The first reason is related to the nature of computation in SOMs. At each timestep, the SOM works on localized regions of the input space to compute its output (the BMU). This allows for better understanding of the dynamics of the underlying process that generates the temporal sequence, especially when combined with visualizing techniques. This is in contrast with MLP-based models, the latter work on highly distributed data in the input space, thus make the interpretability and the visualization of the results more difficult.

Another reason for using SOMs in sequence processing is related to their clustering capability. Unlike feedforward networks, which require specifying the number of units in the model in advance to fit the data space, the number of units in SOMs is less important to specify. SOMs, that can adapt their prototypes, can fit to the best the data distribution to their actual prototypes, no matter was their number. With the same number of neurons, one can adapt the parameters of the competitive learning in order to fit another data distribution. A third reason is that injecting prior knowledge to the model is as easy as loading a prototype vector.

In order to cope with temporal sequence processing tasks, SOMs are used in different ways [Guimaraes 2003]. The first way is by using the basic SOM, but with the pre-processing or post-processing of the temporal data. The second, is by modifying the basic SOM algorithm, basically the learning rule, so that it reflects the temporal dependency between the sequence elements. And the third way is modifying the basic SOM architecture to obtain a new model.

If we leave apart the use of the basic SOM in sequence processing, different SOM-based models add different modifications to the basic static SOM in order to add the capability of representing the temporal context of inputs. Some of these models represent time explicitly using a spatial representation of time through, for example, a time window, and deal with windowed temporal inputs as a spatial input vectors. Windowing can be implemented using tapped delay lines, in this case the necessary short-term memory is implemented by a data structure. Other SOM-based models represent time implicitly through adding some form of feedback implementing a short-term memory mechanism that accounts for the temporal context of the inputs. Here, the memory mechanism is rather seen as a procedure.

In addition to these discussed models, there are other models that use hierarchical architectures that contain multiple SOMs. Each of the mentioned options can be implemented using different methods, resulting in various models, most of them are reviewed in the coming paragraphs.

However, until now, the representational capabilities and the internal representation of structures within these models are unclear [Hammer 2004b]. They are hardly understood in spite of the different mathematical efforts and theoretical analysis in this direction, such as [Hammer 2004b, Tino 2006]. This is in line with the statement presented by researchers like Hammer and Barreto [Hammer 2004b, Barreto 2003] that there is yet no clear winner unsupervised SOM-based model that represents the temporal context in sequence processing in the best way.

SOM-based models were used in various applications including time series classification and forecasting [Ultsch 1996, Koskela 1997, Vesanto 1997], control [Ritter 1989, Simula 1996a], monitoring [Simula 1996a, Kangas 1990b] and data mining [Deboeck 2010, Guimaraes 2001].

5.3 Temporal sequence processing with the basic SOM

Temporal sequence processing can be carried out using the SOM algorithm as proposed by Kohonen. In order to use the static SOM in such temporal tasks, the data is either pre-processed before presenting it to the SOM or the SOM output is post-processed.

5.3.1 Pre-processing: SOM with time-embedding

In these methods, the temporal sequence is pre-processed in such a way that embeds (or hides) the temporal dimension of the sequence elements in another vector, that is presented as input to the basic SOM model, and processed as a static vector in the SOM-traditional way. Time is embedded in the input vectors mainly in two ways, either by concatenating input elements using a tapped delay line, or by extracting some features from the sequence that will be used as a static input vector.

Tapped delay lines are the simplest approach to provide the SOM with an external mechanism of short-term memory, the sequence elements are shifted into the tapped delay line, so that at some timestep, the tapped delay line contains a subsequence of the temporal sequence, or a time window, that is considered as one spatial pattern vector selected from a temporal sequence, the spatial vector is presented as an input to the SOM at that timestep. Such model is used in [Kangas 1990a, c. Chappelier 1996], and is illustrated in figure 5.2(a).

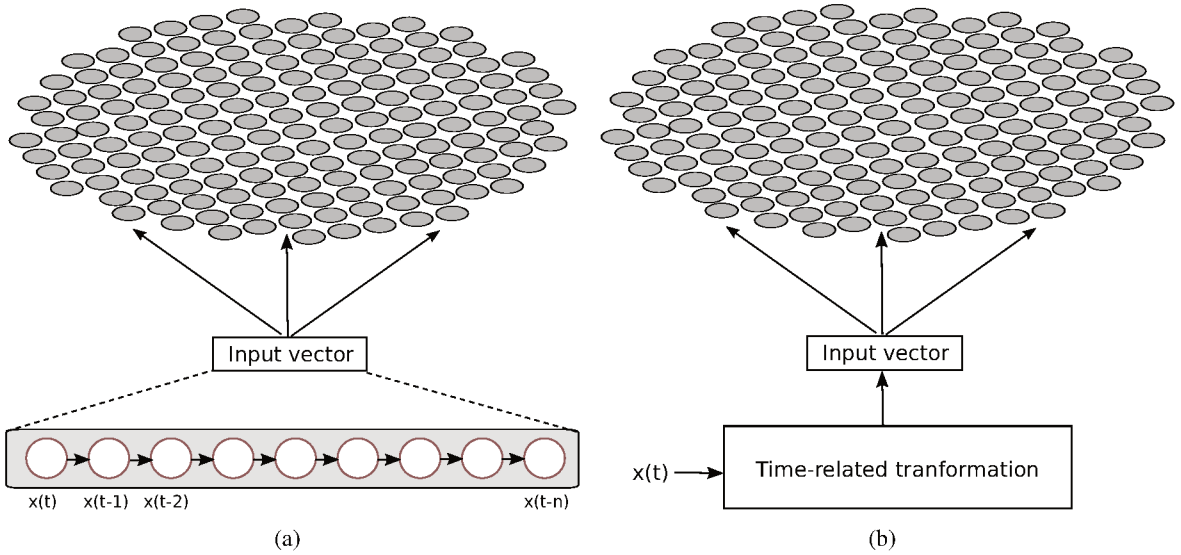


FIGURE 5.2: SOM with pre-processing for temporal sequence processing: (a) Time-delay SOM: at each timestep, the input to the map is the static vector formed by the elements of the subsequence available in the tapped delay line. (b) The static inputs are some feature vectors extracted by some time-related transformation applied on the temporal sequence.

These methods are also called *time-delay SOMs*, because they use the same approach as feed-forward delay networks presented in 4.4. However, they share with these networks the same drawbacks; the length of the delay line should be determined beforehand, thus may not fit the degree of the processed temporal sequences. If the sequence degree is higher than the length of the delay line, then the dynamics of the task will not be captured by the SOM, while if it is smaller, this will result in an unnecessary time consuming processing, besides, it may fail in isolating the context of smaller-degree elements in the sequence.

Other drawbacks are related to the SOM algorithm. First, the SOM computes the BMU on the basis of the Euclidean distance between the input vector and the stored prototype in each unit. Thus, two inputs (time windows) which are temporally close (for example shifted by one step)

could be assigned to spatially distant BMUs. Second, that grouping the elements of a subsequence in the delay line and presenting its context as input vector to the SOM makes it insensitive to the order of elements in the result vector, thus to their statistical dependency. This is because the SOM algorithm computes the BMU on the basis of the Euclidean distance between the input vector and the stored prototype in each unit, and in this, elements order is irrelevant; it is possible to carry out a permutation on the elements of the time window and the result does not change. There exist other distance measures, some of which like the Manhattan distance [Demartines 1992] suffer from the same problem. Other possible distance measures that are sensitive to elements order are the dot product distance [Demartines 1992], and data-related distances [Horio 2008, Hajjar 2013], the latter work uses Mahalanobis distance and varies it within each data cluster on the map.

Another approach to time-embedding in the map input, is by performing time-related transformations as a pre-processing of the data sequence. Depending on the task, if the to-be-processed temporal sequences contain features that can be processed in domains other than the temporal one, then the pre-processing of the temporal sequence consists in extracting these features and using them as static inputs to the basic SOM algorithm. This approach is illustrated in figure 5.2(b) Several transformation can be used, such as using frequency domain features after applying Fourier transformation as in [Kohonen 1988], wavelet-transform as in [Moshou 2004], time-frequency transformation as in [Atlas 1996]. Another technique for embedding time using complex numbers [Mozayyani 1995] was mentioned in 4.3.

Data pre-processing through time-related transformations has the advantage of maintaining the basic SOM algorithm, and it can be used with other static model, not only SOMs. However, this approach is only useful in tasks where the temporal sequences have sufficient informative features in domains other than time.

5.3.2 Post-processing: Trajectories as extrapolation into the temporal domain

In this approach, the sequence elements are processed one after another by the basic SOM as static vectors, then the successive outputs, or part of them, which is the BMUs corresponding to the processed sequence elements is visualized on the map as a trajectory (illustrated in figure 5.3). The trajectory then can be seen as extrapolating the static outputs again into the temporal domain. The visualization of the processing results as a trajectory of BMUs is possible thanks to the two-dimensional lattice of SOM that is used as a representation space. This visualization is not possible in feedforward networks. Representing the result as a trajectory can be perceived as a form of post-processing of processed sequence elements inputs.

In this method, it is not necessary to present the sequence elements in their natural order, instead, it allows for collecting the elements of the temporal sequence and presenting them to the SOM in any order, and the temporal information can be recovered and interpreted in the temporal domain to reflect the structure of the underlying process. The interpretation of results can be facilitated by the shape or direction of the trajectory. The required information can sometimes be obtained if the trajectory is heading toward a specific region, an example work is [Simula 1995] in which the authors classify speech signals using the behavior of the resulting trajectories.

More information can be obtained by combining the trajectory with other information, for example by drawing the trajectory on the map lattice while showing the prototypes values as gray

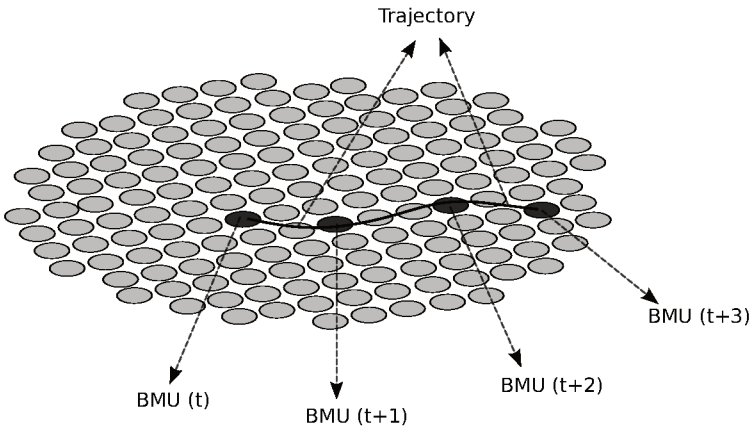


FIGURE 5.3: A trajectory on the map lattice is obtained by connecting the subsequent BMUs.

scale colors. This is possible if the prototypes are scalar values, otherwise it is possible to represent one component of the prototypes vector using gray scale colors. This way, one can obtain more knowledge about the behavior of the underlying process. The latter technique is used in the model proposed in this manuscript for the purpose of result visualization, however, this is not the core temporal method used in our model, this will be explained in the next chapter.

More sophisticated visualization techniques such as U-matrices, and hierarchical clustering visualization are reviewed in [Guimaraes 2003]. However, the trajectory can be used beyond visualization purposes; the trajectory associated with a sequence or a subsequence thereof, can be used as an input to a higher-level module in a more complex processing system such as hierarchical SOM models. For example, in [Srinivasa 1999], a higher-level module works on the produced trajectories and classifies them.

The main drawback of the methods within this approach is that they can't deal with ambiguous sequence elements (introduced in 4.1); elements with the same value, will be assigned to the same BMU, this complicates the reading and interpretability of the trajectory.

5.4 Temporal sequence processing with modified SOMs

Besides to pre-processing and post-processing techniques used for processing temporal sequences with the basic SOM by Kohonen, other methods that alter the standard functioning of the basic SOM were developed and applied for the same purpose.

There are plenty of SOM modifications that aim at considering the temporal context of input elements, two possible approaches besides to some of their variations are mentioned here. The first relies on embedding the context of the current input in the map input vector, the algorithm handles both parts differently, both in computing the matching and in applying the learning rule. The second approach restricts its search for the next BMU in the neighborhood of the past BMU.

5.4.1 Embedding of the context in the map input: the Hypermap

This method implements a different approach for temporal sequence processing than the basic SOM. It associates the sequence elements with their context in a supervised way, through embedding the element and its context in the input vector presented to the map. The used algorithm has different variations, especially in time series prediction applications.

In the method, called the Hypermap [Kohonen 1991], the input to each unit in the map consists of two distinct vectors, the pattern vector $x^{pat}(t)$, and the context vector $x^{cont}(t)$. The input $x^{pat}(t)$ can contain one sequence element, or the concatenation of the elements of a subsequence of length τ_1 centered around t, and the input $x^{cont}(t)$ contains the concatenation of a larger subsequence of length $\tau_2 > \tau_1$ also centered around t. This is illustrated in figure 5.4. To each map unit i is

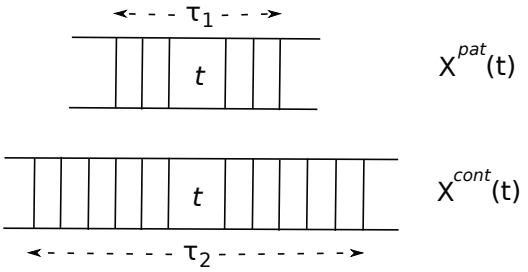


FIGURE 5.4: Inputs to Hyper map: The input to the map is a pair of two vectors, the pattern vector $x^{pat}(t)$ of length τ_1 centered around t, and a context vector with a larger length τ_2 also centered around t.

associated two prototype vectors $\omega_i^{pat}(t)$ and $\omega_i^{cont}(t)$ that are compared to $x^{pat}(t)$ and $x^{cont}(t)$ respectively.

The computation of the BMU in this method occurs in two steps, the first is done by comparing the context vector $\omega_i^{cont}(t)$ to the corresponding prototype vector $\omega_i^{cont}(t)$ in order to select a "best match region", called the *context domain*, denoted $R \subset \mathcal{A}$, obtained by all units that fulfill:

$$d_{\mathbb{R}^{\tau_2}}(x^{cont}(t), \omega_i^{cont}(t)) = \left\| x^{cont}(t) - \omega_i^{cont}(t) \right\| \leq \delta \tag{5.6}$$

with δ is a threshold determined experimentally. The BMU is then selected, in the second step, within the region R, on the basis of the $x^{pat}(t)$ vector:

$$i_{bmu} = \text{argmin}_{i \in R} d_{\mathbb{R}^{\tau_1}}(x^{pat}(t), \omega_i^{pat}(t)) = \text{argmin}_{i \in R} \left\| x^{pat}(t) - \omega_i^{pat}(t) \right\| \tag{5.7}$$

This is illustrated in figure 5.5.

Prototype adaptation is carried out in two periods, in the first period, the context prototypes $\omega_i^{cont}(t)$ are adapted by unsupervised learning using $x_i^{cont}(t)$ vectors, following the same rule as the basic SOM. When learning converges to a stable representation, context weights are frozen at the end of this period. In the second period, only the $\omega_i^{pat}(t)$ vector is adapted in unsupervised way. Another step may be added at this period to adapt $\omega_i^{pat}(t)$ in a supervised way using the LVQ algorithm [Kohonen 2001]. Kohonen applied the Hypermap model on phoneme recognition, in order to recognize patterns that occur in the context of other patterns.

The Hypermap is also used in time series prediction with slight modifications. When used in

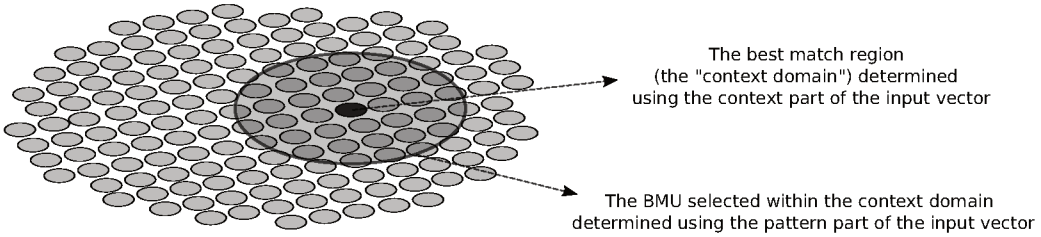


FIGURE 5.5: The Hypermap.

prediction, the pattern part $x^{pat}(t)$ is the "future" vector, as opposed to its context $x^{cont}(t)$ which represents the "past" vector. The two parts of the input vector are used for learning at time t, and can be handled similarly or differently in computing the matching and in learning. However, the two input vectors are arranged in a different way. The context vector $x^{cont}(t)$ is used to hold the subsequence $[x(t-1)x(t-2)\cdots x(t-p)]$ with p the order of embedding, regarded as the past, or the temporal context of the future value $x(t)$ that will be held in $x^{pat}(t)$. The vectors are concatenated in one vector $X(t) = \left[x^{cont}(t)|x^{pat}(t)\right]$ as in figure 5.6, so as the corresponding prototypes $W_i(t) = \left[\omega_i^{cont}(t)|\omega_i^{pat}(t)\right]$.

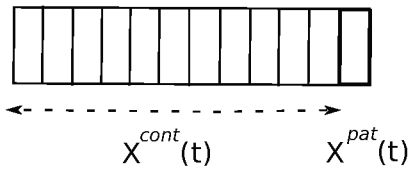


FIGURE 5.6: The Hypermap input in processing time series.

$x^{pat}(t)$ holds the sequence element to predict, which is available during the learning phase, but not available during the exploitation phase. During the learning phase, the context part $x^{cont}(t)$ is loaded with the past sequence elements, and the pattern part $x^{pat}(t)$ is loaded with the future element. So in prediction tasks, the aim of learning is to associate the future value with its past, i.e. its context. This associative mapping, can be seen as extrapolating the sequence into the future.

However, some works like [Koskela 1998] alter this learning behavior. In this work, the search for the BMU is reduced to one step where the BMU is determined only by the context vector, and the learning phase is a one step process where both prototype vectors are updated at the same time.

In the exploitation phase, the future value is unknown, thus the prediction relies only on the past part $x^{cont}(t)$ of the input vector to compute the BMU. When the BMU is found, its prototype part $\omega_i^{pat}(t)$ corresponds to the future value to predict.

Hypermap and its variations were used in phoneme recognition in [Kohonen 1991, Midenet 1994], robot control as in [Walter 1998], and forecasting chaotic time series [Koskela 1998].

Another algorithm that uses the same philosophy as Hypermap in prediction, i.e. using one prototype vector that groups the context with the pattern, was proposed in [Barreto 2001], and used in time series prediction, and in robotic control [Barreto 2003]. The algorithm is called vector-

quantized temporal associative memory (VQTAM), where the BMU is determined by the context part $x^{cont}(t)$ only, hence the notion of "context domain" is not defined in VQTAM, and the BMU is searched within all the map, not in a restricted region. Both the context and the pattern parts of the prototype $W_i(t)$ are updated concurrently, using the basic SOM learning rule. A major difference between VQTAM and the Hypermap approach is that there is no separated learning periods, learning of both prototype parts in VQTAM occurs at each time step until the map reaches a stable representation, not in periods like Hypermap.

During the exploitation phase, the one-step-ahead prediction is obtained by $\omega_i^{pat}(t)$ after finding the BMU in a similar way as in the learning phase, i.e. depending only on the context part $x^{cont}(t)$ of the input. In VQTAM, the prediction can also be computed after smoothing the prototype of the BMU, by computing the average of the $\omega_i^{pat}(t)$ parts of the k neighbor prototypes, with $k > 1$. A variation of VQTAM was proposed in [Lendasse 2005] and improved time-series forecasting performance over a benchmarking task. It implements an auto-regressive model $AR(p)$ in which the elements of the context vector $x^{cont}(t)$ (past p sequence elements) as well as the pattern vector $x^{pat}(t)$ are weighted by a set of parameters, and the algorithm searches to find the best parameters through a cross-validation procedure.

Other SOM-based models that use the same idea as the Hypermap and VQTAM were developed in [Lehtokangas 1996] to associate a *local model*, which is an $AR(p)$ model, to each unit in the map. A set of p parameters are associated to the context part $\omega_i^{cont}(t)$ of units prototypes; one parameter for each element corresponding to the p elements of the sequence held by $x^{cont}(t)$. Training is carried out using a set of q input vectors that are $p+1$ length (additional one for the future value), obtained by a moving window over the time series. As in Hypermap and VQTAM, BMUs are determined using $x^{cont}(t)$ but learning is carried out using the whole input vector $x(t)$. After learning is completed, each parameter of each local model (associated to each unit) is determined by matrix calculations. Obviously this is a lot of computation. During learning, some q local models specialize to q time-windows of the processed time series. In the exploitation phase, the one-step-ahead prediction of the time series is given by future part $\omega_i^{pat}(t)$ that the local model has associated to the BMU, which, during the learning phase, has adapted the best to the actually processed portion of the time series. SOM in this example computes local models, here they are autoregressive models. It is worth noting that the computed models are local not because they are associated to localized units in the map, but because the map units model the behavior of the time series in a localized portion of this time series. SOM-based models are efficient methods in such local modeling, because, with their population of computing units, they can cover several portions of the processed sequence. Several other variant methods that use the same philosophy of the Hypermap are used for local modeling, such as [Walter 1990] for the online learning of local models, and the KSOM model [Barreto 2004] for computing time varying local AR models from prototypes. The model proposed in [Principe 1998] is used for computing "nonlinear" local models applied in system identification and control.

As one expects, the Hypermap algorithm and its variations perform temporal embedding as do time-delay SOMs, thus suffer from the same problems; one should determine the degree of the sequence in advance, and when it is high, the method becomes computationally expensive.

5.4.2 Restricting the BMU search: Kangas map

The Kangas map introduces a slight modification on the basic SOM algorithm, However, it is efficient in processing temporal sequences. Unlike the Hypermaps, in which the context is explicitly embedded in the input vector, the context in this algorithm is defined by the previous BMU. This simple idea proposed by Kangas [Kangas 1992], is basically the same as the basic SOM, but with restricting the search of the BMU within the neighborhood of the previous BMU (determined by some prefixed distance). The matching and the learning of the basic SOM remain unchanged in this algorithm. This makes it even faster than the basic SOM, as computing the matching, and learning is only necessary in a bounded region of the map. Kangas map is illustrated in figure 5.7.

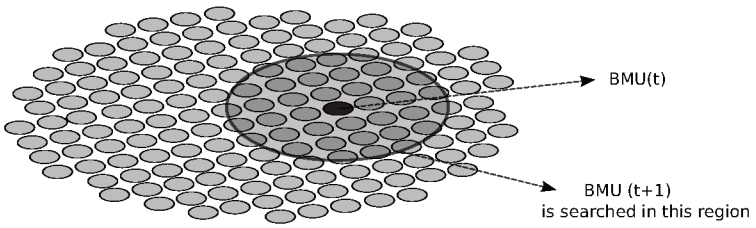


FIGURE 5.7: Kangas Map: The search of the next BMU is restricted to a bounded region around the previous BMU.

Let us consider a temporal sequence which has two subsequences with identical elements, i.e. an ambiguous sequence. The basic SOM maps these two subsequences to the same set of BMUs. However, the Kangas map *may* assign them to different sets of BMUs, as the elements of the second subsequence may be assigned to BMUs in a different map area, depending on the latest BMU arrived before processing the elements of this second subsequence. Hence, the locations of the BMUs depend on how they are arrived, thus, depend on the temporal context of the processed sequence elements. Kangas applied his algorithm in speech recognition [Kangas 1992], but it was also applied for other tasks like texture identification and 3D object recognition [Chandrasekaran 1995, Chandrasekaran 1998].

Kangas introduced an approach to constraint the search of BMUs, applicable regardless of the used algorithm in the map. One drawback of this algorithm is that the correct mapping of ambiguous sequences is not guaranteed. However, other approaches that condition the search of BMUs and solve this problem also exist; instead of limiting the search for the BMU in the neighborhood of the previous one, the model called SARDNET introduced in [James 1995] excludes the previously assigned BMUs from being assigned again in the subsequent searches. This means that the map is forced to assign to a sequence of length l, even if ambiguous, l different BMUs that allow for a trajectory of length l.

5.5 SOM-based context models

The third way to use SOMs for temporal sequence processing is to change the network topology. This can either be done by adding feedback connections, allowing for the appearance of some form of short-term memory, or by using architectures with hierarchical layers, each containing one or

more SOMs. This section presents several SOM networks with feedback, while hierarchical SOMs are presented in the coming section.

It has been shown in the previous chapter (see 4.5) that feedback connections in RNNs create an internal dynamics in the network that allow to implement a short term memory, used in memorizing the context of the to-be-learned sequence elements. This is why models that use SOM with feedback are sometimes called *context models* [Hammer 2004b]. The advantage of SOMs with feedback over other SOMs implementing time embedding techniques, like the one seen in time-delay SOMs, is similar to the advantages of RNNs over time-delay networks: there is no need to specify the length of the delay line in advance; the network can adapt -in the limit of its memory depth- to the length of the context of the processed element.

Unlike RNNs which consist in multiple layer networks, there are few values in SOM that can be fed back. The basic SOM computes no activation to its units, however, in most SOM models with feedback, activation plays a central role in the model.

SOM-based context models differ in how they account for the temporal context.However, whatever the case, all these models should deal with the sequences differently from the basic SOM; they should take into account the order in which the elements are presented. The elements of a sequence are presented one after another to the model, each iteration of the prototype adaptation (learning) should be influenced by the previously presented elements which form the context of the current input element. The integration of the temporal context is done during the BMU selection, so that the BMU at a certain timestep corresponds not only to the current input, but to the current input associated with its temporal context. The temporal context intervenes in the selection of the BMU, by involving the context information in computing the matching prior to BMU selection. This ensures that the BMU is influenced by the temporal context of the current input element.

Basically, SOM-based context models differ by the way the context is represented, and their approach of influencing the BMU selection by the context information. That will be detailed in the coming subsections where different models with different approaches for context representations are presented.

5.5.1 Local feedback: Temporal Kohonen map

This model was the first one to introduce feedback to the SOM. Temporal Kohonen maps (TKM) [Chappell 1993] differ from the basic SOM in that they use the output or activation of the map units for the model functioning. Whereas the tapped delay line provides the SOM with an external short-term memory mechanism, another memory mechanisms can be implemented on the level of each map unit. In TKM, each unit memorizes its past activation, and reuses it in computing its next activation. This is done using a leaky integrator (implemented by a difference equation) to recursively compute the output. For a unit $i \in \mathcal{A}$, the unit computes an activation, which is also the computed matching as follows:

$$y_i(t) = E_i(t) = \beta y_i(t-1) - \frac{1}{2} \|x(t) - \omega_i(t)\|^2 \tag{5.8}$$

with β is a decay factor, where $0 \leq \beta \leq 1$. When talking about the context models, the matching $E_i(t)$, which is basically a distance as mentioned in 5.1 is sometimes called the *recursive distance*,

because it includes information related to the context, which is here the past activation $y_i(t-1)$.

The BMU in TKM is computed as the unit with the highest activation, but the learning rule is the same as the basic SOM. The latter is a drawback in this model because the context information computed by the leaky integrator at the unit output does not affect the prototype adaptation. This results in TKM suffering from the loss of temporal context. Although, learning is slightly affected by the context information, as it is carried out in the neighborhood of the BMU, that the context intervenes in its computation. Another reason for the loss of the temporal context is related to the way in which the matching is computed: the use of the square operator in computing the matching -which is a form of error- loses the information of the direction of the error.

Another drawback with TKM is that the classification of the input sequence is only determined by the index of the last reached BMU. For this reason, besides to its loss of context, TKM may give the same classification for sequences that differ slightly in their elements [Carpinteiro 1998].

5.5.2 Enhanced local feedback: Recurrent SOM

TKM keeps no information about the direction of the error vector, and does not use the context information in learning. In order to overcome this problem, recurrent self-organizing map (RSOM) was proposed by [Varsta 1997b, Varsta 1997a]. Instead of using the leaky integrator in the output, a leaky integrator which preserves the direction of the error signal is used *at the input*. Also, the context information is used in prototype adaptation. The integrator of a unit i computes a recursive distance:

$$y_i(t) = E_i(t) = (1-\beta)y_i(t-1) + \beta(x(t) - \omega_i(t)) \tag{5.9}$$

where β is a memory depth parameter that determines the influence of the past difference relative to the current input, with $0 \leq \beta \leq 1$. This is illustrated in figure 5.8. The vector $y_i(t)$ captures a linear combination of the subsequent input patterns. This way, the temporal context of the input is incorporated in determining the BMU.

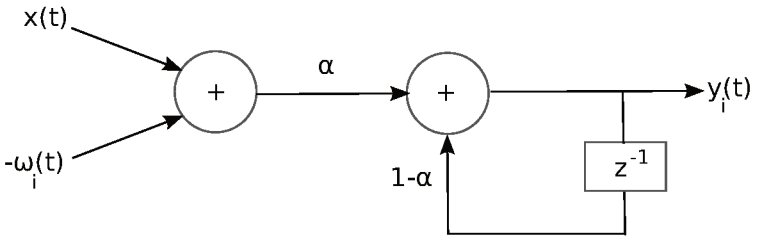


FIGURE 5.8: RSOM leaky integrator

The BMU is selected as:

$$i_{bmu}(t) = \text{argmin}_{i \in \mathcal{A}} \|y_i(t)\| \tag{5.10}$$

Conversely to TKM, the vector $y_i(t)$ of each unit i in the map is used in learning:

$$\omega_i(t+1) = \omega_i(t) + \alpha(t)h(d_{\mathcal{A}}(i, i_{bmu}(t)))y_i(t) \tag{5.11}$$

In both TKM and RSOM, the usage of leaky integrator makes the context information implicit in

computation, without reference to the previous map activity. Also, the approach of both models for accounting to the context is to refer to the unit itself; no other map units are involved in the computation of the activation.

RSOM was used to compute local linear models for time series prediction in [Koskela 1997]. It was also used in [Varsta 1997b] in the classification of EEG-based epileptic signals, for each new input, the prototypes are reset to zero.

Although RSOM overcomes some of the TKM problems, it shares with it the shortcomings of classifying the sequence by the last reached BMU, thus loosing the spatial representation of the temporal sequence on the map.

5.5.3 Total activity as feedback: Recursive SOM

Both TKM and RSOM use a form of local feedback in the computation of the unit output, the leaky integrator approach for presenting the temporal context suffers from the limitation that only past activities related to that very unit can hold the context information, and no information about other units are considered.

However, other SOM-based context models that use a more explicit representation of the context were developed. These models use a global information related to the past map state of activation. They use as feedback (to all units) some output information of all or some of the map units in the previous timestep. This way, each unit in the map becomes "aware" of the previous map state, and can account for a richer temporal context. One important model is the one proposed by Voegtlin [Voegtlin 2002], called recursive SOM or RecSOM. It is a reference model to which most recent temporal SOMs are compared to. The model proposed later in this manuscript is compared to it as well.

RecSOM approach to account for the temporal context is to feed back the activation of *all* the map units in the previous timestep to each unit in the next timestep. The past activations of all the map units $\{y_i(t-1)\}_{i\in\mathcal{A}}$ are arranged in a vector $x^{cont}(t) = y(t-1) \in \mathbb{R}^N$ that is concatenated with the current input $x^{pat}(t) = x(t)$ and the result vector $X(t) = \left[x^{pat}(t)|x^{cont}(t)\right]$ is presented as input to each map unit at time t. This implies that to each unit i is associated a two-part prototype vector that corresponds to both input parts: $W_i(t) = \left[\omega_i^{pat}(t)|\omega_i^{cont}(t)\right]$. The prototype $W_i(t)$ vector is then compared to $X(t)$ to compute the matching for the unit i.

RecSOM goes beyond using the simple local recurrence of leaky integrators as in the earlier models presented previously, and was proved by [Hammer 2004b] to exhibit much richer dynamical behaviors. Within this model structure, the map becomes a non-autonomous dynamical system, its rich-enough dynamics allows to implement a form of short-term memory. In RecSOM, the basic SOM algorithm is applied for both the current input and the context information represented by the previous map activation, and the network learns to associate the current input with the previous map activity; activity which is related to the matching of all units prototypes with the previous map input in the previous timestep. It is by this association that the map units respond to a specific part of the temporal sequence. So each unit i learns to represent a coupled input vector that corresponds to some input sequence element associated with its context expressed as the past map activity, that depends on the already seen elements. It is important here to emphasize that the interior representation of the context in the map, has a different structure from the "pure" context that

consists of elements of a subsequence, but depends on it. The representation of long sequences is learned iteratively, based on previously learned subsequences. Figure 5.9 illustrates RecSOM.

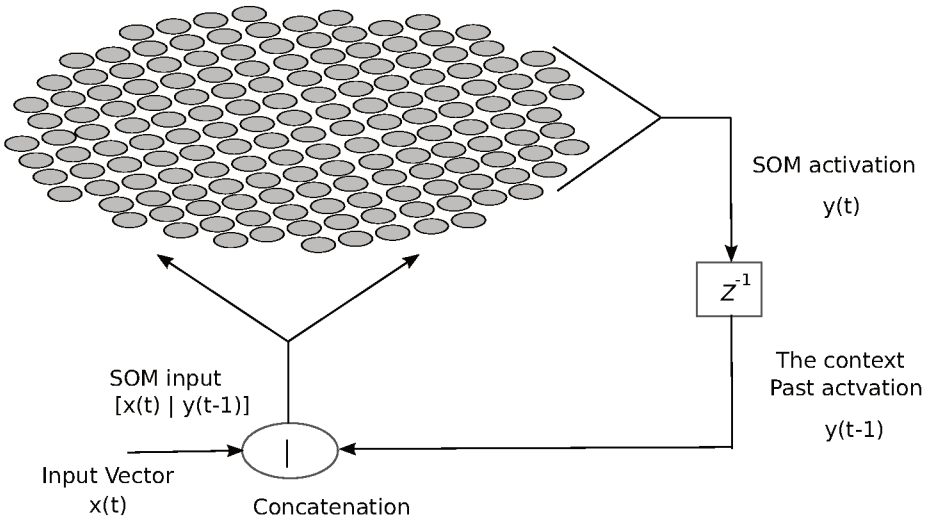


FIGURE 5.9: Recursive self-organizing map (RecSOM): At each timestep t, the input vector $x(t)$ is concatenated with a context value that is the past activation $y(t-1)$ of all maps units, The result vector form the actual input to all units within the map. The map computes the new activity vector $y(t)$ that becomes the new context value for the next input.

The activation vector $y_i(t)$ for a unit i at time t is computed as follows:

$$y_i(t) = \exp(-E_i(t)) \tag{5.12}$$

with the recursive distance $E_i(t)$ is given by:

$$E_i(t) = \alpha \left\| x(t) - \omega_i^{pat}(t) \right\|^2 + \beta \left\| y(t-1) - \omega_i^{cont}(t) \right\|^2 \tag{5.13}$$

where α and β are constant coefficients that control the importance of the actual input and its context respectively.

The BMU at time t is computed as:

$$\begin{aligned} i_{bmu}(t) &= \text{argmax}_{i \in \mathcal{A}} y_i(t-1) = \text{argmin}_{i \in \mathcal{A}} E_i(t) \\ &= \text{argmin}_{i \in \mathcal{A}} \left(\alpha \left\| x(t) - \omega_i^{pat}(t) \right\|^2 + \beta \left\| y(t-1) - \omega_i^{cont}(t) \right\|^2 \right) \end{aligned} \tag{5.14}$$

Learning consists in modifying both parts of the prototype vector depending on their respective parts of the input vector:

$$\omega_i^{pat}(t+1) = \omega_i^{pat}(t) + \gamma h(d_{\mathcal{A}}(i, i_{bmu}(t))) \left(x(t) - \omega_i^{pat}(t) \right) \tag{5.15}$$

$$\omega_i^{cont}(t+1) = \omega_i^{cont}(t) + \gamma h(d_{\mathcal{A}}(i, i_{bmu}(t))) \left(y(t-1) - \omega_i^{cont}(t) \right) \tag{5.16}$$

where γ is a learning rate, $h(d_{\mathcal{A}}(i, i_{bmu}(t)))$ the neighborhood function as in the basic SOM.

At this point, it is convenient to differentiate between recursive SOMs and recurrent SOMs. In recurrent SOMs some output value is simply fed back to the network again, while in recursive SOMs, the feedback signal is treated exactly as the input signal. This is the case in RecSOM, as both the activation feedback and the current input are handled similarly in the feedforward phase of the model functioning.

This RecSOM model belongs to a general class of SOM-based models, called *activation based recursive SOMs* (ARSOMs) that use activation state of the network as a time delayed internal input that serves as the context, while the external input serves as the "content", and both are concatenated in one input vector presented to each unit in the map.

RecSOM was used in many applications, Voegtlin himself used his model to process a corpus of written English, after each word, he resets the context vector, and as in the basic SOM, the radius $\sigma(t)$ of the neighborhood function is decreased with time. He defined the *receptive field* of a unit in the map as the common suffix of all subsequences for which that unit becomes the BMU. The organization of receptive fields on the map is shaped on the only basis of the suffix structure of the processed subsequences, and it doesn't depend on the position of the subsequence in the input sequence, i.e. doesn't depend on its temporal context, this organization is said then to have a Markovian flavor [Tino 2006]. In general, all feedback SOM-based models have their organization with this Markovian flavor [Tino 2007].

The Markovian organization of the activation state space of RecSOM happens only of its representation is stable. Stability in RecSOM, as investigated in [Voegtlin 2002, Tino 2006] means that, under a fixed input, the mapping $y(t) \mapsto y(t+1)$ of the map activation is a contraction, i.e. the autonomous RecSOM dynamics is dominated by a unique attractive fixed point. The authors in [Tino 2006] claim that this condition, if satisfied (related to the values of the β coefficient), leads to a Markovian representation of the temporal context.

Another recursive model, similar to RecSOM in that it uses the activity of all the units outputs in the previous time step as a feedback is proposed in [Hynna 2006]. In this model, called "also" the activation-based recurrent self-organizing map (ARSOM), the only difference from RecSOM is in how it computes the units outputs. It is computed as follows:

$$y_i(t) = F(d_{\mathcal{X}}(x(t), \omega_i^{pat}(t)), \alpha).F(d_{\mathbb{R}^N}(y(t-1), \omega_i^{cont}(t)), \beta) \tag{5.17}$$

where F is a transfer function used for both matchings. The authors tried several transfer functions while changing the radius of the neighborhood function, and found their characteristics that allow the network to outperform RecSOM in some specific temporal tasks. Interestingly, the authors reported that the choice of the optimal settings is also data-dependent, i.e. it depends on the complexity of the input sequence. The latter remark is in line with what claim Hammer an Barreto [Hammer 2004b, Barreto 2003], that there is yet no clear winner SOM-based model that represents the temporal context in sequence processing in the best way.

