Université de Montréal

Optimisation d'une règle d'apprentissage pour réseaux de neurones artificiels

par

Samy Bengio

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Juin 1993

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée:

Optimisation d'une règle d'apprentissage

pour réseaux de neurones artificiels

présentée par:

Samy Bengio

a été évaluée par un jury composé des personnes suivantes:

Jean-Yves Potvin	président-rapporteur
Jan Gecsei	directeur de recherche
Jocelyn Cloutier	codirecteur
Geňa Hahn	membre du jury
Patrice Simard	examinateur externe

Thèse acceptée le: **5 octobre 1993**

Sommaire

Les réseaux de neurones artificiels sont des modèles de calcul parallèles et distribués basés, par analogie avec le fonctionnement du cerveau, sur une interconnexion forte et pondérée d'unités de traitement simples appelées *neurones*. Ces modèles ont la même puissance qu'une machine de Turing, et peuvent donc résoudre tout problème calculable, à condition cependant de trouver une architecture convenable et des poids satisfaisants.

La recherche dans l'espace des poids d'un réseau de neurones artificiels s'effectue à l'aide d'une *règle d'apprentissage*. Plusieurs règles ont été développées récemment mais on éprouve encore beaucoup de problèmes quant au choix des divers paramètres qui régissent le fonctionnement de ces règles. De plus, les règles actuelles sont souvent impuissantes devant la résolution de tâches difficiles.

Plutôt que de chercher à développer une règle d'apprentissage universelle, nous proposons dans cette thèse une nouvelle méthode permettant la recherche de nouvelles règles d'apprentissage paramétriques spécialisées dans la résolution d'une classe restreinte de tâches. Cette recherche s'effectue à l'aide de méthodes d'optimisation traditionnelles telles que la descente du gradient, les algorithmes génétiques et le recuit simulé.

La théorie sur la capacité des systèmes d'apprentissage, telle qu'étendue dans cette thèse au cas des règles d'apprentissage paramétriques, permet de nous guider dans le design de telles règles, notamment par l'utilisation de connaissances sur le domaine des tâches à résoudre pour contraindre la capacité de la règle paramétrique. Elle fournit en effet une borne maximale sur l'erreur de généralisation obtenue par une règle d'apprentissage sur des tâches qui n'étaient pas utilisées pour l'optimisation des paramètres de la règle. Cette borne est fonction du nombre de tâches utilisées pendant la phase d'optimisation et de la capacité de la règle d'apprentissage.

Les expériences réalisées et présentées ici montrent tout d'abord qu'il est effectivement possible de trouver par optimisation une règle d'apprentissage capable de résoudre des tâches complexes. Ces expériences confirment de plus les résultats théoriques sur la capacité des règles d'apprentissage paramétriques.

Une comparaison entre une règle trouvée par optimisation et la règle la plus populaire dans la littérature montre qu'il existe, à l'intérieur d'un cadre défini, de meilleures règles d'apprentissage spécialisées.

Notons finalement qu'une des contributions importantes de ce travail consiste à fournir une méthode permettant l'exploration systématique de l'espace des règles d'apprentissage pour réseaux de neurones artificiels.

Mots-clés:

- modèles connexionnistes,
- réseaux de neurones artificiels,
- apprentissage,
- optimisation,
- généralisation.

Table des matières

Liste des tableaux

Liste des figures

Remerciements

Plusieurs personnes m'ont assisté, d'une façon ou d'une autre, lors de la réalisation de cette thèse. J'aimerais ici tous les en remercier profondément.

Tout d'abord, bien sûr, je voudrais remercier mes deux directeurs de recherche Jan Gecsei et Jocelyn Cloutier. Ils ont été parmi les seuls à accepter, en 1989 et 1990 respectivement de diriger un étudiant en réseaux de neurones au département, à l'époque où ce sujet n'était pas très en vogue à l'Université de Montréal. Ils m'ont par la suite continuellement soutenu et m'ont permis d'assister à plusieurs conférences internationales de très haut calibre.

J'aimerais aussi remercier mon frère Yoshua Bengio. C'est lui qui le premier m'a initié au fantastique domaine des réseaux de neurones. Il a été très patient avec moi dans mes débuts, m'a fourni une volumineuse documentation qui m'a constamment été utile, et par-dessus tout, m'a assisté tout au long de ma recherche, critiquant le fond comme la forme.

Je veux ensuite remercier ici mes proches amis. D'abord Patrick Agin, qui a fait ses premiers pas en réseaux de neurones avec moi, et avec qui j'ai pu avoir des discussions tant mathématiques que philosophiques sur le sujet.

Nous sommes en quelque sorte *frères de neurones*; Michel Langevin, avec qui j'ai partagé le quotidien universitaire et qui m'a aidé à formaliser certains problèmes auxquels j'avais affaire; Lyne Larouche, qui a dû m'endurer quotidiennement; et enfin Jean-Claude Girard qui, de loin, a toujours formé une des pierres angulaires de notre petit groupe d'amis.

Enfin, je voudrais remercier particulièrement les deux principales équipes avec lesquelles j'ai travaillé au département. Tout d'abord, le laboratoire de VLSI qui m'a aimablement fourni l'accès à des ordinateurs puissants ainsi qu'à un bureau, et dont l'équipe d'étudiants, de professionnels et de chercheurs a toujours été de très agréable compagnie. La deuxième équipe, le laboratoire Héron-Multimédia, m'a fourni un emploi à temps partiel qui m'a permis de survivre pendant ces dernières années. J'ai développé avec la plupart des membres de ces deux équipes des liens très solides et je les en remercie ici.

À Célia et Carlo, naturellement...

mais peut-être aussi un peu à l'élève Chaprot, cancre

Chapitre 1

Introduction

De tout temps, l'homme s'est intéressé au mystère de l'*intelligence*. Certains, comme Freud et Piaget au début du siècle cherchaient, à l'aide de leurs théories sur le développement chez l'enfant, à comprendre le fonctionnement de l'intelligence. D'autres, comme Gödel et Turing s'intéressaient plutôt à la question computationnelle: "Comment doit-on construire une machine pour qu'elle ait un comportement intelligent?"

Ces deux voies commencent à s'unir dans les années 40, alors que McCulloch et Pitts ([44]) proposent le modèle du *neurone formel*. Ils montrent que de petites unités computationnelles simples (grossièrement analogues à des neurones biologiques), lorsqu'assemblées en réseau, sont capables de logique mathématique. Plus tard, d'autres vont montrer que ces réseaux sont en fait équivalents à des machines de Turing.

Cependant, pour réaliser une fonction quelconque à l'aide d'un réseau de neurones formels de McCulloch et Pitts, il faut déterminer la valeur des poids des connexions qui unissent les neurones en réseau. À cette époque, il est encore difficile d'imaginer comment un tel réseau de neurones pourrait *apprendre*, c'est-à-dire trouver la valeur idéale de ces poids.

C'est en 1949 que Donald Hebb propose une première *règle d'apprentissage* pour réseaux de neurones. Il suggère qu'un changement raisonnable serait d'augmenter la force d'une connexion lorsque les deux neurones reliés par cette connexion sont actifs simultanément. Depuis, plusieurs modèles de réseaux de neurones et de règles d'apprentissage ont été proposés, le modèle le plus connu étant le perceptron multi-couches utilisant la règle de la rétro-propagation de l'erreur.

Cependant les choix d'une architecture de réseau de neurones particulière et d'une règle d'apprentissage particulière pour la résolution d'une tâche donnée restent problématiques. Ils ne sont pas simples et l'on utilise généralement des heuristiques pour les effectuer.

1.1 Thème de la thèse

Depuis quelque temps, plusieurs chercheurs ont commencé à proposer des algorithmes permettant de choisir l'architecture optimale pour une tâche donnée [20, 41]. *Dans cette thèse, nous nous attaquons plutôt au choix de la règle d'apprentissage et de ses paramètres en fonction du type de tâche que l'on compte résoudre.* Plus précisément, nous proposons une nouvelle méthode permettant d'utiliser des techniques traditionnelles d'optimisation afin d'explorer, pour la première fois, l'espace des règles d'apprentissage, et d'y trouver la règle idéale pour la résolution d'un type de tâche déterminé.

1.2 Plan de la thèse

Le plan de la thèse se présente comme suit. Le chapitre 2 est une revue du domaine des réseaux de neurones artificiels. On y décrit les principales caractéristiques des divers modèles ainsi que quelques résultats théoriques connus reliés à l'apprentissage par réseaux de neurones artificiels. La revue se termine en montrant quelques limites des modèles actuels.

Le chapitre 3 décrit en détails l'idée principale suggérée dans cette thèse, à savoir *l'optimisation d'une règle d'apprentissage*, ainsi que la méthode pour y parvenir. Les travaux connexes reliés à cette idée y sont aussi présentés.

Le chapitre 4 donne une base théorique permettant d'estimer la qualité des règles d'apprentissage paramétriques trouvées par optimisation. Il s'agit d'une extension de la théorie sur la capacité décrite notamment par Vapnik et Chervonenkis [60], appliquée cette fois-ci aux règles d'apprentissage paramétriques.

Plusieurs méthodes d'optimisation sont présentées au chapitre 5: la descente du gradient, le recuit simulé, les algorithmes génétiques ainsi que la programmation génétique. Pour chacune, nous discutons de son utilisation pour l'optimisation de règles d'apprentissage paramétriques.

Le chapitre 6 décrit trois séries d'expériences visant à démontrer la réalisabilité de la démarche proposée. La première série d'expériences a pour but de trouver une règle d'apprentissage pour un ensemble de tâches de conditionnement classique, la deuxième série permet de montrer qu'il est possible de trouver des règles d'apprentissage pour des tâches linéairement non séparables, et la troisième série d'expériences, plus vaste, permet d'une part de vérifier la théorie décrite au chapitre 4, et d'autre part de comparer les différentes méthodes d'optimisation entre elles. De plus, une comparaison entre les meilleures règles trouvées par optimisation et la règle de la rétro-propagation de l'erreur est présentée.

La conclusion rappelle l'intérêt de la démarche, souligne ses limites et propose quelques avenues de recherche futures basées sur les principaux résultats de cette thèse.

Chapitre 2

Les réseaux de neurones artificiels

Dans ce chapitre, nous introduisons un modèle de calcul basé sur une analogie avec le fonctionnement du cerveau: les *réseaux de neurones artificiels* (on dit aussi *modèles connexionnistes*). Au début, un des buts de cette approche était d'expliquer et de comprendre le fonctionnement du cerveau; on avait donc beaucoup à faire avec la neurobiologie et la psychologie. Aujourd'hui, ce noble objectif reste tout-à-fait actuel: nous en sommes toujours aux premiers balbutiements dans notre tentative d'élucider l'un des systèmes les plus complexes que nous connaissions. D'autres objectifs se sont cependant ajoutés. Comme nous le verrons, les réseaux de neurones artificiels peuvent aussi être utilisés pour la résolution de problèmes complexes d'intelligence artificielle tels que la reconnaissance de la parole ou les problèmes de contrôle.

L'approche connexionniste tente donc de s'inspirer du fonctionnement humain pour concevoir des modèles radicalement nouveaux du comportement et du traitement de l'information. Ces modèles se fondent tous sur l'interconnexion forte d'un nombre élevé d'unités de traitement simples (appelés *neurones*). Le comportement général de ce genre de réseau est déterminé par sa structure et par le poids des connexions (ou *synapses*). Chaque connexion a en effet un poids, ou *efficacité synaptique*, qui représente le degré d'influence d'une synapse sur un neurone. L'élégance, la souplesse et la puissance de tels modèles réside dans la capacité d'apprentissage de nouvelles connaissances par modification des poids des connexions à partir d'exemples. Il y a souvent d'ailleurs séparation entre la phase d'apprentissage du réseau et la phase opérationnelle. Dans la phase d'apprentissage, on présente au réseau un grand nombre d'exemples; c'est durant cette phase que le réseau modifie le poids de ses connexions, bref qu'il apprend. Différents algorithmes d'apprentissage ont été proposés et ils ont été le focus de la recherche dans ce domaine durant les trois dernières décennies. Ces algorithmes peuvent être *biologiquement plausibles*, c'est-à-dire qu'ils respectent les contraintes biologiques actuellement connues notamment celle de n'utiliser que l'information locale aux neurones et aux connexions du réseau. Mais il existe aussi des algorithmes d'apprentissage non biologiquement plausibles, qui utilisent des informations extérieures au réseau, et qui bien souvent, sont plus efficaces[1].

Nous présentons dans cette section un survol des modèles connexionnistes, de leur analogie grossière avec le cerveau aux différents aspects théoriques reliés à l'apprentissage, en passant par leurs principales applications et leurs plus grand défauts.

[1]c'est-à-dire qu'ils peuvent résoudre des tâches plus complexes.

2.1 Quelques données sur le cerveau

Les modèles connexionnistes sont donc nés d'une grossière analogie avec le cerveau. Quelques données générales sur le cerveau nous aideront à mieux comprendre cette analogie.

Le cerveau humain est composé d'environ 10^{10} à 10^{12} cellules nerveuses, appelées *neurones*. Chaque cellule est connectée à environ 10^3 autres cellules par le biais de structures spécialisées appelées *synapses*. C'est par l'intermédiaire de ces dernières qu'un neurone reçoit des signaux des autres neurones.

Quand un signal, ou influx nerveux, arrive à une synapse, cela provoque la libération d'un médiateur chimique qui modifie le potentiel électrique de la cellule nerveuse à laquelle se transmet ce signal. Si à la suite de cette modification le potentiel dépasse une certaine valeur de seuil, la cellule émet à son tour un signal qui se propage par son axone (voir figure 2.1) vers d'autres cellules; la synapse est alors dite *excitatrice*. Si au contraire la modification fait diminuer le potentiel, on parle alors de synapse *inhibitrice*.

Le neurone joue donc un rôle d'intégrateur de signaux nerveux, alors que la synapse joue celui de modulateur. Les chercheurs admettent de plus ([15]) que l'apprentissage chez l'animal impliquerait notamment une modification de l'efficacité synaptique. C'est donc sur ces grands principes que sont basés les modèles formels de réseaux de neurones artificiels.

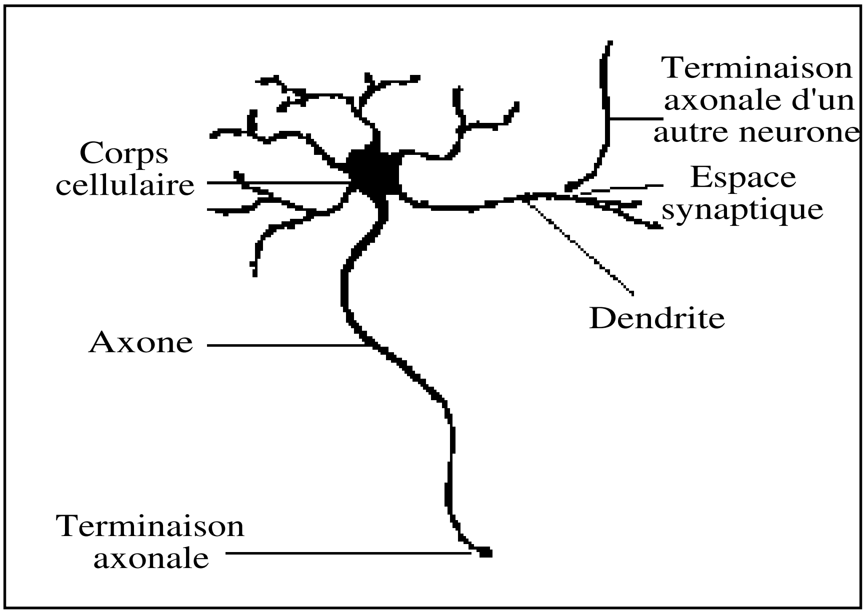

Figure 2.1: Image simplifiée d'un neurone biologique.

2.2 Description du modèle général

On retrouve dans [52] une description générale des modèles connexionnistes. Les différents *ingrédients* que l'on y retrouve sont les suivants:

- *Un environnement*, qui correspond à des vecteurs d'entrée $\mathbf{e}$ présentés séquentiellement au réseau. Ce sont ces vecteurs que le réseau traite, et qui influenceront éventuellement son apprentissage. Certains modèles utilisent aussi des vecteurs de sortie désirée $\mathbf{d}$, afin de guider les changements de poids dans la bonne direction. On associe alors un vecteur de sortie désirée $\mathbf{d}$ à chaque vecteur d'entrée $\mathbf{e}$.

- *Un ensemble d'unités de traitement*, capable de faire certaines opérations simples, avec peu de mémoire. Ces unités sont souvent appelées neurones pour leur similitude grossière avec les neurones du cerveau. Certains modèles (tels que le *perceptron* et ses successeurs [51]) classent les unités selon qu'elles soient des neurones d'entrée, cachés, ou de sortie. Un neurone d'entrée est une unité qui reçoit en entrée une valeur venant de l'environnement. On utilise pour ce faire une fonction de réindexage $m(i)$ qui assigne à chaque neurone d'entrée i un élément du vecteur d'entrée courant $\mathbf{e}$[2]. Un neurone de sortie est une unité qui fournit une valeur à l'environnement. Lorsqu'un vecteur de sortie désirée $\mathbf{d}$ est disponible, on utilise alors une fonction de réindexage $n(j)$ pour assigner à chaque neurone de sortie j un élément de $\mathbf{d}$. Enfin, un neurone caché est un neurone qui n'est ni un neurone d'entrée, ni un neurone de sortie. D'autres modèles (tels que la machine de Boltzmann [1]) fusionnent les neurones d'entrée et de sortie en une seule classe: les neurones visibles.

- *Un état d'activation*, qui est représenté par un vecteur $\mathbf{x}$ de valeurs $x(i)$ pour chaque neurone i. Cet état peut prendre des valeurs discrètes ou continues. Lorsque le neurone i est un neurone d'entrée, $x(i) = e(m(i))$. Dans tous les autres cas, $x(i)$ est calculé par la règle de propagation décrite plus bas.

- *Une fonction de sortie*, $f(\cdot)$ qui permet de calculer pour chaque neurone i une valeur de sortie $y(i)$ en fonction de l'état d'activation $x(i)$:

$$y(i) = f(x(i)) \tag{2.1}$$

  On retrouve plusieurs sortes de fonctions de sortie dans la littérature, mais le plus souvent, on utilise une des trois fonctions suivantes: (a) une fonction signe, (b) une fonction semi-linéaire[3], ou (c) une fonction sigmoïde (voir la figure 2.2).

[2]Ceci permet notamment d'avoir des vecteurs d'entrée de taille différente du nombre de neurones d'entrée.

[3]On dit aussi fonction linéaire par morceaux.

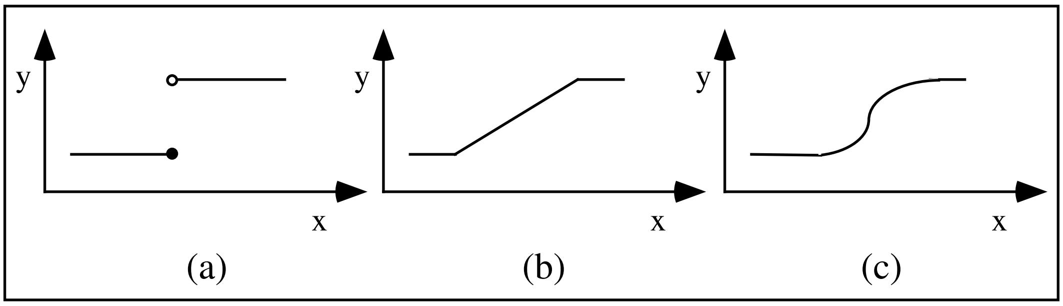

Figure 2.2: Diverses fonctions de sortie utilisées dans les modèles connexionnistes.

- *Un patron de connectivité* entre les unités de traitement, spécifiant les interconnexions du réseau. Certains réseaux ont un patron de connectivité complet (chaque unité est reliée à toutes les autres unités), d'autres ont leurs unités reliées sous forme de couches superposées (les unités d'une couche sont reliées à celles de la couche supérieure). À chaque lien entre deux unités i et j, on associe un poids $w(i,j)$ correspondant à la force de la connexion entre ces deux unités.

- *Une règle de propagation*, qui décrit comment calculer l'état d'activation d'une unité j en fonction des unités i pour lesquels il existe un poids $w(i,j)$. Appelons $source(j)$ l'ensemble de ces unités i. La règle la plus souvent utilisée consiste à calculer la somme des valeurs de sortie $y(i)$ des unités $i \in source(j)$, pondérées par les poids des connexions correspondantes.

$$x(j) = \sum_{i \in source(j)} w(i,j)\, y(i) \tag{2.2}$$

  Cette règle n'est pas utilisée pour calculer l'état d'activation des neurones d'entrée (dans leur cas, $x(j) = e(m(j))$).

- *Une règle d'apprentissage*, qui permet de modifier la force des connexions par l'expérience, c'est-à-dire par la présentation de vecteurs au réseau (voir section 2.3.3 pour quelques exemples).

Certains auteurs formalisent les modèles de réseaux de neurones artificiels par un graphe $G = (\mathcal{S}, \mathcal{A})$, où les sommets $s \in \mathcal{S}$ sont les neurones, et les arêtes $a \in \mathcal{A}$ sont des couples de sommets (s_i, s_j), qui représentent les connexions entre les neurones i et j.

2.3 Classification des modèles

Malgré certaines similitudes communes à tous les réseaux, on retrouve une grande variété de modèles de réseaux de neurones artificiels (voir [30, 31] pour une description plus détaillée). On peut cependant les classifier selon certains critères très généraux [64] tels que le type d'association qu'ils effectuent, le mode de supervision nécessaire pour l'apprentissage dans ces réseaux, et le mode d'apprentissage.

2.3.1 Association

Tous les modèles de réseaux de neurones artificiels sont des *mémoires associatives*, c'est-à-dire qu'ils peuvent effectuer une association entre des vecteurs d'entrée et des vecteurs de sortie. Certains modèles associent les vecteurs d'entrée à eux-mêmes. On dit alors que ce sont des modèles *auto-associateurs*. On les utilise généralement lorsque l'on veut compléter ou améliorer un vecteur d'entrée incomplet ou perturbé aléatoirement. Les modèles *hétéro-associateurs* quant à eux permettent d'effectuer une association entre deux ensembles de vecteurs. La figure 2.3 illustre ces deux types de modèles.

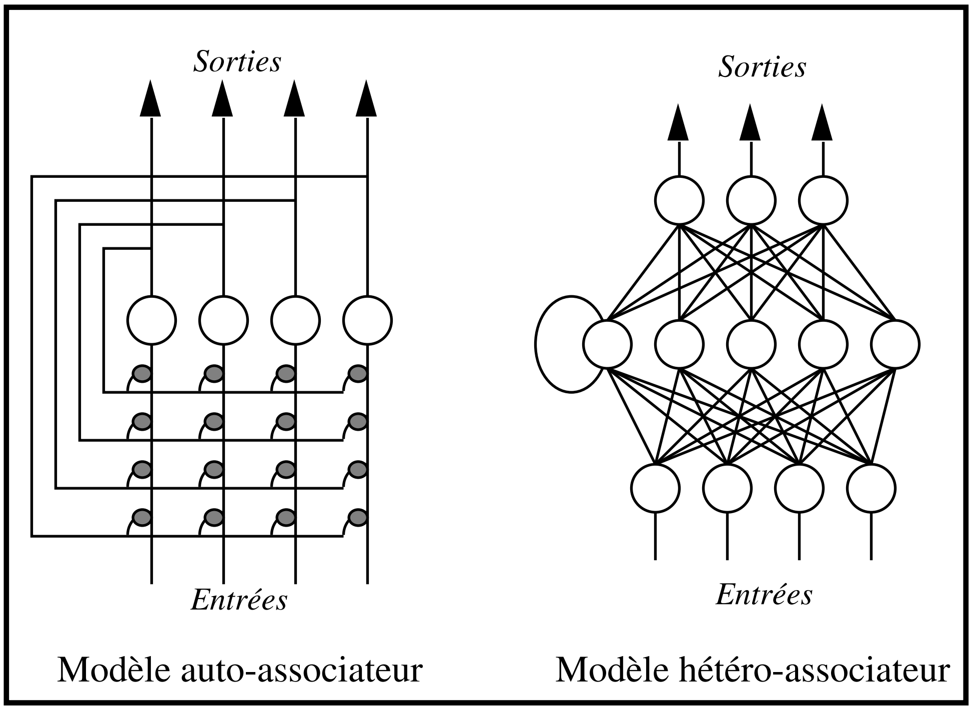

Figure 2.3: Exemples de modèles auto- et hétéro-associateurs.

2.3.2 Supervision

La supervision mesure la quantité d'information qu'un réseau de neurones reçoit dans le but de guider son apprentissage. La supervision peut être complète (on fournit la sortie attendue au réseau, ce sont des *modèles supervisés*), partielle (on note simplement la réussite ou l'échec du réseau, ce sont des *modèles de renforcement*), ou nulle (tout l'information nécessaire à l'apprentissage se trouve dans l'entrée, ce sont des *modèles non-supervisés*).

Les réseaux supervisés (comme le modèle de la rétropropagation de l'erreur [51]) ont besoin de recevoir pour chaque vecteur d'entrée présenté en phase d'apprentissage un vecteur de sortie désirée associé. Ces réseaux sont généralement utilisés pour construire une fonction entre un ensemble de vecteurs d'entrée et un ensemble de vecteurs de sortie.

Les réseaux non-supervisés (comme le modèle *Brain State in a Box* d'Anderson [2]) ne reçoivent aucune information quant à la sortie désirée. Ces réseaux, plus plausibles biologiquement que les modèles supervisés, sont cependant moins puissants: ils ne peuvent effectuer que des tâches simples de catégorisation (il n'est donc pas possible de résoudre des tâches de régression, telles que l'approximation de fonction).

Enfin, les modèles de renforcement [55] sont probablement les modèles les plus plausibles biologiquement et psychologiquement. En effet, on accepte en général le fait que l'être humain reçoive des signaux de renforcement lorsqu'il effectue certaines tâches (par exemple, lorsqu'un enfant se brûle les doigts en touchant à un élément chauffant, ou lorsqu'il fait rire ses parents en faisant une grimace, il reçoit un signal positif ou négatif relatif à son action). Ces signaux sont partiels et bruités (le message sera négatif ou positif, mais sur l'ensemble de l'action de l'enfant). Il est donc difficile d'attribuer le crédit à une partie du réseau, voire même à un neurone ou une connexion en particulier. Ce problème est noté dans la littérature comme celui de l'affectation du crédit (*credit assignment*) [55].

2.3.3 Les mécanismes d'apprentissage

La caractéristique la plus intéressante d'un réseau de neurones artificiels, c'est sa capacité d'apprendre, c'est-à-dire de modifier sa matrice de poids W en fonction de son environnement de telle sorte qu'après un certain temps, le réseau puisse prédire, compléter ou simuler son environnement.

La règle de Hebb

En 1949, Donald Hebb dans *The Organisation of Behavior* [29] suggère qu'un changement raisonnable et de plus biologiquement plausible serait d'augmenter la force d'une connexion lorsque les unités présynaptique (avant la connexion) et postsynaptique (après la connexion) sont actives simultanément (voir figure 2.4). La règle de Hebb peut ainsi être formalisée comme suit:

$$\Delta w(i, j) = \epsilon \, y(i) \, x(j) \tag{2.3}$$

c'est-à-dire que le changement de poids $\Delta w(i, j)$ de la connexion entre le neurone i et le neurone j est proportionnel à la valeur de sortie du neurone présynaptique et à l'état d'activation du neurone postsynaptique. Ici, ϵ est une constante modulant le changement de poids (cette constante, présente dans la plupart des modèles, est souvent appelée *taux d'apprentissage*). L'essence des idées de Hebb se retrouve encore aujourd'hui dans la plupart des modèles d'apprentissage.

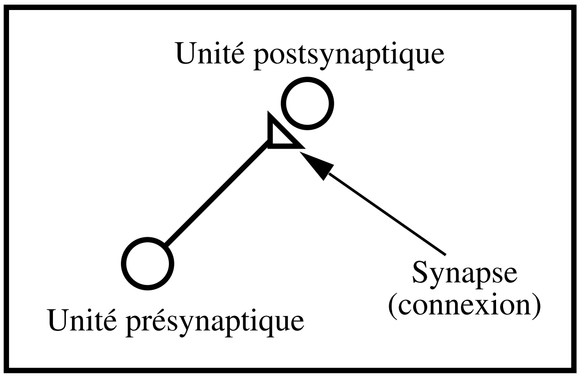

Figure 2.4: Illustration simplifiée d'une synapse.

Le perceptron de base

En 1962, Frank Rosenblatt [50] propose le *perceptron*. C'est un modèle en couches, comportant une couche de neurones d'entrées, et une couche de neurones de sortie. Les connexions sont faites entre la couche d'entrée et la couche de sortie. L'apprentissage dans les réseaux à une seule couche de poids peut se faire par une règle de modification de poids dont la version la plus connue est *la règle delta*.

Cette règle, qui est dérivée de la règle de Hebb, utilise une supervision externe pour guider le réseau vers l'apprentissage désiré. La modification de poids, itérative, obéit à la loi suivante:

$$\Delta w(i,j) = \epsilon \ (d(j) - y(j)) \ y(i) \tag{2.4}$$

C'est-à-dire que le changement de poids $\Delta w(i,j)$ de la connexion entre l'unité i et l'unité j est proportionnel à la différence entre la sortie désirée $d(j)$ et la sortie obtenue $y(j)$ à l'unité j, ainsi qu'à la sortie $y(i)$ de l'unité i.

Dans son livre *Principles of Neurodynamics*, Rosenblatt démontre à l'aide du *théorème de convergence du perceptron*, que si l'on utilise un perceptron pour résoudre un problème donné et qu'il existe effectivement une solution c'est-à-dire une matrice de poids W pour le perceptron qui résolve le problème, alors l'apprentissage par la règle du perceptron[4] convergera nécessairement vers cette solution. Cependant, Minsky [45] montrera plus tard que les problèmes *linéairement non séparables* (voir figure 2.5), notamment le

[4]La règle du perceptron de Rosenblatt est légèrement différente de la règle delta, mais cette différence ne change pas les résultats décrits ici.

ou-exclusif ne possèdent pas de solution (c'est-à-dire une matrice de poids) pour un perceptron de Rosenblatt. Il faudrait des neurones cachés pour emmagasiner de l'information supplémentaire, mais la règle delta ne permet pas de connaître les modifications à apporter aux connexions aboutissant à ces neurones. Plusieurs méthodes ont par la suite été développées pour résoudre ce problème (voir par exemple le modèle décrit dans la section 2.4).

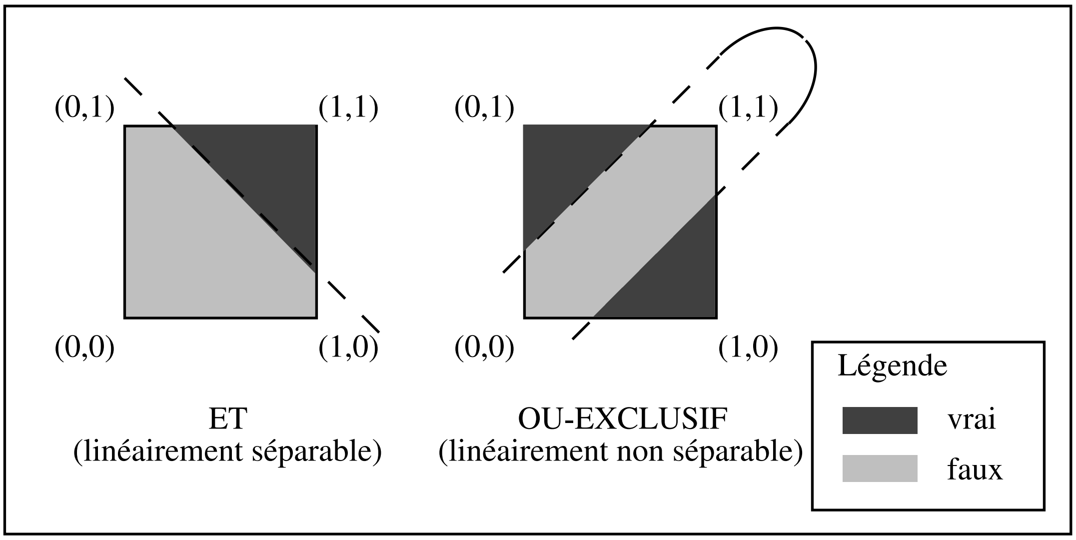

Figure 2.5: Illustration de la notion de séparabilité linéaire. Un problème de classification est dit *linéairement séparable* s'il est possible de séparer l'espace des solutions avec un hyperplan de telle sorte qu'aucune classe de solution ne soit de part et d'autre de l'hyperplan.

Autres modèles

Le modèle du perceptron, ainsi que ses dérivés (comme celui de la section 2.4) sont des modèles supervisés, et hétéro-associateurs. On retrouve dans la littérature plusieurs autres modèles (non-supervisés, auto-associateurs), basés sur divers mécanismes d'apprentissage plus ou moins biologiquement plausibles. On pourra trouver une bonne introduction à ces modèles dans [30].

2.4 Le perceptron multi-couches et la règle de la rétropropagation de l'erreur

En 1986, Rumelhart et ses collègues ([51]) publient une généralisation de la règle delta qui résout le problème soulevé vingt ans plus tôt par Minsky. Cette généralisation, *la règle de la rétropropagation de l'erreur*, ou règle delta généralisée, consiste à propager l'erreur obtenue à une unité de sortie d'un réseau à couches comportant une ou plusieurs couches cachées, à travers le réseau par descente du gradient dans le sens inverse de la propagation des activations, d'où son nom de rétropropagation de l'erreur. La figure 2.6 montre une illustration du principe.

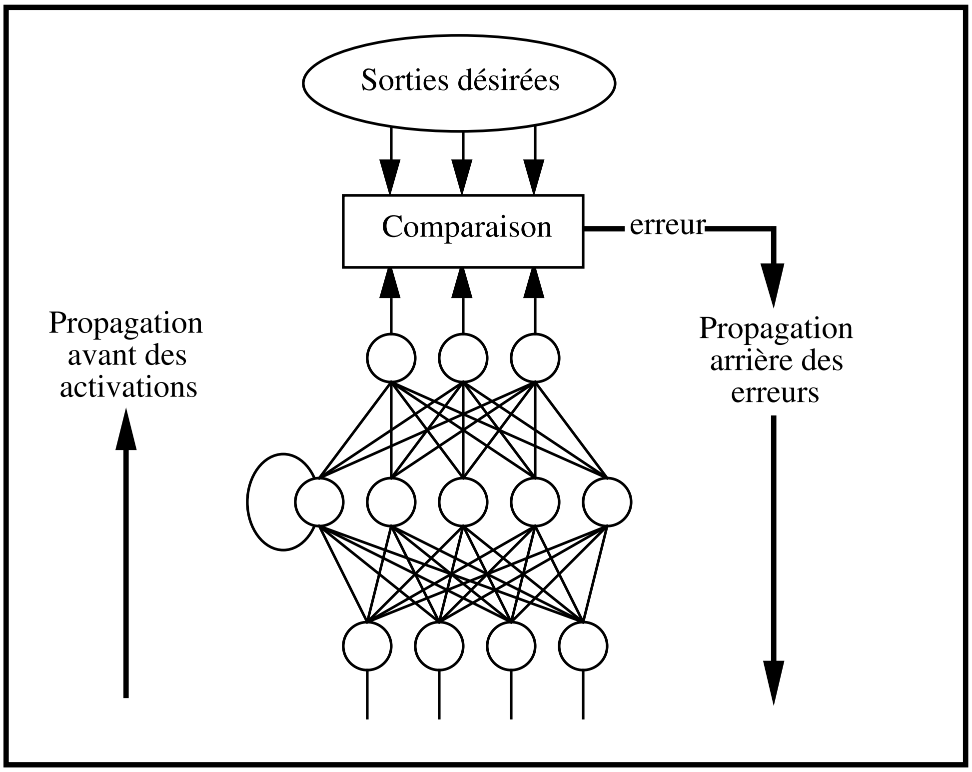

Figure 2.6: Schéma du modèle de la rétropropagation de l'erreur.

Un réseau à couches est composé d'un ensemble de neurones, pouvant être groupés en sous-ensembles distincts (les couches) de telle sorte qu'il n'y ait aucune connexion entre deux neurones d'une même couche. En fait, une amélioration ultérieure de la règle de rétropropagation permet l'utilisation de couches réentrantes et de cycles dans le système.

Comme pour la plupart des modèles, on distingue deux phases dans l'utilisation du modèle de rétropropagation: la phase d'apprentissage et la phase de test. En mode test, lorsque le réseau a appris à modéliser son environnement, le comportement souhaité du réseau est le suivant: on présente un stimulus (ou vecteur d'entrée) au réseau, celui-ci propage vers la sortie les valeurs d'activation correspondantes (en utilisant une règle de propagation), afin de générer, par l'intermédiaire des neurones de sortie, un stimulus (ou vecteur de sortie). Ce stimulus devrait correspondre à la sortie désirée, telle qu'apprise lors de la phase d'apprentissage.

Rumelhart et ses collègues décrivent dans [51] une généralisation de la règle delta dérivée par la méthode de la descente du gradient, afin de calculer la modification des poids des connexions des couches cachées (pour plus de détails, voir l'annexe A; voir aussi [38] pour une dérivation élégante par la méthode des multiplicateurs de Lagrange).

Afin de pouvoir effectuer une descente du gradient sur l'erreur par rapport aux poids du réseau, la fonction de sortie d'un neurone doit être différentiable et non linéaire (sinon, on pourrait réduire le réseau à deux couches seulement). La fonction la plus souvent utilisée est la suivante (c'est la sigmoïde de la figure 2.2):

$$y(j) = f(x(j)) = \frac{1}{1 + e^{-x(j)}} \tag{2.5}$$

La règle delta généralisée dicte alors le changement de poids entre le neurone

i et le neurone j de la façon suivante:

$$\Delta w(i,j) = \epsilon\, \delta(j)\, y(i) \tag{2.6}$$

c'est-à-dire de façon proportionnelle à une mesure d'erreur obtenue à l'unité postsynaptique ($\delta(j)$) et à la valeur obtenue à l'unité présynaptique ($y(i)$). Pour les connexions aboutissant aux neurones de sortie, cette mesure d'erreur est calculée ainsi:

$$\delta(j) = (d(j) - y(j))\ f'(x(j)) \tag{2.7}$$

où $f'(x(j))$ est la dérivée de la fonction d'activation non linéaire $f(x(j))$. Le calcul de l'erreur aux unités cachées se fait ensuite récursivement par descente du gradient. Soit $dest(j)$ l'ensemble des neurones auxquels j se connecte,

$$\delta(j) = f'(x(j)) \sum_{k \in dest(j)} \delta(k)\, w(j,k) \tag{2.8}$$

Lorsque l'on applique la règle delta généralisée sur le réseau de façon itérative pour un ensemble de vecteurs d'entrées (correspondant à l'environnement), le réseau tentera de minimiser l'erreur obtenue à la sortie, et donc de modéliser le mieux possible la fonction désirée entre les entrées et les sorties.