5.5.4 BMU coordinates as compact feedback: SOMSD

The RecSOM approach to account for the context is to use the whole map activity as a feedback, requiring a lot of computation due to its complexity and storage requirements. Although, the feedback information in RecSOM is centered around the BMU, while the activity of other regions in the map is suppressed [Hammer 2004b], this is justified by the exponential transformation given in equation (5.12), that emphasizes regions with high activity and suppresses those with small activity.

Other context models with feedback connections that are less computationally demanding than RecSOM in the unsupervised processing of temporal sequences were also developed. These models use much more compact feedback information. One model is proposed in [Hagenbuchner 2003], called SOM for structured data (SOMSD), it was basically developed for learning general tree structures, however, it can be used on temporal sequences as a special case of trees.

Instead of all the units activity as in RecSOM, in this approach, the feedback information to SOMSD units is the coordinates $r_{i_{bmu}(t-1)}$ of the BMU in the input map space $\mathcal{A}$ at the previous timestep. At time t, $r_{i_{bmu}}(t-1)$ is compared to the context part $\omega_i^{cont}(t)$ of the prototype $W_i(t)$. Here also, the dimension of $\omega_i^{cont}(t)$ is the same as the dimension of $r_{i_{bmu}}(t-1) \in \mathcal{A}$ (embedded in $\mathbb{R}^2$). The recursive distance is given by:

$$E_i(t) = \alpha \left\| x(t) - \omega_i^{pat}(t) \right\|^2 + \beta(d_{\mathcal{A}}(\omega_i^{cont}(t), i_{bmu}(t-1))) \tag{5.18}$$

Then the activation $y_i(t)$ is given in a similar way than RecSOM, as in equation (5.12). Here, by computing $d_{\mathcal{A}}(\omega_i^{cont}(t), i_{bmu}(t-1))$, the context part $\omega_i^{cont}(t)$ is manipulated as if it belongs to $\mathcal{A}$.

The BMU is selected as:

$$i_{bmu}(t) = \text{argmin}_{i \in \mathcal{A}} E_i(t) \tag{5.19}$$

Training adapts the two parts of the prototype vector of the winning neuron and its neighborhood as in the basic SOM:

$$\omega_i^{pat}(t+1) = \omega_i^{pat}(t) + \gamma h(d_{\mathcal{A}}(i, i_{bmu}(t)))(x(t) - \omega_i^{pat}(t)) \tag{5.20}$$

$$\omega_i^{cont}(t+1) = \omega_i^{cont}(t) + \gamma h(d_{\mathcal{A}}(i, i_{bmu}(t)))(r_{i_{bmu}(t-1)} - \omega_i^{cont}(t)) \tag{5.21}$$

The use of a compact feedback makes SOMSD considerably much faster than RecSOM. The cost of the gained computation speed, is that the context information is not as rich like in RecSOM, however, the noise is also reduced in SOMSD compared to RecSOM.

There are other models that implement the same feedback. For example, in [Mcqueen 2002] the author presents a recurrent SOM for language processing. The context part of the input vector is the binary-coded geometrical coordinates of the last BMU in the two-dimensional map space, considered as the feedback to the network. For each input sequence (sentence) he draws the trajectory of the BMUs, and resets the context vector for each new sequence. The recognition of a sentence is based on the "visual signature" obtained by different BMU trajectories.

However, this approach for context representation depends on the topology of the map, because it uses the semantic meaning of the BMU coordinates. For this reason, it is claimed in [Strickert 2005] that it cannot be combined with other models without lattice semantics, like with

the neural gas model [Martinetz 1993], the latter being a topology preserving network which can be applied on unspecified lattice topologies. Besides, the fact that the context integration depends on the distance in the map space $\mathcal{A}$, makes the topological distortion on the map lattice affect the sequence representation, this limits the usability of the model in processing complex temporal sequences.

5.5.5 BMU content as compact feedback: Merge SOM

Merge SOM (MSOM) [Strickert 2003, Strickert 2005] is a model that also uses a compact feedback as SOMSD, but instead of using the semantic meaning of the coordinates of the BMU, in this approach, the feedback signal is a merged content of the BMU prototype vector. The prototype vector associated to each unit i at time t is also two-part, the context part $\omega_i^{cont}(t)$ and the pattern part $\omega_i^{pat}(t)$, both have the same dimension. In order to have the whole prototype $W_{i_{bmu}}(t-1) = \left[\omega_{i_{bmu}}^{pat}(t-1)|\omega_{i_{bmu}}^{cont}(t-1)\right]$ at time $t-1$ fit in the context part $\omega_i^{cont}(t)$ of units prototypes at time t, $\omega_{i_{bmu}}^{pat}(t-1)$ and $\omega_{i_{bmu}}^{cont}(t-1)$ are merged using a linear combination, and the merged value is stored in the context part $\omega_i^{cont}(t)$ of all units prototypes in the next timestep. It is the computation of the linear combination that requires both prototype parts to be of the same dimension.

The recursive distance in MSOM is computed as follows:

$$E_i(t) = (1-\alpha)\left\|x(t) - \omega_i^{pat}(t)\right\|^2 + \alpha\left\|c_{i_{bmu}}(t-1) - \omega_i^{cont}(t)\right\|^2 \tag{5.22}$$

where $c_{i_{bmu}}(t-1)$ is a linear combination:

$$c_{i_{bmu}}(t-1) = (1-\beta)\omega_{i_{bmu}}^{pat}(t-1) + \beta\omega_{i_{bmu}}^{cont}(t-1) \tag{5.23}$$

with $0 \leq \beta \leq 1$.

The BMU is computed as the one with the highest matching value:

$$i_{bmu}(t) = \text{argmin}_{i\in\mathcal{A}} E_i(t) \tag{5.24}$$

Prototype adaptation is the same as in SOMSD equations (5.20) and (5.21) (substituting $c_{i_{bmu}}(t-1)$ by $r_{i_{bmu}(t-1)}$ in (5.21)), but with possibly different learning rate γ for each equation.

In this model, the context does not depend on the lattice topology. It was mainly developed in order to be, unlike RecSOM, computationally affordable, and to be possible to combine, unlike SOMSD, with other lattice structures like neural gas model.

Although the capacity of unsupervised context models are hardly understood, some recent works studied the computational capabilities of feedback models in the form of theoretical models of computation. TKM and RSOM can only simulate finite memory models [Kaminski 1990], because of their restricted recurrence within each unit. SOMSD and MSOM are claimed to be equivalent to finite state automata [Hammer 2004c, Strickert 2005] while RecSOM was claimed to be at least able to simulate a pushdown automata [Hammer 2006].

5.6 Hierarchical SOM models

These models consist essentially of a hierarchy containing basic SOMs or temporal SOMs like the previously presented ones. A hierarchical model is arranged in layers, and in each layer resides one or more SOMs. SOMs in different layers may be synchronized or the SOMs of each layer may work in a different time scale than those of other layers.

Hierarchical models are normally used when the task requires decomposing the problem of sequence processing into smaller subproblems, or when constructing a higher level semantics is desired, then SOMs pertaining to each layer, do each a part of the work. For example, the model can use several SOMs at the first layer to handle different sources of data, then in the upper layer, the result of the processing can be fused by another map. Such models are used in speech recognition, where subsequent layers process phonemes, syllables, and words [Behme 1993].

Hierarchical models differ between them by the number of SOMs used in each layer, their types (basic or temporal), and in the number of layers. They also differ in how information is forwarded from each layer to the other. Information from a lower layer is forwarded in the form of an input vector to the map or maps in the upper layer. The prototypes of the map in the lower layer can be presented as inputs to the maps in the next layer, or the inputs can be a transformation of the prototypes of the maps in the lower layer. Such transformations can be based on the distance between units prototypes, or concatenating subsequent vectors that form a trajectory into one vector presented to the map in the next layer (like in [Srinivasa 1999] as mentioned before).

One example model that computes the distance in time between two subsequent prototypes is a VQTAM-based model that performs double vector quantization (DVQ). It is proposed in [Simon 2004] and is used for long-term time series prediction tasks. This model is achieved by using two basic SOM networks, intended to give a static characterization of the evolution of a time series. The first SOM clusters q subsequences obtained by a sliding time window on the time series as explained before. The current subsequence is held in $x^{cont}(t)$ part of the current input. The second SOM clusters the *deformations* $\Delta x^{cont}(t) = x^{cont}(t+1) - x^{cont}(t)$ associated to each of the q subsequences. Visualizing the q pairs of $(x^{cont}(t), \Delta x^{cont}(t))$ gives useful information about the evolution of the time series between the q time windows.

Another example of prototype transformation is the work of [Simula 1996b], in which two SOMs are used to predict the behavior of an industrial process. The first SOM was used to track the operating points of the process by a trajectory in the map space, the second map is trained on the concatenation of trajectory prototypes to predict the next operating point that represents the expected state of the process. Another example is the model proposed in [Kangas 1990a], that uses a two layer model for phoneme recognition, where the first layer contains a SOM and the second layer contains a time-delay temporal SOM that processes the trajectory of activations produced by the first map.

In some cases, a hierarchical model can combine SOMs with other networks in order to improve the performance of these networks. These models are referred to as *hierarchical hybrid models*. Such a model was used in [Mayberry 1999] which combines SOM with Elman and NARX networks in order to improve their performance concerning the long-term dependency in processing natural language.

Combining SOMs to carry out some task is a frequent practice in literature. Many different hierarchies and processing methods exist, however, SOMs or temporal SOMs as presented before, reside somewhere in all architectures.

5.7 Other self-organizing neural networks for sequence processing

Self-organization is an emergent property that characterizes the self-organizing map by Kohonen, however, they are not the only neural models to exhibit this property. There are other topology-preserving neural models that are used in temporal sequence processing.

SOMs and SOM-based models for temporal processing encode time in their units activation rules, and implement no lateral connections. At the inverse, most of the non-SOM self-organizing models use lateral connections among the output units. It is these connections that are responsible for learning sequential order of inputs, so that the model can capture the temporal dependencies between the input sequence elements. In these models, lateral connections between units are adaptable following a form of the Hebbian learning rule that depends on the activity of units on both vertices of a connection.

One example model that also uses a two-dimensional map is the one proposed by [Kopecz 1995]. In this work, the map $\mathcal{A}$ is provided with two groups of lateral connections, the first one is called "symmetric lateral connections", they are responsible for the spatial localized activations of the map units. A symmetric connection between two units i and j has a weight $\omega^s_{ij}(t) = \omega^s_{ji}(t)$ computed as a function of the distance between the units:

$$\omega^s_{ij}(t) = \alpha.h(d_{\mathcal{A}}(i,j)) \tag{5.25}$$

where α is a constant, and $h(.)$ is the same as in equation (5.5). The second group is called "asymmetric lateral connections", it is responsible for encoding the temporal order of the inputs. An asymmetric connection between two units i and j has a weight $\omega^a_{ij}(t)$. Unlike their symmetric counterparts, these connections are adaptable by an asymmetric Hebbian learning rule, this rule accounts not only for units activations, but on the timing of these activations, thus it is referred to as a "temporal Hebbian rule":

$$\frac{\mathrm{d}\omega^a_{ij}(t)}{\mathrm{d}t} = -\frac{\mathrm{d}\omega^a_{ji}(t)}{\mathrm{d}t} = S(y_i(t))\frac{\mathrm{d}S(y_j(t))}{\mathrm{d}t} \tag{5.26}$$

where $y_i(t), y_j(t)$ are the activations of units i, j respectively. $S(y_i(t)) = 1$ if $y_i(t) > 0$ and $S(y_i(t)) = 0$ otherwise. The activation dynamics of a unit i is given for the sequence learning and recall phases:

$$\frac{\mathrm{d}y_i(t)}{\mathrm{d}t} = -y_i(t) + \sum_{j=1}^{N} \omega^s_{ij}(t)S(y_j(t)) + x_i(t) - h \text{ (for learning)} \tag{5.27}$$

$$\frac{\mathrm{d}y_i(t)}{\mathrm{d}t} = -y_i(t) + \sum_{j=1}^{N} (\omega^s_{ij}(t) + \varepsilon\omega^a_{ij}(t))S(y_j(t)) + x_i(t) - h \text{ (for recall)} \tag{5.28}$$

where h is a constant, $x_i(t)$ is the external input to the unit i, and ε is a parameter that controls the speed of activity change during recall. Once trained, the asymmetric connections allow active regions in the map to trigger other regions, thus reproducing the learned temporal sequence. This method was tested in the learning and recall of artificial sequences, and gave good results, unless the sequence is ambiguous where it fails.

Another self-organizing network used in temporal sequence processing is the reservoir-like recurrent network proposed in [Lazar 2009]. In this model, called self-organizing recurrent network (SORN), the network contains a population of N_E excitatory units and a smaller population of N_I inhibitory units. Each of these unit has a threshold of activation. The connectivity between the two populations is total, i.e. every excitatory unit $x_i^E(t)$ is connected to every inhibitory unit $x_j^I(t)$ by a connection holding the weight ω_{ij}^{IE} and vice versa. The connectivity between excitatory units is sparse and random, while inhibitory units are not connected between them.

A temporal sequence consists of different symbols, in this model, each symbol is assigned to a group of N_U input units, which are the third type of units. These units are set to 1 when the current input element is that sequence symbol, otherwise they are set to 0. Input units are connected to excitatory units only.

The activity of excitatory and inhibitory units are updated as follows:

$$x_i^E(t+1) = \theta\left(\sum_{j=1}^{N_E} \omega_{ij}^{EE} x_j^E(t) - \sum_{k=1}^{N_I} \omega_{ik}^{EI} x_k^I(t) + x_i^U(t) - T_i^E(t)\right) \tag{5.29}$$

$$x_i^I(t+1) = \theta\left(\sum_{j=1}^{N_E} \omega_{ij}^{IE} x_j^E(t) - T_i\right) \tag{5.30}$$

where θ is a threshold function that constrains the activity of units to a binary representation, and T_i^E and T_i^I are the excitatory and inhibitory units thresholds respectively, their values are drawn from a uniform distribution in $\left[0, T_{max}^E\right]$ and $\left[0, T_{max}^I\right]$. As equations show, each unit is driven by the past units activities and by the current input for excitatory units. A unit fires if the total weighted input that it receives is greater than its specific threshold.

The network is trained by unsupervised learning, driven by three plasticity techniques. First, the plasticity of the excitatory connections, called *spike-timing-dependent plasticity* (STDP):

$$\omega_{ij}^{EE}(t+1) = \omega_{ij}^{EE}(t) + \alpha\left(x_i(t)x_j(t-1) - x_i(t-1)x_j(t)\right) \tag{5.31}$$

This rule means that the synaptic weight $\omega_{ij}(t)$ of the connection between unit i and j is strengthened by a fixed amount α when the unit i is active in the time step following the activation of unit j, and is weakened by the same amount if the activation of i precedes the activation of j. This plasticity is intended to reinforce the firing of subsequent units for subsequent input elements.

The second plasticity normalizes connections in order to keep constant the sum of the weights of one type of incoming connections to an excitatory unit:

$$\omega_{ij}^{EE}(t) = \omega_{ij}^{EE}(t) / \sum_j \omega_{ij}^{EE}(t) \tag{5.32}$$

The third plasticity technique consists in modifying the excitatory units thresholds, it is called the *intrinsic plasticity* (IP):

$$T_i^E(t) = T_i^E(t) + \beta(x_i(t) - H_{IP}) \tag{5.33}$$

where H_{IP} is a constant depending on the number of input units N^U and excitatory units N^E. This rule means that the unit that has been active increases its firing threshold to leave place for other units to fire. Similarly, other units lower their thresholds to increase their chance in firing in the next timestep. This IP rule also reinforces the firing of subsequent units for subsequent input elements.

This network is claimed by the authors to learn the temporal structure embedded in the input sequence in unsupervised manner, and to be able to learn ambiguous sequences and assign identical inputs to different representations depending on their temporal context. When a readout is trained on the network, it is claimed by the authors [Lazar 2009] to outperform highly tuned random reservoirs in some sequence prediction tasks.

A third example of self-organizing networks used in temporal sequence processing is the model presented in [Araujo 2002]. It is a relatively complicated model that combines adaptive lateral connections with two additional context mechanisms at the input: A tapped delay line implementing short-term memory, and a set of units that hold a part of the sequence as its identifier. The content of these two sets of context units with the external input form the current input to the network. During the lateral competition, the winner is excluded from subsequent competitions.

The combination of these three mechanisms for temporal processing besides to the winner exclusion technique allows for learning ambiguous sequences. Moreover, they allow the network to learn two sequences concurrently.

The common property between the presented models that differentiates them from SOM-based models is the adaptability of lateral connections, so that the temporal order of the input elements is learned to be captured by the subsequent activations of the output neurons.

5.8 On the cellular nature of SOM-based models

SOMs are fine-grain models that consist of populations of simple processing units. Units are computing in parallel at least in some phase of the computation. Specifically, the matching phase is a parallel one where all the map units compute a matching value based on the distance between the current input and their stored prototypes.

For the SOM to be a cellular model, it should meet, besides to the simplicity and the parallelism of the processing units, the conditions of locality and decentralization. In SOMs, there are no lateral connections, the interaction between units occurs only in the learning phase using the neighborhood function that implements the winner-take-most update strategy. This learning process requires the use of a central processor in order to search for the BMU within the population of units, and in order to apply the prototype update as the neighbor units concerned by the update has no connections to the BMU. The centralized processor is also needed in cases where the learning rate $\alpha(t)$ or the radius $\sigma(t)$ of the neighborhood function are decayed with time during learning.

What is said about SOMs, can also be said about all the presented SOM-based models. For those models, in addition to the mentioned reasons, the central processor is also required for the reset of some map values, like the prototypes, as mentioned for RSOM, RecSOM, or the model presented in [Mcqueen 2002]. Obviously, **SOMs and SOM-based models are not cellular models**, due to the lack of decentralized processing during the learning phase.

In the following, we present the neural field theory that we rely on later in the learning phase of our proposed model introduced in the next chapter, and we show how it makes a SOM-based model a cellular one.

5.9 Neural fields

The *Dynamic Neural Fields Theory* (DNFT) aims at describing the formation and evolution of patterns of activity at the level of populations of neurons, rather than at the level of individual ones. The DNFT approach bridges the gap between two approaches used for the symbolic formalization of the neural activity in the brain.

The first approach is the biophysical models that provide detailed descriptions of electrical or chemical phenomena observed within or between individual neural cells [Hodgkin 1952, FitzHugh 1955, Nagumo 1962, Izhikevich 2004]. While they generally display great accuracy with respect to experimental observations, their computational complexity makes it impractical to use them in large-scale neural modeling (see [Izhikevich 2004] for a review).

The second approach is the statistical modeling of neurons activity; it relies on the Bayesian approach and proposes probabilistic models of neural information processing at the level of networks of neurons [Rombach 2008, Friston 2008, Moran 2013]. While extremely valuable for examining emergent computational properties of neural dynamics, they generally focus on "phenomenological" rather than "functional" descriptions of these processes, thus providing only limited insights about the neural mechanisms underlying these phenomena. Moreover, most of the statistical models have formal descriptions which are not conform with the principles of cellular computing paradigm presented previously.

Unlike these two approaches, DNFT uses an integro-differential mathematical description to define a paradigm of local, parallel, distributed and decentralized computations performed by a population of basic units which exchange information through local connections. These properties make the DNFT compatible with cellular computing general principles of computation. In the following subsections we introduce the mathematical framework of DNFT, as we consider it as an excellent candidate for our motivation in building neural networks which can be both computationally powerful and architecturally appropriate with cellular computing requirements.

5.9.1 The formalism of dynamic neural fields

DNFT addresses the problem of capturing the instantaneous response of a population of neural units to external stimuli and tracking the long-term dynamics resulting from the interactions between groups of units. The focus is set on population-centric description of neural phenomena rather than a unit-centric one. DNFT accounts for mesoscopic (i.e. intermediate-level) characteristics of the

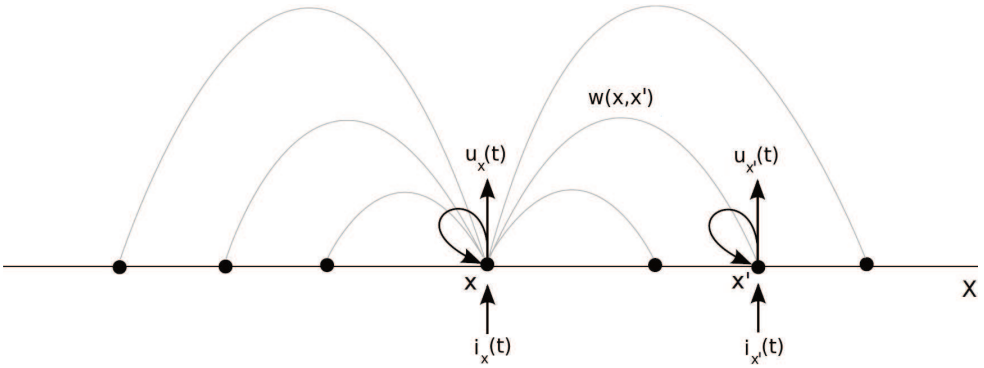


FIGURE 5.10: Notations and general architecture of a dynamic neural field (here depicted for the one-dimensional case). From [Alecu 2011a].

neural units activity, such as the mean firing rate of *groups* of neural units, rather than microscopic variables (such as channel currents related to individual neural units). Attempts in this direction started since the work of Beurle [Beurle 1956], but the theoretical foundations of dynamic neural fields were established in the works of Wilson and Cowan [Wilson 1973] and Amari [Amari 1977].

A dynamic neural field is a spatiotemporal description of the evolution of the activity of a set of neural units, usually denoted by $u_x(t)$. It represents the average firing rate which can be observed at a specific position x in the space and at time t within the set of population of units. Usually, a position in the field refers to a place occupied by a group of neural units, rather than by an individual ones. One can say that this group defines an *abstract neural unit* of computation, as it embodies a computational process performed by a group of neural units. This view allows for the mesoscopic description of the neural phenomena instead of the microscopic one.

The space X within which the field is defined is a subset of a topological space and is considered a continuum, typically, $X \subset \mathbb{R}$ in the one-dimensional space or $X \subset \mathbb{R}^2$ in the two-dimensional space. The units are interconnected through lateral connections as illustrated in figure 5.10. In most of the models, these connections act either in an excitatory or inhibitory manner, in the following way: the connections between short-range neighbor units are excitatory, while the long-range connections to more distant units are inhibitory. The synaptic weight $w(x, x')$ represents the influence of the connection between two units situated in the positions x and x' in the field. The distance $\|x - x'\|$ between them is given according to the metric $\|\cdot\|$ in the topological space X, typically the Euclidean distance is used. This synaptic weight is normally modeled as a difference of Gaussians:

$$w(x, x') = A^+ \mathrm{e}^{-a\|x-x'\|^2} - A^- \mathrm{e}^{-b\|x-x'\|^2} \tag{5.34}$$

with A^+ and A^- are the amplitudes of the excitatory and inhibitory influences and a and b are modulation parameters.

In addition to the activity of the neighboring units, each unit x in the field can be influenced by an external input $i_x(t)$ referred to as the *external stimulation*. With these notations we can introduce

the classical model proposed by Amari in [Amari 1977].

5.9.2 Amari model

The activity of the field at each point $x \in X$ in the space and at a time instance t is defined as follows:

$$\tau \frac{\mathrm{d}u_x(t)}{\mathrm{d}t} = -u_x(t) + \int_{x' \in X} w(x, x')\sigma(u_{x'}(t))\mathrm{d}x' + i_x(t) + h \tag{5.35}$$

Here, τ is a time constant which defines the rate with which the activity of the field can change at each timestep, $\sigma(.)$ is a sigmoid function that models the activation (firing) of a unit, and h is the field's resting potential; a level of activity to which the field converges in absence of any other stimulation (i.e. external or lateral).

The integral term in equation (5.35) implies that in order to update its current activity, a unit must know the activities of all the other units in the field. However, in practice, as the synaptic weight to distant units in the field approaches zero (due to the exponential in equation (5.34)), thus it is sufficient to know the activities (and the corresponding synaptic weights) only for a group of neighboring units, in order to perform the computation. In this case, DNFT can be implemented in a fully parallel, distributed and local manner: each unit of the field updates its activity based only on cumulative local information: the external simulation $i_x(t)$ and the current activity $u_x(t)$ of its neighbors modulated by the synaptic weights, which is called the *lateral contribution*, .

When the ensemble of neural units in the population performs the computation given by the neural field equation (here equation (5.35)), the population evolves in time giving a temporal activity profile that is determined by the local competition between the neural units.

5.9.3 Behaviors of dynamic neural fields

The behavior of the neural field depends on the internal parameters of the model and on the profile of the input stimulation. For example, it has been shown that in absence of external input, the field can (in certain conditions) self-sustain patterns of activities similar to stationary or traveling pulses or waves, or more complex patterns such as breathers or Turing instabilities [Coombes 2005, Folias 2004, Folias 2005, Wyller 2007].

However, in the practical case in which the field is stimulated by some external input (the case most frequently used in DNFT applications and also in the case of our work), the neural field usually displays the so-called *bump dynamics*. Through its competition mechanism, the field gradually increases its activity in places corresponding to the highest local cumulative external stimulation. As a result, the field generates a so-called *bump of activity* in these places as in figure 5.11 (sometimes the bumps resemble to the shape of a bell). The bumps remain unchanged as long as the input stimulation does not change. These bumps represent, in terms of dynamical systems theory, the attractors of the neural system, that is, the stable states toward which the field converges.

When the profile of the input stimulation changes, the profile of the field's activity can change too. In certain cases, the already emerged bumps vanish and reemerge somewhere else in the

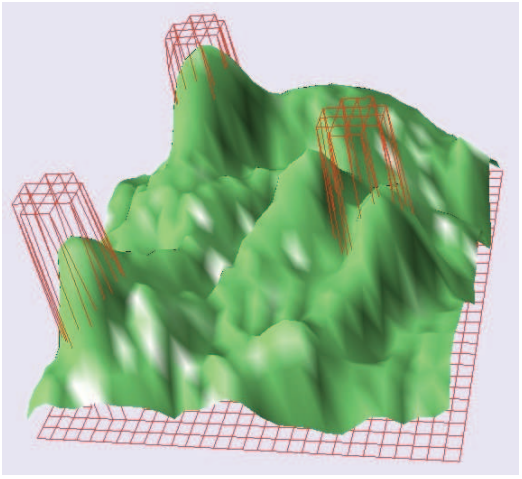


FIGURE 5.11: Formation of bumps in a 2D dynamic neural field (illustrated as a mesh surface), in response to input stimulation (illustrated as a full surface). Illustration obtained using `bijama` simulation software [Frezza-Buet 2012].

field, in other cases, they shift toward the new attractors. However, there can also be situations where, even if the profile of the input stimulation changes, the positions and amplitudes of the already emerged bumps remain the same (in this case, they are self-sustained through the lateral contributions, rather than by external stimulation). Other more subtle effects can appear too, but it is generally difficult to characterize all the dynamic regimes that a field can exhibit.

5.9.4 Applications of dynamic neural fields

A whole area of research in DNFT focused on the study of the behaviors of neural fields in order to find the conditions under which these behaviors can emerge and can remain stable, these studies began early with the proposal of Amari [Amari 1977]. However, they are usually difficult to conduct due to the intrinsic complexity and non-linear character of the model equations.

A second path of research approaches DNFT from a more practical point of view, it aims at applying the computational features and the dynamics of neural fields to the design of new neural mechanisms. This is based on a more general view, called attractor network computational paradigm [Amit 1992], which exploits the properties of the attractors of recurrent dynamical networks to perform complex computations. This paradigm was used in computational neuroscience [Gros 2009] and has been applied to various problems.

Neural mechanisms based on DNFT have been proposed in recent years for visual attention [Fix 2007], short-term and long-term memory [Simmering 2007], and for designing embodied cognitive autonomous robotic agents capable of acquiring multimodal sensory perception and developing motor skills [Iossifidis 2001, Ménard 2005, Erlhagen 2006] or decision-making and planning skills [Toussaint 2006, Trappenberg 2008, Sandamirskaya 2008].

DNFT was initially developed to provide an intermediate level of description for the neural ac-

tivities across the cortical tissue. Due to its laminar anatomical structure and its columnar functional structure [Mountcastle 1957, Mountcastle 1997], the DNFT fits the architectural macroscopic description of the cerebral cortex. According to this theoretical framework, the cortex can be viewed as a set of interconnected computational units arranged in a two-dimensional space. Works such as [Bressloff 2002, Hutt 2003, Sabatini 2004, Meyer 2007] show that DNFT can provide accurate descriptions of neural dynamics observed in neurophysiological studies of the cortical tissue. However, DNFT is not limited to the depiction of neural activities detected in the cerebral cortex, but can also be used to describe the neural dynamics of other regions in the brain. Other studies have shown that DNFT can successfully describe patterns of neural activity in other sub-cortical neural structures as well, such as in the superior colliculus [Schneider 2002] or the striatum [Nakahara 2002].

Neural units in DNFT compute their activity on the basis of the activity of neighbor units in the same layer, thus DNFT is a subclass of recurrent neural networks (RNN). Consequently, classical techniques used in RNN can be applied to the context of DNFT, most notably in the area of training. Although this subject is of central interest in RNN realm, learning based on DNFT has been rather rarely studied. Training a DNFT-driven neural architecture can be achieved by considering adaptable characteristics of the neural field such as the synaptic weights, the resting state potential, the firing profiles, etc., or by the investment of the dynamic properties of the patterns of activity that emerged in response to input stimulation, in order to adjust some other variables of the global architecture. Works such [Gross 1998, Miikkulainen 2005a, Ménard 2005, Gläser 2008b, Gläser 2008a] have been proposed to implement unsupervised or reinforcement learning strategies through neural field-like dynamics. Besides, the model of [Kopecz 1995] presented in 5.7 is a spacial kind of neural field, in which the synaptic weights are adaptive, except that different field equations are used for learning and recall.

In our work, we are interested in applying DNFT to the problem of learning of temporal sequences using unsupervised self-organization. Our work relies on two main works [Ménard 2005] and [Alecu 2011b], both conducted previously in our lab, which deal with the subject of self-organization driven by dynamic neural field. The legacy of the first work is the `bijama` architecture (which we will rely on to build our model), while the legacy of the second is the Binp neural field.

In our work, we use the dynamics of neural field bumps to drive the self-organization process in our learning architecture. In this context, the bump is the analogous of the neighborhood function $h(.)$ of the basic SOM algorithm. We recall that in the basic SOM, the neighborhood function is a winner-take-most (WTM) function facilitating the cooperation mechanism. Its role is to compute the amount with which the prototypes of the units in the map adjust themselves according to the distance from the BMU. In our proposed implementation of the self-organization process, this role of the neighborhood function is replaced by the dynamics of a bump generated through a neural field mechanism stimulated by an external input (which is the matching computed by the map units). Thus, the entire process can be implemented locally, in a completely distributed parallel way (unlike the case of the SOM, which requires centralized processing).

The model of Amari is the first candidate to implement this mechanism. However, in practice, it is very difficult to achieve the appropriate behavior for the field in the context of self-organization. As analyzed in [Alecu 2011b], in order to perform learning, the field has to be extremely sensitive,

and capable of generating a bump even in conditions of very low input stimulation. If this can be achieved, the field then experiences very high levels of excitation within the area of the bump, and very high levels of inhibition elsewhere. When the input changes, these changes are usually too small to counterbalance the influence of the existent excitation/inhibition contributions. As a result, in most of the time, the bump does not follow the changes in the input, being thus unable to implement satisfactorily a WTM policy.

To address this problem, [Alecu 2011b] proposed a new neural field model, adding new features to the model of Amari. This is shown to exhibit more convenient behavior than Amari model, and to make use of the bump dynamics to drive successfully a SOM-like unsupervised learning process. In the next subsection, we introduce this neural field model.

5.9.5 Binp model

Unlike in the model proposed by Amari and presented above, the Binp model (stands for: Back inhibited neural populations) [Alecu 2011b] makes a separation between the excitatory and inhibitory connections. The corresponding weights are then:

$$w^+(x,x') = \mathrm{e}^{-a\|x-x'\|^2} \tag{5.36}$$

$$w^-(x,x') = \mathrm{e}^{-b\|x-x'\|^2} - \mathrm{e}^{-c\|x-x'\|^2} \tag{5.37}$$

This leads to distinguish excitatory and inhibitory lateral contributions, perceived by each unit x in the field, and denoted by $\mathcal{E}_x(t), \mathcal{I}_x(t)$, respectively:

$$\mathcal{E}_x(t) = \sigma^+\left(\int_{x'} w^+(x,x') f(u_{x'}(t))\mathrm{d}x'\right) \tag{5.38}$$

$$\mathcal{I}_x(t) = \sigma^-\left(\int_{x'} w^-(x,x') f(u_{x'}(t))\mathrm{d}x'\right) \tag{5.39}$$

where σ^+, σ^- and f are sigmoid functions.

The Binp field formalism is given by:

$$\tau \frac{\mathrm{d}u_x(t)}{\mathrm{d}t} = i_x(t) + \alpha\,\mathcal{E}_x(t) - \beta\,\mathcal{I}_x(t) - \gamma\,g(i,v) \tag{5.40}$$

$$\frac{\mathrm{d}v_x(t)}{\mathrm{d}t} = h(\mathcal{E}_x(t)) \tag{5.41}$$

Here, τ is a time constant that has the same meaning as in Amari model, while α, β, γ are positive constants. The activity described by the variable v models a delayed version of the perceived local excitation $\mathcal{E}$. Finally, g and h are sigmoid functions, which enable a built-in delayed local feedback inhibition mechanism.

The model is built such that the field is capable of generating a bump quickly in all conditions and in particular, in conditions of very low input stimulation. When a bump is generated, the excitatory contribution increases within the neighborhood of the bump. The auxiliary activity v increases too, but more slowly than the excitation $\mathcal{E}$. Throughout all this time, the bump follows the dynamics in the model of Amari. After a transitory period, it stabilizes to match the locally

highest region of input stimulation. The slower v evolves, the larger this period is. When the input stimulation changes and the new input perceived by the units in the bump region is high enough, the bump remains in its place. However, if the latter input is low, the function $g(i, v)$ is designed to become very high and as a consequence, inhibits locally the field (influenced by the negative term in equation (5.40)). If multiple units become inhibited, this triggers a local feedback inhibition chain reaction which causes the bump to vanish, leaving thus the possibility for the bump to reemerge elsewhere (where the input is significantly higher), while this area is temporarily inhibited.

As expected, this improves the dynamics of the bump compared to the model of Amari. With this mechanism, the new field model can be successfully applied in driving the self-organization process. [Alecu 2011b, Alecu 2011a] present more detailed description of the behaviors of the Binp neural field.

However, Binp model can not implement a WTM policy in all cases. In particular, it allows for a specific undesired behavior to happen. Suppose that a bump has emerged due to a certain external stimulation. If the amount of this external stimulation is sufficient enough, the bump sustains itself due to this external stimulation but also due to lateral excitation (the levels of inhibition outside the bump are high in this case). However, in this situation, the field may become insensitive to the external stimulation received by other units in the map, even if they receive higher stimulation. As a result, instead of inhibiting the bump in the initial location and reemerging it in the new location that corresponds to the higher stimulation, the field maintains the bump location unchanged. While this effect does not seem to be statistically significant in the context of basic SOM self-organization, it becomes an important issue in the implementation of our learning mechanism, which deals with temporal sequences. In the first case, the field activity is used in learning only, thus, this undesired field behavior does not significantly affect the general process of self-organization, while in the second case of a context model, the activity of the field is used not only for learning, but also as a feedback signal. This undesired behavior of the Binp neural field in driving self-organization in context models using feedback leads to erroneous temporal contexts.

Given these considerations, in our work we use a new neural field model, presented below, which solves the issues encountered with the Binp model.

5.9.6 LISnf model

The problem in the Binp model is the self-sustaining of the bump in some situations, although the input stimulation elsewhere in the map can be higher from the input in the bump region. To address this problem, another model was proposed in our lab [1], called the lateral input sensitive neural field (LISnf). In this model, a unit takes into account not only the activity of connected units, but also the input stimulation of these units. Besides to solving this Binp self-sustaining issue, LISnf offers a faster competition mechanism, reliable enough to fit our purpose of driving self-organization by a parallel and local computation mechanism.

LISnf differs from the previous models of Amari and Alecu in that it doesn't use integral equations as will be shown next, instead, it deals with the discrete space of indexed units. This is justified by the motivation for which these models were developed: instead of modeling neural dynamics in

[1] By Hervé Frezza-Buet, to be published.

the brain, LISnf was designed for machine learning purposes. Consequently, instead of relying on continuum X, with LISnf relies on a discrete map space $\mathcal{A}$.

Let us consider the two units p and q in $\mathcal{A}$. We note $\mathcal{D}_R(p)$ the set of units included in a disk of radius R around the unit p. The cardinal of this set, i.e. the number of units within the disk is noted $|\mathcal{D}_R|$. The unit p is connected to the units $q \in \mathcal{D}_R(p)$ within the disk with excitatory connections that will be called on-connections.

Let us also define $\overline{\mathcal{D}}_R(p)$ as all the units in the map that do not belong to $\mathcal{D}_R(p)$. For a unit p, connections with the units $q \in \overline{\mathcal{D}}_R(p)$ are inhibitory connections that will be called off-connections. Connecting p to all units within $\overline{\mathcal{D}}_R(p)$ takes a lot of memory, hence it is connected to a subset of these units with a probability P. The set of off-connections are thus noted $\overline{\mathcal{D}}_R^P(p)$. Both the on- and off-connections are established at the design time and fixed thereafter.

The LISnf dynamics relies on using smoothed versions of the input i and the activity u, these versions are denoted $\bar{i}$ and $\overline{u}$, and computed locally by averaging the values of i and u read through the on-connections of the unit:

$$\bar{i}_p(t) = \frac{\sum_{q \in \mathcal{D}_R(p)} i_q(t)}{|\mathcal{D}_R|},\ \overline{u}_p(t) = \frac{\sum_{q \in \mathcal{D}_R(p)} u_q(t)}{|\mathcal{D}_R|} \tag{5.42}$$

From these preliminary definitions, let us define an ordering relation between a pair $(\bar{i}_p(t), \overline{u}_p(t))$ and the analog values $(\bar{i}_q(t), \overline{u}_q(t))$ at p and q respectively. A function $\text{Inf}_t(p,q)$ implements this order relation, it returns 1 when the pair at q is greater than a pair at p, otherwise it returns 0. The order relation (depicted in figure 5.12) is based on a lexicographic order with a priority given to i over u, and using a tolerance parameter $\varepsilon \geq 0$:

$$\text{Inf}_t(p,q) = \begin{cases} 1 & \text{if } \bar{i}_q(t) > \bar{i}_p(t) + \varepsilon \\ 0 & \text{if } \bar{i}_q(t) < \bar{i}_p(t) - \varepsilon \\ 1 & \text{if } \left|\bar{i}_q(t) - \bar{i}_p(t)\right| \leq \varepsilon \text{ and } \overline{u}_q(t) > \overline{u}_p(t) + \varepsilon \\ 0 & \text{otherwise} \end{cases} \tag{5.43}$$

The LISnf field equation is then a straightforward update rule, that is, as mentioned before, not derived from any differential equation as in Amari and Binp neural fields:

$$u_p(t+1) = [u_p(t) + \delta \Delta u_p(t)]_0^1 \tag{5.44}$$

with

$$\Delta u_p(t) = \left[1 - \alpha \frac{\sum_{q \in \overline{\mathcal{D}}_R^P(p)} \text{Inf}_t(p,q)}{|\mathcal{D}_R|} \right]^{\pm} \tag{5.45}$$

and $\alpha > 0$ and $\delta > 0$, and $[x]^{\pm}$ is a Heaviside function defined as $[x]^{\pm} = 1$ if $x \geq 0$, $[x]^{\pm} = -1$ otherwise, and $[x]_0^1$ is a saturation function with $[x]_0^1 = x$ if $0 \leq x \leq 1$, $[x]_0^1 = 0$ if $x \leq 0$, and $[x]_0^1 = 1$ if $x \geq 1$.

The LISnf model solves the problem encountered with Binp. Each unit within the LISnf field is aware of the activity of other units connected to it, but also of their input stimulation. So when

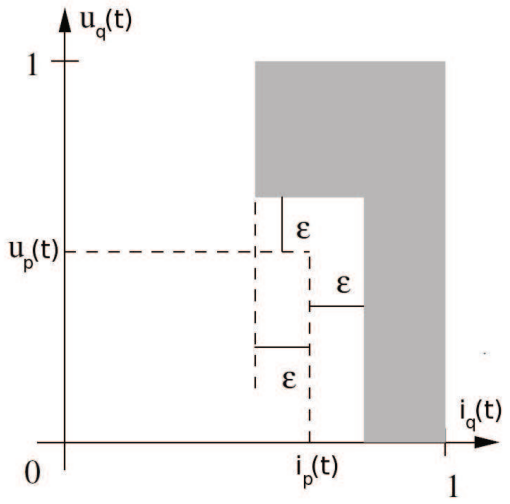


FIGURE 5.12: The order relation implemented by $\text{Inf}_t\,(p,q)$: The gray area represents the value of 1, while white areas represent the 0 value.

the average input stimulation of the units within the off-connections range are sufficiently high to suppress the already existing bump, there would be sufficient points $q \in \overline{\mathcal{D}}_R^P\,(p)$ with $\text{Inf}_t\,(p,q) = 1$. Thus, the value $\Delta u_p(t)$ in equation (5.45) computes to a negative value (and equal to -1). Consequently, this causes the reducing the value of $u_p(t+1)$ in equation (5.44) at the next timestep.

Moreover, due to the order relation defined in equation (5.43), the suppression of the bump can happen even if the average input to the distant units is almost equal (with a range defined by the tolerance ε) to the average input in the bump region, while their activity is larger (at least by ε) than that of the units in the bump region. This means that under some conditions related to the input, sufficient number of activity spikes in the field can be able to inhibit the already existing bump.

Indeed, unlike Binp, the LISnf neural field offers the reliable mechanism to drive the self-organization in our activity-based model for temporal sequence processing, that will be presented in the next chapter. In our simulations, we used the following numerical values the LISnf field parameters: $\alpha = 20$, $\delta = 0.333$, $\varepsilon = 0.02$, and $P = 0.3$.

5.10 Cellular computing with SOM and DNFT

DNFT offers a moderately complex yet powerful formalism which can be used to describe, design and simulate various neural phenomena. As shown above, it has been used with great success in different research works to propose new neural mechanisms, and is a valuable framework capable of describing neurophysiological observations at the mesoscopic level in various nervous structures.

In our research, we are interested in the computation mechanism of DNFT that we judge useful in attaining our goal: conceiving a self-organizing neural model for temporal sequence processing, having the properties that fit the general computational principles of cellular computing.

We have previously shown that the unsupervised computation of the existing SOM-based tem-

poral models do not fulfill the requirements of the cellular computing paradigm, basically because they require a central processor for their computation. We have also presented three different DNFT models, each adds some enhancements on the precedent. All of these neural field models can produce activity bumps that can substitute the neighborhood function used for learning in the basic SOM. As for the combination of a neural field with the SOM, this can be simply done by considering the computed match $E_p(t)$ of each unit p as the input $i_p(t)$ of the neural field.

As explained before, some neural fields fit the process of self-organization more than the others. In particular, the LISnf model is a fast and reliable model compared to Amari and Binp, and offers the desired performance that allows to account on its accuracy in activity-based self-organizing neural models, in which the computation of the map activity is crucial for the correctness of the computation.

Hence, using the LISnf neural field, the computation in the map becomes fully decentralized, as each unit can now compute whichever value based on the values read by its connections (that is the functional locality condition required for the cellular computing paradigm), and there is no more need for the central processor. As for resetting the activities of the weights of the neural field, our proposed model is autonomous in the sense that, when the computation starts, there is no need to reset whichever value.

Moreover, the computation within the neural field framework is parallel like in the basic SOM. As will be shown in the next chapter, our proposed model is implemented according to the `bijama` framework [Ménard 2005], in which equation (5.44) is applied to the map units asynchronously. As explained in 3.3.2, for each update cycle, all the units in the map are evaluated according to some random order, so the update of a unit p is visible to whichever unit q whose evaluation occurs after p. This asynchronous update regime is intended to bring stability to the competition mechanism. For LISnf , there is no buffering, no need to store the weights for the on- and off-connections, this allows for fast computation.

However, one should point out that the use of any of the DNFT models in guiding the self-organization process, is not without drawbacks. The standard self-organization process computes one -and only one- neighborhood function around the BMU. When conducting learning using the neural field mechanism, one should guarantee that there is one, and only one bump within the field. Thus, the emergence of a bump within the field should inhibit the emergence of other bumps elsewhere in the field. This is why, when LISnf was introduced, we considered the off-connections of a unit p to be with all other units in $\overline{\mathcal{D}}_R(p)$, that is, all the map units outside the on-connections region $\mathcal{D}_R(p)$. Later we avoided total connectivity by introducing a probability P that controls the number of the off-connections in order to save memory, so the unit p is now connected to P ($P = 0.3$ in our model) of the total units within $\mathcal{D}_R(p)$ (denoted $\overline{\mathcal{D}}_R^P(p)$ to refer to this partial connectivity).

Although connectivity is not total like in Hopfield networks, it is not that much convenient with the strict condition of topographic locality of the cellular computing paradigm that requires the connections of a unit to be in a bounded region around that unit, typically to its direct neighbors. Other conditions like parallelism, decentralization, functional locality are maintained within the framework of DNFT. However, as mentioned in 4.5.6 for Hopfield networks, the mitigation of the strict condition of topographic locality put by Sipper [Sipper 1998b], makes it possible to see such

models as compatible with the cellular computing paradigm.

Motivated by these considerations and encouraged by the previous success in applying DNFT to aspects closely related to our objectives, we decided to adopt this neural paradigm, namely the LISnf model as a computational foundation for our research work which we introduce next.

CHAPTER 6

A Self-Organizing Cellular Model for Temporal Sequence Processing

Contents

6.1 Introduction

So far, we have seen the difference between coarse-grain and fine-grain computation architectures, and what makes an architecture that carries out computation a cellular computer. We have also seen that artificial neural networks are fine-grain models that often lack the topographic locality and the decentralized properties of cellular computing structures, although, they are powerful models in processing temporal sequences.

Between the different temporal neural models, those implementing recurrence (through feedback connections) exhibit interesting temporal properties and a rich internal dynamics. This, besides to their adaptability, makes them adequate models for dynamical systems representation, modeling and control. Specifically, self-organizing maps were recently used in temporal processing,

and their different approaches for processing temporal sequences were presented in the past chapter. Besides, we introduced the DNFT framework and explained how they help in making SOMs conform with the cellular computing requirements.

The proposed temporal model is a fine-grain architecture for temporal sequence processing, it is a multi-map recurrent architecture that seeks to self-organize as a whole. As a fine-grain architecture that relies on a neural field for distributed computation, this temporal architecture becomes a cellular computing model, which is the objective of this research work.

In order to show the model ability in temporal sequences processing, it is applied to the extraction of a state representation of a dynamical system starting from an ambiguous stream of observations on that system. Before presenting the model, in the coming subsections, we get into the details of its computational constraints and architectural characteristics, and explain the example application that is presented in its context.

6.1.1 A fine-grain architecture

The proposed model is a parallel fine-grain computing architecture that is run on `InterCell` supercomputer, which is a coarse-grain parallel computing architecture that allows for high performance computation. Although coarse-grain, the `bijama` framework which is a fine-grain software dedicated for distributed computation, was run on `InterCell` in order to serve our research objectives. This hardware and software combination is used to implement and run a model with hundreds of cells, however, `bijama` running on `InterCell` allows for running larger-scale fine-grain systems.

Using such parallel hardware architecture to run the systems implemented in `bijama` software package for fine-grain system modeling imposes some constraints on these systems:

- First, the system is strictly connectionist, knowing that connections are the computer science modeling for the interactions between cells within a cellular model. The system consists of a population of units that interact with others using connections, units compute their output values by reading the outputs of the connected units and the external output, if there is any. The whole system is only defined by this connectivity and the units update rules, there is no central processor that supervises the system execution, and no global variables allowed.

- Second, the system is evaluated by the successive update of all its units, chosen in a random order. This means that the update regime is asynchronous.

- And third, the system is driven only by its own dynamics and the external input. After the systems is set out to run, there is no external intervention from the user that goes beyond monitoring the system evolution. This means that, unlike most of the previously seen adaptive neural models, in models implemented in `bijama` there is no separation between learning and exploitation or test phases, there is no reset to any system variable, nor tuning with time to any system parameters. The system is left to drive its dynamics spontaneously in reaction to the external inputs; it is strictly online.

It should be emphasized here that these constraints on systems implemented with `bijama` on `InterCell` are compliant with the properties of fine-grain models.

6.1.2 Distributed winner-take-most

With constraints imposed by `bijama` fine-grain framework, driving the self-organization of the basic SOM in the traditional way as defined by Kohonen is not possible.

In Kohonen SOM, as well as in recursive self-organizing models (like Rec-SOM [Voegtlin 2002]), the winner-take-most (WTM) learning policy requires that the learning rate of the prototype that matches the best the external input (the BMU prototype) be the higher. The learning rates of the prototypes of the BMU neighbors are also non null; they are less than that of the BMU and decrease with units distances from the BMU following a bell-shaped learning kernel (typically a Gaussian). It is this distribution of learning rates that implements the WTM competition learning policy (as already mentioned in 5.9). The application of such policy requires first, finding the BMU which is obtained by a global search procedure on the population level that iterates on all units. This requires the use of a central processor to carry out this procedure. Once the BMU is found, the application of the WTM policy requires, next, computing the learning rates of its neighbors within the bell-shaped kernel, then applying these learning rates to update the prototypes values. Both steps require the use of a central processor within the traditional SOM learning algorithm. Besides, both the search of the BMU and the computation of the learning kernel require the process to be synchronous, which is not the case of `bijama` framework

Within the `bijama` framework, the learning kernel should be computed in a distributed and asynchronous manner that fits its way of computation. The WTM policy of traditional Kohonen SOMs (also used with other recursive models) should be replaced with a distributed and decentralized WTM policy that can be computed and applied asynchronously. DNFT models as presented in the past chapter offer the convenient solution.

Using a neural field to implement the distributed WTM policy, the competition is set up via a dynamical process, as follows: each unit p in the map matches the external input $o(t)$ it receives at time t against its prototype ω_p, and computes a matching value $i_p(t)$. The learning rates distribution $\{u_p(t)\}_{p\in\text{map}}$ computed by the neural field forms a bump (similar to the bell-shape) centered at the position of the highest $i_p(t)$ (corresponding to the BMU). The selection of BMU within a pre-selected region (by the bump) reminds the mechanism of Hypermap (see 5.4.1).

As shown in the previous chapter, some neural fields fit the process of self-organization better than the others. Some neural fields are appropriate for simple self-organization (the case of one map) even though they form bumps that are not all the time centered around the BMU, but the self-organization process itself needs not necessarily such a reliable field. But in the case of a temporal model that includes feedback, such non-reliability can cause the imprecise computation to be amplified leading to an erroneous model behavior. We have shown in 5.9.6 that the LISnf neural field guarantees the necessary accurate behavior, and offers a fast and reliable competition mechanism compared to other neural fields like Amari and Binp.

We have also shown in 5.10 that using neural fields makes self-organizing maps meet the cellular computing properties as they offer the necessary decentralization and functional locality, and somehow sufficient topographic locality that allows to consider the self-organizing map a cellular model.

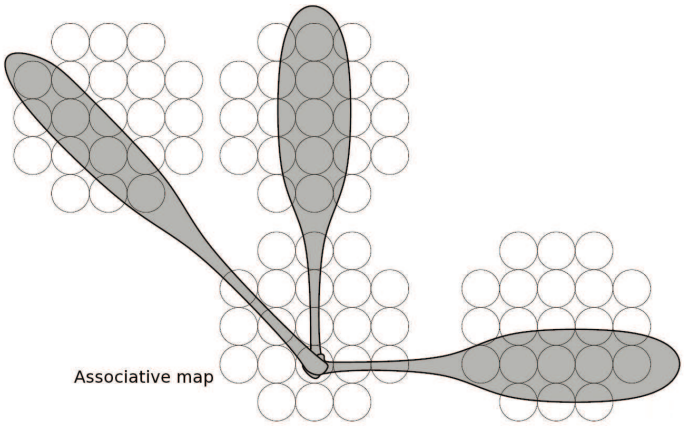


FIGURE 6.1: Multimodal architecture: three self-organizing maps that correspond to three modalities (sensory or motor), interact through an associative map. The partial connectivity has the form of strips; each unit receives connections from remote map units within the strip. Extracted from [Ménard 2005].

6.1.3 A multi-map architecture

The `bijama` framework was originally introduced by Ménard [Ménard 2005] and used for *joint-organization*. The latter signifies the self-organization of maps within a multi-map architecture in which maps are connected to each other, such architectures are normally used for multimodel simulations, like the sensory-motor coordination as in the same work by Ménard. In such architectures, each of the maps corresponds to some modality, either sensory or motor. However, multi-map architectures may also contain *associative maps*, that play the role of an intermediate medium for signal exchange between other maps. In such scenario, each map seeks to self-organize driven by its external inputs, hereafter called *thalamic inputs*, but also driven by inputs received from connected maps (their activity), hereafter called *cortical inputs*. This terminology borrows from the neuroscience of the cerebral cortex, in which mesoscopic neural units (also called micro-columns) receive input signals from the thalamic area as well as from similar units in the cortical tissue. Figure 6.1 illustrates a multimodel architecture that includes an associative map. In `bijama` , it is possible to combine thalamic and cortical inputs in one input that is the actual input to the competition mechanism (in our case, the neural field). The presented model implies a comprehensive illustration of `bijama` methodology.

For the purpose of joint-organization, the inter-map connectivity between the multimodal maps is implemented by connecting each unit in a local map to a set of units in one or more remote maps, this set may contain all the units of the remote map or maps. However, in his work, Ménard showed that partial connectivity may be sufficient for the joint-organization of the model maps. This aims to avoid the combinatorial explosion that comes with the total connectivity and thus, to reduce the computation load. So, the partial connectivity is implemented by connecting each unit in the local map to a set of units in the remote map that has the shape of a *strip*, identified by its width and direction as shown further.

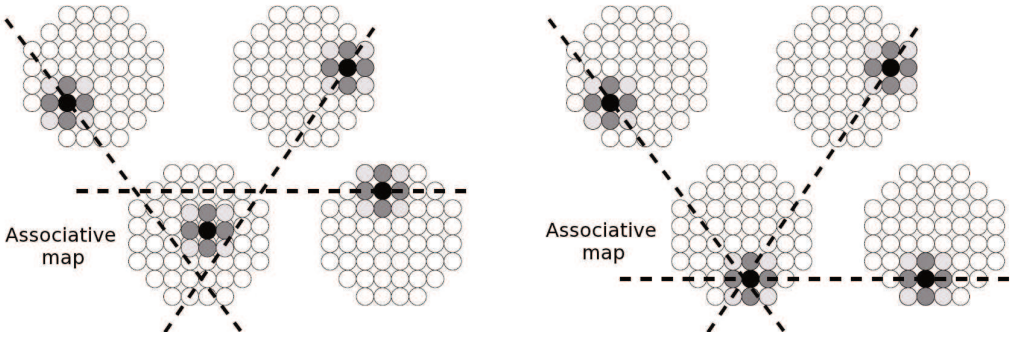


FIGURE 6.2: Activity binding in joint-organization. Left: transitory activity bumps that are not bound. Right: binded stable activity bumps align together. Extracted from [Ménard 2005].

This joint-organization as handled by Ménard is intended to find a concordance between the different map responses to their stimuli, that corresponds to a static task. Concordance is attained by *binding* together the sensory maps activity with the motor map activity through the activity of an associative map. At the start of the run, the activity bumps of the maps (in response to specific sensory inputs) can appear in random places. This situation corresponds to a non-stable activity, because the mutual excitation due to the maps interconnectivity, causes the bump in each map to excite a different region in the other map than the actual bump region, thus inhibiting it due to lateral competition (implemented by a neural field). For this reason, this situation is transient in the context of joint-organization, and the activity bumps tend to bind together, which is a stable situation. Binding the activity bumps means that they reach a position in their maps such that bumps are placed at connected places. Transient and stable activity bumps are depicted in figure 6.2.

This way, activity bumps that correspond to a specific sensory input are for example bound together with a motor activity bump. The binding of activities that corresponds to all input stimuli and motor map activity is the result of joint-organization.

6.1.4 A dynamical recurrent architecture

It has been shown in the past two chapters that several techniques were applied to artificial neural networks in order to encode time. The best method, regarding memory, computation time, and adaptability to varying statistical characteristics of the input temporal sequences (mainly their complexity), is turning neural networks into adaptive dynamical systems through adding recurrent connections. Hence, by their adaptive and dynamic structures, neural models can efficiently represent the temporal context of input sequences.

In specific, we have seen in the past chapter that the recent trend in building adaptive temporal self-organizing maps is by implementing recurrence, with RecSOM being a benchmark in the literature.

When the input to the temporal SOM-based model arrives online, it can be seen as a stream of

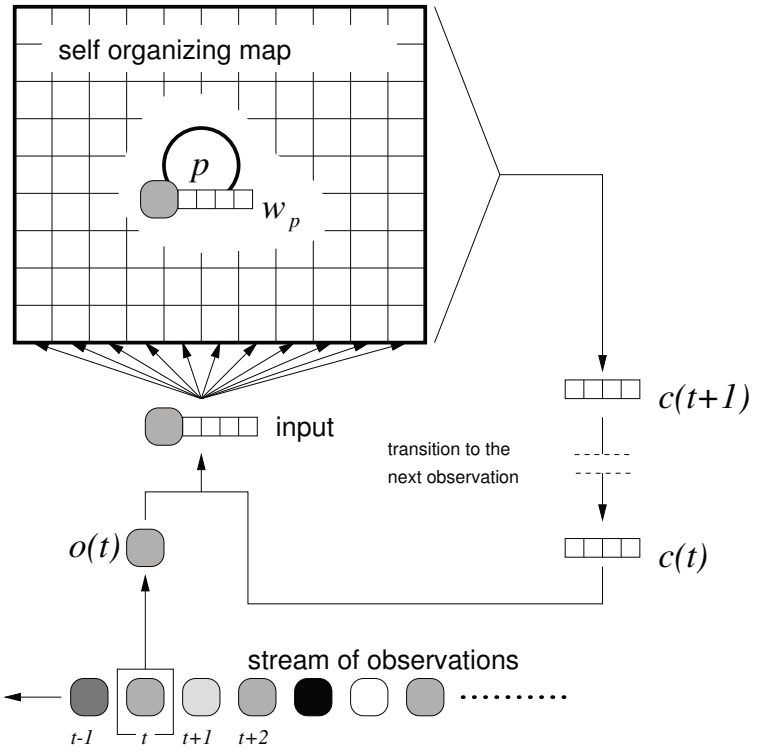


FIGURE 6.3: Recursive self-organizing map principle. At each time, $o(t)$ is received from the input stream. It is combined with a context value $c(t)$ in order to form the actual input to all units within the self-organizing map. The map contains units p, each having a prototype w_p comparable to the input. The activities of units i are used to form $c(t+1)$ with a process that differs between different approaches [Hammer 2004a] (see also the previous chapter). $c(t+1)$ becomes the context value for the next input.

values. As a reminder, the input ξ_t to such a model received at time t is made of two components: the external observed input $o(t)$ and a context value $c(t)$. The architecture is recurrent since $c(t)$ is computed from the state of the map at time $t-1$, when $o(t-1)$ was presented. It has been pointed out in the previous chapter that these recurrent architectures are called recursive when they handle the feedback signal in the same way as the external input signal. Figure 6.3 illustrates the work-flow of RecSOM, but also the workflow of all other recursive self-organizing architectures.

Similarly, the proposed model consists also of a recurrent architecture, moreover, it is also recursive as it handles the feedback signal in the same way as the external input. In fact, the model architecture implements two levels of recurrence, on the map level, and on the architecture level. Some maps in the architecture are computing their activity by performing competition via a neural field. The mechanism of neural fields itself is recurrent as pointed out in the previous chapter. However, the architecture implements another level of recurrence, the connectivity between the architecture map forms a recurrent path as well. As will be shown further, the map connectivity implements a form of feedback that reinjects the activity of a specific map into its dynamics after being processed by other maps.

Indeed, the proposed architecture is a recurrent neural network that shares a common architectural and functional features with some of the previously seen neural models. For example, the proposed model can be regarded as a cellular implementation of RecSOM; it is a recursive model that uses the neural field paradigm in order to compute in a local and decentralized way, although, it is able to implement a competition and cooperation mechanisms. However, although recursive, it is a self-organizing architecture that implements vector quantization on a temporal stream of inputs.

This work builds on the previous research efforts in our lab. It bases on `bijama` framework introduced by Ménard, but also on the research of Alecu in dynamic neural field theory, and goes beyond the latter work to using LISnf neural field that is sensitive to neighbor units inputs. The proposed model makes use of both works and introduces an architecture that is able to process complex temporal sequences and to adapt in such a way that follows their changes. This architecture does this in an autonomous way, moreover, it is cellular, asynchronous, and unsupervised.

6.1.5 Example sequence processing: ambiguity resolving and representation

In order to show the utility of the proposed model in processing temporal sequences, we will introduce it by an example that corresponds to a practical problem encountered in some application domains. For instance, in POMDP (partially observable Markov decision process) problems, where there is an agent that interacts with its environment, the agent observes the environment and performs actions back in the environment. The observations perceived by the agent can be regarded as a stream (or sequence) of inputs, and the subsequent actions it takes can be regarded as a sequence of actions. In such problems, the environment is partially observable by the agent, i.e. the observations perceived by the agent may not be sufficient to determine the exact state of the environment, hence, to determine the action to be taken.

Although of this partial observability, the agent has to estimate the actual environment state, and based on this estimation, to take the appropriate action. However, the estimation of the actual state is not straightforward, because sometimes, the agent receives identical observations that correspond to different environment states, hence they require the agent to perform different actions. This ambiguity in the stream of observations corresponds to a complex sequence, and the agent should resolve the observation ambiguity before taking any action.

As shown further, the proposed model is able to resolve the ambiguity of observations, it receives a stream of observation from the environment, and uses them to build up a representation that assigns to each observation (even repeated or ambiguous) a distinct representation. This can be thought of as building a representation not to the observations themselves, but to the underlying environment states. The model, being a neural architecture for temporal sequence processing, does this by distinguishing identical observations in the input stream by their temporal contexts.

However, by building such representation, the model does not exactly simulate the agent in a POMDP problem. In such problems, the changes in the environment happens due to the actions of the agent performed in the environment, but in the case of the proposed model, there is no feedback to the environment. Thus, the model can be thought of as the part of the agent that builds an internal representation of the environments states that disambiguates the observations.

Although, the model exhibits more flexibility than needed by typical POMDP agents that interact with an environment with a predefined behavior: the model can re-adapt its representation

whenever the defined behavior of the environment changes. When the environment is thought of as a dynamical system, the change of the environment corresponds to the change of an evolution function of the dynamical system.

As the proposed model does not perform actions in the environment, we will handle the environment as an autonomous dynamical system. The actual state of the system at time t is denoted $x(t)$. The evolution of the state through time is controlled by an evolution function noted ϕ_t, thus: $x(t+1) = \phi_t(x(t))$. The interaction between the agent and the dynamical systems is one way, the agent is only observing the system state and gets a stream of observations $o(t) = O(x(t))$ through time (as in figure 6.4-a).

For the sake of illustration, let us consider an example autonomous dynamical system. Figure 6.4-b depicts a wheel divided into gray-scale colored sectors. Each sector is labeled by a letter depending on its color. The wheel is turning counter clock-wise. In this example, we define the actual state $x(t)$ of this dynamical system as the discrete angle of a fixed point on the disk circumference relative to an outer reference point as illustrated in the figure. This system is autonomous, in that the state changes are not driven by any external action.

The agent observes the values of a variable related to the state, this observation is the color (also expressed as a letter) at some fixed position (the observation is denoted $o(t)$ in figure 6.4-b). In this example, the sequence of observations is periodical: as the wheel turns, the same observation sequence is repeated, it is the sequence $CDCEABEFCCFEDCBA$ in the actual case of the figure. Obviously, this sequence of observations is ambiguous, or complex. To illustrate this ambiguity, let's consider the color labeled A. In one wheel turn, this color can be observed in two different time instances $o(t1) = o(t2) = A$, corresponding to two different system states (wheel angles) $x(t1) \neq x(t2)$. In this case, if the agent observing the wheel is required to take distinct actions for each distinct state $x(t1)$ and $x(t2)$, then the observation A will not be sufficient to infer the system state.

In the case of ambiguous observations illustrated in the above wheel example, the architecture is expected to extract two different representations $\hat{x}(t1) \neq \hat{x}(t2)$ corresponding to $x(t1) \neq x(t2)$ although $o(t1) = o(t2)$. The proposed model, which is an unsupervised cellular one that consists of several modified self-organizing maps interconnected in a recurrent architecture, initiates a representation of the system state space over the surface of one of its maps, so that each state x is projected to some specific position $\hat{x}$ on that map. In further map representations, the observation o associated to each $x/\hat{x}$ is shown by the color assigned to the position $\hat{x}$ (see gray-scale units in the map on the right in figure 6.4-b).

Let us recall that the evolution function ϕ is time-related, as depicted in figure 6.4-a. During further experiments, the arrangement of wheel colors is changed during the learning, i.e. the evolution function ϕ is suddenly modified. This allows for testing the ability of recurrent self-organizing process to cope with non-stationary dynamical systems, which corresponds to a non-stationary POMDP problem as well.

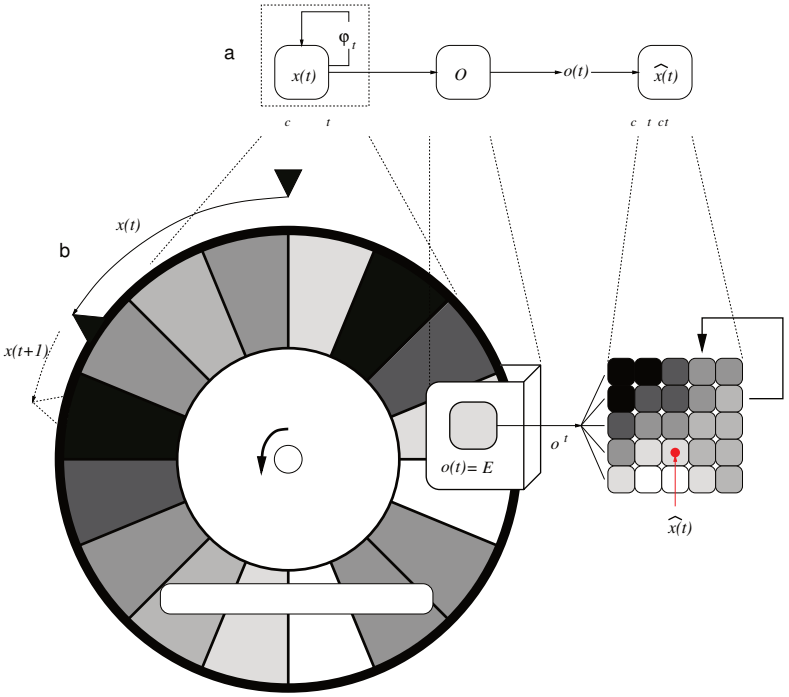


FIGURE 6.4: Dynamical system state extraction. a) A schematic of an autonomous dynamical system. b) An example of an autonomous dynamical system; a turning wheel exhibiting ambiguity when observed. The system state is the angle of rotation, and the observation is the color of the angular sector at some fixed position (also referred to by a letter) . The observer gets identical observations for different states, i.e ambiguous observations.

6.2 Architecture

Having presented the preliminary background on the distributed fine-grain nature of models built using `bijama` framework, the cellular nature of self-organization with neural fields, and the dynamical properties of neural models involving recurrent connections, we are now ready to introduce our cellular model for temporal sequence processing. We choose to present it in the context of the example application of dynamical systems that requires processing ambiguous sequences. The rest of this section delves into describing the model architecture.

The architecture is multi-map, it consists of three interconnected maps that form a recurrent pathway aiming to capture the temporal context of the inputs, i.e. the observations of the autonomous dynamical system. Considering the temporal context of each observation should help distinguishing it from identical observations occurring at a different temporal context. The architecture extracts a representation of the states of the observed dynamical system. This representation is built on the surface of one of the architecture maps.

We start by giving names for the three maps, they are the *input* map, the *delay* map, and the *associative* map (see figure 6.5). Roughly speaking, the input map is the one on the surface of which the state representation will emerge. The other two maps play an intermediate role in information

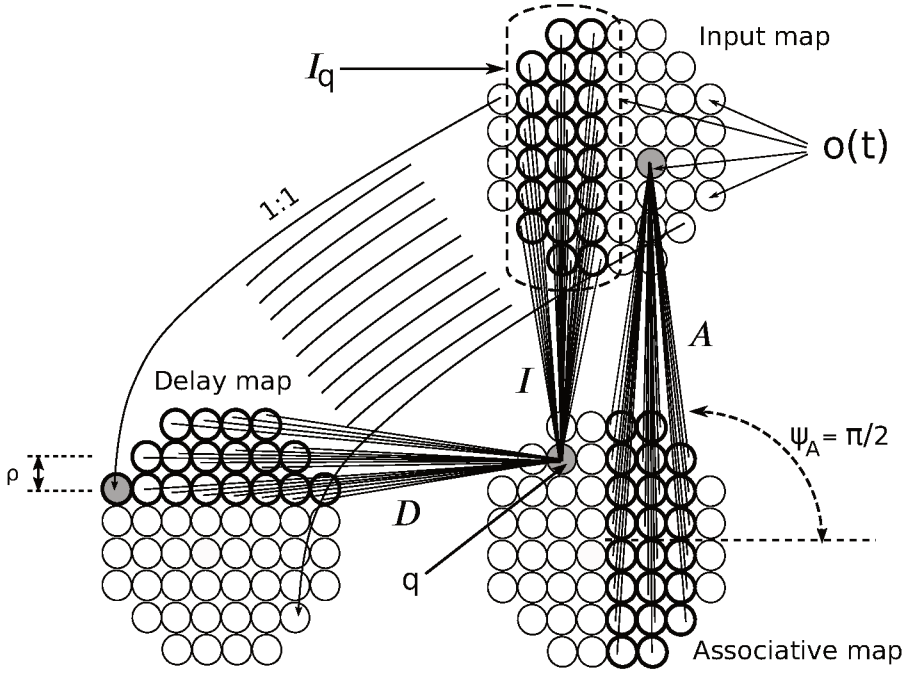


FIGURE 6.5: The model architecture. Input map and delay map are connected via one-to-one connections, other connections are one-to-many connections organized in strips. For a unit q, $\mathcal{I}_q$ is the strip of kind $\mathcal{I}$ handled by q.

flow through the recurrent pathway: the delay map maintains a faithful delayed copy of the input map, and the associative map is an intermediate medium through which the actual and the past state of the input map interact. The activity of the associative map is determined by the input and delay maps, and its activity is reinjected to the input map dynamics. All maps are partially interconnected. Interconnection is two-way only between the input and associative maps. Input and associative maps compute their activity via LISnf neural field, while the activity of the delay map is obtained by simple copy and delay. The input map implements a modified version of Kohonen map, while the associative map computes its activity driven by the activity of the other two maps as explained further.

Concerning the interaction with the dynamical system, it is one way, the architecture receives the observation stream as inputs, no signal returns to the environment. The input map receives at time t the external input $o(t)$ and builds on its surface a spatial coding $\hat{x}(t)$ for the input observation. The dynamics of the input map is driven by both the external input and the associative map activity. In particular, the activity of the associative map is the one that delivers the necessary information about the temporal context of the actually processed input, and offers the necessary signal to bias learning in such a way that the formed representations consider the history of the actual input.

The update of architecture units (including all maps units) is carried out using an asynchronous update strategy; at each update cycle (corresponding to one time step), all the units of the architecture are updated in a random order.

The exchange of information in the architecture is carried out through connections between the

architecture units. Connections are two-type, intra-map connections and inter-map connections. Intra-map connections are necessary for the implementation of the neural field mechanism in a distributed architecture, these connections can be of on- or off-connection types as pointed out when presenting LISnf neural field in 5.9.6. Inter-map connections, that we also call *cortical connections*, are those forming the recurrent pathway between the architecture maps, they are in the form of strips as illustrated in figure 6.5.

The neural field uses the on- and off-connections types to perform distributed lateral competition in order to compute the units activities within a map. The activity of neural fields tend to agglomerate in different map regions, these agglomerations have the shape of a bell or a bump (see the dark meshes in figure 6.7). For the purpose of self-organization, the field should be parametrized in such a way that gives place to the emergence of a single activity bump, in order to be a valid alternative to the neighborhood function. In the proposed model, the LISnf neural field is only acting in the input and associative maps, and it is parametrized in such a way that this requirement is met.

The activity bump that the neural field computes is actually the response of the map to its inputs whether they are external (like thalamic inputs) or internal (like cortical inputs) to the architecture. However, the input to the neural field could be a value computed on the basis of several input types received by the map units (as explained further). When an activity bump emerges in some region in a map, the units in that region are the ones to which learning is applied. Learning occurs in the units prototypes (the case of input map), and in the inter-map connections. Let us emphasize again that driving self-organization by neural fields reveals to be difficult as explained in [Alecu 2011c].

As mentioned before, inter-map connections are arranged in strips. Each unit in some local map is connected to a set of units in the remote map. A local map unit with position p in the two-dimensional map space is connected to a subset of units in the remote map, the units within this subset of remote map units are located within a strip-shaped region in that map (see figure 6.5). For now, we note this strip region with the generic notation $\mathcal{S}_p$, which signifies the existing connections between the unit at the position p in the local map with the units at positions $q \in \mathcal{S}_p$ in the remote map. The set of all strips connecting the units of the local map with the units of the remote map are denoted by $\mathcal{S}$.

Each cortical connection within a strip, that connects a local unit p with a remote one $q \in \mathcal{S}_p$ handles a weight whose current value is $\bar{s}_{pq}(t)$, this weight is referred to as *cortical weight*. The strip $\mathcal{S}_p$ owned by p handles a vector of cortical weights $\bar{S}_p(t) = (\bar{s}_{pq}(t))_{q \in \mathcal{S}_p}$.

Strips are characterized by their width and direction. The width of a strip is expressed by its half width $\rho_{\mathcal{S}}$. The direction $\psi_{\mathcal{S}}$ of a strip refers to the angle of the axis connecting the centers of the local and remote maps relative to the horizontal axis (see figure 6.5). We note the activity of a unit at the position p at time t as $u_p(t)$, and the vector of remote unit activities perceived by p through the strip $\mathcal{S}_p$ as $S_p(t) = (u_q(t))_{q \in \mathcal{S}_p}$.

Now that the generic definitions are presented, we can specialize the generic notation of the strip according to the remote map name, so in figure 6.5, the notation $\mathcal{S}$ is replaced by $\mathcal{A}, \mathcal{I}, \mathcal{D}$ according to the initial of the remote map (the name of the map where the information originates).

In `bijama` framework, units activities are computed using a layered stack of computational modules that defines the functional behavior of units. In the proposed model architecture, units that belong to some map have the same stack composition. Each module in the stack handles a scalar

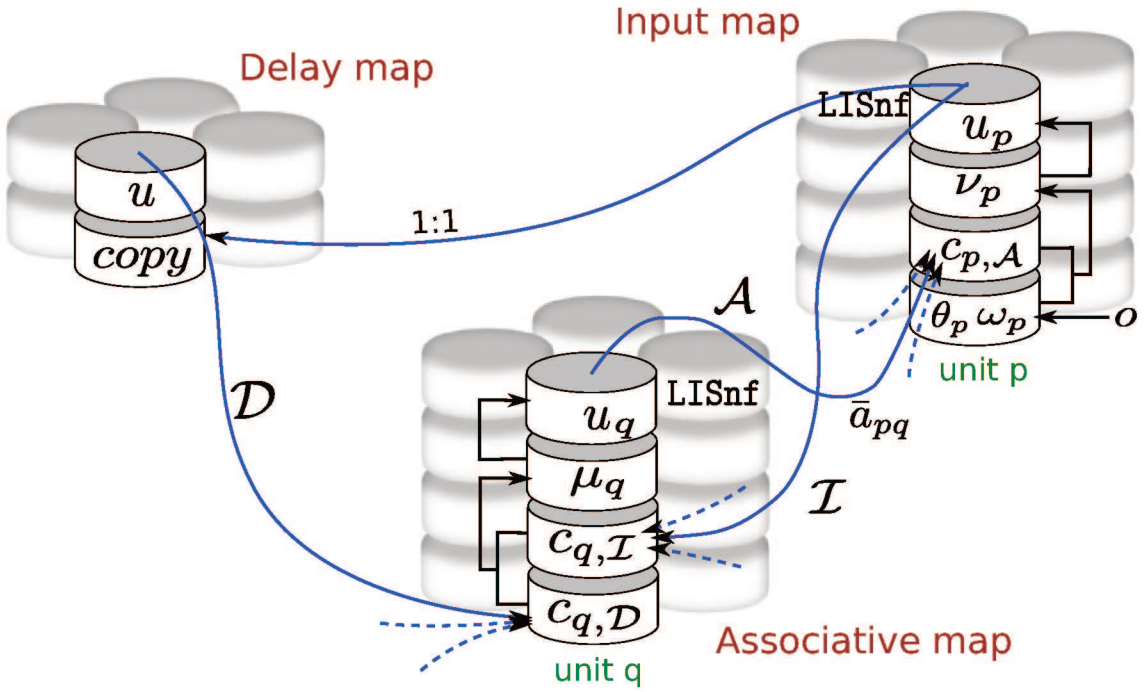


FIGURE 6.6: The layered organization of the three maps computation within `bijama` framework.