2.4.1 Le problème des minima locaux

Comme tous les algorithmes d'optimisation basés sur la descente du gradient, l'algorithme de la rétropropagation est sujet aux minima locaux. En effet, la solution finale trouvée par descente du gradient sera fortement reliée au choix des poids initiaux du réseau. Si les poids sont choisis près d'un minimum local sous-optimal, l'algorithme ne pourra pas trouver la solution désirée. Afin de contourner ce problème, on utilise plusieurs techniques:

- Relancer l'apprentissage plusieurs fois en utilisant des poids initiaux différents, ce qui entraîne un temps de calcul plus élevé.
- Introduire du bruit dans la recherche pour pouvoir sortir des minima locaux. Par exemple, le simple fait d'effectuer le changement de poids après la présentation de chaque vecteur d'entrée plutôt qu'après la présentation de l'ensemble des vecteurs d'entrée[5] introduit du bruit et accélère souvent le processus d'apprentissage [14, 61].

2.5 Résultats théoriques

Plusieurs résultats théoriques reliés aux modèles connexionnistes et à l'apprentissage par l'exemple ont déjà été obtenus, particulièrement en ce qui a trait au pouvoir d'expression des réseaux de neurones artificiels, leur complexité, ainsi que leur capacité de généralisation. Nous décrivons dans cette section quelques-uns de ces résultats.

[5] tel que le stipule l'algorithme original de rétropropagation de l'erreur.

2.5.1 Pouvoir d'expression

Le *pouvoir d'expression* d'un réseau de neurones artificiels est une mesure du nombre de fonctions différentes que celui-ci peut approximer (indépendamment de la méthode d'apprentissage). Plusieurs résultats (dont ceux de [17, 21, 33]) montrent par exemple qu'un réseau de neurones artificiels à couches sans récurrence[6] peut approximer avec une précision arbitraire n'importe quelle transformation continue d'un espace à dimension finie vers un autre espace à dimension finie, s'il possède suffisamment de neurones cachés. En ce sens, on dit d'eux qu'ils sont des *approximateurs universels*. Certains résultats [33] montrent même qu'à l'exception de cas extrêmes, une seule couche cachée est suffisante.

Il faut cependant noter que ces résultats ne fournissent aucun indice sur la méthode à utiliser pour trouver les poids correspondant à l'approximation d'une fonction donnée.

2.5.2 Complexité

Les réseaux de neurones artificiels ayant un si grand pouvoir d'expression, il devient intéressant de connaître les aspects de complexité reliés à ce modèle. Par exemple, plusieurs personnes se sont intéressé à la complexité du problème d'apprentissage par réseaux de neurones artificiels (voir [46] pour une revue). Ainsi, Judd montre dans [35] le résultat important suivant:

> Étant donné un réseau de neurones artificiels arbitraire R et une tâche arbitraire T devant être résolue par R, le pro-

[6] sans connexion d'une couche postérieure vers une couche antérieure.

> blème consistant à décider si dans l'espace de tous les paramètres de R (ses poids, sa structure) il existe une solution qui résout adéquatement T, est équivalent au problème de satisfaisabilité, et donc *NP-complet* (voir [23] pour une introduction à la NP-complétude). Ainsi, la recherche d'une telle solution (c'est-à-dire l'apprentissage) est *NP-ardue*.

Malgré cela, Baum soulève dans [3] la possibilité de trouver une solution (un ensemble de poids) pour T en temps polynomial si on peut utiliser des algorithmes d'apprentissage *constructifs*[7]. Il existe un certain nombre de ces algorithmes (voir notamment [20]) mais aucune preuve de convergence en temps polynomial n'existe actuellement pour ces algorithmes.

D'un point de vue pratique, certaines expériences empiriques (dont [31]) montrent qu'on peut faire apprendre une tâche complexe à un réseau de neurones artificiels en utilisant l'algorithme de la rétropropagation de l'erreur en temps $O\left(W^3\right)$ où W représente le nombre de poids du réseau. En effet, bien qu'il faille un temps exponentiel (sur le nombre de poids) pour obtenir la solution optimale, on peut souvent en pratique se contenter d'une solution sous-optimale satisfaisante obtenue en temps polynomial.

2.5.3 Apprentissage et généralisation

Dans cette section, nous décrivons formellement le problème d'apprentissage supervisé, afin d'introduire la notion de *généralisation* d'un système d'apprentissage.

[7] ayant la possibilité d'ajouter des neurones et des connexions durant l'apprentissage.

Apprentissage par l'exemple

Soit X une variable aléatoire dont la distribution de probabilité P_X est fixe mais inconnue, et supposons qu'il existe une fonction $\phi(x) : X \rightarrow Y$. Par exemple, pour le problème de la reconnaissance de caractères, $x \in X$ peut être une matrice de pixels et $y \in Y$ le caractère alphanumérique associé.

Le but d'un système d'apprentissage est alors de produire une fonction paramétrique $\hat{\phi}(x;\theta) : X \rightarrow Y$ qui minimise

$$\int_{x \in X} J(\hat{\phi}(x;\theta), \phi(x))\; P_X(x)\; dx \tag{2.9}$$

en modifiant adéquatement ses paramètres θ[8]. Ici, J est une fonction de coût scalaire basée sur la différence entre $\hat{\phi}$ et ϕ (telle que le carré de leur différence par exemple). Pour y parvenir avec un système d'apprentissage par l'exemple (ou apprentissage supervisé) on dispose généralement d'un ensemble de N couples $(x_i, \phi(x_i))$, pour $i = 1 \rightarrow N$, chacun pigés dans (X, Y) de façon indépendante et suivant P_X.

La performance d'un tel système se mesure alors en fonction de la différence entre ϕ et $\hat{\phi}$ aux N points choisis:

$$\sum_{i=1}^{N} J(\hat{\phi}(x_i;\theta), \phi(x_i)) \tag{2.10}$$

[8]Dans le cas d'un réseau de neurones artificiels, θ est l'ensemble des poids du réseau.

Généralisation

L'importante notion de *généralisation* peut alors être vue ainsi: après avoir utilisé un système d'apprentissage pour produire $\hat{\phi}$, une approximation de ϕ, en utilisant N exemples de ϕ pigés selon P_X, de quelle manière ϕ et $\hat{\phi}$ diffèrent-elles? Comment $\hat{\phi}$ se comporte-t-elle sur des exemples différents des N connus? On attend bien sûr d'un système qui généralise bien une bonne approximation de ϕ pour tout son domaine, et non seulement les N connus. Pour répondre à cette question, nous introduisons un formalisme important en théorie de l'apprentissage: la notion de *capacité*.

Soit $z = (x, y) \in (X, Y)$ et soit $G(z; \theta)$ l'ensemble des fonctions paramétriques de la forme $g(z; \theta)$. Par exemple dans le cas des réseaux de neurones artificiels, $g(\cdot)$ représente une architecture de réseau et une fonction de coût, θ représente l'ensemble des poids du réseau, et z les vecteurs d'entrée et de sortie désirée fournis au réseau. Ainsi,

$$g(z; \theta) = J(\hat{\phi}(x; \theta), \phi(x)) \tag{2.11}$$

Lorsque $J \in \{0, 1\}$ Vapnik définit dans [59] la *capacité de* $G(z; \theta)$ comme étant le nombre maximum de points $x_1, x_2, \ldots, x_h$ pouvant être séparés à tout coup en deux classes distinctes à l'aide d'une fonction dans $G(z; \theta)$. Par la suite, Vapnik décrit aussi une extension de sa définition pour le cas où $J \in \mathbb{R}$. Dans ce cas, la capacité est définie comme le nombre maximum d'associations $z_1, z_2, \ldots, z_h$ pouvant être apprises à l'aide d'une fonction dans $G(z; \theta)$ avec un coût J inférieur à un seuil choisi pour maximiser la capacité.

Par exemple, la capacité d'un ensemble de fonctions linéaires de la forme

$$\hat{\phi}(x,\theta) = \sum_{i=1}^{n} \theta_i \ \psi_i(x) \tag{2.12}$$

est égale à $n+1$ (le nombre de paramètres plus 1).

Si l'on considère un réseau de neurones artificiels comme un système permettant de choisir une fonction parmi un ensemble déterminé par la structure du réseau, alors, la capacité du système représente le nombre maximum d'exemples que le réseau peut apprendre correctement. Plus un système peut approximer de fonctions différentes, plus sa capacité est élevée. En général, plus le nombre de poids d'un réseau de neurones artificiels est élevé, plus sa capacité augmente. On appelle souvent la capacité d'un système d'apprentissage sa *Dimension Vapnik-Chervonenkis* (ou VCdim), du nom de ses auteurs.

Des résultats importants de [4, 13, 59] relient la généralisation d'un système d'apprentissage à sa capacité de la façon suivante:

> Soit un système d'apprentissage arbitraire ayant une VC-dim égale à h, et ayant correctement appris un ensemble de N exemples d'une fonction arbitraire ϕ ayant une distribution de probabilité fixe mais inconnue, alors on peut obtenir une borne supérieure ϵ sur la différence espérée entre l'erreur d'apprentissage et l'erreur de généralisation[9] en pire cas sur des exemples futurs définie ainsi [59]

[9] L'erreur d'apprentissage mesure la différence entre la valeur de les fonctions ϕ et $\hat{\phi}$ (l'approximation de ϕ) aux N points choisis, alors que l'erreur de généralisation (qu'on appelle aussi l'erreur de test) mesure plutôt la différence espérée sur des points différents des N points choisis pour l'apprentissage.

$$\epsilon \leq \mathcal{O}(\sqrt{\frac{h}{N}} ln \frac{N}{h}) \qquad (2.13)$$

On en déduit ainsi les résultats suivants: pour un nombre fixe d'exemples N, si on commence avec un système d'apprentissage ayant une VCdim h minimale et qu'on l'augmente progressivement[10], ϵ décroit jusqu'à ce qu'une valeur critique de h soit atteinte. Passé ce point, augmenter h aura pour effet d'augmenter ϵ. De plus, pour un h fixe, augmenter N améliorera la généralisation jusqu'à une valeur asymptotique qui dépend de h.

Ces résultats sont très importants car ils sont valides pour n'importe quel système d'apprentissage, n'importe quelle fonction ϕ, et n'importe quelle distribution de probabilité utilisée pour générer les exemples. Cependant, il est souvent compliqué de calculer la VCdim h d'un système d'apprentissage complexe tel qu'un réseau de neurones artificiels multi-couches à fonctions non linéaires.

2.6 Applications des réseaux de neurones artificiels

Bien qu'assez récents, les réseaux de neurones artificiels sont déjà utilisés dans plusieurs applications. Dans cette section, nous passons rapidement en revue les divers domaines ou les réseaux de neurones artificiels ont déjà été appliqués avec succès.

Les réseaux de neurones artificiels sont surtout utilisés pour tout ce qui a

[10] Par exemple, on peut augmenter le nombre de connexions d'un réseau de neurones artificiels.

trait à la reconnaissance de formes (notamment la reconnaissance de caractères manuscrits [40]), la reconnaissance de la parole ([12]), et le traitement de signal ([43, 54]). Dans ces cas-la, on utilise principalement l'algorithme de la rétropropagation de l'erreur[11] et les résultats obtenus pour ces problèmes réputés difficiles sont généralement meilleurs qu'avec toute autre technique traditionnelle.

Ils sont aussi utilisés pour des problèmes reliés à la robotique (par exemple, le système de [49] pour apprendre à conduire) ou dans le domaine des jeux. Ainsi, les programmes NeuroGammon, puis TDGammon de Tesauro [56, 57] sont actuellement les meilleurs programmes de backgammon au monde. TD-Gammon, par exemple, utilise un algorithme d'apprentissage ingénieux basé sur la rétropropagation de l'erreur dans le temps qui permet à un réseau d'apprendre en *jouant* contre lui-même. De plus, on les utilise dans des systèmes hybrides avec des méthodes d'intelligence artificielle *classiques* pour des domaines comme les systèmes experts ([58]) et le langage naturel ([42, 48]).

De manière générale, on sera bien avisé d'utiliser les réseaux de neurones artificiels pour résoudre tout problème pour lequel on possède un grand nombre d'exemples alors qu'il s'avère difficile d'en déduire des règles claires. L'esthétique et la simplicité de l'apprentissage par l'exemple deviennent alors des atouts importants.

[11] Parfois en conjonction avec d'autres méthodes telles que les modèles de Markov[12].

2.7 Problèmes de design reliés à l'apprentissage par réseaux de neurones artificiels

Un des problèmes majeurs des réseaux de neurones artificiels a trait au design. Devant une tâche à résoudre par réseaux de neurones artificiels, le chercheur doit prendre des décisions de design non évidentes et pourtant très importantes: ainsi par exemple, il faut généralement décider de l'architecture (nombre de neurones cachés par exemple, ou nombre de couches cachées, et interconnexion), souvent de façon *ad hoc* ou en utilisant quelques règles heuristiques simples. Le Cun décrit ce problème dans [39] en essayant diverses architectures pour un problème donné et en calculant l'erreur de généralisation pour chacune. En effet, hormis une recherche exhaustive, aucune méthode n'est connue pour déterminer l'architecture optimale pour un problème donné. Or toute la théorie sur les réseaux de neurones artificiels, leur puissance de calcul, ou leur faculté de généralisation ne tient que si l'on utilise l'architecture idéale (ou tout au moins suffisante et nécessaire).

Une solution à ce problème consiste à utiliser des algorithmes *constructifs* tels que [20] qui commencent avec une architecture minimale et ajoutent des neurones et des connexions au fur et à mesure de l'apprentissage. D'autres solutions telles que [41] utilisent plutôt une technique inverse: à partir d'une architecture complète, ils éliminent certains neurones et/ou connexions qui semblent non essentiels.

Depuis peu, on commence à utiliser des méthodes d'optimisation pour chercher l'architecture idéale. Ainsi, plusieurs travaux proposent l'utilisation des algorithmes génétiques ([26, 32]) pour optimiser l'architecture des réseaux de neurones (voir [6, 62] pour des exemples, ou [63] pour une revue).

Un autre problème tient au choix des paramètres des divers algorithmes d'apprentissage. En effet, chaque règle d'apprentissage utilise généralement un certain nombre de paramètres pour guider l'apprentissage. Ainsi, la règle de la rétropropagation de l'erreur est basée notamment sur le taux d'apprentissage. Ce taux varie d'une tâche à l'autre, et encore une fois, on utilise souvent des règles heuristiques simples pour déterminer sa valeur idéale. Dans la même veine que pour le choix des architectures, on utilise maintenant des méthodes comme les algorithmes génétiques pour choisir ces paramètres ([6, 34] et [63] pour une revue).

Dans cette thèse, nous nous proposons d'aller plus loin encore et de chercher, par des méthodes d'optimisation comme les algorithmes génétiques, à trouver de *nouvelles règles d'apprentissage*. Nous décrivons dans le prochain chapitre comment y parvenir.

Chapitre 3

Optimisation de la règle d'apprentissage

Nous avons vu au chapitre 2 que la caractéristique la plus remarquable des réseaux de neurones artificiels est leur capacité d'adaptation à un environnement variable, c'est-à-dire leur faculté *d'apprentissage*.

L'apprentissage dans un réseau de neurones artificiels se fait par l'intermédiaire d'une *règle d'apprentissage*. À l'instar des systèmes nerveux biologiques où l'on admet actuellement que l'apprentissage serait le résultat de la *plasticité synaptique* (modification de l'efficacité d'une synapse), les modèles de réseaux de neurones artificiels utilisent des règles d'apprentissage qui dictent une variation des poids des connexions en fonction de l'environnement.

Il existe une multitude de modèles de réseaux de neurones artificiels et tout autant de règles d'apprentissage (voir [30] pour une revue). Il faut cependant noter plusieurs problèmes reliés aux règles d'apprentissage actuelles.

Tout d'abord, et bien que les chercheurs s'entendent pour noter l'importance de la règle d'apprentissage dans les systèmes connexionnistes, la plupart des modèles utilisent des *heuristiques* pour leur design (notamment pour le choix des paramètres intrinsèques à ces règles). Nous avons d'ailleurs mentionné à la section 2.7 différents problèmes reliés au design des règles d'apprentissage et de leurs paramètres ainsi que les solutions actuellement utilisées.

De plus, on cherche en fait des règles d'apprentissage universelles, capable de résoudre n'importe quelle tâche, ce qui n'existe peut-être pas (ainsi, même dans les systèmes nerveux biologiques, on admet qu'il existe probablement non pas une mais plusieurs règles d'apprentissage, spécialisées pour des tâches cognitives différentes [22]). C'est ainsi que même les meilleures règles d'apprentissage existant actuellement et ayant souvent de solides bases mathématiques éprouvent des problèmes face à la résolution de tâches difficiles (nous avons vu par exemple à la section 2.4 que la règle de la rétropropagation de l'erreur est sensible aux minima locaux et peut être lente à converger pour certains problèmes de grande taille).

3.1 Fonction paramétrique

Nous proposons dans cette thèse une méthode permettant la recherche de nouvelles règles d'apprentissage spécialisées pour la résolution de classes de tâches bien définies. Pour ce faire, nous considérons la règle d'apprentissage comme une fonction paramétrique et à l'aide de diverses méthodes d'optimisation traditionnelles telles que la descente du gradient, le recuit simulé et les algorithmes génétiques[1], nous proposons d'optimiser les paramètres

[1] voir chapitre 5 pour une discussion sur ces méthodes.

de cette règle de sorte qu'elle puisse être utilisée pour résoudre une classe donnée de tâches.

Pour y parvenir, nous devons préalablement établir les hypothèses de travail suivantes:

- la même règle est utilisée par plusieurs neurones (cette contrainte peut être étendue à une règle pour chaque type de neurone ou de synapse[2]). Les modèles actuels de réseaux de neurones artificiels n'utilisent qu'une seule règle d'apprentissage pour toutes les connexions du réseau. Dans les modèles que nous proposons, on pourra envisager plusieurs mécanismes d'apprentissage différents.

- il existe une dépendance (éventuellement stochastique) entre la modification synaptique et certaines informations disponibles à chaque synapse[3]. En d'autres termes, le changement de poids n'est pas totalement le fruit du hasard et dépend de variables pouvant être mesurées localement. Cette hypothèse est nécessaire sans quoi on serait forcé d'admettre l'impossibilité de mécanismes d'apprentissage.

- cette dépendance peut être approximée avec une précision arbitraire par une fonction paramétrique, c'est-à-dire une fonction qui s'exprime sous la forme

$$\Delta w(i,j) = f(x_1, x_2, \ldots, x_n; \theta_1, \theta_2, \ldots, \theta_m) \tag{3.1}$$

[2]Comme on l'a déjà dit, on sait maintenant qu'il existe dans le cerveau différentes sortes de neurones et de synapses, et qu'il est donc probable qu'il y ait plusieurs mécanismes d'apprentissage, bien qu'on ne connaisse pas encore très bien leur fonctionnement [22].

[3]Sans obliger nécessairement que ce soit biologiquement plausible. Ainsi la règle de la rétropropagation de l'erreur, qui n'est pas biologiquement plausible, utilise des informations concernant l'erreur globale du réseau. Ces informations sont néanmoins rendus disponible à chaque synapse.

où $\Delta w(i, j)$ représente la modification de poids à apporter à la connection entre les neurones i et j, les x_k sont les n variables de la fonction et les θ_l sont les m paramètres.

Chalmers ([16]) à d'ailleurs proposé une idée similaire en 1990 sur l'évolution de l'apprentissage dans les réseaux de neurones. Nous présentons à la section 3.6 les résultats principaux de ses travaux, en les mettant en relief avec ce qui est proposé dans cette thèse.

3.2 Design de la règle d'apprentissage

Notre règle d'apprentissage est donc en réalité une fonction d'un certain nombre de variables, disponibles à une synapse, et d'un certain nombre de paramètres, tous fixes d'une synapse à l'autre. Ce sont justement ces paramètres que nous nous proposons d'optimiser pour trouver une règle d'apprentissage capable de résoudre une classe définie de tâches. Il faut cependant régler préalablement plusieurs problèmes de design dont nous discutons dans cette section.

3.2.1 Variables de la règle d'apprentissage

Les variables de la règle d'apprentissage (les x_k de l'équation (3.1)) sont des informations qui peuvent influencer la modification synaptique. On peut considérer par exemple les variables *locales* à chaque synapse (qui représentent des mécanismes physiquement proche de la synapse). La figure 3.1 montre quelques exemples de variables locales qui peuvent influencer l'efficacité synaptique et qui sont basées sur nos connaissances actuelles du fonctionnement du cerveau.