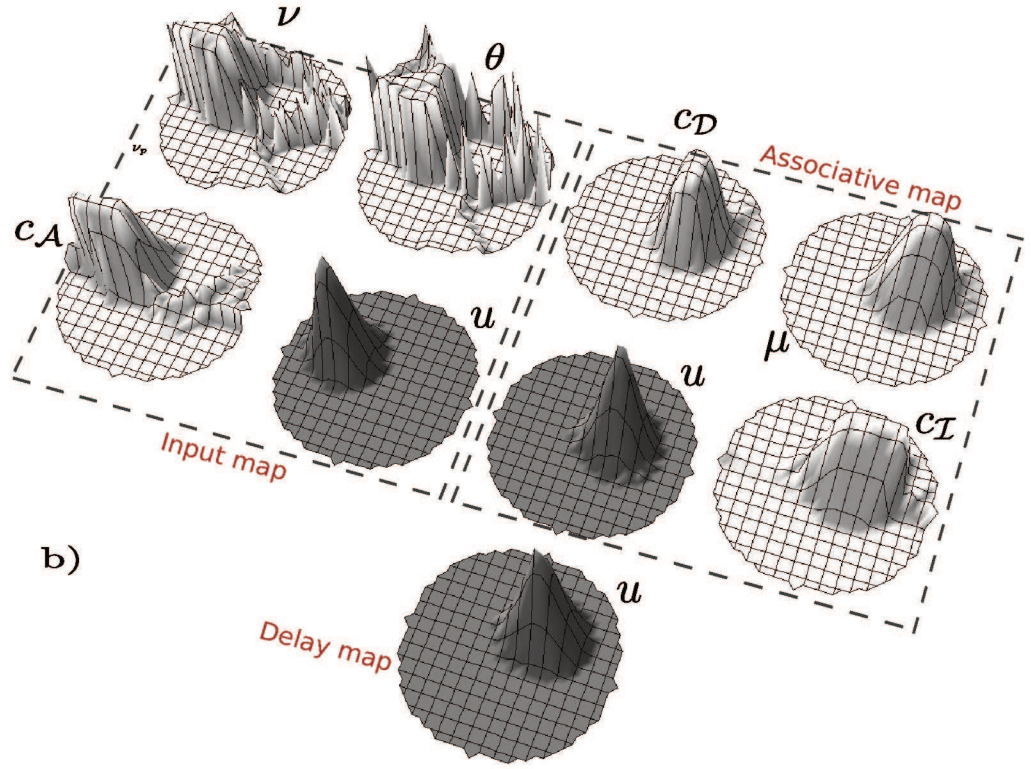


FIGURE 6.7: A snapshot of all layers activities of all maps in the architecture.

value that is either received as an input or computed on the basis of scalars in lower modules in the unit stack. The dependency between modules follows the ascending order, i.e. each module can compute its scalar on the basis of lower modules scalars. With this flexible stack building method, it is possible to define the behavior of a unit with a reasonable freedom, adjusting the number of modules in the stack to the need, but always respecting the ascending modules dependency. The stack of modules for the three maps units and their interconnectivity within the architecture is illustrated in figure 6.6.

The uppermost module of each unit stack is the one holding the unit activity $u_p(t)$. When remote maps read the activity of the local map units, they actually read this module of the unit stack, i.e. this is the module accessed by cortical connections in order to read the unit activity. When a map computes its activity by a neural field (case of input and associative maps), thus, the uppermost module that handles the unit activity is the neural field module.

In the case of the proposed architecture, the neural field module implements the LISnf algorithm with the input i to the neural field (refer to 5.9.6) being the scalar handled by the previous module in the unit stack. As shown in figure 6.6, $i = \mu$ for the associative map and $i = \nu$ for the input map. Both values are not pure inputs, they are computed as combinations of other inputs to the map units.

Let us now describe the composition of the stack of modules for each of the architecture maps, and the computation carried out by each stack module. Figures 6.6 and 6.7 help reading the descriptions that follow.

A unit in the input map receives two inputs, the external input is the elements $o(t)$ of the stream of observations, and the internal input is delivered by connections within the strips $\mathcal{A}$ originating from the associative map. The activity of the input map units is computed starting from these two inputs. The LISnf neural field uses a computed combination of these inputs and computes the activity of the units that form an activity bump in some localized map region. The latter will be shown in the experiments to correspond to a valid representation $\hat{x}(t)$ of the dynamical system state $x(t)$.

The lowermost module in the stack of a unit p in the input map handles the external input $o(t)$. As the external input is called thalamic input, this module is referred to as the *thalamic module*. The thalamic module matches its received input $o(t)$ against a stored prototype $\omega_p(t)$ also called the *thalamic prototype*, and computes the matching value $\theta_p(t)$, also called the *thalamic matching* which decreases with the distance between the compared values:

$$\theta_p(t) = \exp\left(-\frac{(o(t) - \omega_p(t))^2}{2\sigma^2}\right) \tag{6.1}$$

The value of thalamic matching is kept in the range $[0, 1]$, this is also the case for all maps stack modules. Both the values of the thalamic prototype $\omega_p(t)$ and the computed matching $\theta_p(t)$ are stored in the thalamic module. Each computed value of each module in all the units stacks within a map form an activity distribution over the map surface. Figure 6.7 shows the activity of all modules in all maps. In that figure, the computed matching $\theta_p(t)$ for all units of the input map form the thalamic matching distribution noted θ.

The next module in the stack of unit p in the input map is called the *cortical module*. This mod-

ule handles the strip $\mathcal{A}_p$ originating from the associative map. Similarly, it computes the matching value $c_{p,\mathcal{A}}(t)$ between the weight vector $\bar{A}_p(t)$ of all connections within the strip and the vector $A_p(t)$ of remote units activity in the associative map that are within the strip connections. The matching, called the *cortical matching* is computed as follows:

$$c_{p,\mathcal{A}}(t) = \frac{\left\langle A_p(t).\bar{A}_p(t)\right\rangle}{\max\left(\left\|\bar{A}_p(t)\right\|^2, B\right)} \tag{6.2}$$

with B a numerical constant. Let us notice here that a similarity measure based on the Euclidean distance is not a good candidate to compute the cortical matching. If both $A_p(t)$ and $\bar{A}_p(t)$ are zero vectors (due to initialization for example) this will result in a maximum cortical matching value, whereas, it is not actually meant to be an actual matching.

The third module in the input map stack computes a scalar that is a merging of the thalamic and cortical modules values; it merges $\theta_p(t)$ and $c_{p,\mathcal{A}}(t)$ into one scalar $\nu_p(t)$, which is called *cortico-thalamic merging*:

$$\nu_p(t) = \sqrt{\theta_p(t).\left(\beta + (1-\beta).c_{p,\mathcal{A}}(t)\right)} \tag{6.3}$$

with β a constant that calibrates the participation of thalamic and cortical matchings in their merge. Figure 6.8 shows the variation of $\nu_p(t)$ according to $\theta_p(t)$ and $c_{p,\mathcal{A}}(t)$.

The idea behind this rule is that the activity perceived by the cortical connections is not sufficient to activate the unit. A similar principle in neuroscience can be found in Grossberg ART Model for visual attention [Grossberg 1976], which says that the input from remote cortical regions in the brain is not sufficient to activate the cortical unit, but when a primary input exists then cortical inputs can modulate the unit activity. It should be stressed at this point that we are not concerned with any similarity with the brain neuroscience, as our approach is related to computational cellular structures for temporal sequence processing. Instead, we explain the absorbing nature of the thalamic activity in the cortico-thalamic merging from the perspective of temporal processing: essentially, the activity of the associative map that is delivered to the input map via cortical connections is intended to deliver a signal related to the temporal context of the actually processed input, thus it is expected to bias the activity bump within an active thalamic region, but, for the same reason, it should not in any way be sufficient to activate a region where the thalamic activity is null.

The value of $\nu_p(t)$ forms the input of the neural field module, which is the uppermost module in the stack of the input map units. This module computes the unit activity $u_p(t)$ by executing lateral competition between the map units in the way explained in 5.9.6. Thus, this module is referred to as a *competition module*. The units activities are computed following the LISnf field equation in distributed and asynchronous way as explained before.

LISnf behaves in such a way that the map regions that have low thalamic matching θ (see equation (6.3) and figure 6.8) will have low neural field inputs $i = \nu$ even if they have high cortical matching c. At a time step t, the activity $u_p(t)$ of the map units will be distributed in such a way that they have a maximum in the region of the two-dimensional map space where the external input $o(t)$ matches the best the stored prototypes $\omega_p(t)$, i.e. in the region having the higher $\theta_p(t)$. The participation of the cortical matching in the cortical merging is still important though, because

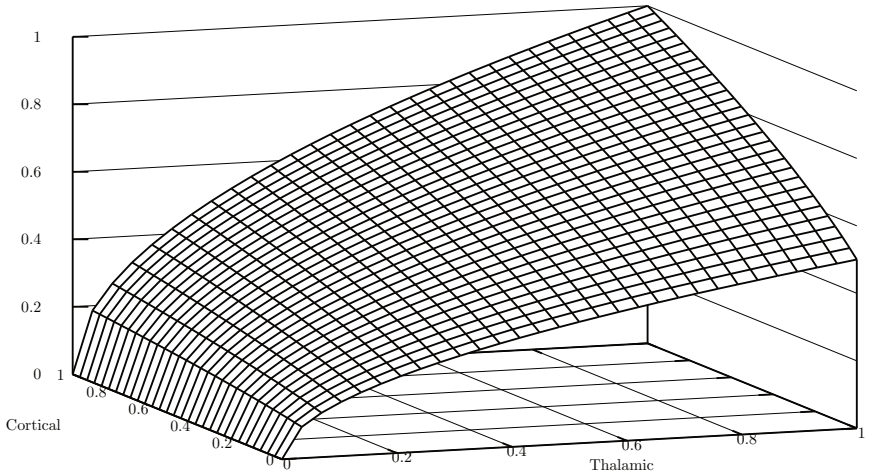


FIGURE 6.8: Cortico-thalamic merging for $\beta = 0.25$.

it determines the bump position within the region that has the best matching $\theta_p(t)$.

Due to lateral competition, the activity of the neural field follows the shape of a bump even if the input distribution to the neural field module has not a bump shape, although, the bump follows the higher input activity. For example, in figure 6.7, although the input to the neural field (the activity distribution ν) does not have the shape of a bump, the lateral competition executed by the neural field resulted in a bump of activity around the higher input values (the activity distribution u). When the bump of activity is computed, it is used to implement the WTM policy of the self-organization algorithm. The activity value $u_p(t)$ of a unit p computed by the neural field is used to modulate the learning of the thalamic prototype of unit p. The learning moves $\omega_p(t)$ towards the input $o(t)$ proportionally to the unit activity value $u_p(t)$:

$$\omega_p(t+1) = \omega_p(t) + \alpha_\omega . u_p(t) . (o(t) - \omega_p(t)) \tag{6.4}$$

with α_ω a fixed thalamic learning rate, the same for all units within the map. As u has the shape of a bump, only the units surrounding the best matching place actually learn.

Besides to adapting the thalamic prototypes, the computed unit activity is also used to adapt the cortical weights within the strip that the unit handles. For each connection within the strip owned by a unit, learning implies moving the cortical weight $\bar{a}_{pq}(t)$ of a cortical connection towards the connection input which is the activity $u_q(t)$ of the unit in the remote map accessed by the connection:

$$\bar{a}_{pq}(t+1) = \bar{a}_{pq}(t) + \alpha_{\mathcal{S}} . u_p(t) . (u_q(t) - \bar{a}_{pq}(t)) \tag{6.5}$$

with $\alpha_{\mathcal{S}}$ a fixed cortical learning rate for all the connections within the architecture. The previous learning rule for the cortical connections implies that learning occurs only in connections to active units in the local map; if $u_p(t)$ is null then no learning occurs in the cortical connections within the strip owned by this unit. Another remark is that the learning mechanism of cortical weights in equation (6.5) is similar to that of thalamic weights in equation (6.4) which is respon-

sible of the self-organization of thalamic weights in Kohonen maps [Kohonen 2001]. The weight $\bar{a}_{pq}(t)$ is moved towards the connection input $u_q(t)$ (originating in the remote map) proportional to the activity $u_p(t)$ of the unit in the local map. This means that what happens to the weights of cortical connections within each strip resembles to what happens to the prototypes in the basic SOM.

The second map in the architecture is the associative map, it receives the actual activity of the input map as well as its delayed activity maintained by the delay map. Both activities are delivered by strips to the associative map. This neural field in this map performs lateral competition to compute its activity which has also the form of a bump. This activity is then reinjected to the input map via the strips $\mathcal{A}$ as mentioned before.

The lowermost module in the stack of a unit q in the associative map is a cortical module that handles the strip $\mathcal{D}_q$ originating in the delay map. The next module is a cortical module that handles the strip $\mathcal{I}_q$ originating from the input map. Both modules compute their matching $c_{q,\mathcal{D}}(t)$ and $c_{q,\mathcal{I}}(t)$ respectively, in the same way as given by equation (6.2).

The third module in the stack of a unit in the associative map merges the scalars in lower modules, and is referred to as *cortico-cortical merging*. It computes the merging $\mu_q(t)$ of the cortical matchings $c_{q,\mathcal{D}}(t)$ and $c_{q,\mathcal{I}}(t)$ as follows:

$$\mu_q(t) = \sqrt{c_{q,\mathcal{I}}(t).c_{q,\mathcal{D}}(t)} \tag{6.6}$$

The value $\mu_q(t)$ is actually the input to the upper module, which is the LISnf neural field module that computes the unit activity $u_q(t)$. When $u_q(t)$ is computed it is then used to modulate the learning of the cortical weights within the strips $\mathcal{D}_q$ and $\mathcal{I}_q$ in the same way as given by equation (6.5). There is no thalamic learning in this map as it does not receive an external input.

The last map in the architecture is the delay map, it receives a copy of the activities of the input map units via one-to-one connections (not strips) and delays them for a specified period T of time. Its delayed activity is injected into the associative map dynamics via the strips $\mathcal{D}$. The stack of a unit p in the delay map has two modules. The first module is called the *copy* module, it copies its activity $u_p(t)$ from the activity of an input map unit $u_q(t)$, where p is a position in the two-dimensional space of the delay map, and q is the same position in the input map two-dimensional space. The second module in the stack is called the *FIFO* module which implements the delay. The FIFO module contains a T-length FIFO queue that exposes its input on its output after T time steps. Thus $u_p(t) = u_q(t - T)$ for each unit p in the delay map and its corresponding unit q in the input map. By means of this delay mechanism, the activity of the delay map at time t is the same as the input map activity at time $t - T$.

Having presented the model architecture, it can be noticed that the architecture parameters are independent from which dynamical system or observation stream it processes, thus it is a model free architecture. After initialization, the architecture parameters (including learning rates) are left unchanged during learning even if the input stream changes. This means that, unlike some other SOM-based models, there is no need for decaying the learning rates, or the width of the learning kernel, as usually done in models based on self-organizing maps [Carpinteiro 1999, Miikkulainen 2005b] (and the previous chapter). This statement, in addition to the fact that there are no imposed initial

conditions on the values of thalamic and cortical weights (they are initialized to random values in the range $[0, 1]$) means that if the architecture succeeds to set up a representation for a specified dynamical behavior, then it can succeed to re-adapt and set up another representation when the system dynamics changes. This confirms that the architecture is system-independent, moreover, it tells that it can cope with non-stationary dynamical systems.

6.3 Experiments

The experiments aim to test the capability of the proposed cellular self-organizing architecture in temporal sequence processing. The architecture is tested in the particular example of extracting a valid representation for the states of a dynamical system. We also test the adaptation of the representation extracted by the architecture to the change in the evolution function ϕ_t of the dynamical system. The proposed model is also compared to the RecSOM [Voegtlin 2002] reference algorithm, which has been implemented for the sake of this comparison.

As shown in the further experiments, both temporal models are able to build a representation of the dynamical system state based on their ability to consider the temporal context of the inputs. This ability is rooted in two properties for both architectures: the self-organization that assigns different representations for distinct input values on the map surface, and recurrence that differentiates them and ensures assigning different representations to ambiguous observations that have the same value $o(t)$ but correspond to different system states by considering their different temporal contexts (recall that ambiguous observations corresponds to distinct $x(t)$).

The experimental scheme presented next starts by verifying the self-organization capability of the fine-grain and cellular implementation of the basic SOM algorithm. The next experiment tests the capability of the proposed architecture to perform self-organization while considering the temporal context of the inputs. This experiment shows how the proposed architecture is able to build up a representation of a dynamical system state even though its receives ambiguous observations as inputs.

Later we compare the behavior of the proposed architecture to the behavior of RecSOM for the same inputs. An experiment simulates the case of a non-stationary dynamical system by changing the evolution function ϕ_t, and shows how both architectures can reorganize to find another representation that fits the new situation. Some other experiments aim to compare the properties of the short-term memory and stability of both the proposed architecture and RecSOM.

6.3.1 Notations and representations

Before introducing the experiments results, let us explain some issues that concern all the introduced experiments. In this section we explain how observations are sampled from the dynamical system, and how they are introduced to the self-organizing architecture. It is also necessary to explain how the representation built up by the architecture and RecSOM is visualized and interpreted.

In order to facilitate the visualization of the architecture behavior, the thalamic prototypes $\omega_p(t)$ are coded in gray-scale colors (as figure 6.9-b). Values that are close to 0 are assigned the black color and values close to 1 are assigned the white color. The colors on the map are used to represent the prototype values for each unit p in the map. When the maps is self-organized, organization

appears as a continuous shade over the map surface. Regions of the map that have nearly uniform prototypes define what we call a *thalamic region*. For example, there are two wide thalamic regions (black and white) in figure 6.16, while figure 6.10 (bottom-right) shows a map tiled with six thalamic regions.

The dynamics of the system is somehow subtle to represent. Consider the state of the dynamical system at a time instance τ to be $x(\tau)$, the observation of the state is then $O\left(x(\tau)\right)$. This observation is the external input to the input map in the architecture (as depicted in figure 6.6, on the right, the observation is the input to the thalamic module). The neural field mechanism requires a sufficient time for the activity bump to form, and a sufficient time to adapt the thalamic prototypes and the cortical weights during the learning process. For this reason, the input $O\left(x(\tau)\right)$ is maintained for several time steps. We found experimentally that when using the LISnf neural field mechanism, a value of $T = 24$ time steps is enough for bump formation to reach a steady state, then for learning to occur. This is an improvement over the previous neural fields like Binp, that needed $T = 100$ time steps for the same phenomena to occur [Alecu 2011c], thus LISnf is quite fast as compared to the Binp neural field.

So, in order to maintain the same input for a T time steps period of time, in all the coming experiments, τ is incremented each T time steps. During this period, the same input $O\left(x(\tau)\right)$ is presented to the input map, this can be expressed as follows: $o(t) = o(t+1) = \cdots o(t+T-1) = O\left(x(\tau)\right)$ and $o(t+T) = o(t+T+1) = \cdots o(t+2T-1) = O\left(x(\tau+1)\right)$, etc., where $o(t)$ is the input for the input map at time t. In order to have the whole architecture parts computing consistently the same input, it is necessary to choose the T value to be the same as the length of the FIFO queue (in the FIFO module) of the delay map, i.e. the delay should correspond exactly to the duration T between two successive observations. In this notation, the time variable used for the transitions in the dynamical system which was noted t in figure 6.4, is now noted τ, since t is now used to refer to tinier time steps used in the architecture evaluation.

Recall what was mentioned in 6.1.5 that the input stream to the architecture is the repetition of a periodic sequence of sampled observations, that is, a repetition of the sequence $O\left(x(\tau)\right)$, $O\left(x(\tau+1)\right), \cdots, O\left(x(\tau+n-1)\right)$ where n is the sequence length.

In order to obtain a meaningful representation of the map response (activity bumps) to the input stream, the positions of the successive activity bumps are plotted on the map surface. So, we plot the response to the successive inputs $O\left(x(\tau)\right)$, $O\left(x(\tau+1)\right)$, $\cdots$, $O\left(x(\tau+l-1)\right)$ with $l > n$. We trace l positions that represent the map response to a larger number than the length of the repeated sequence elements. This is not necessary though, but it aims to visualize the stability of the map response through several repetitions of the sequence. The trace is obtained as follows: for each time τ (at the end of each T time steps), the barycenter of the activity u of all map units is computed and plotted on the map surface, since at this time, the neural field is expected to find a stable bump for each input $O\left(x(\tau)\right)$. The barycenter is computed as follows:

$$G(\tau) = \sum_{p \in \text{input map}} u_p(t) \cdot p \Big/ \sum_{p \in \text{input map}} u_p(t) \tag{6.7}$$

It can be noticed that as the activity of the map has the shape of a bump, then, the computed barycenter for all the map units corresponds to the position of the bump center.

As revealed by the representations extracted in the coming experiments, the computed barycenter represents the map state $\hat{x}(\tau)$ at time τ (that is shown on the right in figure 6.4-b). From the stream of such $\hat{x}(\tau)$ positions at each transition of the dynamical system occurring at time τ, it is possible to trace the recent positions at each time τ. The trace is made of the last l computed barycenters ($l = 50$ in all the experiments), organized in a list $P(\tau) = \{G(\tau - l + 1), G(\tau - l + 2), \cdots, G(\tau)\}$. They are drawn on the map surface in the form of a l-length path depicted in the coming figures as a red poly-line (see figure 6.9-b for example).

The experiments in this section are all run with the same numerical values. In these experiments we simulate the case of sampling noisy observations of the dynamical system. The actually observed value $O(x(\tau))$ (presented during T time steps) is the value represented by a gray-scale color, and corresponds to a letter in the sequence : $A = 0$ (black), $B = 0.2$, $C = 0.4$, $D = 0.6$, $E = 0.8$ and $F = 1$ (white). The noise $noise_o$ is added to the input value, but the result after noise addition is kept in $[0, 1]$. Noise is sampled from a uniform random distribution $\mathcal{U}[-0.05, 0.05]$. Concerning the coming experiments in which we simulate a non-stationary dynamical system, the change of the system evolution function occurs after the presentation of $\tau(s) = 25000$ inputs.

The numerical values of the architecture parameters are fixed to the following in all the experiments: the map radius is $R = 15$ units for all maps, $u_p(0) = 0$, $\omega_p(0), \bar{a}_{pq}(0), \bar{i}_{pq}(0), \bar{d}_{pq}(0)$ are initialized to uniform random values from $\mathcal{U}[0, 1]$, $\sigma = 0.07$, $\alpha_\omega = \alpha_\mathcal{S} = \gamma/T$ (see γ, T below), $B = 10$, $\beta = 0.25$, $\rho_\mathcal{I} = \rho_\mathcal{A} = \rho_\mathcal{D} = 3$. $\psi_\mathcal{I} = 90°$, $\psi_\mathcal{A} = -90°$, $\psi_\mathcal{D} = 0$, $T = 24$, and $l = 50$. The competition module is implemented by the LISnf neural field equation, with the following parameters (as mentioned in 5.9.6): $P = 0.3$, $\delta = 0.333$, $\varepsilon = 0.02$, and $\alpha = 20$.

As we announced before, the RecSOM algorithm is used for the sake of comparison with the proposed architecture. As explained further, its parameters are set such that it behaves in a similar way to the distributed architecture. The numerical values of RecSOM parameters are given here but their meaning is explained later, they are: $\lambda = 10$, $\eta = 0.1$, $\gamma = 0.1$, and $\Omega = 5$.

6.3.2 Distributed winner-take-most self-organization

We start our experiments by testing whether a distributed implementation of the basic SOM algorithm can also self-organize in the same way as the basic SOM introduced by Kohonen [Kohonen 1997]. This implies testing if the winner-take-most (WTM) policy can be realized in a distributed way using the LISnf neural field. For this purpose, we built a distributed version of SOM using `bijama`. The distributed SOM architecture is similar to the architecture presented in figure 6.6, but the latter is reduced to the input map alone.

When a single input map is needed, the two layers corresponding to the integration of the associative map influence are removed (both the cortical and the cortico-thalamic merging modules are dropped), so that the input map consists of the thalamic module and the neural field module on the top of it (figure 6.9-a).

Although SOM self-organization is known to be difficult to obtain from usual neural fields equations [Alecu 2011c], figure 6.9-b shows that LISnf was able to drive the self-organization to obtain one similar to the topology-preserving self-organization that can be obtained by the basic SOM algorithm (colors that correspond to close prototypes values are close on the map surface in figure 6.9-b). In the same figure, there are 6 different thalamic regions (corresponding to thalamic

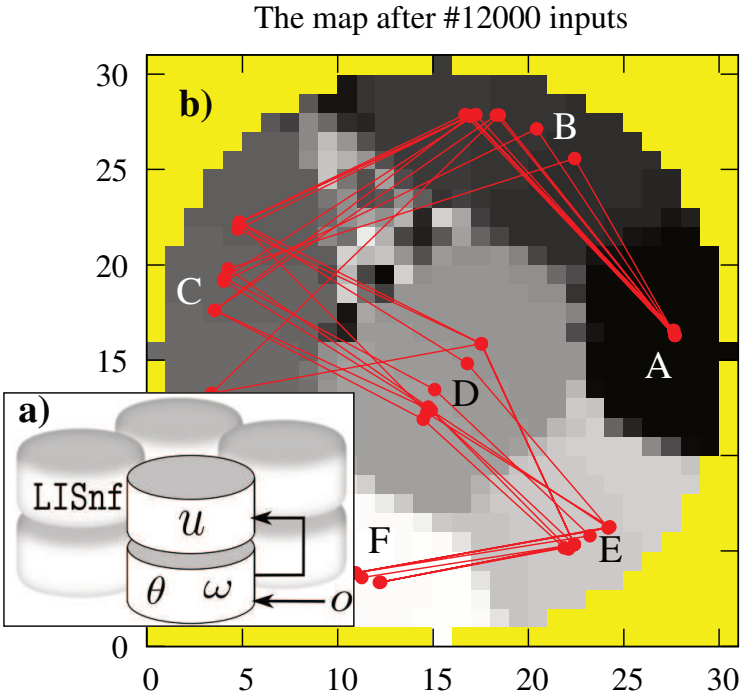


FIGURE 6.9: **a)** Module structure of a basic SOM input map. The module representation is the same as in figure 6.6, and the equations of the modules are the ones detailed in section 6.2. **b)** Distributed WTM self-organization. The gray-scaled values correspond to prototypes $\omega_p(t)$, and the poly-line traces the succession of the $l = 50$ last bump positions in the map. Each point of the poly-line is labeled with the input value presented at that time. The repeated input sequence is the ambiguous $S_1 = ABCDEFEDCB$ sequence.

prototypes) colored with 6 different grayscale colors. This reflects that the input consists of 6 different values: from A to F. The trace of the map activity (bump position) reveals that the input goes from A to F, then from F to A (the input sequence is $S_1 = ABCDEFEDCB$). However, there is no clear distinction between values occurring in each direction although the sequence is ambiguous. For example, within the C region, there is no distinction between the C observation received when the observation stream goes from A to F and the C observation received when it goes from F to A. This is because the map is inherently not able to consider the temporal context of inputs, just like the basic SOM, and is not related to the distributed implementation of the algorithm.

6.3.3 Disambiguation of observation stream

The result of the previous experiment is that the basic SOM can not differentiate input values by their temporal context, instead, it assigns close input values to clusters of points in the same thalamic region, which is not adequate for processing ambiguous temporal sequences. Now we test the proposed recursive architecture (explained in 6.2 and depicted in figures 6.5 and 6.6) that uses the principle of recurrence to see if it is able to consider the history of input values, or their temporal context, to affect assigning representations for each of the stream elements that correspond to the

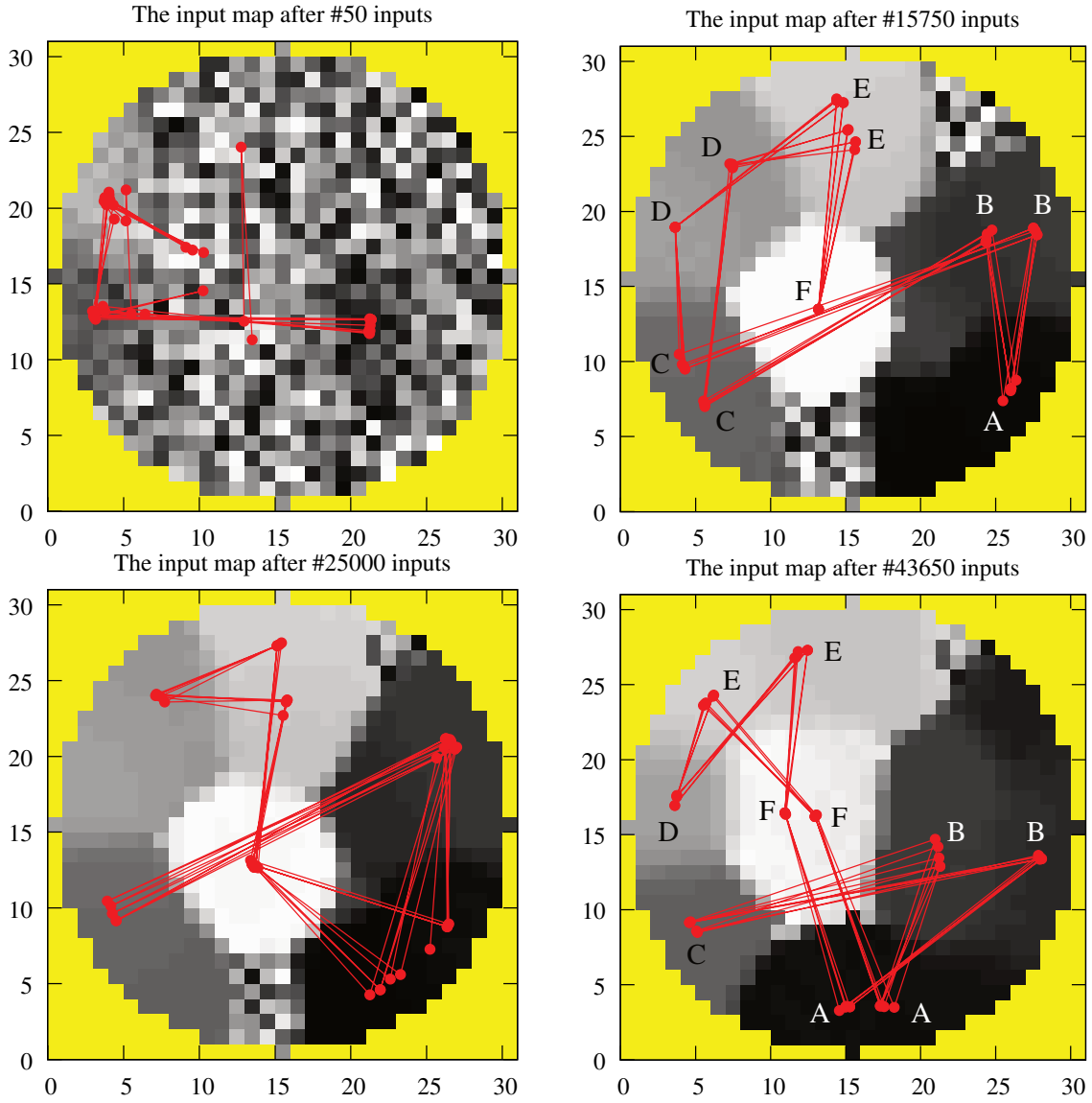


FIGURE 6.10: Status of the input map during the system evolution as a response to two input sequences. The two higher figures represent the map response on the first ambiguous input sequence $S_1 = ABCDEFEDCB$. The two lower figures represent its response on the second ambiguous input sequence $S_2 = ABCBAFEDEF$.

dynamical system observations.

For this experiment, the input is the repetition of the input sequence $S_1 = ABCDEFEDCB$ which is the ambiguous one used with the past experiment, and noise is added to the inputs as well. Let us recall that in the architecture, besides to the thalamic prototypes, the weights of cortical connections within the strips are also adapted continuously during the experiment.

Figure 6.10 (top-left) shows the state of the input map at the experiment start. The thalamic prototypes $\omega_p(t)$ random initialization is still visible, and the random poly-line indicates that there

is no useful representation at the experiment start. After the architecture processes sufficient repetitions of the input sequence, the spatial self-organization of thalamic prototypes occurs as shows the figure 6.10 (top-right).

Let us analyze this resulting representation: each thalamic region in the top-right figure corresponds to a range of observation values. Each region contains the representation of one or two observations that correspond to one or two system states, respectively. Consider for example the black thalamic region that corresponds to observations close to 0. It contains the representation of one system state that corresponds to the observation A, this observation is not ambiguous. The two points marked D express non successive system states corresponding to the same observation value, but occurring in different temporal contexts. By following the poly-line, it can be verified that the order of D representations is conform with its order in the input sequence S_1; it is once preceded by C and once by E. The ambiguities of other observations B, C, E are also resolved and two different state representations are assigned in each thalamic region. The result is that, the architecture resolved the ambiguity of input values, and correctly assigned them to different representations according to the input history of each input value. Differently speaking, the architecture took the input observation stream and built up a representation that assigns them non-ambiguous representations, a representation that maps to the states underlying these observations, although the observations are ambiguous.

Unlike the representation of the experiment in 6.3.2, the built up representation does not only consider the observations values, but also considers their temporal context, and the clusters of dots (the vertices of the poly-line) map one-to-one to the underlying states of the dynamical system. Hence, we can say that the representations $\hat{x}(\tau)$ built up by the architecture on the input map surface can be regarded a bijective mapping to the dynamical system states $x(t)$, thus, the resulted representation is indeed a valid state representation.

It should be mentioned that the duplication of state representations corresponding to the same value $o(\tau)$ is formed progressively while the whole architecture gets organized. A "split" phenomenon happens during the organization that makes the clusters of dots (observation values representations) concentrated around one position split into two separate clusters of dots concentrated around two distinct positions on the map surface.

Again, this happens due to the influence of the recurrent pathway in the architecture that contains a delay mechanism. Recurrence with delay is responsible for implementing a form of short-term memory that helps considering the temporal context. The context information is delivered by the cortical connections within the strips, so that they bias the bump position within each thalamic region in the input map depending on the history of this map activity. This allows to obtain multiple bump positions within the same thalamic region, not just two, as shown in a further experiments.

6.3.4 State representing of non-stationarity dynamical systems

The continuity of the previous experiments concerns investigating the architecture response to a change in the evolution function ϕ_t of the dynamical system, which corresponds to the simulation of a non-stationary dynamical system. The change of the evolution function can be simulated by a gradual or sudden change in the input stream of observations. In this experiment we dramatically change the observation stream during the architecture run, after having set up the first state

representation.

As already mentioned, this experiment is a continuity to the previous one. During the previous experiment, the input stream is switched from repeating the sequence $S_1 = ABCDEFEDCB$ to repeating the sequence $S_2 = ABCBAFEDEF$. This occurs at time $\tau = \tau(s)$, while always adding noise to the inputs.

As mentioned in 6.3.3 that the sequence S_1 is ambiguous, this is also the case for S_2. For example, in S_2, the input value A is preceded once by F (as the sequence is presented periodically) and once by B so it is ambiguous, while C and D are not, because both C and D are always preceded by the same subsequence of values.

Figure 6.10 (bottom-left) shows the state of input map immediately after switching from S_1 to S_2. The past organization of the map which was fitting S_1 does not fit S_2 anymore. For example, the input F which was corresponding to one point representation in the map, is now ambiguous in S_2, thus it should be assigned to two distinct representations as it occurs in two different temporal contexts. This exactly what happens, figure 6.10 (bottom-right) shows that the architecture seeks another organization and finds another representation that forms a new correct mappings to the dynamical system states corresponding to the sequence S_2.

Let us detail two cases. The inputs A and F which were not ambiguous in S_1 are ambiguous in S_2, so in the new organization, the architecture assigned two distinct representations to these input values. During re-organization, the point clusters corresponding to A and F on the map surface split after the sequence S_2 is presented. Contrarily, the inputs C and D which were ambiguous in S_1 and were assigned two distinct representations are not ambiguous anymore in S_2. The re-organization assigned each of them only one representation. Contrarily to the split phenomena, another phenomena happens in this case; during the architecture evolution, the previously duplicated representations merge.

As a result of this experiment, the proposed architecture was able to build up a representation that maps to the state of the dynamical system, but was also able to re-adapt to follow the system non-stationarity when the transition function ϕ_t changes, and set up a new correct representation. It is convenient to notice that switching from S_1 to S_2 is done without any architecture re-configuration or any further parameter resetting, thus the architecture is strictly online.

6.3.5 Comparison with RecSOM

In order to better judge the behavior of the proposed architecture we compare it with the RecSOM model [Voegtlin 2002], usually used in literature as a reference recursive self-organizing model.

RecSOM was introduced in 5.5.3 with a unified notations for SOM-based models in the previous chapter. Although, RecSOM equations are rewritten with a notation conformity with the proposed architecture in order to reflect the existing similarity between the two models. For the purpose of comparison equity, RecSOM parameters are set so that its behavior resembles the most to the proposed architecture. During this revision, we take the opportunity to explain how inputs are introduced to RecSOM, and how its evolution is computed, besides to discussing its representation issues.