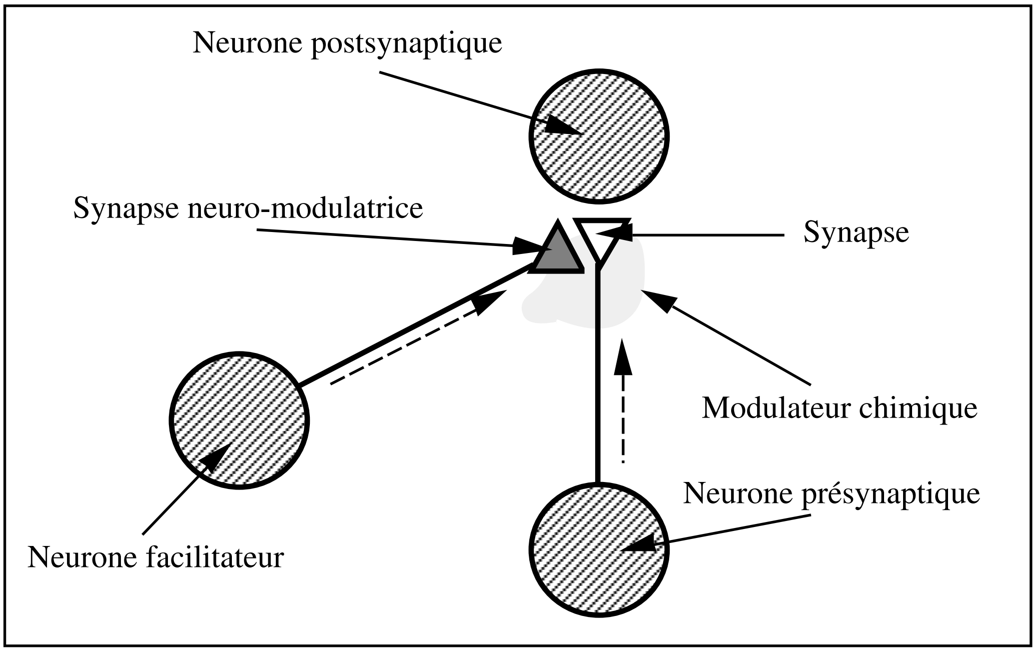

Figure 3.1: Liste non-exhaustive des éléments qui peuvent influencer la modification synaptique.

On remarque notamment les variables suivantes: la valeur de sortie du neurone situé avant la synapse (ou neurone présynaptique) et l'état d'activation du neurone situé après la synapse (ou neurone postsynaptique). Ces deux variables sont utilisées dans la plupart des modèles d'apprentissage hebbien. On sait de plus qu'il existe un certain nombre de modulateurs, tant chimiques qu'électriques, qui peuvent influencer l'efficacité d'une synapse. Ainsi on a pu observer certains neurones (qu'on appelle des neurones *facilitateurs* ou *modulateurs*) qui établissent des connexions directement avec des synapses (plutôt qu'avec d'autres neurones comme c'est le cas habituellement). Ces neurones influencent donc, non pas d'autres neurones, mais bien des synapses. Enfin, il est possible que la valeur actuelle d'une synapse influence sa propre plasticité synaptique.

On peut aussi envisager des variables *non locales* à chaque synapse. Ainsi, les algorithmes de renforcement utilisent notamment une fonction de l'erreur globale du réseau pour calculer le changement de poids à chaque connexion.

3.2.2 Paramètres et forme de la règle d'apprentissage

Après avoir choisi l'ensemble des variables qui peuvent influencer l'efficacité synaptique, on doit ensuite décider du nombre de paramètres et de la façon d'assembler les variables et les paramètres de la règle d'apprentissage paramétrique. On doit premièrement déterminer les opérateurs qui peuvent être utilisés (tels que la multiplication, l'addition, l'exponentiation, etc). Ensuite, et comme il existe un très grand nombre de formes de règles d'apprentissage utilisant les variables, paramètres et opérateurs choisis, il faut concevoir un certain nombre de contraintes pour limiter les choix. La section 3.3 discute de contraintes raisonnables envisagées dans cette thèse. Enfin, comme nous le verrons à la section 5.4, il est aussi possible de faire varier la forme de la règle d'apprentissage au même titre que les paramètres pendant le processus d'optimisation de la règle. Il n'est donc plus nécessaire de fixer la forme, il suffit simplement de déterminer les opérateurs, les variables, et le nombre de paramètres maximum que la règle peut contenir.

3.3 Contraintes de design

Afin de réduire l'espace de recherche des règles d'apprentissage (lors de l'optimisation des paramètres), nous pouvons déterminer un ensemble de contraintes sur la forme et les variables de la règle d'apprentissage paramétrique[4].

3.3.1 Plausibilité biologique

La première contrainte que nous avons choisi dans la plupart des expériences réalisées au chapitre 6 en est une de *plausibilité biologique*. Cela veut dire s'en tenir à des règles qui respectent les connaissances que nous avons actuellement sur les mécanismes synaptiques dans le cerveau. Plusieurs raisons nous ont poussé à choisir cette contrainte:

- Tout d'abord, le principe même d'établir des contraintes vise à réduire l'espace des solutions envisagées. Il faut cependant s'assurer que cette réduction n'écarte pas toutes les solutions intéressantes. La contrainte de plausibilité biologique nous permet justement de réaliser cet objectif. Comme on peut le voir à la figure 3.2, réduire l'espace de cette façon nous suggère tout de même qu'il existe probablement au moins une règle d'apprentissage dans l'espace de recherche, à savoir celle(s) utilisée(s) dans les systèmes biologiques. Notons cependant que plusieurs règles utilisées dans la littérature des réseaux de neurones artificiels (telles que celle de la rétropropagation de l'erreur) sont exclues de ce sous-espace.

- Le second critère en est un de *réalisation matérielle*. Afin que des règles découvertes par ce processus d'optimisation puissent être adéquatement utilisées dans des applications pratiques, et tenant compte de l'aspect fortement parallélisable des réseaux de neurones artificiels, il est judicieux de restreindre l'espace de recherche de façon à n'utiliser

[4]le chapitre 4 donne de plus une raison théorique de l'utilisation de contraintes.

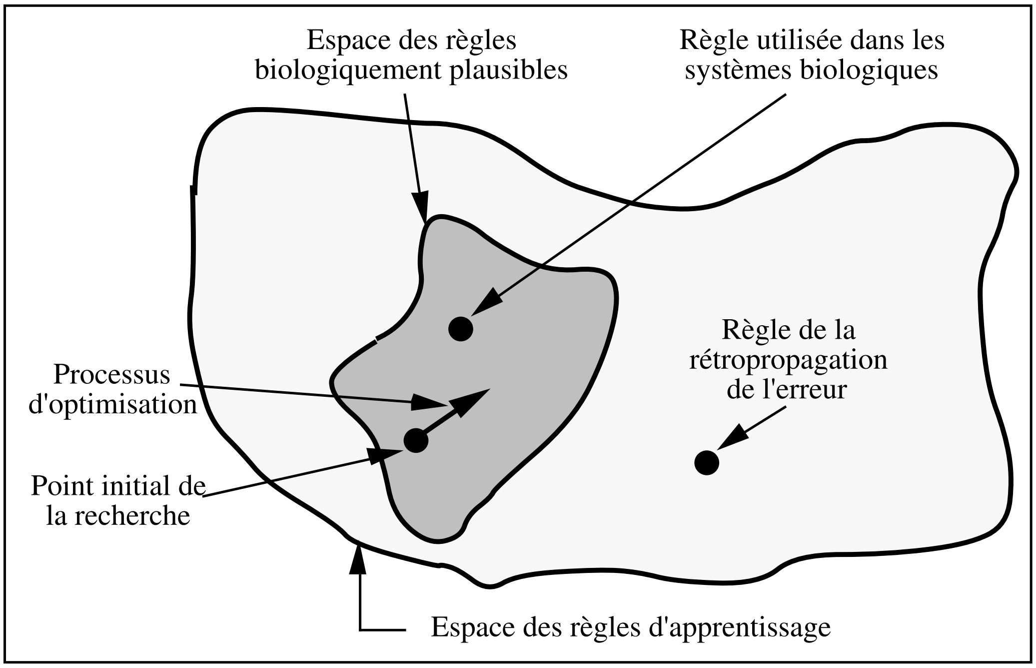

Figure 3.2: Espace des règles d'apprentissage.

que des variables locales à chaque connexion. Cette restriction permet en effet d'envisager facilement une réalisation physique du réseau de neurones pouvant apprendre. Or, cette contrainte de localité est justement atteinte lors du choix de la contrainte de plausibilité biologique.

- Enfin, la contrainte de plausibilité biologique peut nous guider dans le choix de la forme de la règle d'apprentissage. On peut en effet inclure les différents mécanismes de modification synaptique connus dans les systèmes biologiques comme des modules de la règle d'apprentissage. Le rôle des paramètres déterminera alors l'influence particulière de chaque module sur le changement de poids. La figure 3.3 montre comment la règle d'apprentissage paramétrique peut être construite à partir de modules décrivant des mécanismes connus (connaissance *a-priori*) et de modules libres.

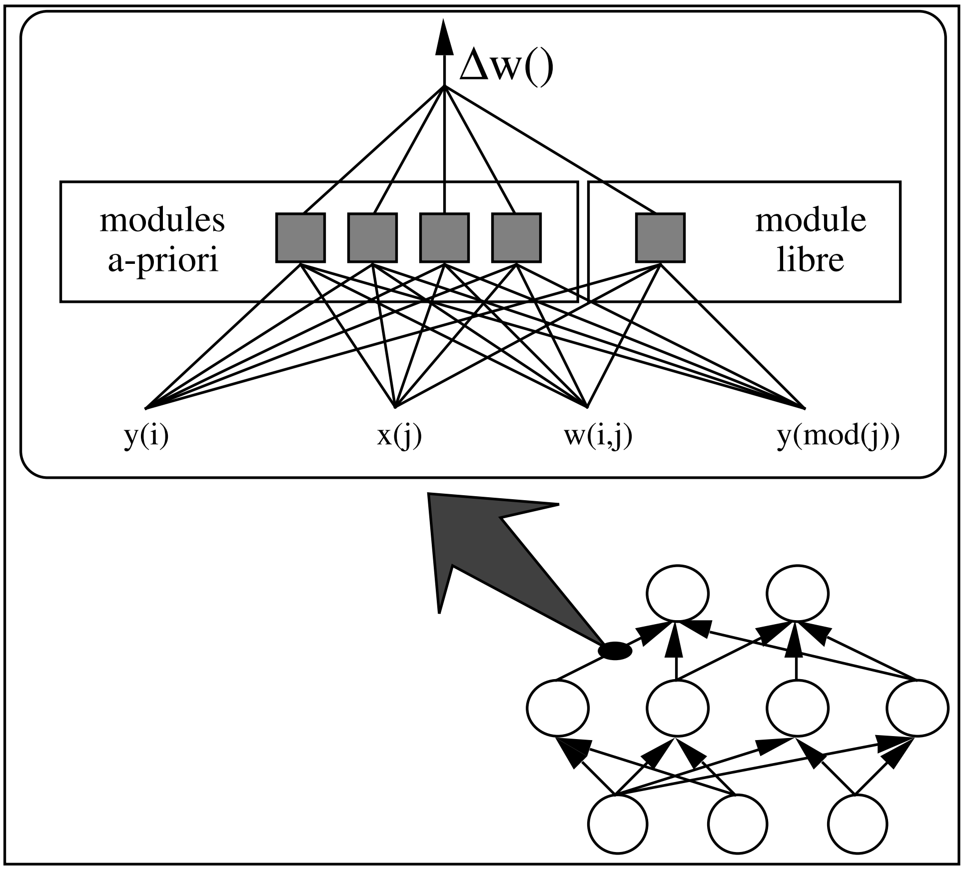

Figure 3.3: Utilisation de connaissances *a-priori* pour le design de règles d'apprentissage paramétriques. À partir de 4 variables (l'activité présynaptique $y(i)$, le potentiel postsynaptique $x(j)$, l'activité d'un neurone modulateur $y(mod(j))$ et le poids de la synapse $w(i,j)$) pouvant influencer la modification synaptique, on construit la forme de la règle en combinant des mécanismes connus et des modules libres. Par exemple, la règle de Hebb peut être représentée par un module calculant $y(i) \cdot x(j)$.

3.3.2 Contrainte de complexité

La seconde contrainte que nous nous imposons est nécessaire: limiter le temps de calcul pour l'apprentissage des tâches. En effet, il est important de se demander quelle est la complexité du problème général d'optimisation d'une règle d'apprentissage paramétrique. Comme on l'a vu à la section 2.5.2, le problème d'apprentissage est *NP-complet*. Il est donc évident que celui de l'optimisation d'une règle d'apprentissage l'est aussi (car une des étapes d'optimisation de la règle est justement l'apprentissage d'une ou plusieurs tâches).

Des résultats empiriques [31] montrent cependant que l'on peut apprendre des tâches complexes avec un réseau de neurones artificiels utilisant la règle de la rétropropagation de l'erreur en temps polynomial[5]. On obtient alors une solution souvent sous-optimale mais suffisante.

C'est donc en fonction de ces résultats pratiques que nous avons choisi de contraindre l'espace des règles d'apprentissage à un sous-espace de règles capables de résoudre une tâche dans un temps polynomial sur le nombre de connexions.

On ne cherche donc plus à trouver *la* meilleure règle d'apprentissage mais bien une règle d'apprentissage biologiquement plausible capable de résoudre une classe définie et restreinte de tâches en temps polynomial, quitte à admettre une erreur d'apprentissage raisonnable (c'est-à-dire se contenter d'une solution *quasi-optimale* satisfaisante).

3.4 Exemples de règles

Dans les expériences décrites au chapitre 6, nous utilisons principalement deux formes différentes de règles d'apprentissage paramétriques. Ces deux règles utilisent les mêmes variables locales. Étant donnée la connexion $i \to j$, il s'agit de l'activité présynaptique $y(i)$, le potentiel postsynaptique $x(j)$, l'activité d'un neurone modulateur $y(mod(j))$, ainsi que le poids de la connection courante $w(i,j)$.

[5] En fait dans l'ordre du nombre de connexions au cube.

La première règle, donnée à l'équation (3.2), est construite en utilisant des connaissances biologiques afin de restreindre le nombre de paramètres à 7:

$$\begin{aligned}
\Delta w(i,j) &= \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) \\
&\quad + \theta_4\, y(i)\, y(mod(j)) + \theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, w(i,j) \qquad (3.2)
\end{aligned}$$

On retrouve dans ces règles les principaux éléments suivants (chacun pondéré par un paramètre):

- $y(i) \cdot x(j)$: la règle de Hebb,
- $y(i) \cdot y(mod(j))$: cette règle est décrite par Hawkins dans son modèle du conditionnement chez l'Aplysia ([27]),
- $y(i) \cdot w(i,j)$: suggéré par Gluck ([25]) pour permettre l'oubli graduel.

La deuxième règle d'apprentissage, donnée à l'équation (3.3), est la somme pondérée (par les paramètres) des produits de toutes les combinaisons possibles des 4 variables locales. Cela donne 16 termes, donc 16 paramètres.

$$\begin{aligned}
\Delta w(i,j) &= \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) + \theta_4\, w(i,j) \\
&\quad + \theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, y(mod(j)) + \theta_7\, y(i)\, w(i,j) \\
&\quad + \theta_8\, x(j)\, y(mod(j)) + \theta_9\, x(i)\, w(i,j) \\
&\quad + \theta_{10}\, y(mod(j))\, w(i,j) + \theta_{11}\, y(i)\, x(j)\, y(mod(j)) \\
&\quad + \theta_{12}\, y(i)\, x(j)\, w(i,j) + \theta_{13}\, y(i)\, y(mod(j))\, w(i,j)
\end{aligned}$$

$$+\theta_{14}\, x(j)\, y(mod(j))\, w(i,j)$$
$$+\theta_{15}\, y(i)\, x(j)\, y(mod(j))\, w(i,j) \tag{3.3}$$

Cette règle, qui inclue notamment celle de l'équation (3.2), pourrait permettre de vérifier si des mécanismes synaptiques simples et nouveaux (différents de ceux proposés par Hebb, Hawkins ou Gluck par exemple) pourraient être envisagés.

3.5 Processus d'optimisation

Soit une classe $\mathcal{T}$ donnée de tâches. Afin de trouver les paramètres optimaux d'une règle d'apprentissage telle que celles suggérées à la section 3.4 capable de résoudre les tâches de $\mathcal{T}$, nous utilisons l'algorithme général suivant:

1. Choisir une forme de règle d'apprentissage paramétrique. Ce choix devrait être guidé par le type de tâche de la classe $\mathcal{T}$. Par exemple, si la tâche est un processus biologique, il serait utile de choisir une règle qui incorpore des mécanismes de modification synaptique biologiques (voir l'exemple des problèmes de conditionnement au chapitre 6). Si par contre la tâche est plus mathématique, on peut choisir d'incorporer dans la règle des éléments permettant d'effectuer des opérations d'optimisation connues comme la descente du gradient (voir les expériences sur la classification au chapitre 6).

2. Initialiser les paramètres de la règle d'apprentissage. Pour ce faire, on peut soit simplement leur affecter une valeur aléatoire à l'intérieur d'un intervalle donné, soit choisir des valeurs qui reflètent des connaissances

a-priori sur la classe $\mathcal{T}$.

3. Choisir, à l'intérieur de $\mathcal{T}$, un sous-ensemble $\mathcal{T}^*$ de tâches représentatives de $\mathcal{T}$. Le chapitre 4 montre l'importance d'un tel choix. Ainsi, si la complexité de $\mathcal{T}^*$ est moindre que celle de $\mathcal{T}$, la règle résultante du processus d'optimisation pourrait ne pas être capable de résoudre toutes les tâches de $\mathcal{T}$ (par ailleurs, si la taille de $\mathcal{T}^*$ est trop grande, le temps d'optimisation pourrait devenir prohibitif).

4. Le processus d'optimisation comme tel suit alors les étapes suivantes:

   (a) Pour chaque tâche de $\mathcal{T}^*$, choisir un réseau de neurones adéquat[6], un ensemble de vecteurs d'apprentissage ainsi qu'un ensemble de vecteurs de test.

   (b) Initialiser les poids des réseaux de façon aléatoire.

   (c) À l'aide de la règle d'apprentissage paramétrique (utilisant les paramètres actuels) et de l'ensemble des vecteurs d'apprentissage, essayer en un nombre fixe d'itérations[7] de résoudre la tâche associée à chaque réseau. Pendant cette phase, les paramètres de la règle (θ) sont fixes alors que les poids des réseaux (W) varient.

   (d) Calculer pour chaque réseau et à l'aide des vecteurs de test, l'erreur de généralisation comme étant une fonction de la différence entre ce que le réseau aurait dû répondre et ce qu'il a effectivement répondu. On peut utiliser par exemple la fonction des moindres carrés, définie ainsi

$$E_k = \frac{1}{2} \sum_{i \in test} \sum_{j \in sortie} (d_{i,j,k} - y_{i,j,k})^2 \tag{3.4}$$

[6]Ce choix n'est cependant pas simple, comme on l'a noté à la section 2.7.
[7]Ce nombre peut dépendre de la tâche.

où E_k est l'erreur de généralisation du réseau de neurone k (correspondant à la tâche k de T^*), $d_{i,j,k}$ est la valeur désirée pour le vecteur de test i au neurone de sortie j du réseau k, et $y_{i,j,k}$ est la valeur effectivement obtenue pour ce même neurone j du réseau k à la présentation du vecteur de test i.

(e) à l'aide de la méthode d'optimisation choisie (voir chapitre 5 pour une présentation de quelques méthodes), modifier les paramètres de la règle en fonction des E_k. On peut par exemple considérer le coût global E^*

$$E^* = \sum_{k \in \mathcal{T}^*} E_k \tag{3.5}$$

Ainsi, chaque réseau de neurones partage la même règle d'apprentissage pendant tout le processus d'optimisation.

(f) recommencer l'algorithme à partir de 4b jusqu'à ce que E^* soit minimale ou que le nombre d'itérations ait dépassé une borne raisonnable.

(g) vérifier la *qualité* de la règle obtenue en essayant de résoudre des tâches de $\mathcal{T}$ qui ne sont pas dans $\mathcal{T}^*$. On vérifie ainsi le pouvoir de *généralisation* de cette règle.

3.6 L'expérience de Chalmers

En 1990, indépendamment de nos travaux, Chalmers publie dans [16] un article sur l'évolution de l'apprentissage dans les réseaux de neurones. Il propose, tout comme nous, de modifier la procédure d'apprentissage des réseaux de neurones artificiels pour qu'elle s'adapte au type de tâches sur lesquelles elle devra être soumise. Nous résumons ici les principaux résultats auxquels est parvenu Chalmers.

1. Il utilise les algorithmes génétiques comme méthode d'optimisation.

2. Le type de tâches pour lequel il cherche une règle d'apprentissage est simple: il s'agit de fonctions booléennes linéairement séparables[8].

3. L'architecture du réseau de neurones qu'il utilise est simple: deux neurones d'entrée directement connectés à un neurone de sortie.

4. Les variables utilisées dans la règle d'apprentissage sont les suivantes, pour la connexion $i \rightarrow j$: l'activité présynaptique $y(i)$, l'activité postsynaptique $y(j)$, la valeur désirée pour le neurone postsynaptique $d(j)$ (n'oublions pas que le réseau n'ayant pas de neurones cachés, le seul neurone postsynaptique est aussi le neurone de sortie, pour lequel nous connaissons la valeur désirée), et le poids de la connexion $w(i,j)$.

   Ainsi, si l'on compare avec les variables proposées dans cette thèse, la seule différence est qu'il utilise directement la valeur désirée alors que nous proposons plutôt d'utiliser la valeur d'un neurone modulateur qui pourrait éventuellement contenir une mesure de l'erreur.

5. La forme de la règle d'apprentissage est une combinaison linéaire des quatre variables et de leur six produits deux-à-deux:

$$\begin{aligned} \Delta w(i,j) &= \theta_0\, w(i,j) + \theta_1\, y(j) + \theta_2\, y(i) + \theta_3\, d(j) \\ &\quad + \theta_4\, w(i,j)\, y(j) + \theta_5\, w(i,j)\, y(i) \end{aligned}$$

[8]voir la figure 2.5 pour une explication sur la séparabilité linéaire et la section 6.2 pour une explication détaillée du problème des fonctions booléennes et la solution présentée dans cette thèse.

$$+\theta_6\, w(i,j)\, d(j) + \theta_7\, y(j)\, y(i)$$
$$+\theta_8\, y(j)\, d(j) + \theta_9\, y(i)\, d(j) \qquad (3.6)$$

6. Les résultats obtenus par la procédure d'optimisation sont intéressants. La meilleure règle d'apprentissage trouvée est

$$\Delta w(i,j) = 4\, y(i)\, (d(j) - y(j)) \qquad (3.7)$$

   ce qui correspond à la règle *delta*, décrite à la section 2.3.3.

7. Il montre aussi que plus le nombre de tâches utilisées pour l'optimisation de la règle est grand, meilleur est le résultat de l'algorithme génétique.

Bien sûr, ces expériences ont été réalisées dans un environnement très simplifié (des fonctions booléennes linéairement séparables) et il est donc dangereux d'en tirer des conclusions hâtives. C'est pourquoi dans cette thèse, nous généralisons l'expérience de Chalmers à plusieurs points de vue:

1. Certaines expériences décrites au chapitre 6 sont faites sur des fonctions linéairement séparables **et** non séparables.
2. D'autres méthodes d'optimisation sont envisagées (et décrites au chapitre 5), telles que le recuit simulé et la descente du gradient.
3. Une théorie sur la généralisation des règles d'apprentissage paramétriques est proposée au chapitre 4 pour expliquer notamment le dernier résultat de Chalmers (le point 7 sur la relation entre le nombre de tâches utilisées pendant l'optimisation et la qualité de la règle résultante).

Chapitre 4

Capacité de généralisation d'une règle d'apprentissage paramétrique

Une règle d'apprentissage devrait être capable d'entraîner plusieurs sortes de réseaux de neurones artificiels, pour résoudre plusieurs sortes de tâches. Le but de la présente thèse est notamment de montrer qu'on peut optimiser une règle d'apprentissage pour qu'elle soit capable de résoudre non pas seulement une tâche mais bien un *ensemble de tâches* ayant des caractéristiques communes.

Pour trouver une telle règle d'apprentissage, nous proposons donc d'optimiser ses paramètres en fonction de son comportement durant l'apprentissage d'un ensemble de tâches. Il peut être intéressant de se demander comment réagira alors notre règle d'apprentissage ainsi optimisée lorsque confrontée à des tâches nouvelles, différentes de celles utilisées pendant l'optimisation de ses paramètres. C'est de cette question que traite ce chapitre (voir aussi [7, 9]).

Nous allons voir ainsi que la *généralisation d'une règle d'apprentissage* (c'est-à-dire sa capacité à résoudre des tâches différentes de celles qui ont servi à son optimisation), peut être reliée à la notion de généralisation d'un système d'apprentissage, telle que nous l'avons décrite à la section 2.5.3, basée elle-même sur le concept de *capacité*.

4.1 Rappel: capacité d'un système d'apprentissage

Nous avons vu à la section 2.5.3 que la capacité d'un système d'apprentissage (ou VCdim) [59] est une mesure de la complexité de ce système et dépend principalement du nombre de paramètres libres du système (les poids dans le cas d'un réseau de neurones). Nous avons aussi montré comment on peut relier la capacité h d'un système d'apprentissage à l'erreur de généralisation qu'il peut faire en pire cas après avoir été adéquatement entraîné sur N exemples d'une tâche donnée. En effet, soit ϵ la différence entre l'erreur d'apprentissage et l'erreur de généralisation, alors:

$$\epsilon \leq \mathcal{O}(\sqrt{\frac{h}{N}} ln \frac{N}{h}) \tag{4.1}$$

où h est la capacité du système et N est le nombre d'exemples utilisés pour l'apprentissage de la fonction à approximer.

Ces résultats fondamentaux guident donc le *design* des systèmes d'apprentissage de la façon suivante:

1. Pour une capacité h donnée, l'erreur de généralisation ϵ décroît lorsque le nombre d'exemples N croît, et

2. Pour un nombre d'exemples N donné, l'erreur de généralisation ϵ décroît lorsque la capacité h croît, jusqu'à ce qu'une valeur critique de h soit atteinte. Passé ce point, ϵ pourrait croître.

4.2 Capacité d'une règle d'apprentissage

Nous définissons ici l'erreur de généralisation d'une règle d'apprentissage paramétrique comme étant l'erreur d'apprentissage en pire cas obtenue par la règle sur une nouvelle tâche, c'est-à-dire une tâche non utilisée pour l'optimisation des paramètres de la règle. On va aussi définir la *capacité d'une règle d'apprentissage paramétrique* comme une mesure de la complexité de cette règle.

Définissons une description de tâche z comme étant l'ensemble (fini ou infini) de couples (e, s) tel que la tâche consiste à associer pour chaque e le s correspondant.

Soit $G(z;\theta)$ l'ensemble des fonctions $g(z;\theta)$ utilisant une règle d'apprentissage paramétrique $f(x;\theta)$ déterminée par sa forme $f(\cdot)$ et ses paramètres θ (x représente l'information locale utilisée par la règle). Le domaine de $g(z;\theta)$ est alors un ensemble de tâches que l'on veut résoudre et z est la description d'une tâche particulière. $g(z;\theta)$ est un nombre réel représentant l'espérance de l'erreur obtenue après avoir entraîné un système d'apprentissage donné avec la règle $f(x;\theta)$ sur la tâche z.

Par exemple, la tâche x pourrait être la fonction booléenne **ET**, déterminée par l'ensemble de couples $\{((0,1),0),((0,0),0),((1,0),0),((1,1),1)\}$. La règle d'apprentissage paramétrique pourrait avoir la forme de l'équation (3.2) avec des valeurs données pour les θ. $g(z;\theta)$ est alors l'erreur d'un réseau de neurones ayant été entraîné avec la règle de l'équation (3.2) sur la tâche z.

Dans ces conditions, nous pouvons appliquer la définition de la capacité donnée par Vapnik dans [59] et déterminer la capacité h d'une règle d'apprentissage paramétrique $G(z;\theta)$ comme étant le nombre maximum de tâches z pouvant être résolues par $G(x;\theta)$ (avec une erreur inférieure à un seuil choisi pour maximiser h).

4.3 Conséquences théoriques

À la lumière de ces résultats, on peut tirer plusieurs conclusions sur la capacité des règles d'apprentissage paramétriques. Ainsi, et suivant les résultats généraux sur la capacité d'un système d'apprentissage, il devient clair que l'espérance de l'erreur obtenue par une règle d'apprentissage sur une nouvelle tâche (c'est-à-dire l'erreur de généralisation de la règle) devrait diminuer lorsqu'on augmente le nombre de tâches utilisées pour l'optimisation des paramètres θ de la règle. Cependant, cette erreur pourrait augmenter lorsque la capacité de la règle, c'est-à-dire une fonction du nombre de paramètres, augmente et que le nombre de tâches est insuffisant.

Ces résultats justifient l'utilisation de connaissances *a-priori* afin de limiter la capacité d'une règle d'apprentissage. De plus, il devient clair que la règle d'apprentissage généralisera mieux sur des tâches similaires à celles utilisées pendant l'optimisation de ses paramètres. En conséquence, on devrait choisir d'optimiser une règle d'apprentissage sur une classe restreinte de tâches, et l'utiliser sur la même classe de tâches. On ne cherche donc plus à découvrir une règle d'apprentissage universelle, mais plutôt *des* règles d'apprentissage spécialisées. Les expériences décrites au chapitre 6 vont donc en ce sens.

Chapitre 5

Méthodes d'optimisation

Comme nous l'avons vu dans le chapitre 3, nous proposons d'utiliser des méthodes d'optimisation traditionnelles pour la recherche d'une nouvelle règle d'apprentissage paramétrique. Dans ce chapitre, nous présentons quatre méthodes d'optimisation utilisées dans les expériences décrites au chapitre 6, soient la descente du gradient, le recuit simulé, les algorithmes génétiques, et finalement la programmation génétique.

Dans chaque cas, la méthode est utilisée pour optimiser un coût E en fonction des paramètres de la règle d'apprentissage. Ce coût peut être défini par exemple par l'équation (3.4), c'est-à-dire en fonction de l'erreur de généralisation d'un réseau de neurones utilisant la règle d'apprentissage donnée pour résoudre un ensemble de tâches donné.

5.1 La descente du gradient

La descente du gradient est une méthode simple d'optimisation locale. Elle peut être appliquée lorsque le critère d'optimisation est une fonction de certains paramètres et qu'il est possible de calculer la dérivée première de ce critère par rapport à ces paramètres.

La règle de la descente du gradient stipule que chaque paramètre qui influence le critère à minimiser doit être modifié itérativement (augmenté ou diminué) dans la direction inverse du gradient. Soit un coût E à optimiser qui dépend des paramètres θ_i. Si l'on veut minimiser E, les paramètres θ_i doivent être modifiés itérativement selon l'équation suivante:

$$\theta_i(t+1) = \theta_i(t) - \epsilon \frac{\partial E(t)}{\partial \theta_i(t)} \tag{5.1}$$

où ϵ est un nombre réel positif relativement petit qui représente le pas de déplacement en direction du minimum le plus proche. La figure 5.1 montre un algorithme simple de descente du gradient.

Cette méthode est cependant très sensible aux minima locaux. Il existe plusieurs variantes de cette méthode, plus rapides et plus résistantes aux minima locaux, par exemple en tenant compte d'informations du second ordre, ou avec un ϵ variable ([5, 34]).

Dans les deux prochaines sections, nous exposons deux méthodes permettant de calculer le gradient désiré pour l'optimisation d'une règle d'apprentissage paramétrique: la première est basée sur l'algorithme de la rétropropagation de l'erreur et la seconde sur la méthode des multiplicateurs de Lagrange.

```
PROCÉDURE DescenteDuGradient
DÉBUT
   s = solution initiale
   c(s) = coût de la solution s
   θ(s) = paramètres de s
   ε = pas de déplacement
   TANTQUE (condition terminale non satisfaite) FAIRE
   DÉBUT
      θ(s*) = θ(s) − ε ∂c(s)/∂θ(s)
      SI (c(s*) < c(s))
         ALORS s = s*
         SINON terminer le programme
   FIN
FIN
```

Figure 5.1: Squelette de l'algorithme de la descente du gradient.

5.1.1 Dérivation par rétropropagation de l'erreur

L'algorithme de la rétropropagation de l'erreur permet d'effectuer une descente du gradient sur les poids d'un réseau de neurones. C'est une méthode simple qui fournit le gradient d'un système d'équations non linéaires sans avoir à le calculer mathématiquement, ce qui n'est pas toujours simple[1] comme nous le verrons dans la prochaine section.

Pour pouvoir utiliser cette méthode pour calculer le gradient par rapport aux paramètres d'une règle d'apprentissage, il faut donc trouver une façon d'exprimer le problème de telle sorte que les paramètres de la règle d'apprentissage soient représentées par les poids d'un réseau de neurones (puisque la rétropropagation de l'erreur trouve le gradient de l'erreur par rapport à des poids).

[1]Pour calculer le gradient d'une variable x par rapport à une autre variable y d'un système d'équations, il faut s'assurer de tenir compte de toutes les influences de la variable y sur la variable x. Nous présentons à la prochaine section une méthode mathématique permettant de trouver toutes les influences de y sur x.

Règle d'apprentissage transformée en réseau de neurones

Un réseau de neurones à couches peut approximer n'importe quelle fonction, en autant qu'il possède suffisamment de neurones cachés (voir section 2.5.1). La figure 5.2 montre un exemple d'un petit réseau de neurones calculant le changement de poids d'une connexion selon la règle de Hebb. On sait que la règle de Hebb dépend de deux variables, $y(i)$ et $x(j)$, la valeur de sortie du neurone présynaptique et l'état d'activation du neurone postsynaptique. Ces deux variables sont donc les entrées du réseau de la figure 5.2. Ils sont connectés à un neurone (via des connexions fixes égales à 1) dont l'opération consiste à faire le produit de ses entrées. Ce dernier neurone est connecté au neurone de sortie via une connexion dont la valeur sera interprétée comme la constante de Hebb (voir l'équation (2.3) par exemple). On voit donc que la sortie de ce petit réseau de neurones correspond bien au changement de poids à effectuer sur des connections d'un réseau de neurones utilisant la règle de Hebb comme procédure d'apprentissage.

De la même manière, on peut définir des règles d'apprentissage paramétrique plus complexes. Ainsi, la figure 5.3 montre l'architecture d'un réseau de neurones calculant le changement de poids selon la règle définie au chapitre 3 par l'équation (3.2). Cette fois-ci, les entrées sont les différentes variables locales disponibles à la synapse: le poids $w(i,j)$, la valeur de sortie de l'unité présynaptique $y(i)$, l'état d'activation de l'unité postsynaptique $x(j)$, et la valeur de sortie d'un neurone modulant le neurone postsynaptique $y\left(mod(j)\right)$. Encore une fois, on utilise des neurones *produit* pour effectuer des calculs intermédiaires. Dans la figure 5.3, les connexions grises correspondent aux 7 paramètres de l'équation (3.2), alors que les connexions noires sont fixées à 1.

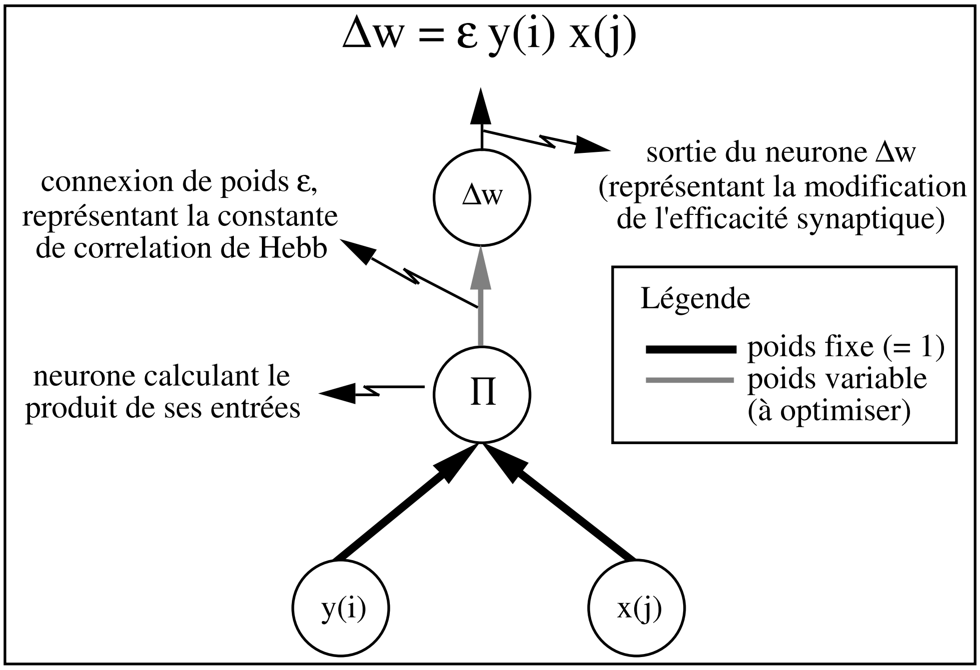

Figure 5.2: Architecture d'un réseau de neurones effectuant le calcul correspondant à la règle de Hebb.

Intégration

Le réseau de neurones calculant le changement de poids selon une règle d'apprentissage paramétrique doit être intégré au réseau de neurone chargé de résoudre la ou les tâches avec cette règle d'apprentissage paramétrique. Pour ce faire, on utilise la *technique des poids partagés* (introduite pour la première fois dans [51]).

Cette technique consiste à forcer deux ou plusieurs connexions d'un même réseau de neurones à conserver le même poids tout au long de l'apprentissage. Lorsqu'utilisé adéquatement, cela permet de réduire la *capacité* d'un réseau de neurones, et donc éventuellement d'augmenter sa généralisation (voir section 2.5.3). On a souvent recours à cette technique lorsque deux neurones doivent calculer la même fonction (reconnaître la même caractéristique par

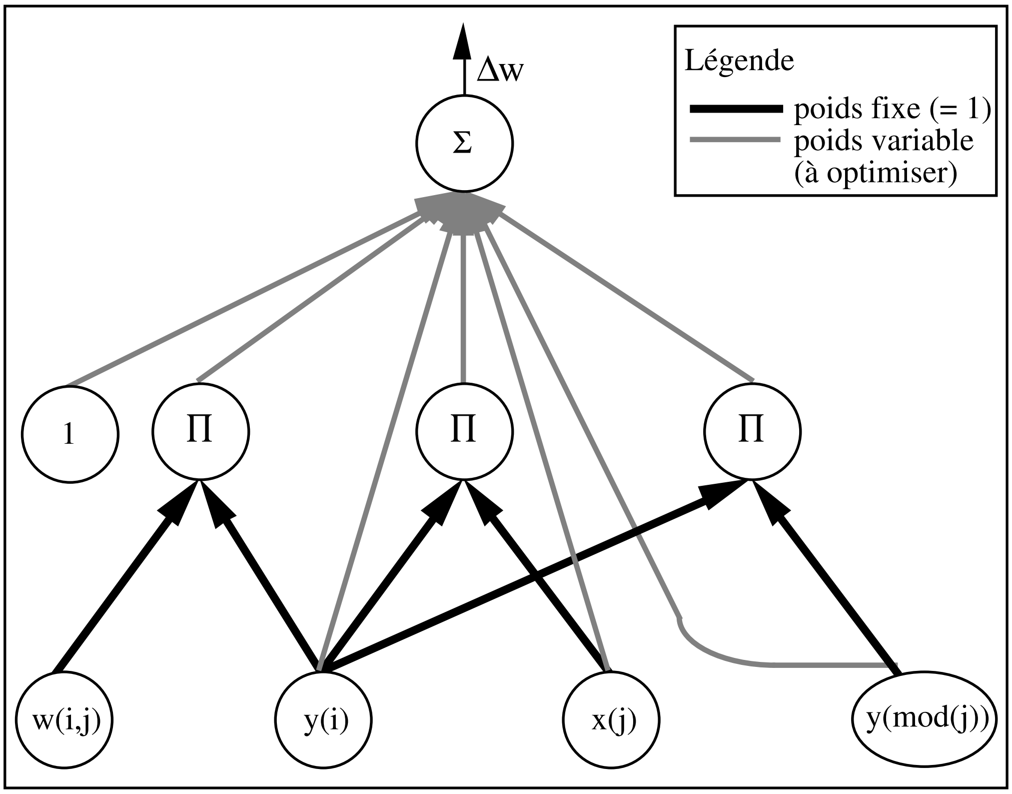

Figure 5.3: Architecture d'un réseau de neurones effectuant le calcul correspondant à une règle ayant 7 paramètres. Les neurones cachés Π effectuent le produit de leurs entrées.

exemple) pour des entrées différentes[2].