The structure of RecSOM is similar to figure 6.3, the context information $c(t)$ is the map activity

at the previous time step delivered by feedback connections. A RecSOM unit p receives two inputs; the external input vector $o(\tau)$ which is compared to a feed-forward weight $\omega_p(t)$ (or prototype), and the activity vector of all units in the map at the previous time step which is compared to a weight vector $\bar{A}_p(\tau)$. The activity of a unit p is a matching value computed as follows:

$$\nu_p(\tau) = \exp\left(-\lambda \left\|o(\tau) - \omega_p(\tau)\right\|^2 - \eta \left\|A(\tau-1) - \bar{A}_p(\tau)\right\|^2\right) \tag{6.8}$$

with $A(\tau)$ the vector of all activities $\{\nu_q(\tau)\}_{q\in\text{map}}$ of the map units.

In this experiments, the RecSOM parameters λ and η are set such that their respective contributions in the exponential are of the same order, and so that the computed matching $\nu_p(\tau)$ has a sensitivity similar to what is obtained from equations (6.3) (and (6.1), (6.2) as well, indeed equation (6.8) is equivalent to that three equations).

Like in the basic SOM, the BMU is defined as the unit that minimizes $\nu_q(\tau)$, noted $k(\tau) = \text{argmin}_{p\in\text{map}}\nu_p(\tau)$. The learning rules used to update the feed-forward and the recurrent weights are:

$$\Delta\omega_p(\tau) = \gamma h_{pk(\tau)}(o(\tau) - \omega_p(\tau)) \tag{6.9}$$

$$\Delta\bar{A}_p(\tau) = \gamma h_{pk(\tau)}(A(\tau-1) - \bar{A}_p(\tau)) \tag{6.10}$$

with h_{pq} is a decreasing function of $d(p,q)$, the latter is the Euclidean distance between the positions of units p and q.

In his experiments, Voegtlin [Voegtlin 2002] used a narrow neighborhood function h, so that the learning of feed-forward and recurrent weights occurs only in the winner unit. This implements a winner-take-all policy, also called *hard competition*, rather than a WTM policy (thus called *soft competition*). This results in K-means-like clustering of the inputs rather than self-organization. For the sake of appropriate comparison of RecSOM with the proposed architecture, we slightly modified the original neighborhood function used in [Voegtlin 2002], we replaced the original exponential function h_{pk} by an arc of cosine. The chosen function is null everywhere except within a limited radius Ω, simulating by this, the activity bump (the dark gray distributions in figure 6.7) computed by the neural field in the proposed model. This choice is also more practical than using an exponential as it avoids applying extremely small learning rates on far units from the BMU, thus reduces the computation load. So, h_{pk} is defined as follows:

$$h_{pk} = \begin{cases} \cos(d(p,k)\frac{\pi}{2}/\Omega) & \text{if } d(p,k) \leq \Omega \\ 0 & \text{if } d(p,k) > \Omega \end{cases} \tag{6.11}$$

with $\Omega = 5$ is the bump width set such that it fits the experimentally observed bumps width in the proposed architecture. For the sake of comparison equity also, the learning rate γ is chosen to be equal to the accumulated effect of the learning rates in the proposed architecture during T time step, Indeed, $o(\tau)$ and $o(\tau+1)$ are separated by T time steps in the case of the proposed architecture, but with only one time step in RecSOM. Thus we put $\gamma = \alpha_\omega T$ with α_ω the thalamic learning rate defined in equation (6.4).

In his experiments, Voegtlin [Voegtlin 2002] applied parameter decaying, but in our experiment

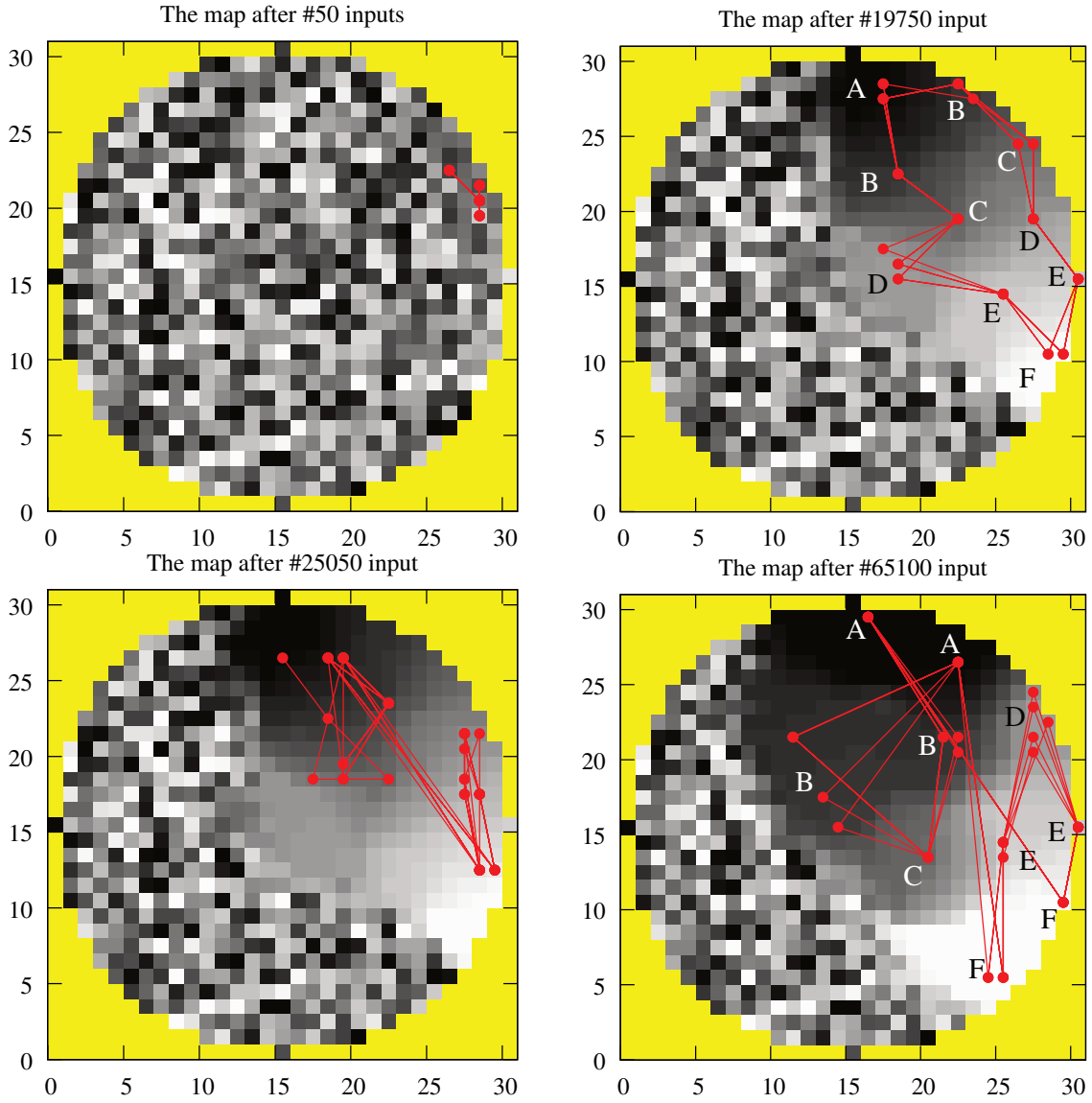


FIGURE 6.11: Status of RecSOM during the system evolution as a response for both input sequences. The higher two figures represent the map response on the first input sequence $S_1 = ABCDEFEDCB$. The two lower figures represent its response on the second input sequence $S_2 = ABCBAFEDEF$.

we deal with RecSOM in the same way as we dealt with the proposed architecture in sections 6.3.3 and 6.3.4. In this experiment, RecSOM is tested with S_1 and S_2 like in those sections, without parameter resetting, and with noise as well.

The representation in this experiment is extracted by the same logic as in past experiments, except that there is one difference in the case of RecSOM, that is, new inputs are introduced to the map at each time step, because in RecSOM, the soft competition is not driven by a neural field that requires few time steps to relax to a steady state, instead it is carried out using a central processor.

Thus, the representation is extracted in the same way explained in 6.3.1, by simply putting $T = 1$.

The results of sequence learning using RecSOM is shown in figure 6.11. Figure 6.11 (top left) shows the initial state of RecSOM. As can be seen, the feed-forward weights $\omega_p(t)$ are initialized to random values as well. Figure 6.11 (top right) shows the map state when the representation corresponding to S_1 observation stream is extracted. It is analog to figure 6.10 (top right) of the proposed architecture. This figures shows that, similarly to the proposed architecture, RecSOM was able to set up a representation that resolves the observation ambiguity and form a bijective mapping to states of observed dynamical system.

Figure 6.11 (bottom-left) shows the map state immediately after switching to the next input sequence S_2. Here, and similarly to figure 6.10 (bottom-left), the past organization that was fitting S_1 does not fit S_2 anymore. Figure 6.11 (bottom-right) shows that RecSOM was able to re-organize and find another representation that forms a new correct mappings to the dynamical system states corresponding to the observation sequence S_2.

To set it clear, the result from this experiment is that RecSOM was able, like the proposed architecture, to extract a correct representation that maps to the dynamical system state, and able to track the non-stationarity in the dynamical system, and set up a new correct representation when the dynamical system transition function ϕ_t changes.

One additional issue to discuss about this experiment results from comparing figure 6.11 with figure 6.10. It could be noticed that all the map surface in the proposed architecture was recruited during the self-organization process, while in RecSOM, only a part of the map was recruited. The representations of different values of inputs tend to be farther in the proposed architecture than in RecSOM due the neural field mechanism.

As pointed out earlier, the internal dynamics of feedback SOM-based models is not clear yet, we are not sure of the source of the difference in surface recruitment. Although, we experimentally watched that the competition driven by the neural field in the proposed architecture tends to explore the available input map surface before arriving to a stable representation, whereas RecSOM, searches to find a sufficiently correct representation and then stops exploring. The origins of the exploration tendency watched in the proposed architecture, could be related to the effect of the positive feedback to the input map, that is known in control theory to result in instability, interpreted here as exploration. Anyway, in a limited neural population, if the exploration itself is not limited, it could be a source of instability, the latter issue is discussed in the next section.

6.4 Representation and stability issues

Experiments in the past section have shown that the proposed architecture was able to process a stream of observations on a dynamical system and set up a representation that maps to the underlying system states, coping with observations ambiguity. They have also shown that this recurrent self-organizing system can adapt its dynamics with the changes of the evolution of the systems dynamics.

However, the proposed architecture is itself a dynamical system. Contrarily to the observed dynamical system, which is autonomous, the architecture is not, because its dynamics is affected by an external input, which is the stream of observations of the modeled dynamical system.

As a dynamical system (and a complex system that exhibits an emergent self-organizing behavior), the architecture exhibits a complex dynamics compared to the simplistic example dynamical system, because first, it is non-autonomous, and second, because it is adaptive and online, which makes its state subject to a more radical change. The architecture dynamics becomes simpler when it reaches a stable representation. However, representation stability is not always guaranteed as shown in the coming experiments, and in light of the complexity of the architecture dynamical system and the difficulty of understanding recursive self-organizing architectures, formal justification is hard to find, hence qualitative justifications are proposed.

Recurrence, that turns the architecture to be a dynamical system, endows it with the short-term memory mechanism which, as explained in 4.2.3, is characterized by two measures: depth and resolution.

The following subsections tackle the short-term memory and stability issues. The forthcoming experiments use the same numerical values of model parameters as in the previous experiments, except that inputs are presented without noise (i.e. $noise_o = 0$) hereafter.

6.4.1 Depth and resolution of the short-term memory

The presented architecture shares with RecSOM and many other recurrent neural networks, the principle of recurrence with one time step delay $(\tau - 1)$ (see figure 6.3), that corresponds to the time separating the presentation of two successive inputs. Contrarily to what one may intuitively think, these models can represent a temporal context with a depth larger than one, because recurrence allows the model to take into account the direct past state of the model, but the latter also includes information about past model states as well. Thus, having a one step delay does not restrict the architecture to consider only the previous input as a temporal context. An example about that is the actual experiment; when a sequence of observations $AAAAAFF$ is presented, the fifth element A is characterized by the precedence of four As before, and the first F by the precedence of five As.

Snapshots of the behavior of the proposed architecture are shown in figure 6.12. It can be seen that only one region representing all As, and other region for all Fs are emerged. Then, each region splits to create new state representations, that split in their turn in order to fit the number of ambiguous observations in each region and assign them the adequate state representations. Subsequent stages of split are illustrated in figure 6.12. The RecSOM algorithm behaves similarly concerning representation splits, however, it has a deeper short-term memory. In order to test the memory depth, we show the results of experiments carried out on sequence lengths where the proposed architecture starts to reach the limit of its representation power . Figure 6.13 shows the representation set up by RecSOM for sequences containing 7 and 8 successive As. The representations set up by the proposed architecture for the same sequences are shown in figure 6.14. Figure 6.14 (right) shows that the architecture started to suffer in representing the sequence with depth 8, unlike RecSOM in figure 6.13 (right) which represents it correctly. This figure is captured as the best snapshot resembling to a correct representation. In this very experiment, the depth of short-term memory seems to be sufficient to capture the temporal context, but the resolution is not sufficient to correctly represent it. This is because the architecture have already distinguished 8 different clusters, but didn't cluster them correctly.

Just to mention, in our experiments, we found that the implemented RecSOM can con-

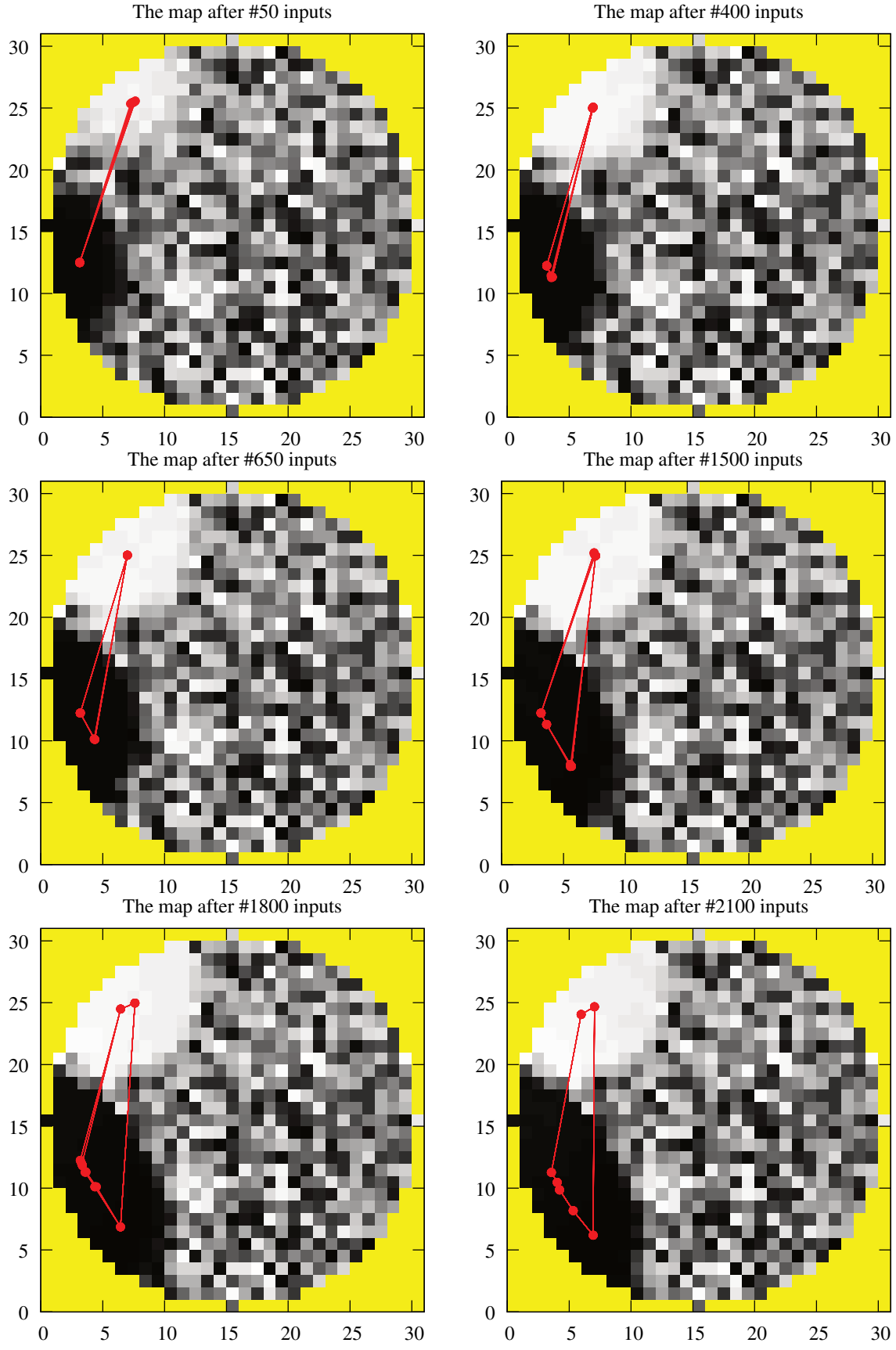


FIGURE 6.12: The mapping of the $AAAAFF$ sequence: The figures show how subsequent splits emerge. In this experiment the short-term memory depth is 5.

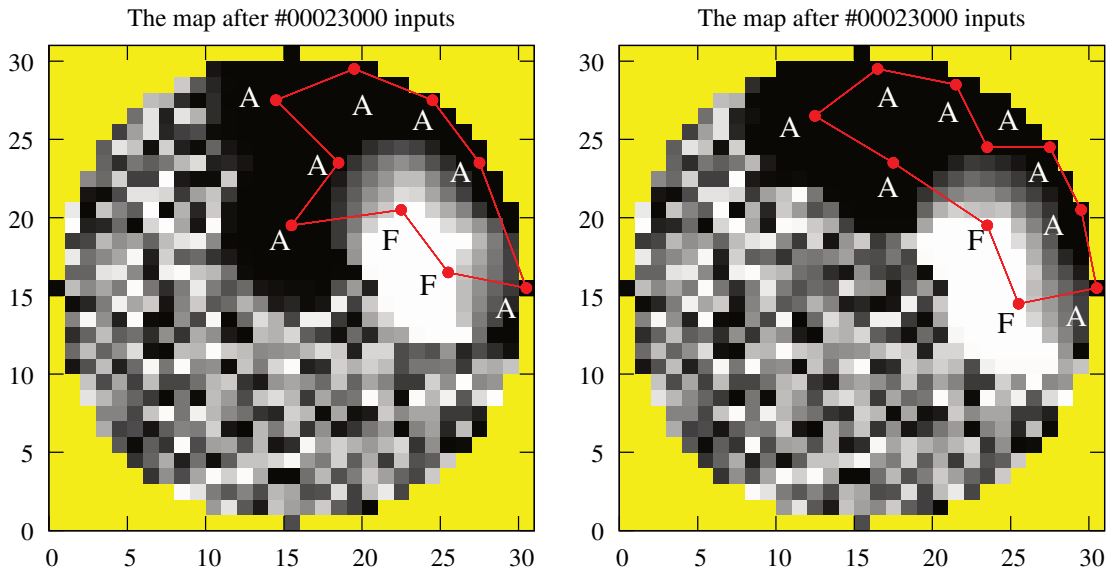


FIGURE 6.13: RecSOM mapping for sequence $AAAAAAAFF$ (left) and $AAAAAAAAFF$ (right).

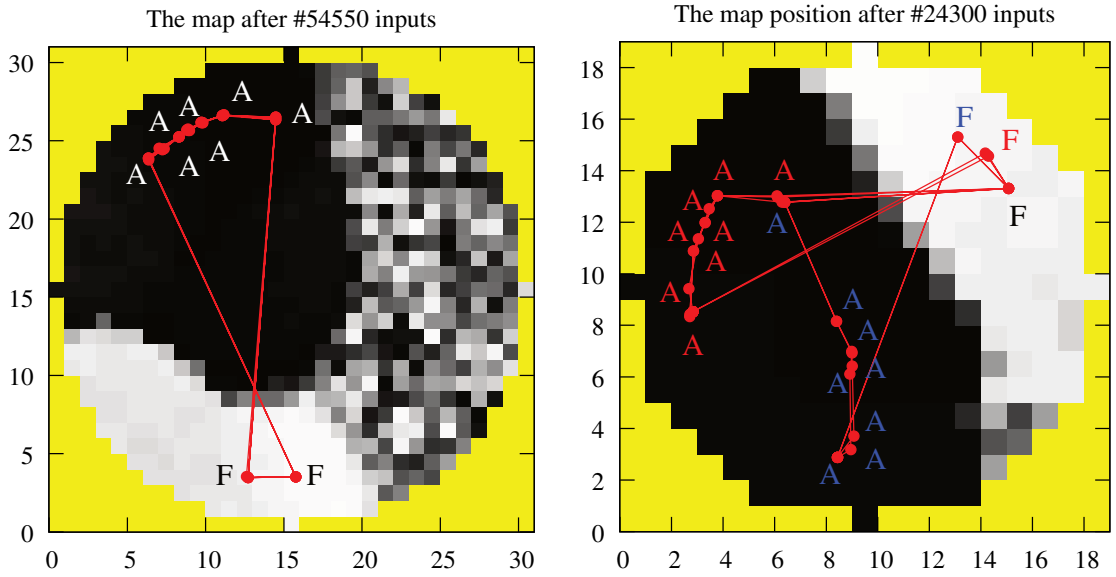


FIGURE 6.14: Architecture mapping for sequence $AAAAAAAFF$ (left) and $AAAAAAAAFF$ (right).

sider sequences up to a degree 13 (requiring a memory depth 13), while in the original work [Voegtlin 2002], the average reached depth (Voegtlin calls it the "quantizer depth") is 7.08 (average length of the receptive field, the latter is defined in 5.5.3), however, in that experiment the input was a corpus of English language, so the map should represent the temporal context of much more inputs, which makes both situations not comparable.

One important remark concerning experiments in this subsection, is that it appears that the or-

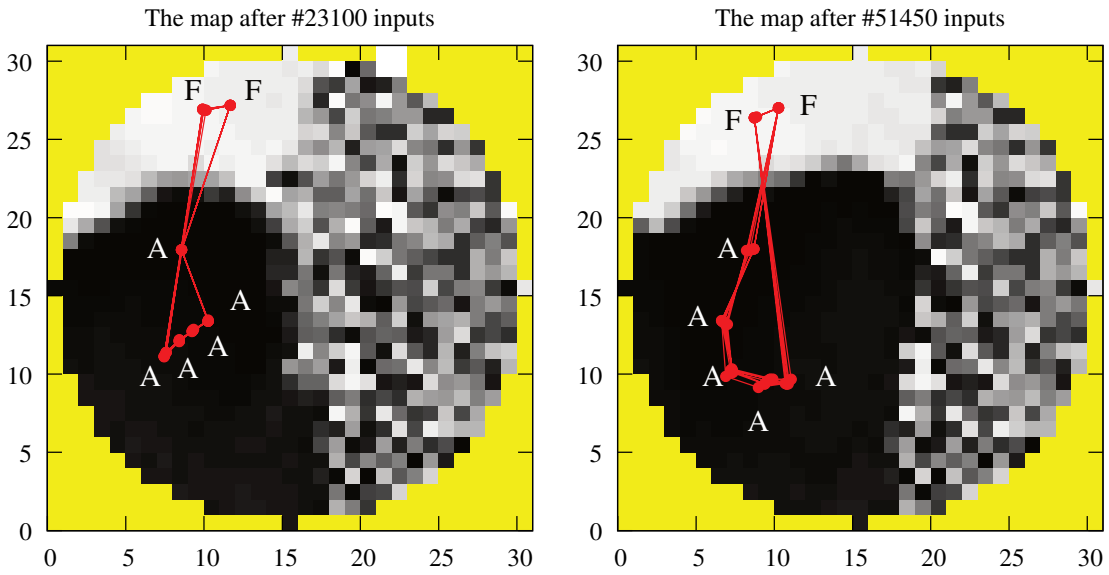


FIGURE 6.15: Prototype organization by the architecture by time. The figure shows how the extracted representation slides during different stages of the system evolution. This is the continuation of the experiment in figure 6.12.

ganization of prototypes by RecSOM is stable, while it continuously "slides" with the architecture. In order to illustrate this, figure 6.15, shows the continuity of the map evolution of the same sequence used for figure 6.12. It shows that different mappings of the same sequence can be observed at different stages of the architecture evolution. This unstable mapping is discussed in the next subsection.

6.4.2 Mapping instabilities

In the previous experiments, some instabilities have been observed at the level of the mapping of the states of the external dynamical system over the input map surface. This corresponds to continuous changes in the topology preserving mapping. Topology preservation which characterizes the basic SOM, is not always guaranteed in the case of temporal SOM-based models, it could sometimes break out. This phenomena is studied in [Tino 2006] for RecSOM. As that work shows, tackling this subject formally is complicated, but it is more complicated in the case of the proposed architecture that is multi-map and uses a neural field and the inter-map partial connectivity. Besides, as mentioned in 5.5.3, the instability issues addressed in RecSOM are indeed a different problem.

In order to reveal more about the instability problem in the proposed architecture, we present a straightforward example. In a new experiment, the architecture is provided with the repetition of a short sequence of observations $AAFF$ as input. As expected from the architecture, it can be seen in figure 6.16 that it is able to map the four states underlying this sequence of observation to four positions in the map surface in spite of observation ambiguity, but the same figure shows that the mapping continuously drifts over the map surface, but the mapping remains correct (this also what happens in figure 6.15). Sometimes, during the drift, separate states can merge inappropriately but

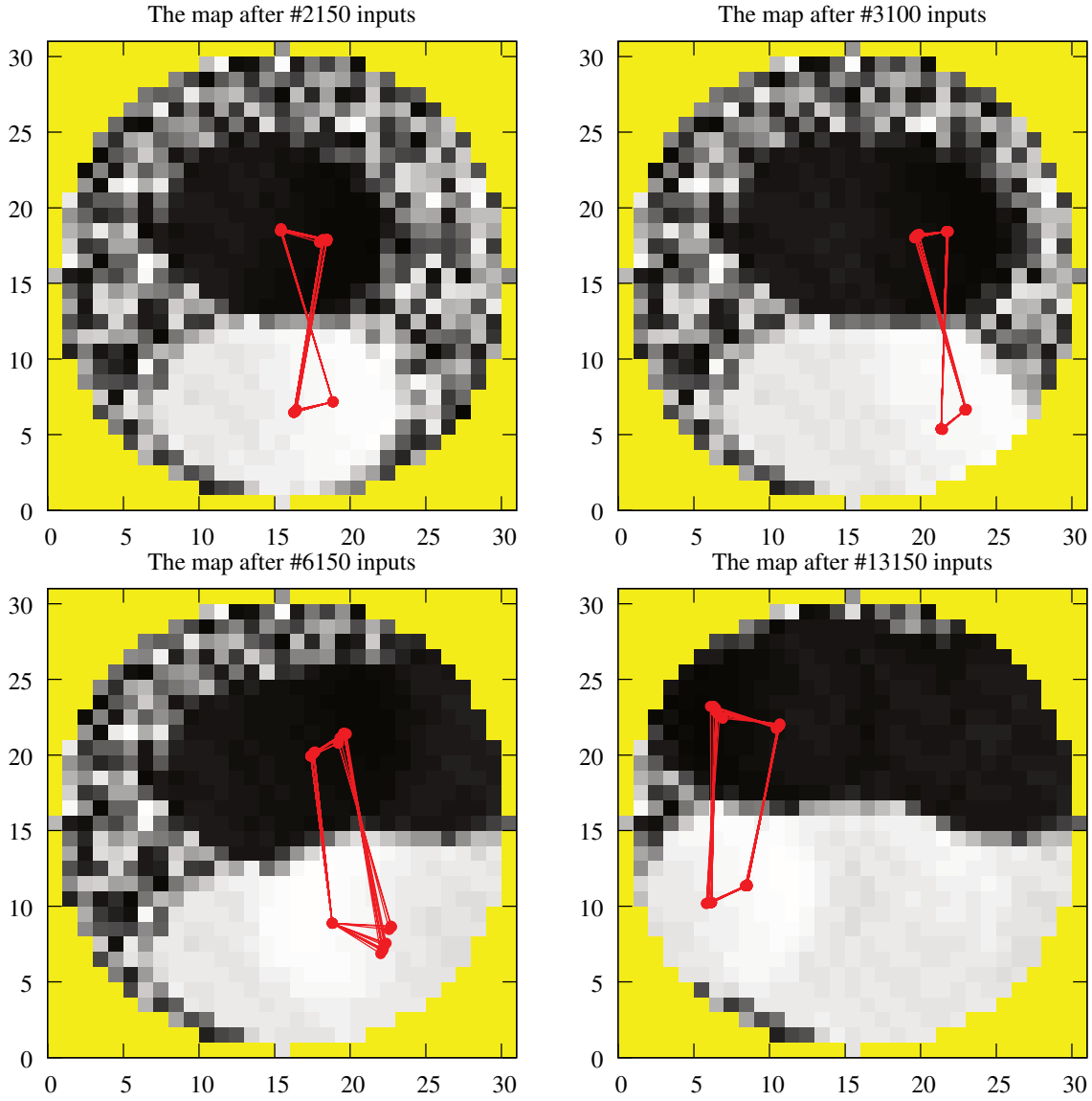


FIGURE 6.16: A straightforward unstable mapping example. The representation of the sequence is $AAFF$ is drifting on the map surface.

re-split after a while. RecSOM taking the same sequence as input does not exhibit any drift (see figure 6.17).

In order to investigate the reasons for this occurring instability, we consider again the input sequence $ABCDEFEDCB$, as in figure 6.10 (top right), but the architecture is run this time with a smaller maps size. Recalling that there is no noise in these experiments, the result is shown in figure 6.18(a). As shown in this figure, the second input map snapshot is taken after presenting sufficiently large number of inputs after the first snapshot. It is obvious that this time, the architecture formed a stable representation. Although, the mapping is not correct; this is because the input map is not wide enough to allow for the splitting of C and D regions. With the same maps size, we

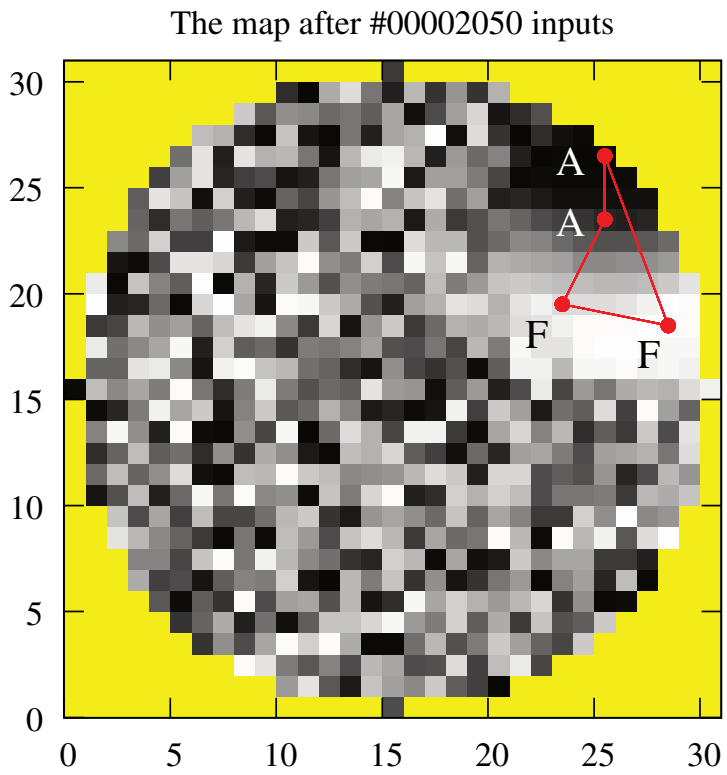
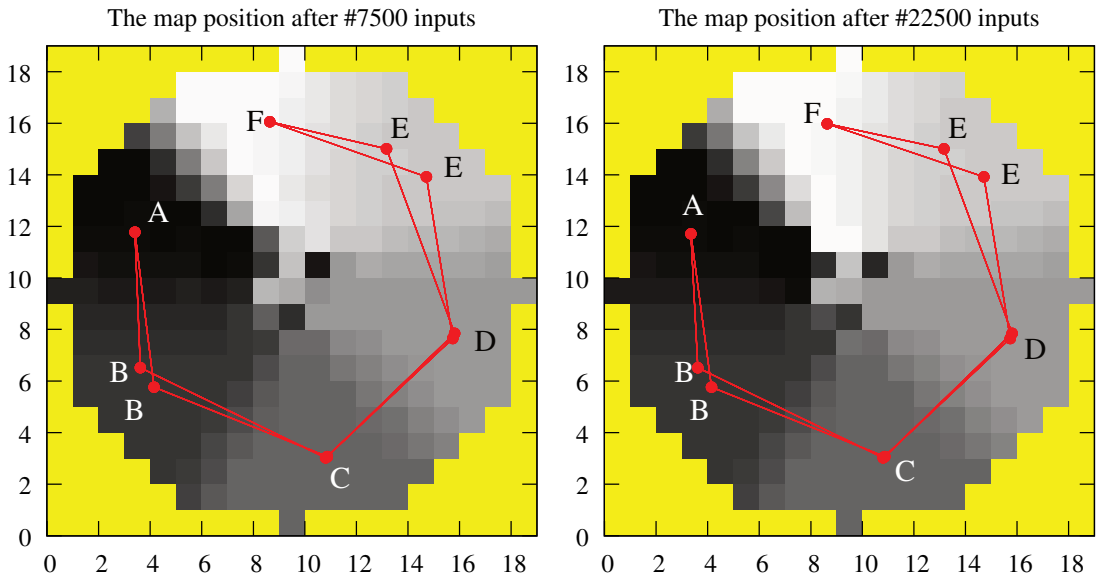


FIGURE 6.17: RecSOM set up a stable representation for the sequence $AAFF$.

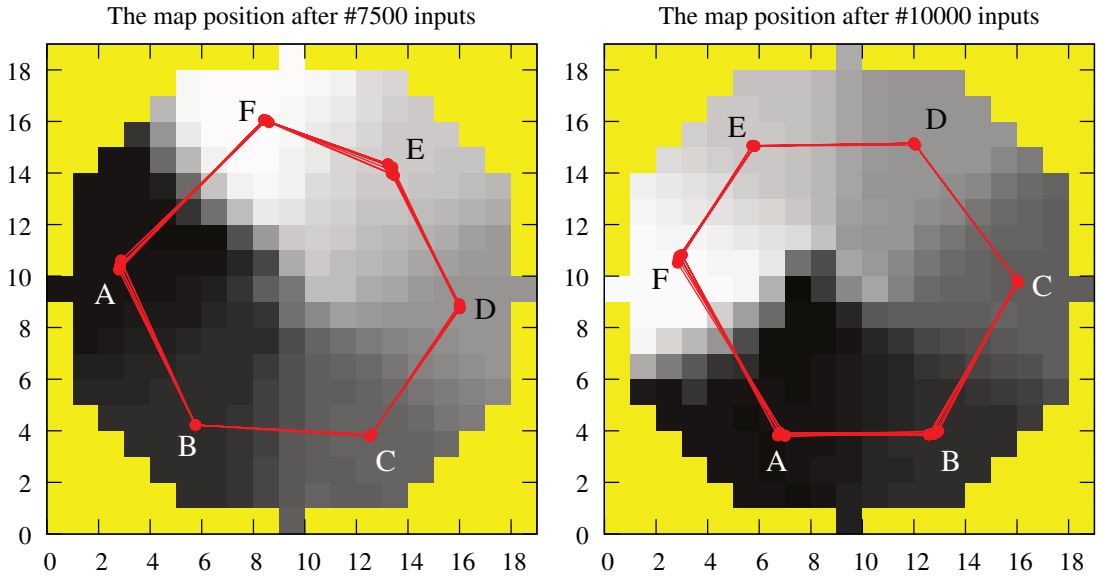
run another simulation with a simpler sequence $ABCDEF$, which is not ambiguous, the result is shown in figure 6.18(b). Contrarily to the case illustrated in figure 6.18(a), the mapping is correct but instability re-appears; the representation is rotating slowly.

Here, a qualitative explanation of this difference in behavior stability is proposed. We have observed during the experiments run on the proposed architecture that the organization of the network looks like an expansive process: the position of the poly-line vertices evolve as if they were repulsive. This can be watched in figure 6.12, where the distance between the nodes tends to increase when the points in some region split.

This mapping expansion occurs also in figure 6.18(a) for the sequence $ABCDEFEDCB$, but the mapping is kept confined within the small map so that it is bounded by the lack of surface available for the representation. It seems that, during the expansion in this case, the direction of vertices movement is related to the order of elements in the input sequence. When elements order contain a change of direction, or a U-turn, the mapping tends to be stable. This could be due to the effect of the self-organization of cortical weights within the strips between the maps (As one can imagine, the visualization of their dynamics is very difficult), we think it is possible that cortical weights learning (that also results in a bump of high cortical weights within each strip) makes the higher weight region (the bump within the strip connections) drifting in one side then in the opposite side when the input sequence makes the U-turn (at A and F). Thus, the higher activity of cortical weights within the inter-map strips is swinging in the case of $ABCDEFEDCB$ (figure 6.18(a)),



(a) Sequence $ABCDEFEDCB$



(b) Sequence $ABCDEF$

FIGURE 6.18: Mapping instability on a smaller map. See text for detail.

instead of continuously drifting (in a circular way) in the case of $ABCDEF$ (figure 6.18(b)). As a result, the fact that points A and F in figure 6.18(a) are trying to move in opposite directions is thought to give the stability in the confined space of the map, while for the sequence $ABCDEF$, there is no direction change and the vertices of the poly-line turn coherently with the sequence of inputs (counter-clockwise in figure 6.18(b)). Unfortunately, tackling mapping instability revealed to be difficult, and the presented explanation is only qualitative, a more formal approach of the mapping instability is in perspective.

6.5 Discussion

In this chapter, we introduced a multi-map recurrent self-organizing architecture, built using the `bijama` framework for fine-grain modeling, and following the neural field paradigm for decentralized lateral competition, making of it a cellular computing model.

Let us recall that at the beginning of this manuscript, we set our research objectives to attain a cellular computing architecture capable of processing temporal sequences. The previous chapters included preliminary introductions to understand the model proposed in this chapter, including fine-grain and cellular computing paradigms, the theoretical models of computation, the neural paradigms for processing temporal data, the paradigms of SOM-based models conceived for temporal sequence processing, and the dynamic neural field theory for distributed computation. Based on these preliminary introductions, in this chapter we introduced the model.