Dans notre cas, il s'agit de créer un nouveau réseau de neurones intégrant:

- celui calculant le changement de poids selon la règle d'apprentissage paramétrique (appelé ci-après le *réseau-règle*), et
- celui devant résoudre la ou les tâches (appelé ci-après le *réseau-tâche*).

Pour ce faire, on remplace chaque connexion du *réseau-tâche* par une co-

[2]Ainsi, un système de reconnaissance de caractères utilise souvent la technique des poids partagés pour forcer des neurones à effectuer le même travail indépendamment de la position spaciale à laquelle les neurones sont affectés. Cela permet par exemple de reconnaître une lettre même si elle est légèrement transformée. On parle alors d'invariance par rapport à la rotation ou la translation de l'objet à reconnaître.

pie du *réseau-règle*. Et pour s'assurer que le *réseau-règle* utilisera les mêmes paramètres (les poids du *réseau-règle*) pour toutes les connexions du *réseau-tâche*, on utilisera la technique des poids partagés en forçant les connexions du nouveau réseau à conserver les mêmes poids pour les connexions correspondantes (d'une synapse du *réseau-tâche* à l'autre). La figure 5.4 montre comment créer ce nouveau *super-réseau*.

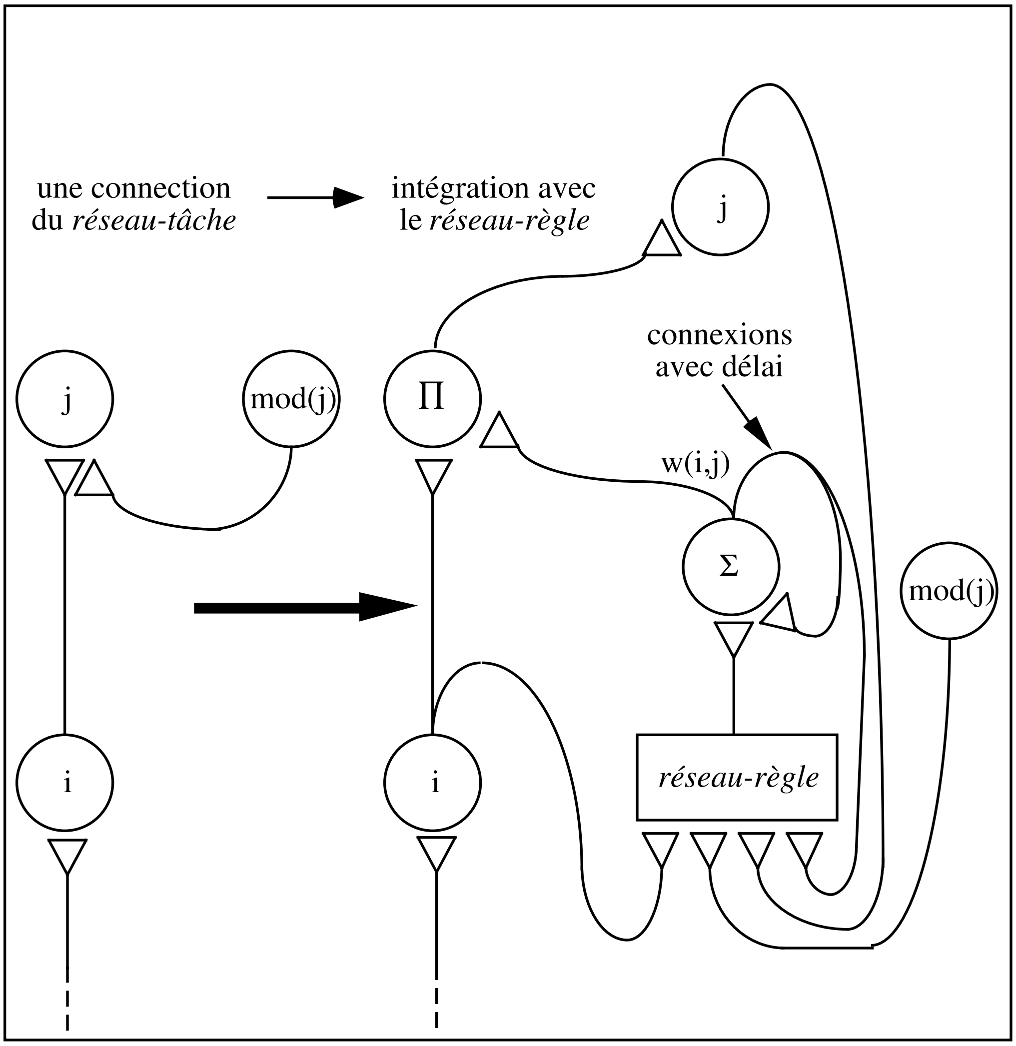

Figure 5.4: Processus d'intégration du *réseau-règle* à toutes les connexions d'un réseau de neurones devant résoudre une tâche donnée en utilisant une règle paramétrique (le *réseau-tâche*). Le neurone Π effectue le produit de ses entrées, alors que le neurone $\sum$ effectue la somme pondérée de ses entrées. À l'exception des connexions contenues dans le *réseau-règle*, toutes les autres sont fixées à 1 et ne varient pas.

Dans cette figure, les neurones Π effectuent le produit des valeurs de sortie de leurs neurones d'entrée, et le neurone $\sum$ effectue la somme pondérée des valeurs de sortie de ses neurones d'entrée. Le poids de la connexion $i \to j$ modulée par $mod(j)$ du *réseau-tâche* est alors conservé dans le nouveau réseau par le neurone $\sum$.

À cause de la rétroaction de l'erreur, le nouveau réseau de neurones est récurrent. Une fois construit, on peut alors appliquer l'algorithme de la rétropropagation de l'erreur à travers le temps afin d'effectuer une descente du gradient par rapport aux paramètres de notre règle d'apprentissage. Même si le nouveau réseau est très gros, il ne contient en fait que peu de poids (grâce à la technique des poids partagés), précisément un nombre égal au nombre de paramètres de la règle d'apprentissage doit être conservé.

Cependant, pour des tâches importantes, cette méthode devient difficilement applicable. En effet, la taille du nouveau réseau de neurones (en nombre de neurones) est dans $\mathcal{O}(N_r \cdot W_t)$ où N_r est le nombre de neurones du *réseau-règle* et W_t le nombre de connexions du *réseau-tâche*. On arrive alors rapidement à un réseau de neurones ayant plusieurs centaines, voire milliers de neurones, ce qui devient inutilisable en pratique (surtout en ce qui a trait à la gestion de la structure de données et au temps de convergence). Il faut donc dans ces cas envisager une autre technique pour effectuer la descente du gradient sur les paramètres: la méthode des multiplicateurs de Lagrange[3].

[3]Notons que les deux méthodes donnent bien sûr le même résultat. La méthode de la rétropropagation de l'erreur est plus simple à utiliser pour des petites tâches, alors que celle des multiplicateurs de Lagrange est nécessaire pour des tâches de plus grosse taille.

5.1.2 Dérivation par la méthode des multiplicateurs de Lagrange

Dans cette section[4], nous présentons une méthode basée sur le formalisme du Lagrangien pour dériver les formules nécessaires pour effectuer une descente du gradient sur les paramètres d'une règle d'apprentissage. Le Cun, dans [38], utilise aussi la méthode du Lagrangien pour étudier les mécanismes de l'algorithme de la rétropropagation de l'erreur.

Le Lagrangien est la somme d'une fonction objective et d'un certain nombre de termes exprimant les contraintes du système, chacun multiplié par une variable, appelée *multiplicateur de Lagrange*. Il suffit alors de dériver cette fonction par rapport à toutes les variables du système et de forcer ces dérivées à zéro pour obtenir un optimum (éventuellement local) de la fonction objective, ou lorsqu'il ne se déduit pas directement, le gradient permettant de s'en approcher.

Notation

Considérons un réseau de neurones ayant des neurones modulateurs composé de N neurones et d'une connectivité quelconque. Considérons de plus que ce réseau se verra présenter I vecteurs d'entrée successifs pour lesquels il existe un vecteur de sortie désirée d.

Dénotons par $x_e(k)$ l'état d'activation du neurone k à la présentation du $e^{ième}$ vecteur d'entrée. De la même façon, $y_e(k)$ représente la sortie du neurone k, $w_e(j,k)$ le poids de la connexion entre le neurone j et le neurone k, et

[4]Cette section a fait l'objet de la publication suivante: [7].

$d_e(k)$ la valeur désirée du neurone de sortie k. Les neurones modulateurs sont notés de la façon suivante: $mod(k)$ représente le neurone qui module toutes les connexions arrivant au neurone k, et si $l = mod(k)$, alors $k = inv(l)$. Enfin, $source(k)$ représente l'ensemble des neurones connectés au neurone k, $dest(k)$ l'ensemble des neurones où k se connecte, et $sortie$ représente l'ensemble des neurones de sortie.

Formulation du problème

Les équations de base régissant l'apprentissage dans un réseau de neurones avec règle d'apprentissage paramétrique et neurones modulateurs sont les suivantes. Tout d'abord, la règle de propagation dicte comment calculer l'état d'activation d'un neurone en fonction de la valeur de sortie des autres neurones.

$$x_e(k) = \sum_{j \in source(k)} w_e(j,k) \cdot y_e(j) \tag{5.2}$$

Ensuite, une fonction de sortie permet de calculer la sortie d'un neurone en fonction de son état d'activation.

$$y_e(k) = f\left(x_e(k)\right) \tag{5.3}$$

où $f(\cdot)$ est une fonction continue (on utilise souvent une fonction sigmoïde).

Enfin, une règle d'apprentissage dicte la variation des poids dans le temps. Dans notre cas, cette règle est remplacée par une fonction paramétrique comme celle de l'équation (3.2).

$$\begin{aligned} w_{e+1}(j,k) &= w_e(j,k) + \\ & r\sum_i \theta_i M_i(j,k,x_e(k),w_e(j,k),y_e(mod(k)),y_e(j)) \end{aligned} \quad (5.4)$$

où r est le taux d'apprentissage, M_i est une fonction de plusieurs valeurs locales (par exemple $w_e(j,k)$, $y_e(j)$, etc), et θ_i est le paramètre contrôlant l'influence du module M_i (par exemple, $M_i = y(j) \cdot x(k)$ est un module exprimant la règle de Hebb).

Comme dans tout système d'apprentissage supervisé, le réseau doit évoluer de façon à minimiser un coût donné. Ici, nous utilisons le critère des moindres carrés:

$$\sum_{k \in sortie} E(y_e(k)) = \frac{1}{2} \sum_{k \in sortie} (d_e(k) - y_e(k))^2 \quad (5.5)$$

Dérivation du gradient

Le Lagrangien de ce système d'équations s'écrit alors comme suit:

$$\begin{aligned} & L(x,y,w,\theta,\alpha,\beta,\gamma) = \\ & \quad \sum_{e=1}^{I} \sum_{k \in sortie} E(y_e(k)) \\ & \quad + \sum_{e=1}^{I} \sum_{k=1}^{N} \alpha_e(k)(x_e(k) - \sum_{j \in source(k)} w_e(j,k) \cdot y_e(j)) \\ & \quad + \sum_{e=1}^{I} \sum_{k=1}^{N} \beta_e(k)(y_e(k) - f(x_e(k))) \end{aligned}$$

$$+\sum_{e=1}^{I-1}\sum_{k=1}^{N}\sum_{j\in source(k)}\gamma_e(j,k)(w_{e+1}(j,k)-w_e(j,k)-$$
$$r\sum_i \theta_i M_i(j,k,x_e(k),w_e(j,k),y_e(mod(k)),y_e(j))) \qquad (5.6)$$

Le Lagrangien est donc la somme de la fonction objective (équation (5.5)) et des contraintes du système (équations (5.2) à (5.4)). Les vecteurs de variables α, β, et γ représentent les multiplicateurs de Lagrange. Pour découvrir le gradient de l'erreur par rapport aux paramètres θ_i, il suffit de dériver le Lagrangien (5.6) par rapport à toutes les variables du système: $x_e(k)$, $y_e(k)$, $w_e(j,k)$, θ_i, $\alpha_e(k)$, $\beta_e(k)$, $\gamma_e(j,k)$, et à les forcer à 0 (si possible). Après quelques substitutions, on peut isoler chaque variable, ce qui donne les équations suivantes.

La première équation nous donne $\alpha_e(k)$ lorsque $\frac{\partial L}{\partial x_e(k)} = 0$.

$$\alpha_e(k) = \beta_e(k)\cdot f'(x_e(k)) + \sum_{j\in source(k)}(\gamma_e(j,k)\cdot r\sum_i \theta_i \frac{\partial M_i(j,k)}{\partial x_e(k)}) \qquad (5.7)$$

La deuxième équation nous donne $\beta_e(k)$ lorsque $\frac{\partial L}{\partial y_e(k)} = 0$.

$$\begin{aligned}\beta_e(k) \quad = \quad & \sum_{j\in dest(k)}(\alpha_e(j)\cdot w_e(k,j)) \\ & + \sum_{j\in source(inv(k))}(\gamma_e(j,inv(k))\cdot r\sum_i \theta_i \frac{\partial M_i(j,inv(k))}{\partial y_e(k)}) \\ & + \sum_{j\in dest(k)}(\gamma_e(k,j)\cdot r\sum_i \theta_i \frac{\partial M_i(k,j)}{\partial y_e(k)}) \\ & + \delta_k^{sortie}(d_e(k)-y_e(k)) \end{aligned} \qquad (5.8)$$

où δ_k^{sortie} vaut 1 lorsque $k \in sortie$ et 0 autrement.

L'équation (5.9) nous donne $\gamma_{e-1}(j,k)$ lorsque $\frac{\partial L}{\partial w_e(j,k)} = 0$.

$$\gamma_{e-1}(j,k) = \alpha_e(k) \cdot y_e(j) + \gamma_e(j,k) \cdot \left(1 + r \sum_i \theta_i \frac{\partial M_i(j,k)}{\partial w_e(j,k)}\right) \tag{5.9}$$

Et l'équation (5.10) nous fournit ce que l'on cherchait: le gradient par rapport aux paramètres.

$$\frac{\partial L}{\partial \theta_i} = \sum_{e=1}^{I} \sum_{k=1}^{N} \sum_{j \in source(k)} -\gamma_e(j,k) \cdot r M_i(j,k) \tag{5.10}$$

Pour finir, en dérivant par rapport aux variables $\alpha_e(k)$, $\beta_e(k)$ et $\gamma_e(j,k)$, on retrouve les équations de base du système. Ainsi, lorsque $\frac{\partial L}{\partial \alpha_e(k)} = 0$,

$$x_e(k) = \sum_{j \in source(k)} w_e(j,k) \cdot y_e(j) \tag{5.11}$$

lorsque $\frac{\partial L}{\partial \beta_e(k)} = 0$,

$$y_e(k) = f\left(x_e(k)\right) \tag{5.12}$$

et lorsque $\frac{\partial L}{\partial \gamma_e(j,k)} = 0$,

$$w_{e+1}(j,k) = w_e(j,k) +$$

$$r\sum_i \theta_i M_i(j,k,x_e(k),w_e(j,k),y_e(mod(k)),y_e(j)) \quad (5.13)$$

Afin de minimiser le coût défini dans l'équation (5.5), on peut alors modifier θ_i itérativement par

$$\forall i \quad \theta_i = \theta_i - \epsilon \frac{\partial L}{\partial \theta_i} \tag{5.14}$$

où ϵ représente le pas de déplacement et $\frac{\partial L}{\partial \theta_i}$ le gradient par rapport aux paramètres θ_i. L'équation (5.14) peut se réécrire grâce à (5.10):

$$\forall i \quad \theta_i = \theta_i + \epsilon \sum_e \sum_k \sum_{j \in source(k)} \gamma_e(j,k) \cdot r M_i(j,k) \tag{5.15}$$

L'équation (5.15), combinée aux équations (5.7) à (5.9), nous permet donc de calculer le gradient de l'erreur par rapport aux paramètres θ_i, et ainsi optimiser une règle d'apprentissage paramétrique par descente du gradient. En effet, on peut, à l'aide des équations (5.7) à (5.9), calculer récursivement à travers le temps (les I vecteurs d'entrée) tous les α, β, et γ nécessaires dans l'équation (5.15)[5].

5.2 Le recuit simulé

Le recuit simulé est une méthode d'optimisation inspirée par des principes de la thermodynamique, qui permet de trouver le minimum (ou le maximum)

[5]Le calcul étant récursif, les valeurs initiales (ou plutôt finales puisque l'on recule dans le temps) des $\alpha_e(k)$, $\beta_e(k)$, et $\gamma_e(j,k)$ sont fixées à 0.

global d'une fonction de coût donnée. Plusieurs analyses de cette méthode ont démontré sa convergence vers un optimum global lorsque certaines conditions sont respectées.

La thermodynamique est une branche de la physique et de la chimie qui étudie les relations entre l'énergie thermique (chaleur) et mécanique (travail), et les lois générales des phénomènes impliquant des échanges ou des transformations thermiques. Elle s'intéresse par exemple aux qualités physique d'un métal ou d'un alliage. La méthode du *recuit* est originalement une opération thermique destinée à améliorer les qualités mécaniques d'un métal, d'un alliage.

Cette méthode consiste à chauffer un métal afin de lui permettre de changer d'état facilement, puis lentement, à descendre la température afin que le métal se stabilise dans l'état le plus stable, qui correspond à l'état où ses qualités mécaniques sont les meilleures. On peut montrer en fait que pour une température donnée T, un solide pourra atteindre l'équilibre thermique (l'état le plus stable) caractérisé par la probabilité d'être dans un état d'énergie E selon la distribution de Boltzmann (appelée aussi distribution de Gibbs) ([37]):

$$P\{E_t = E\} = \frac{1}{Z(T)} e^{\left(-\frac{E}{K_B T}\right)} \tag{5.16}$$

où $Z(T)$ est un facteur de normalisation (qui permet de faire de cette distribution une distribution de probabilité, c'est-à-dire que l'intégrale de $P(E)$ donne 1), et K_B est la constante de Boltzmann.

Cette distribution de probabilité a la propriété suivante: lorsque la température diminue, la distribution de Boltzmann se concentre sur les états ayant l'énergie la plus basse jusqu'au moment où, à température proche de 0, seuls les états d'énergie minimale ont une probabilité non-nulle d'arriver. Cependant, si la température diminue trop rapidement, c'est-à-dire si l'on ne permet pas au solide d'atteindre l'équilibre thermique à chaque température, le solide peut se stabiliser dans un état indésirable.

Le *recuit simulé* ([53]) est une adaptation de cette méthode de recuit permettant de l'utiliser dans la résolution de problèmes d'optimisation. On fait le parallèle entre énergie d'un système et fonction de coût d'un problème d'optimisation. L'algorithme consiste à se déplacer dans l'espace des solutions de façon itérative en choisissant une prochaine solution voisine de la solution courante de façon aléatoire selon la formule de Gibbs. De plus, la température, à l'instar de la méthode du recuit, sera choisie initialement assez haute, puis diminuée selon un horaire bien précis. Lorsque la température est élevée, l'algorithme acceptera des solutions qui peuvent détériorer la fonction de coût (et donc s'échapper de minima locaux), alors que vers la fin de la procédure, à basse température, on se dirigera uniquement vers le minimum local le plus proche qui, si l'horaire de refroidissement à été adéquatement choisi (et que nous ne sommes pas trop malchanceux), sera le minimum global de la fonction de coût. La figure 5.5 montre le squelette d'un algorithme effectuant la procédure du recuit simulé.

Plusieurs auteurs ([24, 37]) ont montré que sous certaines conditions, l'algorithme du recuit simulé converge éventuellement vers le minimum global de la fonction de coût. Une raison intuitive de la convergence est la suivante: considérons la figure 5.6 illustrant un minimum local. La boule représente notre solution courante et initialement, elle peut se retrouver n'importe où sur le graphique. Si on la laisse aller vers le bas (sous la gravité et sans inertie) elle aura autant de chance de tomber au point A qu'au point B. Si on perturbe ensuite le système, on a plus de chance de faire déplacer la balle du

```
PROCÉDURE RecuitSimulé
DÉBUT
   i = 0
   s = solution initiale
   c(s) = coût de la solution s
   T(·) = la cédule de refroidissement. (T(i) diminue lorsque i augmente)
   TANTQUE (condition terminale non satisfaite) FAIRE
   DÉBUT
      choisir aléatoirement s*, un voisin de s
      SI (c(s*) < c(s)) ALORS s = s*
      SINON avec probabilité P = e^{-(c(s*)-c(s))/T(i)} FAIRE s = s*
      i = i + 1
   FIN
FIN
```

Figure 5.5: Squelette de l'algorithme du recuit simulé.

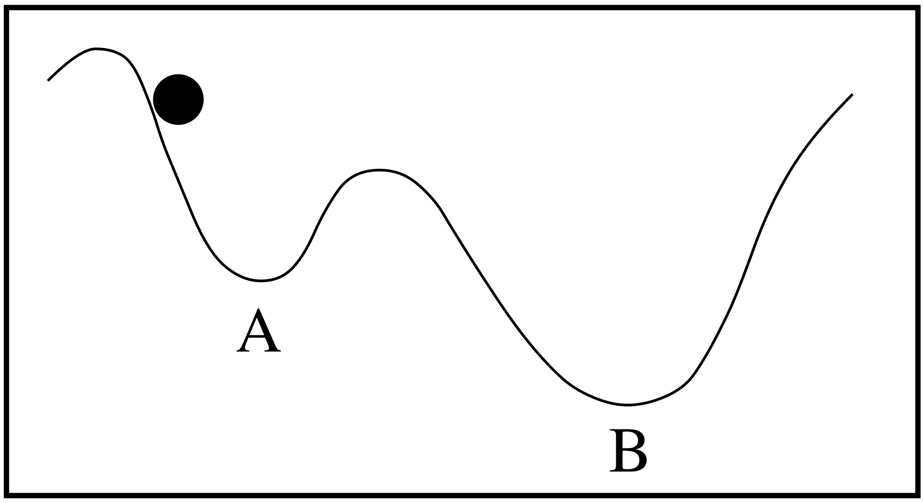

Figure 5.6: Illustration du problème des minima locaux. Si la boule noire est lâchée au point actuel, elle aboutira par gravité (qui représente la descente du gradient) au minimum local A, différent du minimum global B.

point A vers le point B que l'inverse puisque l'énergie nécessaire pour aller de A à B est plus petite que pour aller de B à A (la côte est moins grande), mais les deux cas sont possibles. Maintenant, si on continue à perturber le système mais moins brutalement, une transition de A vers B sera plusieurs fois plus probable qu'une transition de B vers A. En réduisant lentement la température (la brutalité de la perturbation), la probabilité que la balle se trouve en B (donc au minimum global) augmente graduellement.

Le recuit simulé est une méthode d'optimisation qui est donc moins sensible au minima locaux que la simple descente du gradient (grâce à l'introduction du bruit généré par le facteur température). Néanmoins, cet avantage est obtenu au prix d'un temps de convergence souvent prohibitif. En effet, pour s'assurer d'obtenir un minimum global, l'horaire de refroidissement doit souvent être démesurément long. Souvent, on modifiera l'horaire, au risque de tomber tout de même dans un minimum local, mais avec l'effet d'accélérer significativement la procédure.

5.3 Les algorithmes génétiques

Les algorithmes génétiques [26, 32] sont un exemple de méthode d'optimisation globale inspirée par les processus évolutionistes darwiniens: *sélection, reproduction, mutation.*

Considérons une population d'individus représentant chacun une solution potentielle à un problème donné. Chaque individu est codé (représenté) par un ensemble de *chromosomes* artificiels (on utilise généralement une séquence de *bits* pour représenter ces chromosomes). Le processus d'optimisation passe alors par les stades suivants:

1. *Sélection*: On évalue chaque individu selon l'efficacité de la solution qu'il représente face à la fonction de coût à optimiser. On choisit les meilleurs individus selon ce critère.

2. *Reproduction*: Les meilleurs individus sont utilisés pour créer une nou-

velle population complète d'individus. Cette étape est réalisée à l'aide d'opérateurs de reproduction telle que le croisement (*crossover*) qui crée un nouvel individu basé sur un mélange des chromosomes de deux individus *parents* (voir figure 5.7 pour un exemple simplifié de croisement).

3. *Mutation*: Chaque individu de la nouvelle population a une mince chance de subir une modification aléatoire de sa séquence de chromosomes.

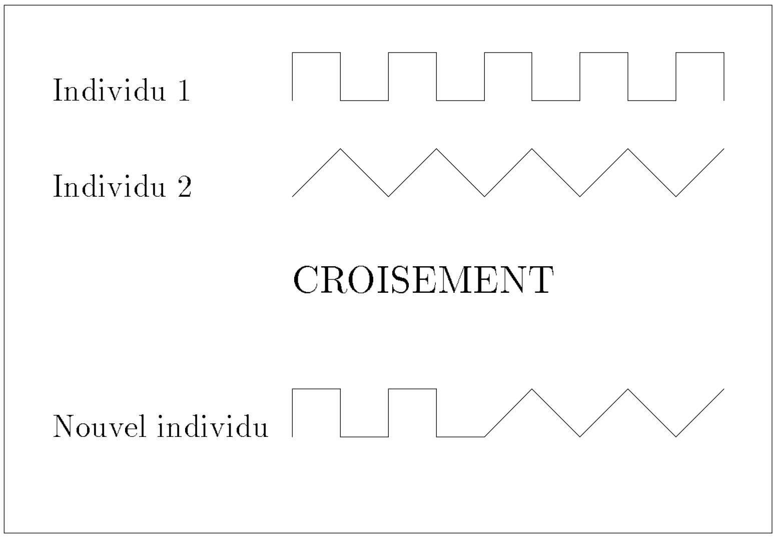

Figure 5.7: Exemple de l'utilisation du croisement pour créer un nouvel individu à partir de deux parents. Le nouvel individu possède une partie des *chromosomes* de chacun de ses parents.

Ces trois stades permettent de créer une nouvelle population. On recommence ce processus pendant plusieurs générations (voir algorithme à la figure 5.8), et à l'instar de l'évolution darwinienne, la population s'améliore graduellement.

On peut se demander si cet algorithme converge toujours vers une solution (ou plutôt vers une population de solutions) optimale(s). Il n'existe

```
PROCÉDURE AlgoGénétique
DÉBUT
   t = 0
   initialiser Population(t)
   évaluer les individus de Population(t)
   TANTQUE (condition terminale non satisfaite) FAIRE
   DÉBUT
      t = t + 1
      sélection de Population(t) à partir de Population(t − 1)
      reproduction de Population(t)
      mutation de Population(t)
      évaluer les individus de Population(t)
   FIN
FIN
```

Figure 5.8: Squelette d'un algorithme génétique.

pas actuellement de preuve de convergence pour les algorithmes génétiques mais plusieurs résultats théoriques ont néanmoins été réalisés récemment. Ainsi, Holland démontre dans [32] que des parties de solution (qu'il appelle *schéma*) *meilleures* (*pires*) au sens de la fonction de coût, croissent (décroissent) exponentiellement d'une génération à l'autre. Ce résultat explique que la population aura toujours tendance à s'améliorer car les *meilleurs* schémas éliminent peu à peu les *pires* schémas.

D'une perspective pratique, Davis ([18]) propose quelques changements à l'algorithme initial qui améliorent son efficacité:

- normaliser la fonction de coût linéairement,
- interdire les duplicatas (les copies identiques) dans une population,
- ajuster les divers paramètres d'une génération à l'autre. Ainsi, le taux de croisement est souvent réduit en cours de route alors que celui de mutation augmente,
- enfin, Davis propose de créer des opérateurs supplémentaires qui tien-

nent compte de chaque problème. Pour cela, il suggère de s'inspirer des autres techniques d'optimisation disponibles pour le problème à régler.

Dans les expériences réalisées au chapitre 6, nous avons utiliser les algorithmes génétiques sans aucune des améliorations suggérées par Davis. L'opérateur de croisement consistait pour chaque bit à choisir aléatoirement le bit correspondant d'un des deux parents. L'opérateur de mutation consistait à inverser aléatoirement certains bits de la solution.

Un des avantages des algorithmes génétiques tient au fait qu'ils ne sont pas sensibles au minima locaux car la méthode est globale et gère un grand nombre de solutions simultanément. De plus, ces algorithmes sont facilement parallélisables. Enfin, comme pour le recuit simulé, il n'est pas nécessaire de pouvoir dériver la fonction objective par rapport à ses paramètres, ce qui en simplifie souvent l'implantation.

Cependant, si la méthode est utilisée sur un processeur séquentiel (ce qui est souvent le cas), le temps de convergence pourrait être très élevé car il faut gérer à chaque itération (génération) non pas une mais plusieurs solutions.

5.4 La programmation génétique: une alternative intéressante

Toutes les méthodes d'optimisation présentées précédemment ont besoin que la forme de la règle d'apprentissage paramétrique soit fixée à l'avance, déterminée par un ensemble de contraintes telles que par exemple le type de tâches à résoudre. Le problème avec cette technique, c'est qu'on peut passer à côté de règles très intéressantes simplement parce qu'elles ne se trouvent pas dans l'espace des règles d'apprentissage généré par la forme choisie.

Une nouvelle méthode d'optimisation, la *programmation génétique* [36], pourrait peut-être nous aider à palier à ce défaut. Alors que les algorithmes génétiques permettent, par des techniques évolutionnistes, de trouver un ensemble de *paramètres optimaux* d'une fonction paramétrique prédéterminée, la programmation génétique utilise les mêmes techniques de base, mais permet plutôt de trouver une *fonction optimale* pour un problème d'optimisation donné.

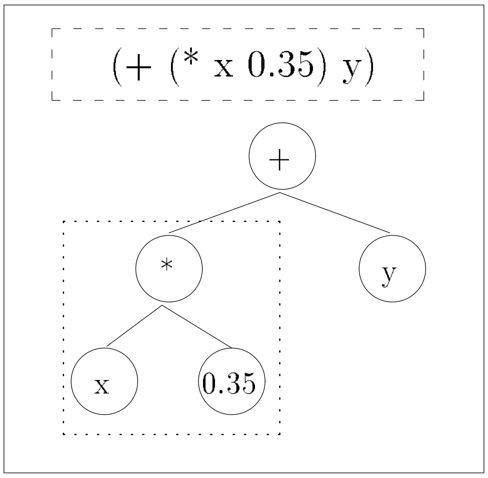

Figure 5.9: Exemple d'une fonction exprimée sous forme d'arbre. Les noeuds représentent des opérateurs et les feuilles représentent des variables ou des constantes.

En effet, tout en conservant les principes fondamentaux des algorithmes génétiques (population, sélection, reproduction, mutation), la programmation génétique considère l'individu comme un programme, une fonction, représentée généralement et par commodité par un arbre. Ainsi, l'arbre de la figure 5.9 représente la fonction

$$0.35x + y \tag{5.17}$$

Dans le cas de la programmation génétique, le but est de trouver la fonction (le programme) qui solutionne le mieux un problème donné. Pour ce faire, on considère tout d'abord une population d'individus (dans ce cas, des fonctions). Initialement ces individus sont créés aléatoirement avec l'aide d'un ensemble d'opérateurs ({et, ou, non} pour des fonction binaires, {+, -, *, /} pour des fonctions arithmétiques, etc) adéquats ainsi qu'un ensemble de terminaux (qui peuvent être des constantes ou des variables).

Ensuite, à l'aide de certains opérateurs de reproduction, on crée une nouvelle génération d'individus à partir des meilleurs individus de la population initiale. Ainsi, de génération en génération, la population tend à s'améliorer globalement lorsque les opérateurs de reproduction sont adéquatement choisis. Ceux-ci sont à peu près les mêmes que ceux utilisés par les algorithmes génétiques: le croisement (ou *crossover*) et la mutation.

L'opérateur de croisement (figure 5.10) consiste à échanger aléatoirement deux sous-arbres de deux individus parents pour créer deux nouveaux individus. L'opérateur de mutation (figure 5.11) consiste à remplacer aléatoirement un sous-arbre d'un individu par un nouveau sous-arbre. Le résultat est un nouvel individu.

La programmation génétique peut être utilisée dans le cadre de l'optimisation de règles d'apprentissage. Dans ce cas, le but sera non pas de trouver un ensemble de paramètres d'une règle d'apprentissage paramétrique, mais bien de trouver une nouvelle règle. Ainsi, cette méthode à l'avantage de ne pas restreindre la recherche à un espace de règles contraint par une forme pré-établie comme c'est le cas pour les autres méthodes d'optimisation en-

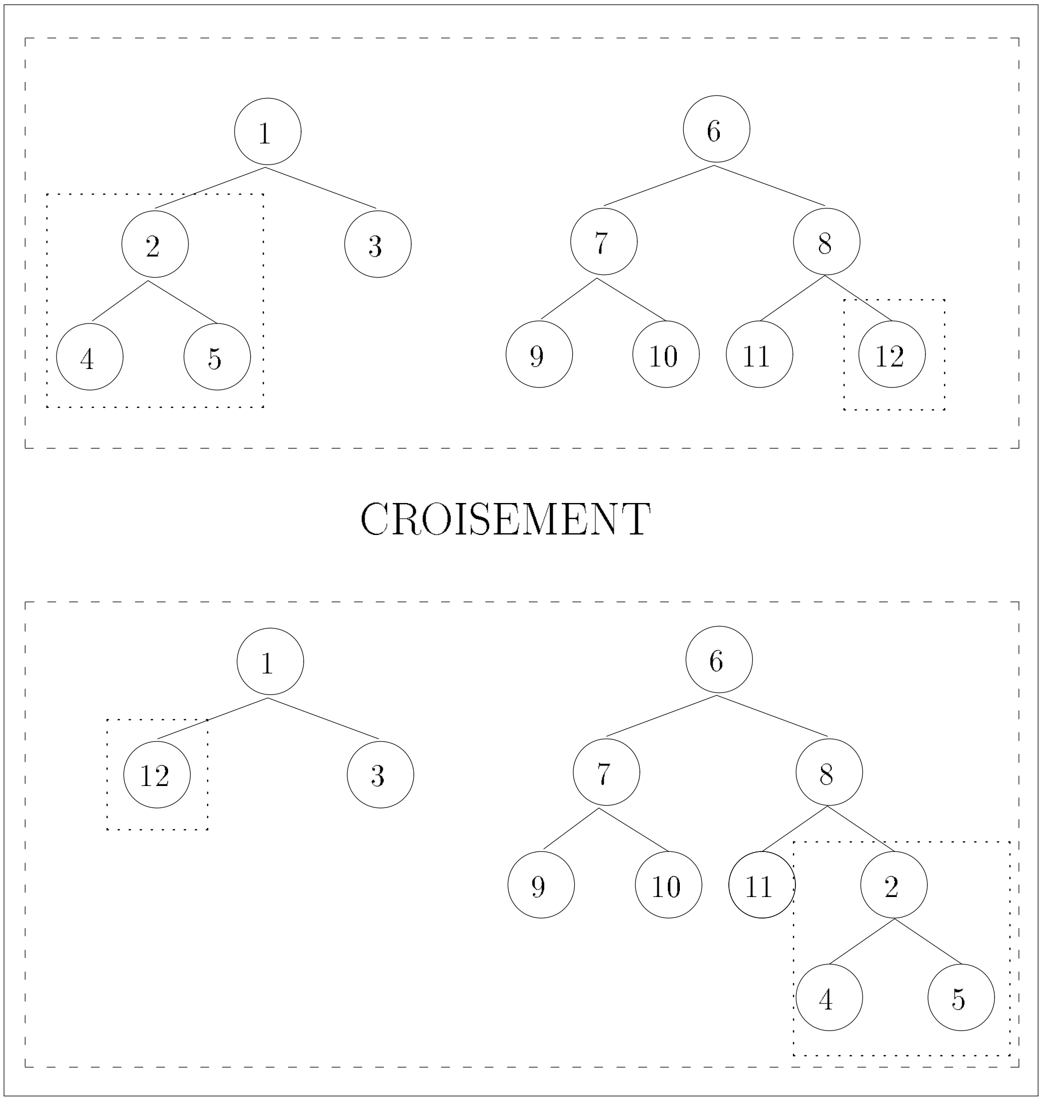

Figure 5.10: Exemple de croisement entre deux arbres. Le sous-arbre dont la racine est le noeud 2 est interchangé avec le sous-arbre dont la racine est le noeud 12.

visagées. Cet avantage est malheureusement contrebalancé par un besoin accru en temps de calcul pour converger éventuellement vers une solution intéressante.

De plus, il devient difficile d'envisager la capacité[6] d'une population de règles

[6]Dans le sens où nous l'avons décrit au chapitre 4.

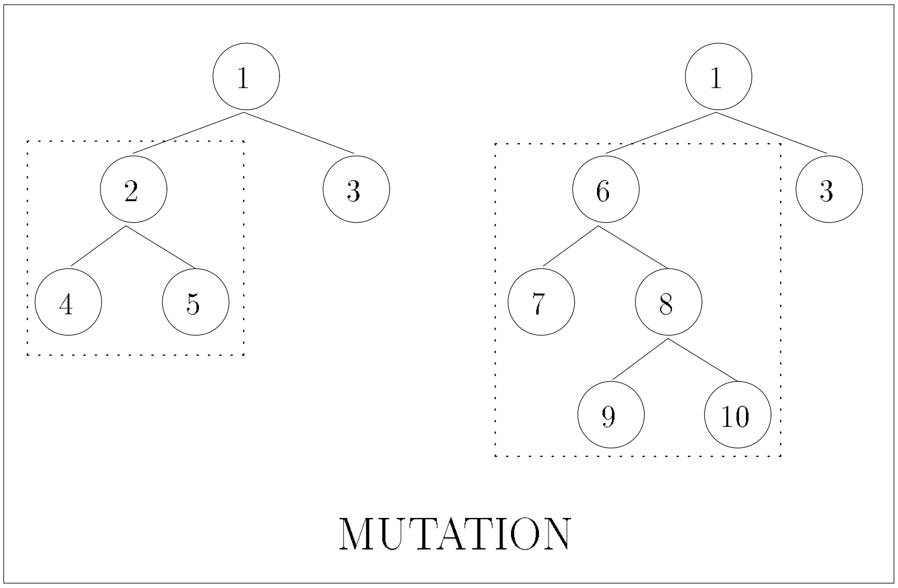

Figure 5.11: Exemple de mutation entre deux arbres. Le sous-arbre dont la racine est le noeud 2 est changé par un nouveau sous-arbre dont la racine est le noeud 6.

d'apprentissage ainsi créées. On imagine ainsi aisément qu'à force de croisements et de mutations, les individus, quoique de taille initiale circonscrite, peuvent atteindre des tailles énormes (à moins d'empêcher par un système de pénalité la formation d'individus de grande taille). N'oublions pas en effet que plus la taille de la règle est grande, plus sa capacité augmente, et plus l'erreur de généralisation pourrait s'aggraver.