During the way, we made the best effort to criticize our proposed model. One central criticism concerns the topographic locality required by cellular computing as set by Sipper [Sipper 1998b], and said that our proposed model can be regarded as cellular with a kind of mitigated perception concerning this very point, and explained that applying the WTM policy using a distributed paradigm of neural field requires inhibiting all possible bumps from emerging elsewhere in the input map. This required overtaking the topographic locality condition, all while meeting the functional locality and decentralized nature of cellular computing. With this consideration, the model reveals to be a cellular model (in the mitigated sense) capable of temporal sequence processing. Another criticism concerns the occasionally encountered instability of the extracted representation.

Before discussing stability issues furthermore, let us first step back and try to look at the architecture, also at the dynamical system, from the perspective of theoretical models of computation. The autonomous dynamical system can be regarded as an autonomous automata (defined in 3.2.1), which is an FSM that has no output, and its input set contains only one element. This element could be thought of as an internal triggering signal that controls the state transition. In the case of the example wheel dynamical system, it is the force turning the wheel. The states of this autonomous automaton are observed and introduced to the architecture.

In its turn, the proposed architecture implements a neural state machine. Indeed, it takes as input the stream of observations on the autonomous dynamical system, and rebuilds a representation of its internal states, each steady state (that corresponds to a stable bump) can be regarded as a state of the neural state machine. As an FSM it has a finite dynamic memory. It is finite in the architecture, because it has a limited memory depth with which the resolution is sufficient to correctly extract the representation of the input observation. From the point of view of a state machine, we account only for the reliable memory depth, which it is limited as in all realistic models. Same can be said on RecSOM.

Depending on the perspective, the FSM that the architecture implements can be regarded as semiautomaton or a Moore-type state machine with output. Specifically, from an internal perspective to the architecture, the activity of the input map is considered as a part of the state, i.e. not an output. In this case, it is a finite neural state machine without output, i.e. a semiautomaton (also defined in 3.2.1). From an external perspective to the architecture, the input map activity (or even the position of successive bumps) is considered as an output. This output is computed after the input is

processed and the next state is reached, thus it is a Moore-type neural state machine, recalling that computing the output after the new state is reached makes the FSM a Moore-type machine.

The idea that motivated bijama was, in the first place, to provide a construction sets for complex distributed fine-grain multi-modal computing systems. The bijama framework along with the InterCell high performance parallel computer, allow for the definition and run of large systems of such characteristics. In this work, we introduced and tested a new kind of building brick in bijama construction set, bringing to it, recurrent self-organization with a cellular implementation, and the ability to process complex sequences that require the accounting for the temporal context of the model inputs. This ability was introduced in the form of an example application, aiming to build a mapping to the state of an autonomous dynamical system. It was chosen to be autonomous for the sake of simulations simplicity. Another reason is that, during this research work, the architecture was thought of as a part of a cognitive agent able to autonomously cope with POMDP problems, the latter requiring the agent to perform actions in the environment. However, and generally speaking, there is nothing that prevents the proposed architecture to cope with the problem of setting up a mapping to the states of non-autonomous dynamical system, if the input to the latter system is delivered by another source than the agent itself.

We think that this work is a natural continuity to previous work in our lab. In the mentioned previous works, Kohonen self-organizing maps has been adapted to bijama , and this was recalled by experiments in figure 6.9 (using a different neural field). For comparison, RecSOM algorithms was adapted to behave similarly to the proposed cellular and distributed architecture. The proposed model could be somehow thought of as a cellular version of RecSOM, adapted to fine-grain and cellular computing requirements. In fact the recurrent path is not exactly the same; the binding mechanism requires strips in both directions between the input and the associative maps.

Nevertheless, bringing the RecSOM paradigm into bijama implies two major modifications on its algorithm. The first modification is the use of a neural field for the winner-take-most competition process, instead of applying a neighborhood function around the best matching unit. This point is critical as investigated in [Alecu 2011c], as bijama implements distributed computing and offers no central processing capability. This is why LISnf neural field was used in this work, selected as it offers more accurate winner-take most policy than other neural fields.

The second major modification, which is desired but not obligatory in bijama , is related to the computation load of the architecture. This modification requires changing the way of delivering the temporal context itself ($c(t)$ in figure 6.3). For RecSOM, the matching distribution $A(\tau)$ of the whole map is used as a context presented to all units (equation (6.8)). This matching distribution is analog to the distribution of $\nu_p(\tau)$ in the input map of the proposed architecture (equation (6.3)), but in the architecture, a more reduced information is used as a context. Indeed, the u distribution (figures 6.6 and 6.7) is rather used as a context, which is analog to $h_{pk(\tau)}$ values in equation (6.10). Moreover, as opposed to RecSOM, the map units do not share the same context information, since they only view the remote u values from their limited connection strip, which means that in some timestep, units receive a partial context information, different from what other units receive. The partial context information could justify the lack of memory resolution compared to RecSOM. Although, the architecture was able to keep a sufficient memory resolution at an important depth, this is because the important information within the map surface is restricted to the bump region,

which can be coped with by using partial connectivity especially after the binding mechanism emerges. Nevertheless, the main concern here, related to adjusting RecSOM to the requirements of large scale modeling with `bijama`, is that such partial connectivity allows for the design of big maps, since in this case, it will be less subject to combinatorial explosion than the all-to-all recurrent connectivity of RecSOM.

From our experiments, it appears that the implementation difference between RecSOM and the proposed cellular, temporal and self-organizing architecture, does not affect the behavior expected from such models. However, with the proposed architecture, an instability is sometimes observed when mapping the dynamical system states over the input map surface. In fact, with the basic SOM, and with SOM-base temporal models (including the proposed model and RecSOM), several mappings of the inputs are actually possible (figure 6.14-right exhibits this effect), and the organization converges to one of them. However, with the proposed architecture, when instability is observed, it seems that these possibilities are continuously visited, following a smooth drift, except if the size of the map or the nature of the input sequence constraints that drift.

As far as we know, this kind of instability (which is different from instability definition for RecSOM), has not been reported or tackled in the literature of self-organizing models. It appears that addressing more complex computational self-organizing systems, like the proposed architecture, unveils such complex dynamics. The representation visualization method proposed in this work, i.e. the path of successive bump positions in the figures, allowed to capture the dynamics and exhibit the instability phenomena. However, more formal tools for analyzing what happens in such neural architecture are lacking in this work, and in literature.

Such a drift in the extracted representation is not without undesirable effects. During the drift, correctly differentiated region may collapse, and re-split afterwards. Although this does not happen with all map-sizes and sequence configurations, and although if this happens, it sometimes happens during small periods relative to the simulation evolution, this instability remains problematic. As the architecture could be used as a building block for more complex larger-scale systems, unstable map states may prevent other blocks that read the map activity from being able to learn from bump positions within the self-organizing modules (the input map). In other words, it will be difficult in this case to an artificial structure to differentiate the change in the bump positions due to a drift or instability, from the change due to the non-stationarity of the dynamical system providing the inputs.

However, as suggested by the experiments shown in figure 6.18, it is possible that adding more constraints leads the system to stabilization. Hence, it is possible that coupling the architecture to several other self-organizing modules in a larger system, which is feasible in `bijama`, impose more ties that constraint the architecture dynamics, and possibly leads to a stable representation. Such test is relevant, even if the instability of mapping is not solved, in order to see if this instability disappears when several self-organizing modules are coupled together. Indeed, we live in universe that consists of a huge, maybe infinite number of interacting cellular structures, it is not sure that they will be stable in the situation of absolute isolation.

CHAPTER 7

Conclusion

Science allowed the Human to get more control on the elements of nature around him, and during the centuries, he has developed mathematical models for the surrounding phenomena. The utility of mathematical models often relies on the ability of computing them, in order to make a practical use of the results of computation.

Computation was developed step by step since the early ages, but its development has witnessed a remarkable acceleration in the past 80 years, that ended by our actual information age, which requires increasing computational power every new day.

In order to increase the available computation power, coupling the available computing hardware in parallel structures was the solution for few decades. However, the urge for unveiling new computation power and methods was motivated by two factors. First, until now, the attained computational power is not sufficient to compute all the existing mathematical models, some of which obliges scientists to work on simplifications of these models or to partition tasks that take long time to run. Although, the existing computational models did not meet their ambitions, for example like carrying out a simulation of a realistic model of the human brain. Second, not every natural phenomena has yet a mathematical model, as for example, complex systems whose interesting emergent behavior results from the interaction of their basic elements. Such systems should be simulated in order to be understood, they are typically of large size, and require looking for alternative ways to compute them.

Both reasons led the scientists to revise the existing computational models, and study their limitations and perspectives. They found that for reasons related to the laws of physics, the development of the hardware implementations following the actual computation paradigms and using the available technologies will not meet the increasing need of science such as physics and economy, not to mention astronomy. Hence, they started to discuss another computing paradigm that can offer more computational power and cope better with complex problems. This paradigm is motivated by the actual information in physics and biology, that allows to regard the interaction of the elements in nature as a sort of computation that occurs on the level of basic physical and biological cells. This idea was proposed in computer science since about 15 years, and is sometimes referred to as “fine grain computing”, and some other times as “cellular computing”.

In chapter 3, and before introducing the cellular computing paradigm, we made an effort to trace the story of the actual computation paradigm since its very start, which dates back to the first efforts of Alain Turing and John Von Neumann. The latter came with a sequentially computing machine which became the corner stone for the later parallel structures of computations until today. The efforts of Turing led to the definition of computability and computable functions, which later allowed Chomsky to put his hierarchy of formal grammar and the theoretical models of computation that compute them. The reason why these works are discussed, is that they are general, and can be

applied in studying whichever paradigm of computation.

We then presented a comparative study of parallel computation paradigms, and distinguished between the prevailing parallel computation paradigm, the "coarse-grain" and the "fine-grain" paradigms of parallel computation. We first revised coarse-grain models that encompass almost all the actually existing computational structures that consist of agglomerations of Von Neumann machines coupled in parallel.

Then we discussed fine-grain models that rely on populations of simple processors working in parallel, and presented their three major families in computer science: cellular automata, cellular neural networks, and artificial neural networks, and discussed their properties and their capability of universal computation.

We then presented the cellular computing paradigm that adopts the principles of computation in nature that give rise to emergent behavior, and proposed to distinguish cellular models from fine-grain ones by regarding the former as a sub-family of the latter, adding the decentralized and local computation conditions (where locality encompasses two types: the functional locality and the topographic locality) to parallelism and simplicity of fine-grain models. Indeed, there are very few research works concerning the fine-grain parallel models, and a deeper comparative studies of models is required.

Until recently, such large-scale nature-inspired parallel models was not possible to simulate due to the lack of the sufficient computation power. Such domain is in its infancy, and is far from being used for general-purpose computing. This led scientists like Wolfram to recommend to start studying the cellular structures and document what they do and build a kind of a centralized knowledge library that could be a reference for future research. This research work is an effort in this direction.

We have set our research goal to construct a cellular computing model based on artificial neural networks, and to use them in processing temporal sequences. Neural models for this kind of tasks already exist, but not in a cellular form. Due to the difficulty of implementation, they only incorporate small populations of neurons with a limited extensibility into large scale systems. Endowing neural models with the cellular computing properties allow for implementing unprocessed large-scale neural models.

Unlike the already existing cellular models (cellular automata and cellular neural networks, studied in chapter 3), neural networks have powerful adaptive properties that make them fit with temporal problems such as the online interaction with dynamical systems.

Neural models for temporal sequence processing were revisited in 4, after studying and differentiating temporal sequences and time series tasks. The major goal of that revision is to reveal that none of the existing models is cellular. Indeed, these models either contain highly non-local connectivity, or their computation is not decentralized, which makes their large-scale implementation on hardware and coarse grain parallel computers very difficult. Among the different existing models, those recurrent ones implementing feedback turning the network to a dynamical system offer good temporal processing capabilities with smaller-size architectures that need less computation.

Chapter 5 was dedicated to the state of the art of a special kind of neural models that was not until recently used in temporal tasks, which are self-organizing neural models. It was shown that most of these models are based on the self-organizing map by Kohonen. Here too, models that

implement feedback connections are more powerful and have interesting characteristics, but also, it was shown that they do not fit with the requirements of the cellular computing paradigm. In this chapter, we also introduced the theory of dynamical neural fields, and showed how they are able to drive the competition mechanism in self-organizing maps in a distributed and decentralized way, i.e. in a way conform with cellular computing requirements. Although known to be difficult to correctly drive self-organization, we selected to our model a new neural field algorithm, developed in our lab, that can cope with this task.

In chapter 6, we presented the model that we proposed. It is a multi-map neural model that consists of a modified self-organization map, coupled in a recurrent path with two intermediate maps. The first one performs delay and keeps a delayed copy of the original map activity. The second one acts as a signal exchange medium between the actual and the delayed states of the self-organizing map activity. This recurrent path, implemented by inter-map strips of connections avoids the combinatorial explosion of total connectivity. The activity of each map is computed by distributed lateral competition carried out by the neural field, making of the architecture a cellular computing system. Considering the recurrence implemented by the recurrent path but also recurrence implemented on the level of each map by the neural field mechanism, the model turns out to be a dynamical system. The neural model is also adaptive, learning occurs in the self-organizing map prototypes, and in the inter-map connections, which makes the system dynamics pretty complex. The difficulty of a formal study of such model dynamics makes it indeed a complex system, especially because it consists of a population of interacting units that exhibit an emergent behavior, namely self-organization, which is a remarked pattern in other complex systems in nature. Understanding less complex temporal models based on self-organizing maps was reported in literature to be difficult as well.

The proposed model is implemented using `bijama` framework dedicated for modeling fine-grain systems, which is run on `InterCell` supercomputer (a coarse-grain parallel architecture). The model is inherently unsupervised, and its update is indeed distributed and asynchronous due to `bijama` properties. The model ability to process temporal sequences was tested on the example of processing a stream of observation on some dynamical system. The system is chosen such that the sequence of observations be non-Markovian or ambiguous, i.e. some different states result in identical observations. The model was shown to be able to process the observation stream presented as its input, and resolve its ambiguity, such that it forms a representation for each observation on the surface of the self-organizing map, while considering the temporal context of observations. Differently speaking, the model was able to assign different representations to identical observations that occur in different temporal contexts. Indeed, recurrence in the proposed architecture resulted in an internal dynamics that implements a short-term memory used to hold the context information. By assigning different representations to identical observations, we traced the succession of individual observation representations, and showed how the result forms a temporal representation that maps to the observed states of the dynamical system, i.e. the architecture was able to extract a representation of the dynamical system state space starting from an ambiguous stream of observations on the system.

We also simulated the case of non-stationary dynamical system, by changing the sequence of observations, and showed how the proposed model was able to set up a new representation that

maps to the new system state. It was shown that the proposed approach is online, and model free as it does not need to consider any information related to the observed dynamical system. The model is also autonomous as no resetting nor turning to any of the model parameters is required after the start of the run.

In order to judge the behavior of the proposed model, we implemented RecSOM algorithm, which is used in literature as a benchmark model for self-organizing temporal models. The implementation of RecSOM was adjusted to reach the maximum equity for comparison. It has been shown that the proposed model, although its cellular implementation, performs in a similar way to RecSOM in temporal sequence processing. However, some behavioral remarks were drawn. The first remarks concern the depth and resolution of the short-term memory. It has been shown that RecSOM overcomes our proposed model, basically due to the loss of information because of the use of partial connectivity for signal feedback instead of total connectivity as in RecSOM, but it was discussed that such total connectivity does not remain practical when implementing larger-scale models. The second remark concerns an observed instability of the extracted representation by the proposed model with some characteristics of the input stream. As far as we know, this instability of this kind has not been reported in literature, and might be unveiled at the occasion of implementing such a complex cellular architecture. Because of the difficulty of studying temporal models based on self-organizing maps, we presented a qualitative study of this instability facilitated by the proposed representation visualization method.

Incoming works will be twofold. First, a better understanding of the mapping instability will be investigated, from very simple examples as the one in figure 6.16. Second, coupling several recurrent self-organizing modules in bigger architectures has to be tested. This is feasible since `bijama` is designed for a cluster implementation. Such test is relevant, even if the instability of mapping is not solved, in order to see if this instability disappears when several modules are coupled. It is suggested by experiments (showed in figure 6.18) that having more constraints stabilizes the system.

A possible work that can use the proposed model, is dynamic modeling used in some applications like system identification. System identification resembles to time series prediction in that both are the engineering embodiment to the old problem of function approximation. Each of these problems seek to quantify the system that created the time series by estimating its parameters [Principe 1998].

The traditional way was to estimate the system parameters by linear models [Box 1976], by assuming that the time series is generated by a linear system excited by noise. In this approach, the inaccuracy of time series is attributed to the stochastic nature of noise, which can not be modeled. The dynamic modeling approach appeared in mid 1990's [Principe 1995], in which the time series is viewed as the output of a deterministic, autonomous dynamical systems. In this approach the variability in time series is not attributed to the nature of the excitation, instead, it is linked with the high order dynamics and nonlinear nature of dynamical systems. In dynamic modeling, the model could be a either a non-linear system, or a linear system with time-varying parameters in order to avoid the trivial dynamics of invariant linear systems. Modeling a dynamical system could be thought of as finding a model that works as an inverse function of the original system.

Dynamic modeling consists of two steps. The first is to transform the observed time series into

a trajectory in a reconstruction space. The second step is to build the predictive model that gives the future values of the time series starting from the trajectory in the reconstruction space. For this purpose, any adaptive nonlinear system could be trained as a predictor. This step is straightforward, while the first step is the most important one.

The evolution function of the dynamical system is normally unknown, thus the observations on the system are used to build the dynamic model. The sampled observation stream is proved by Taken [Takens 1981] to be sufficient to create a trajectory in the Euclidean space, all while preserving the dynamical invariants of the original dynamical system. However, there are conditions that lie the size of the sampled stream with the dimensions of the system attractors in the geometric space. Using the observation stream, it is possible to recover information about the dynamical system in the Euclidean space, which is the construction space. Recovery is carried out by constructing a mapping from the dynamical system space (a manifold) to the Euclidean reconstruction space, the process is called an embedding (and the theorem is called Taken's embedding theorem)). For this purpose, usually a delay line of $N-1$ length is used to represent the N system observations. The process of dynamic modeling is illustrated in figure 7.1.

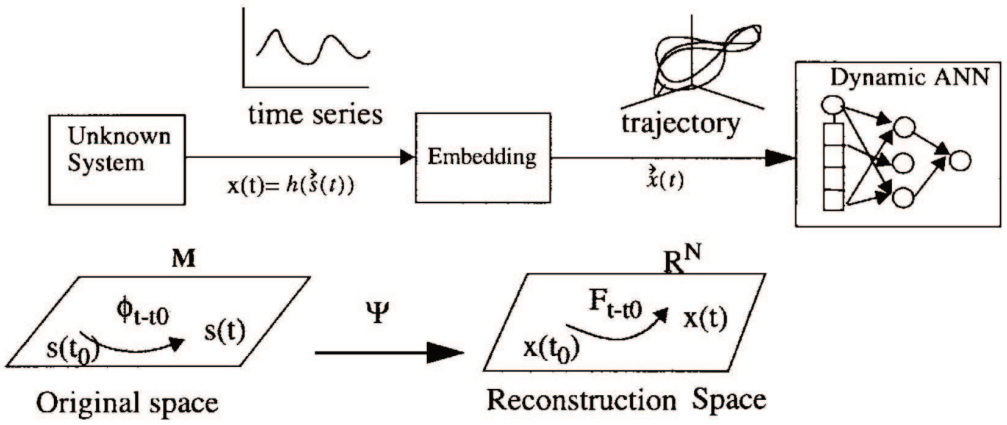


FIGURE 7.1: Nonlinear modeling. Extracted from [Principe 1998].

The proposed model in chapter 6 performs the first step of dynamic modeling differently. It builds a mapping to the dynamical system state using vector quantization biased by the temporal context of input observations. In fact, it does more than the embedding in the reconstruction space: it represents the trajectory in the reconstruction space.

In the proposed model, observations of the dynamical system are not buffered, but they are rather taken when they are available and are presented to the model one by one online. Thanks to recurrence, there is no need to an explicit delay line for embedding, instead, a short-term memory is constructed intrinsically in the model structure. In order to complete the dynamic modeling, for the purpose of system identification for example, it is sufficient to train some temporal neural network that takes as inputs the activity of the map and predicts the next values in the time series.

Concerning the perspectives from this research work, our long-term goal is to design controllers in the field of cognitive robotics, the work presented in this manuscript is only a first step to introduce temporal representation in distributed self-organizing architecture. For this first step, our experiments were limited to the passive observation of some dynamical system producing am-

biguous observations. An extension to POMDP, i.e. to the design of a controller from ambiguous observations, is more compliant with the cognitive robotics goal, but is not straightforward from our "passive" architecture. Indeed, in POMDP problems, the change in the environment state happens as a reaction to actions performed by the agent itself.

Action selection policies that select an action for each environment state are implemented by reinforcement learning (RL). Solving POMDP problems by RL requires first of all solving the problem of partial observability. The agent observes an ambiguous stream of observation that confuses its perception of the real environment state and thus confuses its action selection. Hence, before applying some RL algorithm like SARSA, the agent should maintain a consistent representation of the environment state. The proposed architecture in this work offers this possibility, and makes the application of RL algorithms straightforward, if liberated from the distributed computing constraints.

During our research work, we tried to build such complete POMDP controller, and found that it is possible to "engineer" such a controller for an environment on which we have prior knowledge, by coupling additional maps responsible for action selection. Unfortunately, we were not satisfied by this solution as it contradicts with the autonomous and model-free (or, environment-free) nature of the proposed architecture. Indeed, the optimal robotic agent should be able to navigate in environments on which it has no prior knowledge.

The availability of high performance computing resources [Gustedt 2011], as well as a methodology (`bijama` here) for the design of complex systems, open the field of the simulation of complex computational systems, made of simple nonlinear units that interact massively. Such complexity is difficult to handle, due to the emerging dynamics such as self-organization as well as unexpected population effects, sometimes undesirable. Nevertheless, the nature of such complexity fits the nature of the information processing performed by our brains, and the difficulties that we have to face when we build such systems from scratch, may also help to understand the relevance of the brain organization under the light of computational arguments.

Through this manuscript, we presented our contribution into cellular computing paradigm, using a neural network implementation. This contribution subscribes to the fields of computation and parallel computing, sub-disciplines of computer science, but also subscribes to the interdisciplinary computational neuroscience. We expect that the infant cellular computing field of research will attire more attention in the coming years, with scientists seeking to obtain more computational power, but also, a nature-inspired computation that helps to cope better with nature and facilitates people's life, continuing a story that started in Sumer, 2400 BC.

Bibliography

[Abelson 2000] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman and Ron Weiss. *Amorphous computing*. Commun. ACM, vol. 43, no. 5, 2000. (Cited on page 81.)

[Ackley 1985] H. Ackley, E. Hinton and J. Sejnowski. *A learning algorithm for Boltzmann machines*. Cognitive Science, pages 147–169, 1985. (Cited on pages 108 and 121.)

[Adleman 1994] LM Adleman. *Molecular computation of solutions to combinatorial problems*. Science, vol. 266, no. 5187, pages 1021–1024, 1994. (Cited on page 40.)

[A.E. 1995] Howe A.E. and Cohen P.R. *Understanding planner behavior*. Artificial Intelligence, vol. 76, pages 125–166, 1995. (Cited on page 95.)

[Alecu 2011a] L. Alecu. *Une approche neuro-dynamique de conception des processus d'auto-organisation*. PhD thesis, Université Henri Poincaré, Nancy 1, France, 2011. http://tel.archives-ouvertes.fr/tel-00606926. (Cited on pages 156 and 161.)

[Alecu 2011b] L. Alecu, H. Frezza-Buet and F. Alexandre. *Can self-organization emerge through dynamic neural fields computation?* Connection Science, vol. 23, no. 1, pages 1–31, 2011. http://hal.inria.fr/inria-00537799/PDF/cs10.pdf. (Cited on pages 159, 160 and 161.)

[Alecu 2011c] Lucian Alecu, Hervé Frezza-Buet and Frédéric Alexandre. *Can self-organization emerge through dynamic neural fields computation?* . Connection Science, vol. 23, no. 1, pages 1–31, 2011. (Cited on pages 177, 184, 185 and 201.)

[Alquezar 1995] R. Alquezar and A. Sanfeliu. *An Algebraic Framework to Represent Finite State Machines in Single-Layer Recurrent Neural Networks*. Neural Computation, vol. 7, pages 93–1, 1995. (Cited on page 129.)

[Amari 1977] Shun-ichi Amari. *Dynamics of Pattern Formation in Lateral-Inhibition Type Neural Fields*. Biological Cybernetics, vol. 27, pages 77–87, 1977. (Cited on pages 156, 157 and 158.)

[Amari 1998] S.-I. Amari and A. Cichocki. *Adaptive blind signal processing-neural network approaches*. Proceedings of the IEEE, vol. 86, no. 10, pages 2026–2048, 1998. (Cited on page 101.)

[Amit 1992] D.J. Amit. Modeling brain function: The world of attractor neural networks. Cambridge University Press, 1992. (Cited on page 158.)

[Araujo 2002] Aluizio F. R. Araujo and Guilherme De A. Barreto. *Context in Temporal Sequence Processing: A Self-Organizing Approach and its Application to Robotics*. IEEE Transactions on Neural Networks, vol. 13, pages 45–57, 2002. (Cited on pages 95 and 154.)

[Arbib 2003] M.A. Arbib. The handbook of brain theory and neural networks. A Bradford book. MIT Press, 2003. (Cited on page 95.)

[Arena 1997] P. Arena, R. Caponetto, L. Fortuna and G. Manganaro. *Cellular neural networks to explore complexity*. Soft Computing, vol. 1, no. 3, pages 120–136, 1997. (Cited on pages 77, 78, 85 and 91.)

[Atlas 1996] L. Atlas, L. Owsley, J. McLaughlin and G. Bernard. *Automatic feature-finding for time-frequency distributions*. In Time-Frequency and Time-Scale Analysis, 1996., Proceedings of the IEEE-SP International Symposium on, pages 333–336, 1996. (Cited on page 138.)

[Back 1991] A. D. Back and A. C. Tsoi. *Fir and iir synapses, a new neural network architecture for time series modeling*. Neural Comput., vol. 3, no. 3, pages 375–385, September 1991. (Cited on page 113.)

[Backus 1978] John Backus. *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. Commun. ACM, vol. 21, no. 8, pages 613–641, August 1978. (Cited on page 37.)

[Balsi 2001] Marco Balsi, Alessandro Maraschini, Giada Apicella, Sonia Luengo, Jordi Solsona and Xavier Vilasis-Cardona. *Cellular Neural Networks for Mobile Robot Vision*. In José Mira and Alberto Prieto, editeurs, Bio-Inspired Applications of Connectionism, volume 2085 of *Lecture Notes in Computer Science*, pages 484–491. Springer Berlin Heidelberg, 2001. (Cited on page 90.)

[Barney 2013] Blaise Barney. *Introduction to Parallel Computing*. `https://computing.llnl.gov/tutorials/parallel_comp/`, 2013. [Online; accessed 21-june-2013]. (Cited on pages 59, 61 and 63.)

[Barreto 2001] G.de.A. Barreto and A. F R Araujo. *A self-organizing NARX network and its application to prediction of chaotic time series*. In Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on, volume 3, pages 2144–2149 vol.3, 2001. (Cited on pages 115 and 141.)

[Barreto 2003] Guilherme De A. Barreto, Aluizio F. R. Araújo and Stefan C. Kremer. *A Taxonomy for Spatiotemporal Connectionist Networks Revisited: The Unsupervised Case*. Neural Computation, vol. 15, pages 1255–1320, 2003. (Cited on pages 105, 110, 126, 136, 141 and 148.)

[Barreto 2004] Guilherme A. Barreto, Joao C. M. Mota, Luis G. M. Souza and Rewbenio A. Frota. *Nonstationary time series prediction using local models based on competitive neural networks*. In Proceedings of the 17th international conference on Innovations in applied artificial intelligence, IEA/AIE'2004, pages 1146–1155. Springer Springer Verlag Inc, 2004. (Cited on page 142.)

[Barreto 2007] GuilhermeA Barreto. *Time Series Prediction with the Self-Organizing Map: A Review*. In Barbara Hammer and Pascal Hitzler, editeurs, Perspectives of Neural-Symbolic Integration, volume 77 of *Studies in Computational Intelligence*, chapitre 6, pages 135–158. Springer Berlin Heidelberg, 2007. (Cited on page 135.)

[Beer 1996] Randall D. Beer. *Toward the Evolution of Dynamical Neural Networks for Minimally Cognitive Behavior*, 1996. (Cited on page 76.)

[Behme 1993] Holger Behme, WolfDieter Brandt and HansWerner Strube. *Speech Recognition by Hierarchical Segment Classification*. In Stan Gielen and Bert Kappen, editeurs, ICANN 93, pages 416–419. Springer London, 1993. (Cited on page 151.)

[Beigy 2010] H. Beigy and M.R. Meybodi. *Cellular Learning Automata With Multiple Learning Automata in Each Cell and Its Applications*. Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on, vol. 40, no. 1, pages 54–65, 2010. (Cited on page 85.)

[Ben 1999] Marta Garcia Ben, Elena J. Martinez and Victor J. Yohai. *Robust Estimation in Vector Autoregressive Moving-Average Models*. Journal of Time Series Analysis, vol. 20, no. 4, pages 381–399, July 1999. doi: 10.1111/1467-9892.00144. (Cited on page 95.)

[Bengio 1989] Y. Bengio, R. Cardin, R. De Mori and E. Merlo. *Programmable execution of multi-layered networks for automatic speech recognition*. Commun. ACM, vol. 32, no. 2, pages 195–199, February 1989. (Cited on page 98.)

[Bengio 1992] Yoshua Bengio, Renato de Mori and Marco Gori. *Learning the dynamic nature of speech with back-propagation for sequences.* Pattern Recognition Letters, vol. 13, no. 5, pages 375–385, 1992. (Cited on pages 113 and 127.)

[Bengio 1994] Y. Bengio, P. Simard and P. Frasconi. *Learning long-term dependencies with gradient descent is difficult*. Neural Networks, IEEE Transactions on, vol. 5, no. 2, pages 157–166, 1994. (Cited on pages 76 and 114.)

[Bengio 1996] Y. Bengio and P. Frasconi. *Input-output HMMs for sequence processing*. Neural Networks, IEEE Transactions on, vol. 7, no. 5, pages 1231–1249, 1996. (Cited on page 96.)

[Benjamin 1996] S C Benjamin and N F Johnson. *A Possible Nanometer-scale Computing Device Based on an Adding Cellular Automaton*. Rapport technique cond-mat/9610035, Oct 1996. (Cited on page 40.)

[Berlekamp 1982] E.R. Berlekamp, J.H. Conway and R.K. Guy. Winning ways for your mathematical plays: Games in general. Winning Ways, for Your Mathematical Plays. Academic Press, 1982. (Cited on pages 66 and 67.)

[Beroule 1987] D. Beroule. *Guided propagation inside a topographic memory*. In 1st International Conference on neural networks, pages 469–476, 1987. (Cited on page 98.)

[Beurle 1956] R. L. Beurle. *Properties of a mass of cells capable of regenerating pulses*. Philosophical Transactions of the Royal Society London B, vol. 240, pages 55–94, 1956. (Cited on page 156.)

[Bishop 1995a] Christopher M. Bishop. Neural networks for pattern recognition. Oxford University Press, Inc., New York, NY, USA, 1995. (Cited on page 102.)

[Bishop 1995b] C.M. Bishop. Neural networks for pattern recognition. Neural Networks for Pattern Recognition. Oxford University Press, Incorporated, 1995. (Cited on page 76.)

[Blair 1996] Alan D. Blair and Jordan B. Pollack. *Analysis of Dynamical Recognizers*. Neural Computation, vol. 9, pages 1127–1142, 1996. (Cited on page 129.)

[Blas 2005] Andrea Di Blas, Arun Jagota and Richard Hughey. *Optimizing neural networks on {SIMD} parallel computers*. Parallel Computing, vol. 31, no. 1, pages 97 – 115, 2005. (Cited on page 91.)

[Bogdan 1996] Martin Bogdan, Wolfgang Rosenstiel and Lehrstuhl Fr Technische. *Classification of Nerve Signals using Kohonen's Self-Organizing Map*, 1996. (Cited on page 135.)

[Boulinier 2005] Christian Boulinier, Franck Petit and Vincent Villain. *Synchronous vs. Asynchronous Unison*. In Sébastien Tixeuil and Ted Herman, editeurs, Self-Stabilizing Systems, volume 3764 of *Lecture Notes in Computer Science*, pages 18–32. Springer Berlin Heidelberg, 2005. (Cited on page 57.)

[Bouré 2012] Olivier Bouré, Nazim Fatès and Vincent Chevrier. *Probing robustness of cellular automata through variations of asynchronous updating*. Natural Computing, vol. 11, no. 4, pages 553–564, 2012. (Cited on page 70.)

[Box 1976] G.E.P. Box and G.M. Jenkins. Time series analysis: forecasting and control. Holden-Day series in time series analysis and digital processing. Holden-Day, 1976. (Cited on page 206.)

[Bressloff 2002] P. C. Bressloff, J. D. Cowan, M. Golubitsky, P. J. Thomas and M. C. Wiener. *What geometric visual hallucinations tell us about the visual cortex*. Neural Computation, vol. 14, pages 473–491, 2002. (Cited on page 159.)

[Brette 2007] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Andrew P. Davison, Sami El Boustani and Alain Destexhe. *Simulation of networks of spiking neurons: A review of tools and strategies*. Journal of Computational Neuroscience, vol. 2007, pages 349–398, 2007. (Cited on page 84.)

[c. Chappelier 1996] J. c. Chappelier and A. Grumbach. *A Kohonen Map for Temporal Sequences*. In Proceedings of NEURAP'95, pages 104–110, 1996. (Cited on page 137.)

[Cafagna 2003] D. Cafagna and G. Grassi. *Two-cell cellular neural networks: generation of new hyperchaotic multiscroll attractors*. In Neural Networks, 2003. Proceedings of the International Joint Conference on, volume 2, pages 924–929 vol.2, 2003. (Cited on page 78.)

[Calonge 1997] T. Calonge, L. Alonso, R. Ralha and A.L. Sánchez. *Parallel implementation of non recurrent neural networks*. In José M.L.M. Palma and Jack Dongarra, editeurs, Vector and Parallel Processing VECPAR'96, volume 1215 of *Lecture Notes in Computer Science*, pages 313–325. Springer Berlin Heidelberg, 1997. (Cited on page 91.)

[Campo 2010] Angel Campo and Jose Santos. *Evolution of adaptive center-crossing continuous time recurrent neural networks for biped robot control.* In ESANN, 2010. (Cited on page 125.)

[Campoy 2009] Pascual Campoy. *Dimensionality reduction by self organizing maps that preserve distances in output space*. In Proceedings of the 2009 international joint conference on Neural Networks, IJCNN'09, pages 2976–2982, Piscataway, NJ, USA, 2009. IEEE Press. (Cited on page 135.)

[Caponetto 1998] R. Caponetto, M. Lavorgna, A. Martinez and L. Occhipinti. *Cellular neural network simulator for image processing applications*. In Cellular Neural Networks and Their Applications Proceedings, 1998 Fifth IEEE International Workshop on, pages 360–365, 1998. (Cited on page 78.)

[Carpinteiro 1998] Otávio Augusto S. Carpinteiro and S. Carpinteiro. *A Hierarchical Self-Organizing Map Model for Sequence Recognition*, 1998. (Cited on page 145.)

[Carpinteiro 1999] Otávio Augusto S. Carpinteiro. *A Hierarchical Self-Organizing Map Model for Sequence Recognition*. Neural Process. Lett., pages 209–220, 1999. (Cited on page 182.)

[Carrasco 1996] Rafael C. Carrasco, Mikel L. Forcada and Laureano Santamaria. *Inferring stochastic regular grammars with recurrent neural networks*. In Laurent Miclet and Colin Higuera, editeurs, Grammatical Interference: Learning Syntax from Sentences, volume 1147 of *Lecture Notes in Computer Science*, pages 274–281. Springer Berlin Heidelberg, 1996. (Cited on page 129.)

[Carruccio 2006] E. Carruccio and I. Quigly. Mathematics and logic in history and in contemporary thought. Transaction Pub, 2006. (Cited on page 31.)

[Catfolis 1994] Thierry Catfolis. *Mapping a complex temporal problem into a combination of static and dynamic neural networks*. SIGART Bull., vol. 5, no. 3, pages 23–28, 1994. (Cited on page 99.)

[Chandrasekaran 1995] V. Chandrasekaran, M. Palaniswami and Terry M. Caelli. *Spatio-temporal feature maps using gated neuronal architecture.* IEEE Trans. Neural Netw. Learning Syst., vol. 6, no. 5, pages 1119–1131, 1995. (Cited on page 143.)

[Chandrasekaran 1998] V. Chandrasekaran and Zhi-Qiang Liu. *Topology constraint free fuzzy gated neural networks for pattern recognition*. Neural Networks, IEEE Transactions on, vol. 9, no. 3, pages 483–502, 1998. (Cited on page 143.)

[Chang 2004] Chuan-Yu Chang. *Spatiotemporal-Hopfield neural cube for diagnosing recurrent nasal papilloma*. In Networking, Sensing and Control, 2004 IEEE International Conference on, volume 2, pages 1301–1306 Vol.2, 2004. (Cited on page 122.)

[Chappelier 2001] Jean-Cedric Chappelier, Marco Gori and Alain Grumbach. *Time in Connectionist Models*. In Sequence Learning: Paradigms, Algorithms, and Applications, pages 105–134. Springer-Verlag, 2001. (Cited on pages 99 and 106.)

[Chappell 1993] Geoffrey J. Chappell and John G. Taylor. *The temporal Kohonen map*. Neural Netw., vol. 6, pages 441–445, March 1993. (Cited on page 144.)

[Chen 1990] S. Chen, S. A. Billings and P. M. Grant. *Non-linear system identification using neural networks*. International Journal of Control, vol. 51, pages 1191–1214, 1990. (Cited on page 113.)

[Cheng 2007] Tao Cheng and Jiaqiu Wang. *Application of a Dynamic Recurrent Neural Network in Spatio-Temporal Forecasting*. In VasilyV. Popovich, Manfred Schrenk and KyrillV. Korolenko, editeurs, Information Fusion and Geographic Information Systems, Lecture Notes in Geoinformation and Cartography, pages 173–186. Springer Berlin Heidelberg, 2007. (Cited on page 112.)

[Cheung 1987] Kwan F. Cheung, Les E. Atlas and Robert J. Marks II. *Synchronous vs asynchronous behavior of Hopfield's CAM neural net*. Appl. Opt., vol. 26, no. 22, pages 4808–4813, 1987. (Cited on page 121.)

[Cheung 2006] O.Y.H. Cheung, P.H.W. Leong, E.K.C. Tsang and B.E. Shi. *A Scalable FPGA Implementation of Cellular Neural Networks for Gabor-type Filtering*. In Neural Networks, 2006. IJCNN '06. International Joint Conference on, pages 15–20, 2006. (Cited on page 90.)

[Chi Leung 2011] Patrick Hui Chi Leung. Artificial neural networks applications. InTech, 2011. (Cited on page 89.)

[Chomsky 1956] Noam Chomsky. *Three models for the description of language*. IRE Transactions on Information Theory, vol. 2, pages 113–124, 1956. (Cited on pages 52 and 53.)

[Choudhury 2008] Pabitra Pal Choudhury, Sudhakar Sahoo and Mithun Chakraborty. *Implementation of Basic Arithmetic Operations Using Cellular Automaton*. In Proceedings of the 2008 International Conference on Information Technology, ICIT '08, pages 79–80. IEEE Computer Society, 2008. (Cited on page 70.)

[Chua 1988a] Leon O Chua and L Yang. *Cellular neural networks: Theory*. IEEE Transactions on Circuits and Systems, vol. 35, pages 1257–1272, 1988. (Cited on page 76.)

[Chua 1988b] L.O. Chua and L. Yang. *Cellular neural networks: applications*. Circuits and Systems, IEEE Transactions on, vol. 35, no. 10, pages 1273–1290, 1988. (Cited on page 90.)

[Chua 1993] L.O. Chua and T. Roska. *The CNN paradigm*. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on, vol. 40, no. 3, pages 147–156, 1993. (Cited on page 78.)

[Chua 1995] L.O. Chua, Martin Hasler, George S. Moschytz and Jacques Neirynck. *Autonomous cellular neural networks: a unified paradigm for pattern formation and active wave propagation*. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on, vol. 42, no. 10, pages 559–577, 1995. (Cited on page 78.)

[Chuanwu 2008] Zhang Chuanwu. *Performance Analysis of the CPLD/FPGA Implementation of Cellular Automata*. In Embedded Software and Systems Symposia, 2008. ICESS Symposia '08. International Conference on, pages 308–311, 2008. (Cited on page 90.)

[Cimagalli 1993] V. Cimagalli, M. Bobbi and M. Balsi. *MODA: moving object detecting architecture*. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, vol. 40, no. 3, pages 174–183, 1993. (Cited on page 90.)

[Clement L. 2003] Nagpal R. Clement L. *Self-assembly and self-repairing topologies*. In Workshop on Adaptability in Multi-Agent Systems, 2003. (Cited on page 82.)

[Cook 2004] Matthew Cook. *Universality in Elementary Cellular Automata*. Complex Systems, vol. 15, no. 1, pages 1–40, 2004. (Cited on pages 44 and 67.)

[Coombes 2005] S. Coombes. *Waves, bumps, and patterns in neural field theories*. Biological Cybernetics, vol. 93, no. 2, pages 91–108, 2005. (Cited on page 157.)

[Copeland 1999] B. Jack Copeland and Diane Proudfoot. *Alan Turing's Forgotten Ideas in Computer Science*. Scientific American, vol. 278, no. 4, pages 98–103, April 1999. (Cited on page 33.)

[Copeland 2004] B.Jack Copeland. *Hypercomputation: philosophical issues*. Theoretical Computer Science, vol. 317, no. 1-3, pages 251–267, June 2004. (Cited on pages 33 and 34.)

[Cruse 2006] Holk Cruse. Neural Networks as Cybernetic Systems. 2006. (Cited on page 108.)

[Crutchfield 1988] James P. Crutchfield and Karl Young. *Computation at the onset of chaos*. In The Santa Fe Institute, Westview, pages 223–269. Press, 1988. (Cited on page 126.)

[Daniel 2013] Ramiz Daniel, Jacob R. Rubens, Rahul Sarpeshkar and Timothy K. Lu. *Synthetic analog computation in living cells*. Nature, vol. 497, no. 7451, pages 619–623, May 2013. (Cited on page 40.)

[Das 1992] Sreerupa Das, C. Lee Giles and Guo zheng Sun. *Using Prior Knowledge in an NNPDA to Learn Context-Free Languages*. In Advances in Neural Information Processing Systems, pages 65–72. Morgan Kaufmann, 1992. (Cited on page 129.)

[Dayan 2005] Peter Dayan and L. F. Abbott. Theoretical neuroscience: Computational and mathematical modeling of neural systems. The MIT Press, 2005. (Cited on page 73.)

[Deboeck 2010] G. Deboeck and T. Kohonen. Visual explorations in finance: With self-organizing maps. Springer Finance. Springer, 2010. (Cited on page 136.)

[Demartines 1992] P Demartines and F Blayo. *Kohonen Self-Organizing Maps:Is the Normalization Necessary?* Complex Systems, 1992. (Cited on pages 133 and 138.)

[Diaconescu 2008] Eugen Diaconescu. *The use of NARX neural networks to predict chaotic time series*. WSEAS Trans. Comp. Res., vol. 3, no. 3, pages 182–191, 2008. (Cited on pages 99 and 115.)

[Ding 2011] Jingyuan Ding. *Cellular Automata based Artificial Financial Market*. In Alejandro Salcido, editeur, Cellular Automata - Simplicity Behind Complexity. InTech, 2011. (Cited on page 90.)

[Dominey 2000] Peter Ford Dominey and Franck Ramus. *Neural network processing of natural language: I. Sensitivity to serial, temporal and abstract structure of language in the infant*, 2000. (Cited on page 116.)

[dong Jin 2002] Hui dong Jin, Wing ho Shum, Kwong-Sak Leung and Man-Leung Wong. *Expanding Self-organizing Map for Data Visualization and Cluster Analysis*. Information Sciences, vol. 163, pages 157–173, 2002. (Cited on page 135.)

[Dorffner 1996] Georg Dorffner. *Neural Networks for Time Series Processing*. Neural Network World, vol. 6, pages 447–468, 1996. (Cited on pages 93, 103 and 113.)

[Downey 2012] Allen B. Downey. Think complexity: Complexity science and computational modeling. O'Reilly Media, 2012. (Cited on page 66.)

[Doya 1993] Kenji Doya. *Bifurcations of Recurrent Neural Networks in Gradient Descent Learning*. IEEE Transactions on Neural Networks, vol. 1, pages 75–80, 1993. (Cited on page 115.)

[Du 2010] K.-L. Du. *Clustering: A neural network approach*. Neural Networks, vol. 23, no. 1, pages 89 – 107, 2010. (Cited on page 76.)

[E.F. Moore 1956] E.F. Moore. *Gedanken-Experiments on Sequential Machines*. In C.E. Shannon and J. MacCarthy, editeurs, Automata Studies, pages 129–153, Princeton, New Jersey, 1956. Princeton University Press. (Cited on page 54.)

[Elman 1988] J.L. Elman and D. Zisper. *Learning the hidden structure of speech*. Journal of the Accostical Society of America, 1988. (Cited on page 104.)

[Elman 1990] Jeffrey L. Elman. *Finding structure in time*. COGNITIVE SCIENCE, vol. 14, no. 2, pages 179–211, 1990. (Cited on page 110.)

[Elman 1991] Jeffrey L. Elman. *Distributed representations, simple recurrent networks, and grammatical structure*. In Machine Learning, pages 195–225, 1991. (Cited on page 99.)

[Embrechts 2009] Mark J. Embrechts, Luís A. Alexandre and Jonathan D. Linton. *Reservoir computing for static pattern recognition.* In ESANN, 2009. (Cited on page 99.)

[Erlhagen 2006] W. Erlhagen and E. Bicho. *The dynamic neural field approach to cognitive robotics*. Journal of Neural Engineering, vol. 3, pages 36–54, 2006. (Cited on page 158.)

[Fahlman 1990] Scott E. Fahlman. *The Recurrent Cascade-Correlation Architecture*. In ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 3, pages 190–196. Morgan Kaufmann Publishers Inc., 1990. (Cited on page 125.)

[Fatès 2008] Nazim Fatès and Lucas Gerin. *Examples of Fast and Slow Convergence of 2D Asynchronous Cellular Systems*. In Hiroshi Umeo, Shin Morishita, Katsuhiro Nishinari, Toshihiko Komatsuzaki and Stefania Bandini, editeurs, Cellular Automata, volume 5191 of *Lecture Notes in Computer Science*, pages 184–191. Springer Berlin Heidelberg, 2008. (Cited on page 70.)

[Fatès 2011] Nazim Fatès. *Stochastic Cellular Automata Solve the Density Classification Problem with an Arbitrary Precision.* In Thomas Schwentick and Christoph Durr, editeurs, STACS, volume 9 of *LIPIcs*, pages 284–295. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. (Cited on page 70.)

[Fischer 1997] M M Fischer. *Computational neural networks: a new paradigm for spatial analysis*, 1997. (Cited on page 76.)

[FitzHugh 1955] R. FitzHugh. *Mathematical models of threshold phenomena in the nerve membrane*. Bull. Math. Biophysics, vol. 17, pages 257–278, 1955. (Cited on page 155.)

[Fix 2007] J. Fix, N. Rougier and F. Alexandre. *A top-down attentional system scanning multiple targets with saccades*. In From Computational Cognitive Neuroscience to Computer Vision: CCNCV 2007, 2007. (Cited on page 158.)

[Flynn 1996] M.J. Flynn. *Parallel processors were the future... and may yet be*. Computer, 1996. (Cited on page 86.)

[Fodor 1978] J Fodor. *RePresentations. Philosophical Essays on the Foundations of Cognitive Science*. The MIT Press, 1978. (Cited on page 35.)

[Folias 2004] S. E. Folias and P. C. Bressloff. *Breathing pulses in an excitatory neural network*. SIAM Journal of Applied Dynamical Systems, vol. 3, no. 3, pages 378–407, 2004. (Cited on page 157.)

[Folias 2005] S. E. Folias and P. C. Bressloff. *Breathers in two-dimensional neural media*. Physical Review Letters, vol. 95, 2005. (Cited on page 157.)

[Forcada 1994] Mikel L. Forcada and Rafael C. Carrasco. *Learning the Initial State of a Second-Order Recurrent Neural Network During Regular-Language Inference*, 1994. (Cited on page 129.)

[Forrest 1987] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace and G. V. Wilson. *Implementing Neural Network Models on Parallel Computers*. vol. 30, no. 5, pages 413–419, 1987. (Cited on page 91.)

[Fortuna 2001] L. Fortuna, P. Arena, D. Balya and A. Zarandy. *Cellular neural networks: a paradigm for nonlinear spatio-temporal processing*. Circuits and Systems Magazine, IEEE, vol. 1, no. 4, pages 6–21, 2001. (Cited on pages 78, 81, 85 and 91.)

[Frasconi 1998] Paolo Frasconi, Marco Gori and Alessandro Sperduti. *A General Framework for Adaptive Processing of Data Structures*. IEEE TRANSACTIONS ON NEURAL NETWORKS, vol. 9, pages 768–786, 1998. (Cited on page 125.)

[Frezza-Buet 2012] H. Frezza-Buet. *BIJAMA (Biologically Inspired Joint Associative MAps) software library*. Website, 2012. `http://malis.metz.supelec.fr/spip.php?article149`. (Cited on page 158.)

[Friston 2008] K. Friston. *Mean-Fields and Neural Masses*. PLoS Comput Biol, vol. 4, no. 8, page e1000081, 2008. (Cited on page 155.)

[Fu 2006] Karl Fu and Yang Cai. *Spatiotemporal Data Mining with Cellular Automata*. In VassilN. Alexandrov, GeertDick Albada, PeterM.A. Sloot and Jack Dongarra, editeurs, Computational Science - ICCS 2006, volume 3991 of *Lecture Notes in Computer Science*, pages 1001–1004. Springer Berlin Heidelberg, 2006. (Cited on page 90.)

[Funahashi 1993] Ken-ichi Funahashi and Yuichi Nakamura. *Approximation of dynamical systems by continuous time recurrent neural networks.* Neural Networks, no. 6, 1993. (Cited on page 108.)

[Furber 2009] Steve Furber and Andrew Brown. *Biologically-Inspired Massively-Parallel Architectures - computing beyond a million processors*. In Proc. 9th International Conference on the Application of Concurrency to System Design (ACSD'09),, July 2009. (Cited on pages 62 and 76.)

[Gallagher 1999] John C. Gallagher and Randall D. Beer. *Evolution and Analysis of Dynamical Neural Networks for Agents Integrating Vision, Locomotion, and Short-Term Memory*. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99, pages 1273–1280. Morgan Kaufmann, 1999. (Cited on page 76.)

[Gardner 1983] Martin Gardner. Wheels, life, and other mathematical amusements. Freeman, 1983. (Cited on page 44.)

[Gas 1993] B. Gas and R. Natowicz. *Extending discrete Hopfield networks for unsupervised learning of temporal sequences*. In Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of

1993 International Joint Conference on, volume 3, pages 2714–2718 vol.3, 1993. (Cited on page 122.)

[Gerstner 1992] Wulfram Gerstner and J.Leo Hemmen. *Universality in neural networks: the importance of the mean firing rate*. Biological Cybernetics, 1992. (Cited on page 74.)

[Giles 1991] C.L. Giles, D. Chen, C. B. Miller, H.H. Chen, G.Z. Sun and Y.-C. Lee. *Second-order recurrent neural networks for grammatical inference*. In Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on, volume ii, pages 273–281 vol.2, 1991. (Cited on page 124.)

[Giles 1992] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun and Y. C. Lee. *Learning and extracting finite state automata with second-order recurrent neural networks*. Neural Comput., vol. 4, no. 3, pages 393–405, May 1992. (Cited on page 129.)

[Gilli 1994] M. Gilli. *Stability of cellular neural networks and delayed cellular neural networks with nonpositive templates and nonmonotonic output functions*. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on, vol. 41, no. 8, pages 518–528, 1994. (Cited on page 78.)

[Girau 2000] Bernard Girau. *Neural networks on FPGAs: a survey*. In Second ICSC Symposium on Neural Computation - NC'2000, Berlin, Germany, 2000. (Cited on page 76.)

[Gläser 2008a] C. Gläser, F. Joublin and C. Goerick. *Enhancing topology preservation during neural field development via wiring length minimization*. In ICANN '08: Proceedings of the 18th International Conference on Artificial Neural Networks, Part I, pages 593–602. Springer-Verlag, 2008. (Cited on page 159.)

[Gläser 2008b] C. Gläser, F. Joublin and C. Goerick. *Homeostatic development of dynamic neural fields*. In Proc. International Conference on Development and Learning 2008, pages 121–126, 2008. (Cited on page 159.)

[Gobovic 1994] D. Gobovic and M.E. Zaghloul. *Analog cellular neural network with application to partial differential equations with variable mesh-size*. In Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on, volume 6, pages 359–362 vol.6, 1994. (Cited on page 78.)

[Gollas 2005] F. Gollas and R. Tetzlaff. *Modeling complex systems by reaction-diffusion cellular nonlinear networks with polynomial weight-functions*. In Cellular Neural Networks and Their Applications, 2005 9th International Workshop on, pages 227–231, 2005. (Cited on page 78.)

[Gori 1998] M. Gori, M. Maggini, E. Martinelli and G. Soda. *Inductive inference from noisy examples using the hybrid finite state filter*. Neural Networks, IEEE Transactions on, vol. 9, no. 3, pages 571–575, 1998. (Cited on page 129.)

[Gorman 1988] R. Paul Gorman and Terrence J. Sejnowski. *Analysis of hidden units in a layered network trained to classify sonar targets.* Neural Networks, vol. 1, no. 1, pages 75–89, 1988. (Cited on page 98.)

[Goudreau 1994] Mark W. Goudreau, C. Lee Giles, Srimat T. Chakradhar and D. Chen. *First-Order vs. Second-Order Single Layer Recurrent Neural Networks*. IEEE TRANSACTIONS ON NEURAL NETWORKS, vol. 5, no. 3, pages 511–513, 1994. (Cited on page 124.)

[Gros 2009] C. Gros. *Cognitive computation with autonomously active neural networks: an emerging field*. Cognitive Computation, vol. 1, pages 77–90, 2009. (Cited on page 158.)

[Gross 1998] H.-M. Gross, V. Stephan and M. Krabbes. *A neural field approach to topological reinforcement learning in continuous action spaces*. In Proc. of WCCI-IJCNN'98, Anchorage, pages 1992–1997. IEEE Press, 1998. (Cited on page 159.)

[Grossberg 1976] S. Grossberg. *Adaptive pattern classification and universal recoding: I. Parallel development and coding of neural feature detectors*. Biological Cybernetics, vol. 23, pages 121–134, 1976. 10.1007/BF00344744. (Cited on page 180.)

[Guimaraes 2001] G. Guimaraes, J. h. Peter, T. Penzel and A. Ultsch. *A method for automated temporal knowledge acquisition applied to sleep-related breathing disorders*. Artificial Intelligence in Medicine, page 237, 2001. (Cited on page 136.)

[Guimaraes 2003] Gabriela Guimaraes, Victor Sousa Lobo and Fernando Moura-Pires. *A taxonomy of Self-organizing Maps for temporal sequence processing.* Intell. Data Anal., vol. 7, no. 4, pages 269–290, 2003. (Cited on pages 136 and 139.)

[Gustedt 2011] Jens Gustedt, Stephane Vialle, Hervé Frezza-Buet, D'havh Boumba Sitou, Nicolas Fressengeas and Jérémy Fix. *InterCell: a Software Suite for Rapid Prototyping and Parallel Execution of Fine Grained Applications*. In Kristjan Jonasson (Ed), editeur, Applied Parallel and Scientific Computing, 10th International Conference, PARA 2010, Proceedings, Part I, volume 7133 of *LNCS*. Springer, Heidelberg, 2011. (Cited on page 208.)

[Hagenbuchner 2003] M. Hagenbuchner, A. Sperduti and Ah Chung Tsoi. *A self-organizing map for adaptive processing of structured data*. Neural Networks, IEEE Transactions on, vol. 14, no. 3, pages 491–505, 2003. (Cited on page 149.)

[Hajjar 2013] Chantal Hajjar and Hani Hamdan. *Interval data clustering using self-organizing maps based on adaptive Mahalanobis distances*. Neural Networks, vol. 46, no. 0, pages 124 – 132, 2013. (Cited on pages 133 and 138.)

[Hammer 2004a] Barbara Hammer, Alessio Micheli, Alessandro Sperduti and Marc Strickert. *A general framework for unsupervised processing of structured data*. Neurocomputing, vol. 57, pages 3–35, 2004. (Cited on page 172.)

[Hammer 2004b] Barbara Hammer, Alessio Micheli, Alessandro Sperduti and Marc Strickert. *Recursive self-organizing network models*. Neural Networks, vol. 17, no. 8-9, pages 1061–1085, 2004. (Cited on pages 136, 144, 146, 148 and 149.)

[Hammer 2004c] Barbara Hammer, Alessio Micheli, Alessandro Sperduti and Marc Strickert. *Recursive self-organizing network models*, 2004. (Cited on page 150.)

[Hammer 2006] Barbara Hammer and N. Neubauer. *On the capacity of unsupervised recursive neural networks for symbol processing*. Workshop proceedings of NeSy'06, 2006. (Cited on page 150.)

[Haykin 1998a] Simon Haykin. Neural networks: A comprehensive foundation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 1998. (Cited on page 72.)

[Haykin 1998b] Simon Haykin. Neural networks: A comprehensive foundation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 1998. (Cited on page 127.)

[Hebb 1949] D.O. Hebb. The organization of behavior: a neuropsychological theory. Wiley book in clinical psychology. Wiley, 1949. (Cited on page 118.)

[Hebb 2002] D.O. Hebb. The organization of behavior: A neuropsychological theory. Taylor & Francis, 2002. (Cited on page 72.)

[Hecht-Nielsen 1989] R. Hecht-Nielsen. *Theory of the backpropagation neural network*. In Neural Networks, 1989. IJCNN., International Joint Conference on, pages 593–605 vol.1, 1989. (Cited on page 91.)

[Hertz 1991] J. Hertz, A. Krogh and R.G. Palmer. Introduction to the theory of neural computation. Addison-Wesley, 1991. (Cited on page 121.)

[Heskes 2001] T. Heskes. *Self-organizing maps, vector quantization, and mixture modeling*. Neural Networks, IEEE Transactions on, vol. 12, no. 6, pages 1299–1305, 2001. (Cited on page 135.)

[Hochreiter 2001] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi and Jürgen Schmidhuber. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*, 2001. (Cited on page 114.)

[Hodges 1997] Andrew Hodges. *Turing, Alan, 1912-1954. Artificial intelligence.* Phoenix, London, 1997. (Cited on page 35.)

[Hodges 2002] Hodges. *What would Alan Turing have done after 1954?* `http://www.turing.org.uk/philosophy/lausanne1.html`, 2002. [Online; accessed 01-April-2013]. (Cited on page 34.)

[Hodges 2011] Andrew Hodges. *Alan Turing*. In Edward N. Zalta, editeur, The Stanford Encyclopedia of Philosophy. Summer 2011 édition, 2011. (Cited on page 34.)

[Hodgkin 1952] A.L. Hodgkin and A.F. Huxley. *A quantitative description of membrane current and its application to conduction and excitation in nerve*. Journal of Physiology, vol. 117, no. 4, pages 500–44, 1952. (Cited on page 155.)

[Hopfield 1982] John J. Hopfield. *Neural networks and physical systems with emergent collective computational abilities*. Proceedings of the National Academy of Sciences of the USA, vol. 79, no. 8, pages 2554–2558, 1982. (Cited on page 119.)

[Horio 2008] K. Horio, T. Koga and T. Yamakawa. *Self-organizing map with distance measure defined by data distribution*. In Automation Congress, 2008. WAC 2008. World, pages 1–6, 2008. (Cited on pages 133 and 138.)

[Horne 1995] BillG. Horne, HavaT. Siegelmann and C.Lee Giles. *What NARX networks can compute*. In Miroslav Bartosek, Jan Staudek and Jirí Wiedermann, editeurs, SOFSEM '95: Theory and Practice of Informatics, volume 1012 of *Lecture Notes in Computer Science*, pages 95–102. Springer Berlin Heidelberg, 1995. (Cited on page 115.)

[Horne 1998] Bill G. Horne, C. Lee Giles, Pete C. Collingwood, School Of Computing, Man Sci, Peter Tino and Peter Tino. *Finite State Machines and Recurrent Neural Networks – Automata and Dynamical Systems Approaches*. In Neural Networks and Pattern Recognition, pages 171–220. Academic Press, 1998. (Cited on page 129.)

[Hornik 1989] K. Hornik, M. Stinchcombe and H. White. *Multilayer feedforward networks are universal approximators*. Neural Netw., 1989. (Cited on page 72.)

[Hutt 2003] A. Hutt, M. Bestehorn and T. Wennekers. *Pattern formation in intracortical neuronal fields*. Network: Computation in Neural Systems, vol. 14, pages 351–368, 2003. (Cited on page 159.)

[Hynna 2006] KevinI. Hynna and Mauri Kaipainen. *Activation-Based Recursive Self-Organising Maps: A General Formulation and Empirical Results*. Neural Processing Letters, vol. 24, no. 2, pages 119–136, 2006. (Cited on page 148.)

[Iossifidis 2001] I. Iossifidis and A. Steinhage. *Controlling an 8 dof manipulator by means of neural fields*. In Int. Conf. on Field and Service Robotics (FSR 2001). Yleisjaljennos-Painoporssi, 2001. (Cited on page 158.)

[Izhikevich 2004] E.M. Izhikevich. *Which model to use for cortical spiking neurons?* Neural Networks, IEEE Transactions on, vol. 15, no. 5, pages 1063–1070, 2004. (Cited on page 155.)

[Jacobsson 2005] Henrik Jacobsson. *Rule Extraction from Recurrent Neural Networks: a Taxonomy and Review*. Neural Computation, vol. 17, pages 1223–1263, 2005. (Cited on pages 76 and 127.)

[Jacquemin 1994] Christian Jacquemin. *A temporal connectionist approach to natural language*. SIGART Bull., vol. 5, no. 3, pages 12–22, 1994. (Cited on page 98.)

[Jaeger 2001] H. Jaeger. *The "echo state" approach to analysing and training recurrent neural networks*. GMD Report 148, GMD - German National Research Institute for Computer Science, 2001. (Cited on page 118.)

[Jaeger 2002a] H. Jaeger. *Short term memory in echo state networks, Technical report*. Gmd report, German National Research Center for Information Technology, 2002. (Cited on page 117.)

[Jaeger 2002b] H. Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach*. Rapport technique, 2002. (Cited on pages 114 and 115.)

[Jaeger 2005] H. Jaeger. *Reservoir riddles: suggestions for echo state network research*. In Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on, volume 3, pages 1460–1462 vol. 3, 2005. (Cited on page 117.)

[James 1995] Daniel L. James and Risto Miikkulainen. *SARDNET: A Self-Organizing Feature Map For Sequences*. In G. Tesauro, D. S. Touretzky and T. K. Leen, editeurs, Advances in Neural Information Processing Systems 7 (NIPS'94), pages 577–584, Denver, CO, 1995. Cambridge, MA: MIT Press. (Cited on page 143.)

[Jiang 1997] J. Jiang. *A parallel computing and neural network implementation of LBG image vector quantization*. In Image Processing and Its Applications, 1997., Sixth International Conference on, volume 1, pages 27–31 vol.1, 1997. (Cited on page 91.)

[Jordan 1986] M Jordan. *Serial Order: A Parallel Distributed Processing Approach*. Rapport technique, 1986. (Cited on page 110.)

[Jordan 1990] Michael I. Jordan. *Artificial neural networks*. chapitre Attractor dynamics and parallelism in a connectionist sequential machine, pages 112–127. IEEE Press, Piscataway, NJ, USA, 1990. (Cited on page 105.)

[Jr. 2006] Luís O. Rigo Jr. and Valmir C. Barbosa. *Two-dimensional cellular automata and the analysis of correlated time series*. Pattern Recognition Letters, vol. 27, no. 12, pages 1353 – 1360, 2006. (Cited on page 85.)

[Juang 2004] Chia-Feng Juang, Shiuan-Jiun Ku and Hao-Jung Huang. *Fuzzy temporal sequence processing by recurrent neural fuzzy network*. In Systems, Man and Cybernetics, 2004 IEEE International Conference on, volume 6, pages 5847–5851 vol.6, 2004. (Cited on page 96.)

[Kaminski 1990] Michael Kaminski and Nissim Francez. *Finite-memory automata*. In Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on, pages 683–688 vol.2, 1990. (Cited on page 150.)

[Kang 2009] Pilsung Kang, Naresh K. Selvarasu, Naren Ramakrishnan, Calvin J. Ribbens, Danesh K. Tafti and Srinidhi Varadarajan. *Modular, Fine-Grained Adaptation of Parallel Programs*. In Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, pages 269–279, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 57.)

[Kangas 1990a] J. Kangas. *Time-delayed self-organizing maps*. In Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pages 331–336 vol.2, 1990. (Cited on pages 137 and 151.)

[Kangas 1990b] J.A. Kangas, T.K. Kohonen and J.T. Laaksonen. *Variants of self-organizing maps*. Neural Networks, IEEE Transactions on, vol. 1, no. 1, pages 93–99, 1990. (Cited on page 136.)

[Kangas 1992] J. Kangas. *Temporal Knowledge in locations of activations in a self-organizing map*. In Artificial Neural Networks 2, pages 117–120. I. Aleksander and J. Taylor, 1992. (Cited on page 143.)

[Kilian 1996] Joe Kilian and Hava T. Siegelmann. *The dynamic universality of sigmoidal neural networks*. Inf. Comput., pages 48–56, 1996. (Cited on page 74.)

[Kim 2006] Kyung-Joong Kim and Sung-Bae Cho. *Evolved neural networks based on cellular automata for sensory-motor controller*. Neurocomputing, vol. 69, 2006. (Cited on page 74.)

[Klapper-Rybicka 2001] Magdalena Klapper-Rybicka, Nicol N. Schraudolph and Jürgen Schmidhuber. *Unsupervised Learning in LSTM Recurrent Neural Networks*. pages 684–691. Springer-Verlag, 2001. (Cited on page 124.)

[Kleene 1967] S.C. Kleene. *Mathematical logic*. Wiley, page 232, 1967. (Cited on page 33.)

[Kohonen 1982] Teuvo Kohonen. *Self-organized formation of topologically correct feature maps*. Biological Cybernetics, vol. 43, no. 1, pages 59–69, 1982. (Cited on page 132.)

[Kohonen 1988] T. Kohonen. *The 'neural' phonetic typewriter*. Computer, vol. 21, no. 3, pages 11–22, 1988. (Cited on page 138.)

[Kohonen 1991] Teuvo Kohonen. *The Hypermap Architecture*. In T. Kohonen, K. Mäkisara, O. Simula and J. Kangas, editeurs, Artificial Neural Networks, volume II, pages 1357–1360, Amsterdam, Netherlands, 1991. North-Holland. (Cited on pages 140 and 141.)

[Kohonen 1997] Teuvo Kohonen. Self organizing maps. Springer, 1997. Second Edition. (Cited on page 185.)

[Kohonen 2001] T. Kohonen, M. R. Schroeder and T. S. Huang, editeurs. Self-organizing maps. Springer-Verlag, Inc., 3rd édition, 2001. (Cited on pages 140 and 182.)

[Kolen 2001] J.F. Kolen and S.C. Kremer. A field guide to dynamical recurrent networks. Wiley, 2001. (Cited on pages 55, 56, 74 and 114.)

[Kopecz 1995] K. Kopecz. *Unsupervised learning of sequences on maps with lateral connectivity*. In International Conference on Artificial Neural Networks (ICANN), pages 431–436, 1995. (Cited on pages 152 and 159.)

[Koskela 1996] Timo Koskela, Mikko Lehtokangas, Jukka Saarinen and Kimmo Kaski. *Time Series Prediction with Multilayer Perceptron, FIR and Elman Neural Networks*. In Proceedings of the World Congress on Neural Networks, pages 491–496. Press, 1996. (Cited on pages 100, 103, 108 and 112.)

[Koskela 1997] T. Koskela, M. Varsta, J. Heikkonen and K. Kaski. *Time Series Prediction Using Recurrent SOM with Local Linear Models*. Int. J. of Knowledge-Based Intelligent Engineering Systems, pages 60–68, 1997. (Cited on pages 136 and 146.)

[Koskela 1998] T. Koskela, M. Varsta, J. Heikkonen and K. Kaski. *Temporal sequence processing using recurrent SOM*. In Knowledge-Based Intelligent Electronic Systems, 1998. Proceedings KES '98. 1998 Second International Conference on, volume 1, pages 290–297 vol.1, 1998. (Cited on page 141.)

[Krap 1982] R. M. Krap and R. Lipton. *Turing machines that take advice*. Enseignement Mathematique, 1982. (Cited on page 74.)

[Kremer 1995] S.C. Kremer. *On the computational power of Elman-style recurrent networks*. Neural Networks, IEEE Transactions on, vol. 6, no. 4, pages 1000–1004, 1995. (Cited on page 128.)

[Kremer 1996] Stefan Charles Kremer. *A theory of grammatical induction in the connectionist paradigm*. PhD thesis, Edmonton, Alta., Canada, 1996. AAINN10604. (Cited on page 127.)

[Kremer 1999] Stefan C. Kremer. *Identification of a specific limitation on local-feedback recurrent networks acting as Mealy-Moore machines.* IEEE Transactions on Neural Networks, vol. 10, no. 2, pages 433–438, 1999. (Cited on page 113.)

[Kremer 2001] Stefan C. Kremer. *Spatiotemporal Connectionist Networks: A Taxonomy and Review*. Neural Comput., vol. 13, no. 2, pages 249–306, 2001. (Cited on page 110.)

[Kugel 2002] Peter Kugel. *Computing Machines Can't Be Intelligent (...and Turing Said So)*. Minds Mach., vol. 12, no. 4, pages 563–579, 2002. (Cited on page 35.)

[Kurogi 1991] S. Kurogi. *Speech recognition by an artificial neural network using findings on the afferent auditory system*. Biological Cybernetics, vol. 64, no. 3, pages 243–249, 1991. (Cited on page 90.)

[Lang 1990] Kevin J. Lang, Alex H. Waibel and Geoffrey E. Hinton. *A time-delay neural network architecture for isolated word recognition*. Neural Netw., vol. 3, no. 1, pages 23–43, 1990. (Cited on pages 98 and 104.)

[Langton 1990] Chris G. Langton. *Computation at the edge of chaos: phase transitions and emergent computation*. Phys. D, vol. 42, no. 1-3, pages 12–37, June 1990. (Cited on page 67.)

[Lapedes 1987] A. Lapedes and R. Farber. *Nonlinear signal processing using neural networks*. Report No.LA-UR-87-2662, 1987. (Cited on page 104.)

[Lazar 2009] Andrea Lazar, Gordon Pipa and Jochen Triesch. *SORN: a self-organizing recurrent neural network*. Frontiers in Computational Neuroscience, vol. 3, no. 23, 2009. (Cited on pages 118, 153 and 154.)

[Lee 2002] John Aldo Lee and Michel Verleysen. *Self-organizing maps with recursive neighborhood adaptation.* Neural Networks, vol. 15, no. 8-9, pages 993–1003, 2002. (Cited on page 135.)

[Lehtokangas 1996] Mikko Lehtokangas, Jukka Saarinen, Kimmo Kaski and Pentti Huuhtanen. *A network of autoregressive processing units for time series modeling*. Appl. Math. Comput., vol. 75, no. 2-3, pages 151–165, March 1996. (Cited on page 142.)

[Lendasse 2005] A. Lendasse, D. Francois, V. Wertz and M. Verleysen. *Vector Quantization: A Weighted Version For Time-Series Forecasting*, 2005. (Cited on page 142.)

[Leontaritis 1985] I. J. Leontaritis and S. A. Billings. *Input-output parametric models for non-linear systems Part I: deterministic non-linear systems*. International Journal of Control, vol. 41, no. 2, pages 303–328, 1985. (Cited on page 113.)

[Li 2002] Xia Li and Anthony Gar-On Yeh. *Neural-network-based cellular automata for simulating multiple land use changes using GIS*. International Journal of Geographical Information Science, 2002. (Cited on page 74.)

[Li 2008] Fengjun Li. *Function Approximation by Neural Networks*. In Fuchun Sun, Jianwei Zhang, Ying Tan, Jinde Cao and Wen Yu, editeurs, Advances in Neural Networks - ISNN 2008, volume 5263 of *Lecture Notes in Computer Science*, pages 384–390. Springer Berlin Heidelberg, 2008. (Cited on page 76.)

[Lin 1996] Tsungnan Lin, B.G. Horne, P. Tino and C.L. Giles. *Learning long-term dependencies in NARX recurrent neural networks*. Neural Networks, IEEE Transactions on, vol. 7, no. 6, pages 1329–1338, 1996. (Cited on pages 128 and 129.)

[Lin 2009] Xianghong Lin and Tianwen Zhang. *Event-driven simulation of integrate-and-fire models with spike-frequency adaptation*. Journal of Electronics (China), vol. 26, no. 1, pages 120–127, 2009. (Cited on page 84.)

[Long 2005] Lyle N. Long and Ankur Gupta. *Scalable Massively Parallel Artificial Neural Networks*. 2005. (Cited on page 76.)

[Lopez-Rodriguez 2005] Domingo Lopez-Rodriguez, Enrique Merida Casermeiro and Juan Miguel Ortiz de Lazcano-Lobato. *Hopfield Network as Associative Memory with Multiple Reference Points.* In Cemal Ardil, editeur, IEC (Prague), pages 62–67. Enformatika, Canakkale, Turkey, 2005. (Cited on page 76.)

[Luitel 2012] B. Luitel and G.K. Venayagamoorthy. *Decentralized Asynchronous Learning in Cellular Neural Networks*. Neural Networks and Learning Systems, IEEE Transactions on, vol. 23, no. 11, pages 1755–1766, 2012. (Cited on page 85.)

[Lukoševičius 2009] Mantas Lukoševičius and Herbert Jaeger. *Reservoir computing approaches to recurrent neural network training*. Computer Science Review, vol. 3, no. 3, pages 127–149, 2009. (Cited on pages 108, 116, 117 and 118.)

[Ma 1998] Sheng Ma and Chuanyi Ji. *Fast training of recurrent networks based on the EM algorithm.* IEEE Transactions on Neural Networks, vol. 9, no. 1, pages 11–26, 1998. (Cited on page 125.)

[Maass 1996] Wolfgang Maass. *Networks of Spiking Neurons: The Third Generation of Neural Network Models*. Neural Networks, vol. 10, pages 1659–1671, 1996. (Cited on page 74.)

[Maass 2002] Wolfgang Maass, Thomas Natschläger and Henry Markram. *Real-time computing without stable states: a new framework for neural computation based on perturbations.* Neural Comput, vol. 14, no. 11, pages 2531–2560, 2002. (Cited on page 116.)

[Maass 2006] Wolfgang Maass, Prashant Joshi and Eduardo D. Sontag. *Principles of real-time computing with feedback applied to cortical microcircuit models*. In Proceedings of Advances in Neural Information Processing Systems, pages 835–842. MIT Press, 2006. (Cited on page 116.)

[Mahajan 2009] Yogesh Mahajan and Parvatham Venkatachalam. *Neural Network Based Cellular Automata Model for Dynamic Spatial Modeling in GIS*. In Computational Science and Its Applications - ICCSA 2009, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009. (Cited on page 74.)

[Manolios 1994] Peter Manolios and Robert Fanelli. *First Order Recurrent Neural Networks and Deterministic Finite State Automata*. Neural Computation, vol. 6, pages 1155–1173, November 1994. (Cited on page 129.)

[Margolus 1993] Norman Margolus. *CAM-8: a computer architecture based on cellular automata*. In Pattern Formation and Lattice-Gas Automata, 1993. (Cited on page 86.)

[Markgraf 2007] Joey Markgraf. *The Von Neumann bottleneck*. `http://aws.linnbenton.edu/cs271c/markgrj/`, 2007. [Online; accessed 01-April-2013]. (Cited on page 37.)

[Martinetz 1993] Thomas M. Martinetz, Stanislav G. Berkovitsch and Klaus J. Schulten. *"Neural-Gas" Network for Vector Quantization and its Application to Time-Series Prediction*. IEEE Transactions on Neural Networks, vol. 4, pages 558–569, 1993. (Cited on page 150.)

[Maurer 2005] André Maurer, Micha Hersch and Aude G. Billard. *Extended hopfield network for sequence learning: Application to gesture recognition*. In Proceedings of the ICANN 2005, 2005. (Cited on page 122.)

[Mayberry 1999] Marshall R. Mayberry and Risto Miikkulainen. *SARDSRN: A Neural Network Shift-Reduce Parser*. In Proceedings of the 16th International Joint Conference on Artificial Intelligence, pages 820–825. Morgan Kaufmann, 1999. (Cited on page 151.)

[Mayer 2006] H. Mayer, F. Gomez, D. Wierstra, I. Nagy, A. Knoll and J. Schmidhuber. *A System for Robotic Heart Surgery that Learns to Tie Knots Using Recurrent Neural Networks*. In Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, pages 543–548, 2006. (Cited on page 124.)

[Mcqueen 2002] T. A. Mcqueen, A. A. Hopgood and T. J. Allen. *A recurrent self-organizing map for temporal sequence processing*. In Proceedings of Fourth International Conference in Recent Advances in Soft Computing (RASC2002, 2002. (Cited on pages 149 and 155.)

[Mealy 1955] George H. Mealy. *A Method for Synthesizing Sequential Circuits*. Bell System Technical Journal, vol. 34, no. 5, pages 1045–1079, 1955. (Cited on page 54.)

[Ménard 2005] O. Ménard and H. Frezza-Buet. *Model of multi-modal cortical processing: Coherent learning in self-organizing modules*. Neural Networks, vol. 18, no. 5-6, pages 646–655, 2005. (Cited on pages 158, 159, 164, 170 and 171.)

[Meyer 2007] J. Meyer. *Modelling primary visual cortex dynamics with a dynamic neural field based on voltage sensitive dyes*. PhD thesis, Ruhr-University Bochum, 2007. (Cited on page 159.)

[Midenet 1994] Sophie Midenet and Alain Grumbach. *Learning Associations by Self-Organization: The LASSO model*. Neurocomputing, vol. 6, no. 3, pages 343 – 361, 1994. (Cited on page 141.)

[Miikkulainen 2005a] R. Miikkulainen, J. A. Bednar, Y. Choe and J. Sirosh. Computational maps in the visual cortex. Springer, 2005. (Cited on page 159.)

[Miikkulainen 2005b] Risto Miikkulainen, James A. Bednar, Yoonsock Choe and Joseph Sirosh. Computational maps in the visual cortex. Springer, 2005. (Cited on page 182.)

[Mishtal 2012] A. Mishtal and I. Arel. *Jensen-Shannon Divergence in Ensembles of Concurrently-Trained Neural Networks*. In Machine Learning and Applications (ICMLA), 2012 11th International Conference on, volume 2, pages 558–562, 2012. (Cited on page 126.)

[Misra 2010] J.a Misra and I.b Saha. *Artificial neural networks in hardware: A survey of two decades of progress*. Neurocomputing, vol. 74, no. 1-3, pages 239–255, 2010. (Cited on page 91.)

[Mitchell 1996] Melanie Mitchell. *Computation in cellular automata: A selected review*. Nonstandard Computation, pages 385–390, 1996. (Cited on pages 64 and 65.)

[Miyoshi 2004] Seiji Miyoshi, Hiro-Fumi Yanai and Masato Okada. *Associative memory by recurrent neural networks with delay elements.* Neural Networks, vol. 17, no. 1, pages 55–63, 2004. (Cited on page 122.)

[Moon 1998] Chiung Moon, Yin-Zhen Li and M. Gen. *Evolutionary algorithm for flexible process sequencing with multiple objectives*. In Evolutionary Computation Proceedings, 1998.

IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on, pages 27–32, 1998. (Cited on page 96.)

[Moran 2013] R.J. Moran, D.A. Pinotsis and K.J. Friston. *Neural Masses and Fields in Dynamic Causal Modelling*. Frontiers in Computational Neuroscience, vol. 7, no. 57, 2013. (Cited on page 155.)

[Moshou 2004] Dimitrios Moshou and Herman Ramon. *Financial Applications of Wavelets and Self-organizing Maps*. In Shu-Heng Chen and PaulP. Wang, editeurs, Computational Intelligence in Economics and Finance, Advanced Information Processing, pages 234–249. Springer Berlin Heidelberg, 2004. (Cited on page 138.)

[Mountcastle 1957] V. B. Mountcastle. *Modality and topographic properties of single neurons of cat's somatic sensory cortex.* J. Neurophysiol., vol. 20, no. 4, pages 408–434, 1957. (Cited on page 159.)

[Mountcastle 1997] Vernon B. Mountcastle. *The columnar organization of the neocortex*. Brain, vol. 120, pages 701–722, 1997. (Cited on page 159.)

[Mozayyani 1995] N. Mozayyani, V. Alanou, J. F. Dreyfus and G. Vaucher. *A Spatio-Temporal Data-Coding Applied to Kohonen Maps*. In Fogelman F. Soulié and P. Gallinari, editeurs, Proc. ICANN'95, Int. Conf. on Artificial Neural Networks, volume II, pages 75–79, Nanterre, France, 1995. EC2. (Cited on pages 101 and 138.)

[Mozer 1994] Michael C. Mozer. *Neural Net Architectures for Temporal Sequence Processing*. pages 243–264. Addison-Wesley, 1994. (Cited on pages 99 and 100.)

[Mozer 1995] Michael C. Mozer. *Backpropagation*. chapitre A focused backpropagation algorithm for temporal pattern recognition, pages 137–169. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995. (Cited on page 105.)

[Nagumo 1962] S. Nagumo J. Arimoto and S. Yoshizawa. *An active pulse transmission line simulating nerve axon*. Proc IRE, vol. 50, pages 2061–2070, 1962. (Cited on page 155.)

[Nakahara 2002] Hiroyuki Nakahara, Shun-ichi Amari and Okihide Hikosaka. *Self-organization in the basal ganglia with modulation of reinforcement signals*. Neural Computation, vol. 14, pages 819–844, 2002. (Cited on page 159.)

[Nakamura 2010] Katsuhiko Nakamura and Keita Imada. *Incremental Learning of Cellular Automata for Parallel Recognition of Formal Languages*. In Discovery Science, volume 6332 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg, 2010. (Cited on page 85.)

[Natarajan 2008] Rama Natarajan, Quentin J. M. Huys, Peter Dayan and Richard S. Zemel. *Encoding and decoding spikes for dynamic stimuli*. Neural Comput., vol. 20, no. 9, 2008. (Cited on page 98.)

[Neary 2006] Turlough Neary and Damien Woods. *P-completeness of cellular automaton Rule 110*. In International Colloquium on Automata Languages and Programming (ICALP), volume 4051 of LNCS, pages 132–143. Springer, 2006. (Cited on page 70.)

[Nerrand 1993] O. Nerrand, Pierre Roussel-Ragot, Léon Personnaz, Gérard Dreyfus and S. Marcos. *Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms.* Neural Computation, vol. 5, no. 2, pages 165–199, 1993. (Cited on page 110.)

[Neumann 1966] John Von Neumann. Theory of self-reproducing automata. University of Illinois Press, Champaign, IL, USA, 1966. (Cited on page 64.)

[Nicolosi 2010] L. Nicolosi, R. Tetzlaff, F. Abt, A. Blug and H. Hofler. *Cellular Neural Network (CNN) based control algorithms for omnidirectional laser welding processes: Experimental results*. In Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on, pages 1–6, 2010. (Cited on page 90.)

[Omlin 1992] Christian W. Omlin and C. Lee Giles. *Training Second-Order Recurrent Neural Networks using Hints*. In Proceedings of the Ninth International Conference on Machine Learning, pages 363–368. Morgan Kaufmann, 1992. (Cited on page 124.)

[Oohira 2003] Takayuki Oohira, Koichiro Yamauchi and Takashi Omori. *Meta-learning for Fast Incremental Learning*. In Okyay Kaynak, Ethem Alpaydin, Erkki Oja and Lei Xu, editeurs, Artificial Neural Networks and Neural Information Processing – ICANN/ICONIP 2003, volume 2714 of *Lecture Notes in Computer Science*, pages 157–164. Springer Berlin Heidelberg, 2003. (Cited on page 92.)

[Paasio 1997] A. Paasio, A. Dawidziuk and Veikko Porra. *VLSI implementation of cellular neural network universal machine*. In Circuits and Systems, 1997. ISCAS '97., Proceedings of 1997 IEEE International Symposium on, volume 1, pages 545–548 vol.1, 1997. (Cited on page 78.)

[Park 1991] J. Park and I. W. Sandberg. *Universal approximation using radial-basis-function networks*. Neural Comput., vol. 3, no. 2, pages 246–257, 1991. (Cited on page 101.)

[Park 2008] Han-Saem Park and Sung-Bae Cho. *A Fuzzy Rule-Based System with Ontology for Summarization of Multi-camera Event Sequences*. In Leszek Rutkowski, Ryszard Tadeusiewicz, LotfiA. Zadeh and JacekM. Zurada, editeurs, Artificial Intelligence and Soft Computing ICAISC 2008, volume 5097 of *Lecture Notes in Computer Science*, pages 850–860. Springer Berlin Heidelberg, 2008. (Cited on page 96.)

[Pavlidis 2003] N.G. Pavlidis, D.K. Tasoulis and M.N. Vrahatis. *Financial forecasting through unsupervised clustering and evolutionary trained neural networks*. In Evolutionary Computation, 2003. CEC '03. The 2003 Congress on, volume 4, 2003. (Cited on page 95.)

[Peres 2006] R.T. Peres and C.E. Perreira. *Preliminary Results on Noise Detection and Data Selection for Vector Quantization*. In Neural Networks, 2006. IJCNN '06. International Joint Conference on, pages 3617–3621, 2006. (Cited on page 132.)

[Polk 2002] T.A. Polk and C.M. Seifert. Cognitive modeling. A Bradford Press. MIT Press, 2002. (Cited on pages 120, 121, 122 and 123.)

[Preston 1985] K. Preston and M.J.B. Duff. Modern cellular automata: Theory and applications. Advanced Applications in Pattern Recognition. Springer, 1985. (Cited on page 89.)

[Principe 1995] Jose C. Principe and Jyh ming Kuo. *Dynamic Modelling of Chaotic Time Series with Neural Networks*. In Advances in Neural Information Processing Systems, 7, pages 311–318. MIT Press, 1995. (Cited on page 206.)

[Principe 1998] J.C. Principe, Ludong Wang and M.A. Motter. *Local dynamic modeling with self-organizing maps and applications to nonlinear system identification and control*. Proceedings of the IEEE, vol. 86, no. 11, pages 2240–2258, 1998. (Cited on pages 142, 206 and 207.)

[Przytula 1993] K.W. Przytula and V.K.P. Kumar. Parallel digital implementations of neural networks. PTR Prentice Hall, 1993. (Cited on page 91.)

[Qian 1996] F. Qian and H. Hirata. *A parallel learning cellular automata for combinatorial optimization problems*. In Evolutionary Computation, 1996., Proceedings of IEEE International Conference on, pages 553–558, 1996. (Cited on page 85.)

[Qin 2005] Zheng Qin, Junying Chen, Yu Liu and Jiang Lu. *Evolving RBF Neural Networks for Pattern Classification*. In Yue Hao, Jiming Liu, Yuping Wang, Yiu-ming Cheung, Hujun Yin, Licheng Jiao, Jianfeng Ma and Yong-Chang Jiao, editeurs, Computational Intelligence and Security, volume 3801 of *Lecture Notes in Computer Science*, pages 957–964. Springer Berlin Heidelberg, 2005. (Cited on page 76.)

[Ramanathan 2009] Kiruthika Ramanathan, Luping Shi, Jianming Li, KianGuan Lim, MingHui Li, ZhiPing Ang and TowChong Chong. *A Neural Network Model for a Hierarchical Spatio-temporal Memory*. In Mario Köppen, Nikola Kasabov and George Coghill, editeurs, Advances in Neuro-Information Processing, volume 5506 of *Lecture Notes in Computer Science*, pages 428–435. Springer Berlin Heidelberg, 2009. (Cited on page 90.)

[Ray 1996] Sylvian R. Ray and Hillol Kargupta. *A Temporal Sequence Processor Based on the Biological Reaction-Diffusion Process*. Complex Systems, vol. 9, pages 305–327, 1996. (Cited on page 92.)

[Raytchev 2001] B. Raytchev and H. Murase. *Unsupervised face recognition from image sequences based on clustering with attraction and repulsion*. In Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, volume 2, 2001. (Cited on page 95.)

[Ritter 1989] Helge J. Ritter, Thomas M. Martinetz and Klaus J. Schulten. *Topology-Conserving Maps for Learning Visuomotor-Coordination*. Neural Networks, vol. 2, pages 159–168, 1989. (Cited on page 136.)

[Rolf Pfeifer 2010] Rudolf Fuchslin Rolf Pfeifer Dana Damian. Neural networks. University of Zurich, 2010. (Cited on pages 112 and 113.)

[Rombach 2008] N. Rombach, K.J. Murphy, C.R. Reilly and K. Friston. *Bayesian estimation of synaptic physiology from the spectral responses of neural masses*. Neuroimage, vol. 42, no. 1, pages 272–84, 2008. (Cited on page 155.)

[Roska 1990] T. Roska and L.O. Chua. *Cellular neural networks with nonlinear and delay-type template elements*. In Cellular Neural Networks and their Applications, 1990. CNNA-90 Proceedings., 1990 IEEE International Workshop on, pages 12–25, 1990. (Cited on page 78.)

[Roska 1993] T. Roska and L.O. Chua. *The CNN universal machine: an analogic array computer*. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, vol. 40, no. 3, pages 163–173, 1993. (Cited on page 78.)

[Roska 1999] Tamás Roska. *Computer-Sensors: Spatial-Temporal Computers for Analog Array Signals, Dynamically Integrated with Sensors*. Journal of VLSI signal processing systems for signal, image and video technology, vol. 23, no. 2-3, pages 221–237, 1999. (Cited on page 78.)

[Roska 2002] T. Roska. *Computational and computer complexity of analogic cellular wave computers*. In Cellular Neural Networks and Their Applications, 2002. (CNNA 2002). Proceedings of the 2002 7th IEEE International Workshop on, pages 323–338, 2002. (Cited on page 90.)

[Rumelhart 1985] D.E. Rumelhart, G.E. Hinton, R.J. Williams and San Diego. Institute for Cognitive Science University of California. Learning internal representations by error propagation. ICS report. Institute for Cognitive Science, University of California, San Diego, 1985. (Cited on page 114.)

[Sabatini 2004] S. P. Sabatini, F. S. and L. Secchi. *A continuum-field model of visual cortex stimulus-driven behaviour: emergent oscillations and coherence fields*. Neurocomputing, vol. 57, pages 411–433, 2004. (Cited on page 159.)

[Sahin 2006] Suhap Sahin, Yasar Becerikli and Suleyman Yazici. *Neural Network Implementation in Hardware Using FPGAs*. In Irwin King, Jun Wang, Lai-Wan Chan and DeLiang Wang, editeurs, Neural Information Processing, volume 4234 of *Lecture Notes in Computer Science*, pages 1105–1112. Springer Berlin Heidelberg, 2006. (Cited on page 91.)

[Sandamirskaya 2008] Y. Sandamirskaya and G. Schöner. *Dynamic field theory of sequential action: A model and its implementation on an embodied agent*. In Development and Learning, 2008. ICDL 2008. 7th IEEE International Conference on, pages 133 –138, 2008. (Cited on page 158.)

[Sandholm 2007] Thomas Sandholm. *Autoregressive Time Series Forecasting of Computational Demand*. CoRR, vol. abs/0711.2062, 2007. (Cited on page 95.)

[Scanzio 2010] S. Scanzio, S. Cumani, R. Gemello, F. Mana and P. Laface. *Parallel implementation of artificial neural network training*. In Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on, pages 4902–4905, 2010. (Cited on page 91.)

[Schneider 2002] S. Schneider and W. Erlhagen. *A neural field model for saccade planning in the superior colliculus: speed-accuracy tradeoff in the double-target paradigm*. Neurocomputing, vol. 44-46, pages 623–628, 2002. (Cited on page 159.)

[Schönfisch 1999] Birgitt Schönfisch and André de Roos. *Synchronous and asynchronous updating in cellular automata*. Biosystems, vol. 51, no. 3, pages 123 – 143, 1999. (Cited on pages 58 and 70.)

[Schraudolph 1993] Nicol N. Schraudolph and Terrence J. Sejnowski. *Unsupervised Discrimination of Clustered Data via Optimization of Binary Information Gain*. In Advances in Neural Information Processing Systems, pages 499–506. Morgan Kaufmann, 1993. (Cited on page 124.)

[Seiffert 2001] Udo Seiffert and Bernd Michaelis. *Multi-Dimensional Self-Organizing Maps on Massively Parallel Hardware*. In Advances in Self-Organising Maps, pages 160–166. Springer London, 2001. (Cited on page 76.)

[Sejnowski 1987] Terrence J. Sejnowski and Charles R. Rosenberg. *Parallel Networks that Learn to Pronounce English Text*, 1987. (Cited on page 101.)

[Setti 1996] G. Setti and P. Thiran. *Biological pattern formation with cellular neural networks*. In Cellular Neural Networks and their Applications, 1996. CNNA-96. Proceedings., 1996 Fourth IEEE International Workshop on, pages 279–284, 1996. (Cited on page 78.)

[Shiffman 2012] D. Shiffman. The nature of code. Theoklesia, LLC, 2012. (Cited on page 80.)

[Siegelmann 1994] Hava T. Siegelmann and Eduardo D. Sontag. *Analog Computation Via Neural Networks*. Theoretical Computer Science, vol. 131, pages 331–360, 1994. (Cited on page 75.)

[Siegelmann 1995a] Hava T. Siegelmann. *Computation Beyond the Turing Limit*. Science, vol. 268, no. 5210, pages 545–548, April 1995. (Cited on page 44.)

[Siegelmann 1995b] Hava T. Siegelmann and Eduardo D. Sontag. *On The Computational Power Of Neural Nets*. JOURNAL OF COMPUTER AND SYSTEM SCIENCES, vol. 50, no. 1, 1995. (Cited on pages 127 and 128.)

[Siegelmann 1996] Hava T. Siegelmann. *The simple dynamics of super Turing theories*. Theoretical Computer Science, vol. 168, no. 2, pages 461–472, 1996. (Cited on page 128.)

[Siegelmann 1999] Hava T. Siegelmann. Neural networks and analog computation: Beyond the turing limit. Progress in Theoretical Computer Science. Birkhäuser, 1999. (Cited on page 74.)

[Siegelmann 2003] HavaT. Siegelmann. *Neural and Super-Turing Computing*. Minds and Machines, vol. 13, no. 1, pages 103–114, 2003. (Cited on page 75.)

[Sima 1999] Jiri Sima, Pekka Orponen and Teemu AnttiPoika. *Some Afterthoughts on Hopfield Networks*. In Jan Pavelka, Gerard Tel and Miroslav Bartošek, editeurs, SOFSEM'99: Theory and Practice of Informatics, volume 1725 of *Lecture Notes in Computer Science*, pages 459–469. Springer Berlin Heidelberg, 1999. (Cited on page 119.)

[Sima 2003] Jiri Sima. *Energy-Based Computation with Symmetric Hopfield Nets*. In Limitations and Future Trands in Neural Computation, pages 45–69, 2003. (Cited on page 119.)

[Simmering 2007] V. R. Simmering, A. R. Schutte and J. P. Spencer. *Generalizing the dynamic field theory of spatial cognition across real and developmental time scales*. Brain Research, 2007. (Cited on page 158.)

[Simon 2004] Geoffroy Simon, Amaury Lendasse, Marie Cottrell, Jean-Claude Fort and Michel Verleysen. *Double quantization of the regressor space for long-term time series prediction: method and proof of stability.* Neural Networks, vol. 17, no. 8-9, pages 1169–1181, 2004. (Cited on page 151.)

[Simula 1995] O. Simula and J. Kangas. *Process Monitoring and Visualisation Using Self-Organizing Maps*, 1995. (Cited on page 138.)

[Simula 1996a] O. Simula, E. Alhoniemi, J. Hollmen and J. Vesanto. *Monitoring And Modeling Of Complex Processes Using Hierarchical Self-organizing Maps*. In Circuits and Systems, 1996. ISCAS '96., Connecting the World., 1996 IEEE International Symposium on, volume Supplement, pages 73–76, 1996. (Cited on page 136.)

[Simula 1996b] Olli Simula, Esa Alhoniemi, Jaakko Hollmen and Juha Vesanto. *Monitoring and modeling of complex processes using hierarchical self-organizing maps*. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'96), volume Supplement, pages 737–6, 1996. (Cited on page 151.)

[Sipper 1997] M. Sipper. Evolution of parallel cellular machines: the cellular programming approach. Lecture notes in computer science. Springer, 1997. (Cited on page 85.)

[Sipper 1998a] M. Sipper. *Computing with Cellular Automata: Three Cases for Nonuniformity*. Physical Review E, vol. 57, no. 3, pages 3589–3592, March 1998. (Cited on page 84.)

[Sipper 1998b] Moshe Sipper. *Simple + parallel + local = cellular computing*. In Parallel Problem Solving from Nature — PPSN V, volume 1498 of *Lecture Notes in Computer Science*, pages 653–662. Springer Berlin Heidelberg, 1998. (Cited on pages 40, 64, 81, 164 and 200.)

[Sipper 1999] M Sipper. *The emergence of cellular computing*, 1999. (Cited on pages 40, 41, 82 and 90.)

[Slavova 2012] A. Slavova and P. Zecca. *Cellular Neural Networks modeling of tsunami waves*. In Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on, pages 1–6, 2012. (Cited on page 78.)

[Somervuo 1999] Panu Somervuo and Teuvo Kohonen. *Self-Organizing Maps and Learning Vector Quantization for Feature Sequences.* Neural Processing Letters, vol. 10, no. 2, pages 151–159, 1999. (Cited on page 76.)

[Squier 1994] Richard K. Squier and Ken Steiglitz. *Programmable Parallel Arithmetic Cellular Automata using a Particle Model*, 1994. (Cited on page 70.)

[Srikant 1995] Ramakrishnan Srikant and Rakesh Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements*. In Research Report RJ 9994, IBM Almaden Research, 1995. (Cited on page 95.)

[Srinivasa 1999] N. Srinivasa and N. Ahuja. *A topological and temporal correlator network for spatiotemporal pattern learning, recognition, and recall*. Trans. Neur. Netw., vol. 10, no. 2, pages 356–371, March 1999. (Cited on pages 139 and 151.)

[Stannett 2006] Mike Stannett. *The case for hypercomputation.* Applied Mathematics and Computation, vol. 178, no. 1, pages 8–24, 2006. (Cited on page 33.)

[Storkey 1999] Amos Storkey, Romain Valabregue and Hopfield Learning Rule. *The Basins of Attraction of a new Hopfield Learning Rule*, 1999. (Cited on page 122.)

[Strader 2008] B.P. Strader. Simulating spatial partial differential equations with cellular automata. California State University, San Bernardino, 2008. (Cited on page 90.)

[Strickert 2003] Marc Strickert and Barbara Hammer. *Neural Gas for Sequences*. In PROCEEDINGS OF THE WORKSHOP ON SELF-ORGANIZING NETWORKS (WSOM), PAGES 53-58, KYUSHU INSTITUTE OF TECHNOLOGY, pages 53–57, 2003. (Cited on page 150.)

[Strickert 2005] Marc Strickert and Barbara Hammer. *Merge SOM for temporal data*. Neurocomput., vol. 64, pages 39–71, March 2005. (Cited on pages 149 and 150.)

[Su 1998] Mu-Chun Su, Hi Huang, Chia-Hsien Lin, Chen-Lee Huang and Chi-Da Lin. *Application of neural networks in spatio-temporal hand gesture recognition*. In Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on, volume 3, pages 2116–2121 vol.3, 1998. (Cited on page 90.)

[Sulehria 2007] Humayun Karim Sulehria and Ye Zhang. *Hopfield neural networks: a survey*. In Proceedings of the 6th Conference on 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases - Volume 6, AIKED'07, pages 125–130. World Scientific and Engineering Academy and Society (WSEAS), 2007. (Cited on page 108.)

[Sum 2007] J. Sum, Chi-Sing Leung and Lipin Hsu. *Fault tolerant learning using Kullback-Leibler divergence*. In TENCON 2007 - 2007 IEEE Region 10 Conference, pages 1–4, 2007. (Cited on page 126.)

[Sun 1998] G.Z. Sun, C.L. Giles and H.H. Chen. *The neural network pushdown automaton: Architecture, dynamics and training*. In C.Lee Giles and Marco Gori, editeurs, Adaptive Processing of Sequences and Data Structures, volume 1387 of *Lecture Notes in Computer Science*, pages 296–345. Springer Berlin Heidelberg, 1998. (Cited on page 129.)

[Sun 2001] R. Sun and C.L. Giles. *Sequence learning: from recognition and prediction to sequential decision making*. Intelligent Systems, IEEE, vol. 16, no. 4, pages 67–70, 2001. (Cited on page 96.)

[Sutskever 2010] Ilya Sutskever and Geoffrey E. Hinton. *Temporal-Kernel Recurrent Neural Networks.* Neural Networks, vol. 23, no. 2, pages 239–243, 2010. (Cited on pages 98, 108, 114 and 118.)

[Sutton 1988] Richard S. Sutton. *Learning to predict by the methods of temporal differences*. In MACHINE LEARNING, pages 9–44. Kluwer Academic Publishers, 1988. (Cited on page 125.)

[Sutton 1998a] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction*, 1998. (Cited on page 73.)

[Sutton 1998b] R.S. Sutton and A.G. Barto. Reinforcement learning: An introduction. A Bradford Book, 1998. (Cited on pages 95 and 96.)

[Sutton 2005] Richard S. Sutton and Brian Tanner. *Temporal-difference networks*. In Advances in Neural Information Processing Systems 17, pages 1377–1384. MIT Press, 2005. (Cited on page 125.)

[Suzuki 2013] Kenji Suzuki. Artificial neural networks: Architectures and applications. InTech, 2013. (Cited on page 89.)

[Takens 1981] Floris Takens. *Detecting strange attractors in turbulence*. In David Rand and Lai-Sang Young, editeurs, Dynamical Systems and Turbulence, Warwick 1980, volume 898 of *Lecture Notes in Mathematics*, pages 366–381. Springer Berlin Heidelberg, 1981. (Cited on page 207.)

[Tani 2008] Jun Tani, Ryunosuke Nishimoto and Rainer W. Paine. *Achieving "organic compositionality" through self-organization: reviews on brain-inspired robotics experiments.* Neural Networks, vol. 21, no. 4, pages 584–603, 2008. (Cited on page 125.)

[Technology 2013] Linear Technology. *Wireless Sensor Networks - Dust Networks*. `http://www.linear.com/products/wireless_sensor_networks`, 2013. [Online; accessed 3-august-2013]. (Cited on page 81.)

[Tetzlaff 2002] Ronald Tetzlaff. *Cellular Neural Networks And Their Applications*. Proceedings of the 7th IEEE International Workshop, 2002. (Cited on page 90.)

[Tino 1995] Peter Tino and Jozef Sajda. *Learning and Extracting Initial Mealy Automata With a Modular Neural Network Model*. NEURAL COMPUTATION, vol. 7, no. 4, pages 822–844, 1995. (Cited on page 129.)

[Tino 2006] Peter Tino, Igor Farkas and Jort van Mourik. *Dynamics and Topographic Organization of Recursive Self-Organizing Maps*. Neural Computation, vol. 18, no. 10, pages 2529–2567, 2006. (Cited on pages 132, 136, 148 and 196.)

[Tino 2007] Peter Tino, Barbara Hammer and Mikael Bodén. *Markovian Bias of Neural-based Architectures With Feedback Connections*. In Barbara Hammer and Pascal Hitzler, editeurs, Perspectives of Neural-Symbolic Integration, volume 77 of *Studies in Computational Intelligence*, pages 95–133. Springer Berlin Heidelberg, 2007. (Cited on page 148.)

[Toomarian 1991] N. Toomarian and J. Barhen. *Fast temporal neural learning using teacher forcing*. In Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on, volume i, pages 817–822 vol.1, 1991. (Cited on page 112.)

[Tosic 2004] Predrag T. Tosic. *A perspective on the future of massively parallel computing: fine-grain vs. coarse-grain parallel models comparison & contrast*. In Proceedings of the 1st conference on Computing frontiers, CF '04, pages 488–502, New York, NY, USA, 2004. ACM. (Cited on pages 70, 75, 82, 85 and 86.)

[Toussaint 2006] M. Toussaint. *A sensorimotor map: modulating lateral interactions for anticipation and planning*. Neural Computation, vol. 18, no. 5, pages 1132–1155, 2006. (Cited on page 158.)

[Trappenberg 2008] T. Trappenberg. Decision making and population decoding with strongly inhibitory neural field models. Psychology Press, 2008. (Cited on page 158.)

[Tsoi 1997] Ah Chung Tsoi and Andrew Back. *Discrete time recurrent neural network architectures: A unifying review*. Neurocomputing, vol. 15, no. 3 4, 1997. (Cited on pages 110 and 127.)

[Tsoi 1998] AhChung Tsoi. *Recurrent neural network architectures: An overview*. In C.Lee Giles and Marco Gori, editeurs, Adaptive Processing of Sequences and Data Structures, volume 1387 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 1998. (Cited on page 110.)

[Turing 1936] A. M. Turing. *On Computable Numbers, with an application to the Entscheidungsproblem*. Proc. London Math. Soc., vol. 2, no. 42, pages 230–265, 1936. (Cited on page 32.)

[Turing 1948] A. M. Turing. *Intelligent Machinery*. Report, inst-NPL, 1948. (Cited on page 71.)

[Ultsch 1996] A. Ultsch, G. Guimaraes and W. Schmid. *Classification and prediction of hail using self-organizing neural networks*. In Neural Networks, 1996., IEEE International Conference on, volume 3, pages 1622–1627 vol.3, 1996. (Cited on page 136.)

[Valafar 1993] F. Valafar, O.K. Ersoy and Purdue University. School of Electrical Engineering. A parallel implementation of backpropagation neural network on maspar mp-1. TR-EE. School of Electrical Engineering, Purdue University, 1993. (Cited on page 91.)

[Varsta 1997a] Markus Varsta, Jukka Heikkonen and Jose del R. Millan. *Context Learning with the Self-Organizing Map*, 1997. (Cited on page 145.)

[Varsta 1997b] Markus Varsta, José Millán and Jukka Heikkonen. *A recurrent self-organizing map for temporal sequence processing*. In Artificial Neural Networks ICANN'97, pages 421–426. Springer Berlin / Heidelberg, 1997. (Cited on pages 145 and 146.)

[Vesanto 1997] Juha Vesanto and Juha Vesanto. *Using the SOM and local models in time-series prediction*. In Helsinki University of Technology, pages 209–214, 1997. (Cited on page 136.)

[Vesanto 2000] J. Vesanto and E. Alhoniemi. *Clustering of the self-organizing map*. vol. 11, pages 586–600, 2000. (Cited on page 135.)

[Vilasis-Cardona 2005] X. Vilasis-Cardona and M. Vinvoles-Serra. *On cellular neural network learning*. In Circuit Theory and Design, 2005. Proceedings of the 2005 European Conference on, volume 1, pages I/153–I/156 vol. 1, 2005. (Cited on page 85.)

[Viola 1996] Paul Viola, Nicol N. Schraudolph and Terrence J. Sejnowski. *Empirical Entropy Manipulation for Real-World Problems*. In Neural Information Processing Systems 8, pages 851–857. MIT Press, 1996. (Cited on page 124.)

[Vitali 2011] Stefania Vitali, James B. Glattfelder and Stefano Battiston. *The network of global corporate control*. 2011. (Cited on page 43.)

[Voegtlin 2002] Thomas Voegtlin. *Recursive self-organizing maps*. Neural Networks, vol. 15, no. 8-9, pages 979–991, 2002. (Cited on pages 146, 148, 169, 183, 189, 190 and 195.)

[Vries 1992] Bert De Vries and Jose C. Principe. *The Gamma Model- A New Neural Model for Temporal Processing*. Neural Networks, vol. 5, pages 565–576, 1992. (Cited on pages 99 and 106.)

[Wahlberg 1991] B. Wahlberg. *System identification using Laguerre models*. Automatic Control, IEEE Transactions on, vol. 36, no. 5, pages 551–562, 1991. (Cited on page 106.)

[Waibel 1989] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano and K.J. Lang. *Phoneme recognition using time-delay neural networks*. Acoustics, Speech and Signal Processing, IEEE Transactions on, vol. 37, no. 3, pages 328–339, 1989. (Cited on page 104.)

[Walter 1990] J. Walter, H. Riter and K. Schulten. *Nonlinear prediction with self-organizing maps*. In Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pages 589–594 vol. 1, 1990. (Cited on page 142.)

[Walter 1998] J.A. Walter. *PSOM network: learning with few examples*. In Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on, volume 3, pages 2054–2059 vol.3, 1998. (Cited on page 141.)

[Wang 1995] D. Wang and Budi Yuwono. *Anticipation-based temporal pattern generation*. Systems, Man and Cybernetics, IEEE Transactions on, vol. 25, pages 615–628, 1995. (Cited on page 96.)

[Wang 1998] DeLiang Wang. *The handbook of brain theory and neural networks*. chapitre Temporal pattern processing, pages 967–971. MIT Press, Cambridge, MA, USA, 1998. (Cited on page 93.)

[Weiss 2001] Ron Weiss and Jr. Knight ThomasF. *Engineered communications for microbial robotics*. In Anne Condon and Grzegorz Rozenberg, editeurs, DNA Computing, volume 2054 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2001. (Cited on page 81.)

[Weiss 2003] Ron Weiss, Subhayu Basu, Sara Hooshangi, Abigail Kalmbach, David Karig, Rishabh Mehreja and Ilka Netravali. *Genetic circuit building blocks for cellular computation, communications, and signal processing*. In Natural Computing, pages 47–84, 2003. (Cited on page 81.)

[Werblin 1997] F.S. Werblin and A. Jacobs. *The cellular neural network as a retinal camera for visual prosthesis*. In Neural Networks,1997., International Conference on, volume 4, pages 2327–2332 vol.4, 1997. (Cited on page 90.)

[Werfel 2005] Justin Werfel. *Building patterned structures with robot swarms*. In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05, pages 1495–1502. Morgan Kaufmann Publishers, 2005. (Cited on page 82.)

[Weston 2007] James Weston and Peter Lee. *Cellular Automata Based Binary Arithmetic for use on Self Repairing, Fault Tolerant Hardware*. In Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems, AHS '07, pages 732–739. IEEE Computer Society, 2007. (Cited on page 70.)

[Wikipedia 2013] Wikipedia. *Flynn's taxonomy*. `http://en.wikipedia.org/wiki/Flynn%27s_taxonomy`, 2013. [Online; accessed 17-june-2013]. (Cited on page 61.)

[Williams 1988] R.J. Williams, D. Zipser and San Diego. Institute for Cognitive Science University of California. A learning algorithm for continually running fully recurrent neural networks. ICS report. Institute for Cognitive Science, University of California, San Diego, 1988. (Cited on pages 112, 113 and 122.)

[Williams 1989] Ronald J. Williams and David Zipser. *A Learning Algorithm for Continually Running Fully Recurrent Neural Networks*, 1989. (Cited on page 129.)

[Williams 1992a] Ronald J. Williams. *Some Observations on the Use of the Extended Kalman Filter as a Recurrent Network Learning Algorithm*. Rapport technique, 1992. (Cited on page 125.)

[Williams 1992b] Ronald J. Williams. *Training recurrent networks using the extended Kalman filter*. In Neural Networks, 1992. IJCNN., International Joint Conference on, volume 4, pages 241–246 vol.4, 1992. (Cited on page 110.)

[Wilson 1973] H. R. Wilson and J. D. Cowan. *A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue*. Kybernetik, vol. 13, pages 55–80, 1973. (Cited on page 156.)

[Wolfram 1984] Stephen Wolfram. *Universality and complexity in cellular automata*. Physica D: Nonlinear Phenomena, vol. 10, no. 1-2, pages 1–35, 1984. (Cited on page 67.)

[Wolfram 1986] S. Wolfram. Theory and applications of cellular automata: Including selected papers, 1983-1986. Advanced Series on Complex Systems. World Scientific Publishing Company, Incorporated, 1986. (Cited on page 89.)

[Wolfram 2002] Stephen Wolfram. A new kind of science. Wolfram Media, 2002. (Cited on pages 42, 43, 51 and 67.)

[Wyller 2007] J. Wyller, P. Blomquist and G. T. Einevoll. *Turing instability and pattern formation in a two-population neuronal network model*. Physica D: Nonlinear Phenomena, vol. 225, no. 1, pages 75 – 93, 2007. (Cited on page 157.)

[Xavier-de Souza 2005] Joos Vandewalle Xavier-de Souza Johan A. K. Suykens. *Cellular Learning Automata With Multiple Learning Automata in Each Cell and Its Applications*. International Journal of Circuit Theory and Applications, 2005. (Cited on page 85.)

[Xiao-hua 2009] Liu Xiao-hua, Yuan Da and Li Jin-Jiang. *Image Contour Extraction Based on CNN and Active Contour Model*. In Natural Computation, 2009. ICNC '09. Fifth International Conference on, volume 2, pages 14–17, 2009. (Cited on page 40.)

[Xiao 2003] Xiang Xiao, E.R. Dow, R. Eberhart, Z.B. Miled and R.J. Oppelt. *Gene clustering using self-organizing maps and particle swarm optimization*. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, pages 10 pp.–, 2003. (Cited on page 135.)

[Yin 2002] Hujun Yin. *ViSOM - a novel method for multivariate data projection and structure visualization*. Neural Networks, IEEE Transactions on, vol. 13, no. 1, pages 237–243, 2002. (Cited on page 135.)

[Zegers 2003] Pablo Zegers and Malur K. Sundareshan. *Trajectory Generation and Modulation Using Dynamic Neural Networks*. IEEE Transactions on Neural Networks, 2003. (Cited on page 90.)

[Zhang 1998] Guoqiang Zhang, B. Eddy Patuwo and Michael Y. Hu. *Forecasting with artificial neural networks:: The state of the art*. International Journal of Forecasting, vol. 14, no. 1, pages 35 – 62, 1998. (Cited on page 76.)

[Zuse 1969] K. Zuse. Rechnender Raum. Friedrich Vieweg & Sohn, 1969. (Cited on pages 51 and 68.)

[Zuse 1970] K. Zuse. Calculating space. MIT technical translation. Massachusetts Institute of Technology, Project MAC, 1970. (Cited on page 51.)