Pour les expériences décrites au chapitre 6, nous avons utilisé la programmation génétique sans miseaupoint particulière. Les opérateurs de croisement et de mutation sont exactement ceux décrits ici. Afin de contrôler la capacité des règles engendrées, nous avons simplement empêché la formation de règles ayant plus de 40 paramètres différents.

Chapitre 6

Expérimentation

Afin de démontrer la réalisabilité d'une méthode permettant l'optimisation de règles d'apprentissage paramétriques, nous avons effectué plusieurs expériences dans des domaines variés, en utilisant différentes formes de règles d'apprentissage et différentes méthodes d'optimisation. Le présent chapitre décrit trois séries d'expériences distinctes. On peut aussi retrouver la description et les résultats de ces expériences dans [11, 10, 8, 7, 9].

Pour la première série d'expériences, nous avons choisi un cadre *biologique* en utilisant une règle d'apprentissage paramétrique *biologiquement plausible*: il s'agit d'expériences traitant le conditionnement classique. La deuxième série d'expériences concerne des tâches booléennes. Il s'agit alors de vérifier si l'on peut utiliser une règle d'apprentissage biologiquement plausible pour résoudre des tâches *difficiles*. Enfin, la troisième série d'expériences, sur des tâches de classification, a pour but de vérifier si les aspects théoriques de généralisation des règles d'apprentissage paramétriques sont en accord avec la théorie sur la capacité décrite au chapitre 4.

6.1 Conditionnement classique

Dans cette série d'expériences, nous avons tenté de découvrir une règle d'apprentissage capable de reproduire certains phénomènes de conditionnement classique chez l'animal. Ces phénomènes, d'abord décrits par Pavlov ([47]) ont été depuis très étudiés et plusieurs modèles descriptifs existent. Pour nos expériences, nous utilisons le modèle décrit par Hawkins [27].

6.1.1 Description du problème

On distingue deux sortes de *stimuli* pouvant provoquer une réaction chez un animal:

- **Les stimuli inconditionnels** qui provoquent à tout coup une réaction chez l'animal (par exemple, si l'on observe la salivation chez le chien, le fait de lui présenter un plat de nourriture le fera saliver à tout coup).
- **Les stimuli conditionnels** qui, à moins d'être conditionnés comme nous le verrons plus tard, produisent généralement une faible réaction de la part de l'animal (en reprenant l'exemple du chien, le fait d'allumer une lumière ne fait pas particulièrement saliver le chien).

Les phénomènes de conditionnement que nous avons étudiés et cherché à reproduire sont les suivants:

- **L'habituation**: initialement, un stimulus conditionnel SC_1 (par exemple, une lumière rouge) produit une réponse faible sur l'animal (par exemple, ce dernier salive un peu). En présentant de façon répétitive SC_1, la réponse de l'animal devient de plus en plus faible (celui-ci s'habitue et tend à ne plus tenir compte du stimulus).

- **Le conditionnement**: on fait suivre un stimulus conditionnel SC_1 par un stimulus inconditionnel SI (par exemple, une lumière rouge suivie d'un plat de nourriture). La réponse à SC_1 augmente graduellement (l'animal salive avant de voir la nourriture, dès que la lumière rouge s'allume).

- **Le blocage**: après le conditionnement de SC_1, on présente de façon répétitive à l'animal le stimulus conditionnel SC_1 simultanément avec un second stimulus conditionnel SC_2 (par exemple, une lumière verte), le tout suivi d'un stimulus inconditionnel SI. Dans ce cas, SC_2 ne devient pas conditionné (l'animal ne salivera pas à la seule présentation de la lumière verte).

- **Le conditionnement de second ordre**: après le conditionnement de SC_1, on peut conditionner SC_2 en présentant à l'organisme SC_2 suivi de SC_1 (l'animal commence à saliver en voyant la lumière verte, sachant qu'elle est suivie de la lumière rouge, elle-même suivie de la nourriture). SC_2 est alors conditionné par SC_1.

- **L'extinction**: après le conditionnement de SC_1, une présentation répétée de SC_1 non suivie de SI réduira la réponse de l'animal à son niveau original (si on ne fournit plus de nourriture après la lumière rouge, l'animal tend à moins saliver avec le temps).

Hawkins décrit dans [28, 27] un réseau de neurones biologique simple observé dans un mollusque nommé *Aplysia*. Ce réseau de quelques neurones est capable de reproduire les 5 phénomènes de conditionnement décrits précédemment. Hawkins propose plusieurs mécanismes d'apprentissage pour expliquer les phénomènes observés. Nous nous proposons de trouver une règle d'apprentissage capable de reproduire de façon analogue à l'Aplysia les phénomènes de conditionnement classique (et donc, de trouver les poids optimaux du réseau de neurones pour reproduire les 5 phénomènes).

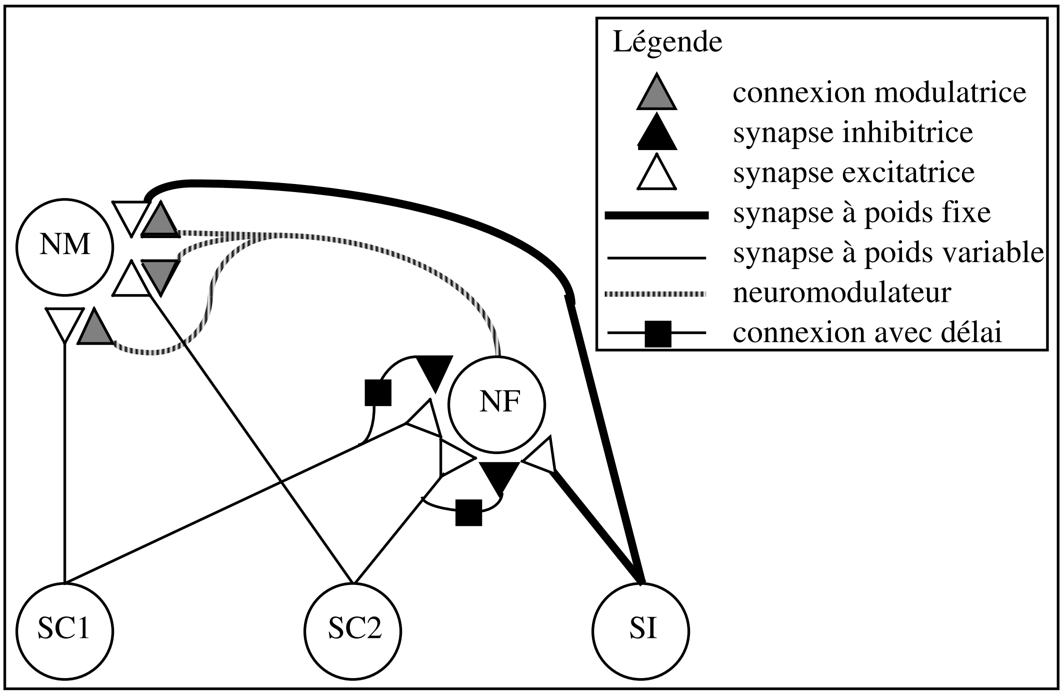

Figure 6.1: Architecture d'un réseau de neurones pour le conditionnement classique, inspiré des observations de Hawkins sur l'*Aplysia*. Les neurones SC_1, SC_2 et SI représentent des stimuli, le neurone moteur NM représente la sortie du réseau, alors que le neurone facilitateur NF module les connexions aboutissant au neurone NM. Cette architecture représente réellement une partie du circuit neuronal de l'*Aplysia*.

La figure 6.1 reproduit l'architecture du réseau de neurones de l'*Aplysia* tel que décrit par Hawkins, que nous utiliserons aussi pour nos expériences. Dans ce réseau les neurones SC_1 et SC_2 représentent des stimuli conditionnels, le neurone SI est un stimulus inconditionnel, le neurone NF est un neurone facilitateur qui module les connections du neurone moteur NM, qui représente la réponse de l'animal aux divers stimuli. Les stimuli influencent le neurone moteur (par des connexions vers NM) et ces connexions sont elles-mêmes influencées (ou modulées) par le neurone facilitateur qui, grâce aux connexions à délai arrivant à lui, peut tenir compte notamment de deux états consécutifs du système (par exemple le fait que SC_1 soit actif au temps t et SI actif au temps $t+1$).

6.1.2 Forme de la règle d'apprentissage

Pour résoudre les 5 tâches de conditionnement, nous avons choisi une forme de règle d'apprentissage simple, utilisant les 4 variables locales disponibles à toute connexion (i,j): la valeur de sortie du neurone présynaptique $y(i)$, l'activité du neurone postsynaptique $x(j)$, la valeur de sortie du neurone modulateur (facilitateur) $y\,(mod(j))$, ainsi que la valeur de la connexion $w(i,j)$. Il s'agit de la règle décrite à la section 3.4 par l'équation (3.2):

$$\begin{aligned}\Delta w(i,j) \;&=\; \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) + \\ &\quad\;\; \theta_4\, y(j)\, y(mod(j)) + \theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, w(i,j)\end{aligned} \tag{6.1}$$

Comme il est mentionné à la section 3.4, cette règle, qui ne contient que 7 paramètres, est construite en utilisant des connaissances biologiques *a priori* afin de réduire sa capacité.

6.1.3 Représentation des données

Chaque tâche de conditionnement peut être prise séparément pour construire une séquence de tuples correspondant à la tâche (telle que décrite par Hawkins). Chaque tuple (SC_1, SC_2, SI, d) correspond à la réponse désirée d lors de la présentation des stimuli SC_1, SC_2 et SI. La figure 6.2 donne un exemple d'une partie d'une telle séquence permettant de modéliser la tâche de conditionnement. Dans cette séquence, la colonne SC_2 à été volontairement omise, car inutile pour la tâche de conditionnement.

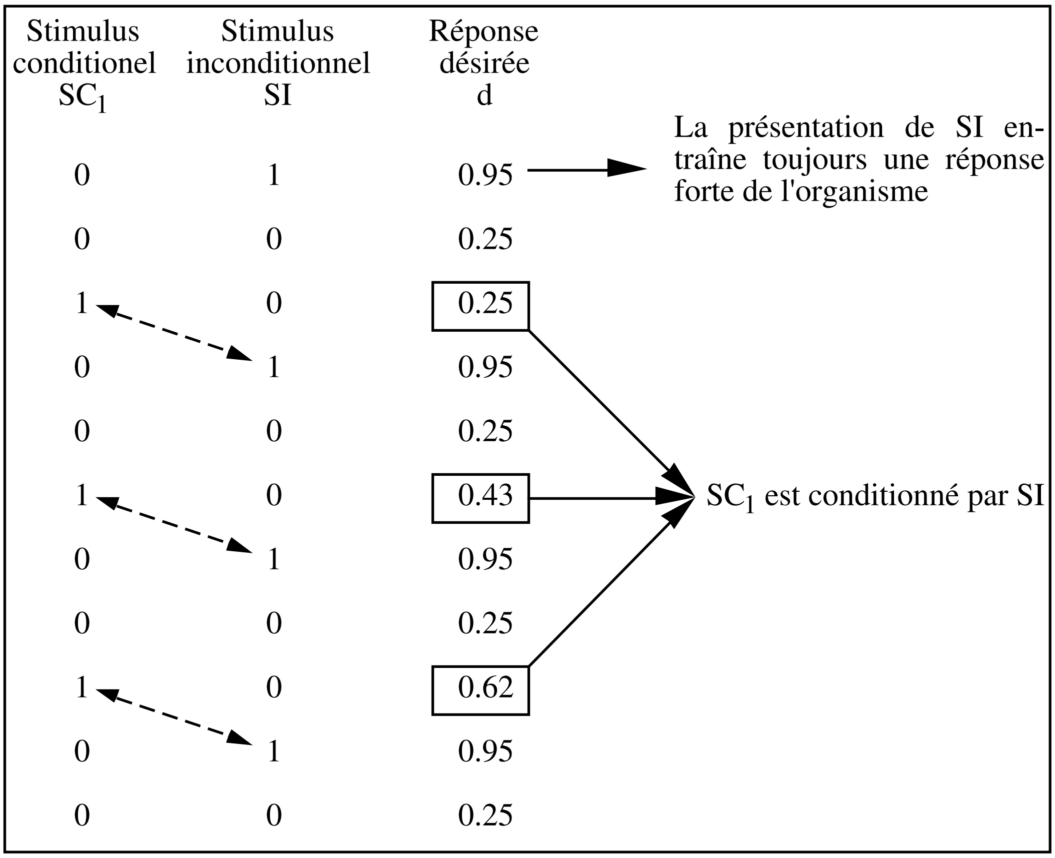

Figure 6.2: Représentation de la tâche de conditionnement par une séquence pour chaque stimuli (dans ce cas, SC_1 et SI). La vraie séquence utilisée pour les expériences est bien plus longue et plus graduelle, car il faut plus de trois présentations de SC_1 suivi de SI pour conditionner SC_1.

Ainsi, on peut y voir qu'à mesure que la séquence progresse, la réponse désirée lors de la présentation du stimulus conditionnel SC_1 augmente (passant de 0.25 à 0.43 à 0.62), tel que le stipule l'observation chez l'animal.

6.1.4 Méthode d'optimisation utilisée

Pour cette expérience, seule la descente du gradient fut utilisée. De plus, le réseau de neurones étant assez petit, la méthode décrite à la section 5.1.1 pour calculer le gradient par rétropropagation de l'erreur à travers un réseau de neurones élargi a été choisie.

Le coût minimisé est alors celui utilisé dans l'algorithme de la retropropagation de l'erreur: *le critère des moindres carrés.* On fournit au réseau une séquence de tuples (SC_1, SC_2, SI, d) où d représente la sortie désirée, soit la réponse de l'animal à la présentation d'un ou plusieurs des stimuli SC_1, SC_2, et SI (voir figure 6.2). Ainsi, pour obtenir une règle d'apprentissage pour résoudre la tâche de conditionnement, le réseau de neurones élargi (intégrant le réseau de la figure 6.1 et celui de la figure 5.3) minimise alors le coût suivant:

$$E_{cond} = \frac{1}{2} \sum_{cond} (d_{NM} - y_{NM})^2 \tag{6.2}$$

où d_{NM} représente la réponse désirée pour un tuple donné, et y_{NM} la réponse obtenue au neurone NM.

Pour trouver une règle d'apprentissage capable de résoudre les 5 tâches de conditionnement classique, le réseau de neurones doit alors être entraîné simultanément sur les 5 séquences correspondantes[1]. Le coût à minimiser est donc en fait la somme des coûts des 5 séquences:

$$E_{global} = E_{habituation} + E_{cond} + E_{blocage} + E_{cond2^e ordre} + E_{extinction} \tag{6.3}$$

[1]Le réseau étant récurrent, il est entraîné par rétropropagation de l'erreur dans le temps [51].

De plus, une règle pourrait être jugée satisfaisante même si le coût obtenu (6.3) n'atteint pas 0. C'est en effet l'aspect qualitatif des courbes que nous tentons de reproduire. Les phénomènes de conditionnement observés d'un animal à l'autre varient généralement quant à la durée du phénomène. Ainsi, le temps de conditionnement devient moins important que le simple fait que l'animal ait effectivement été conditionné.

6.1.5 Résultats

En initialisant les paramètres de la règle d'apprentissage (les θ_i) à des valeurs aléatoires dans l'intervalle $[-1, 1]$, puis en appliquant l'algorithme de la rétropropagation de l'erreur pour effectuer une descente du gradient sur les paramètres de la règle, nous avons pu trouver un ensemble θ tel que les 5 phénomènes de conditionnement puissent être appris avec une règle de la forme de (6.1) et un réseau de neurones tel que celui de la figure 6.1 initialisé avec des poids aléatoires (dans l'intervalle $[-1, 1]$). La règle trouvée est la suivante:

$$\begin{aligned} \Delta w(i,j) \quad = \quad & 0.098 + 0.019\ y(i) - 0.089\ x(j) + 0.197\ y(mod(j)) + \\ & 0.293\ y(j)\ y(mod(j)) + 0.527\ y(i)\ x(j) - 0.419\ y(i)\ w(i,j) \end{aligned} \tag{6.4}$$

On remarque que les poids les plus élevés (en valeur absolue) sont ceux modulant les groupes inspirés de Hebb, Hawkins et Gluck. Une analyse complète de ce résultat n'est cependant pas facile et pourrait être réalisée dans des travaux futurs.

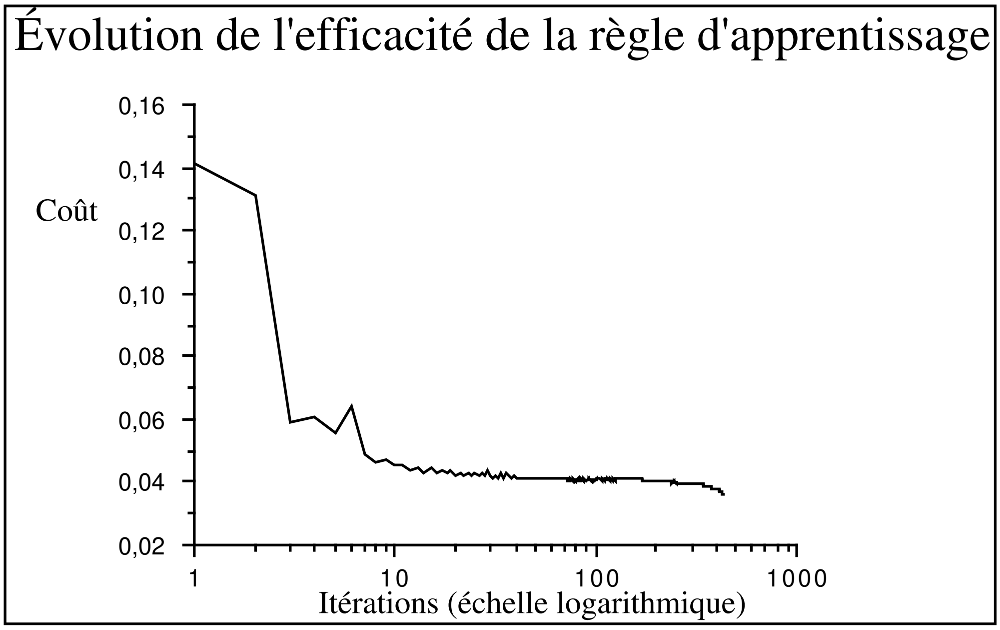

Figure 6.3: Évolution de l'efficacité de la règle d'apprentissage pendant l'optimisation de ses paramètres.

La figure 6.3 montre l'évolution du coût (tel que défini par l'équation (6.3)) lors de l'optimisation des paramètres de la règle d'apprentissage. Ainsi, le coût initial est très élevé (n'apparaît pas sur la figure), mais dès les premières itérations, le coût diminue graduellement jusqu'à obtenir une valeur acceptable. Après 850 itérations[2], l'algorithme de la rétropropagation atteint un plateau et le coût ne descend à peu près plus. La figure 6.4 montre les 5 tâches de conditionnement telles qu'apprises par notre nouvelle règle d'apprentissage, comparées à celles observées par Hawkins.

6.1.6 Discussion

On remarque sur la figure 6.4 que les courbes désirées et obtenues ne sont pas toujours superposées (surtout la courbe du conditionnement), mais comme

[2]Quelques minutes de temps calcul sur un Sparc 2.

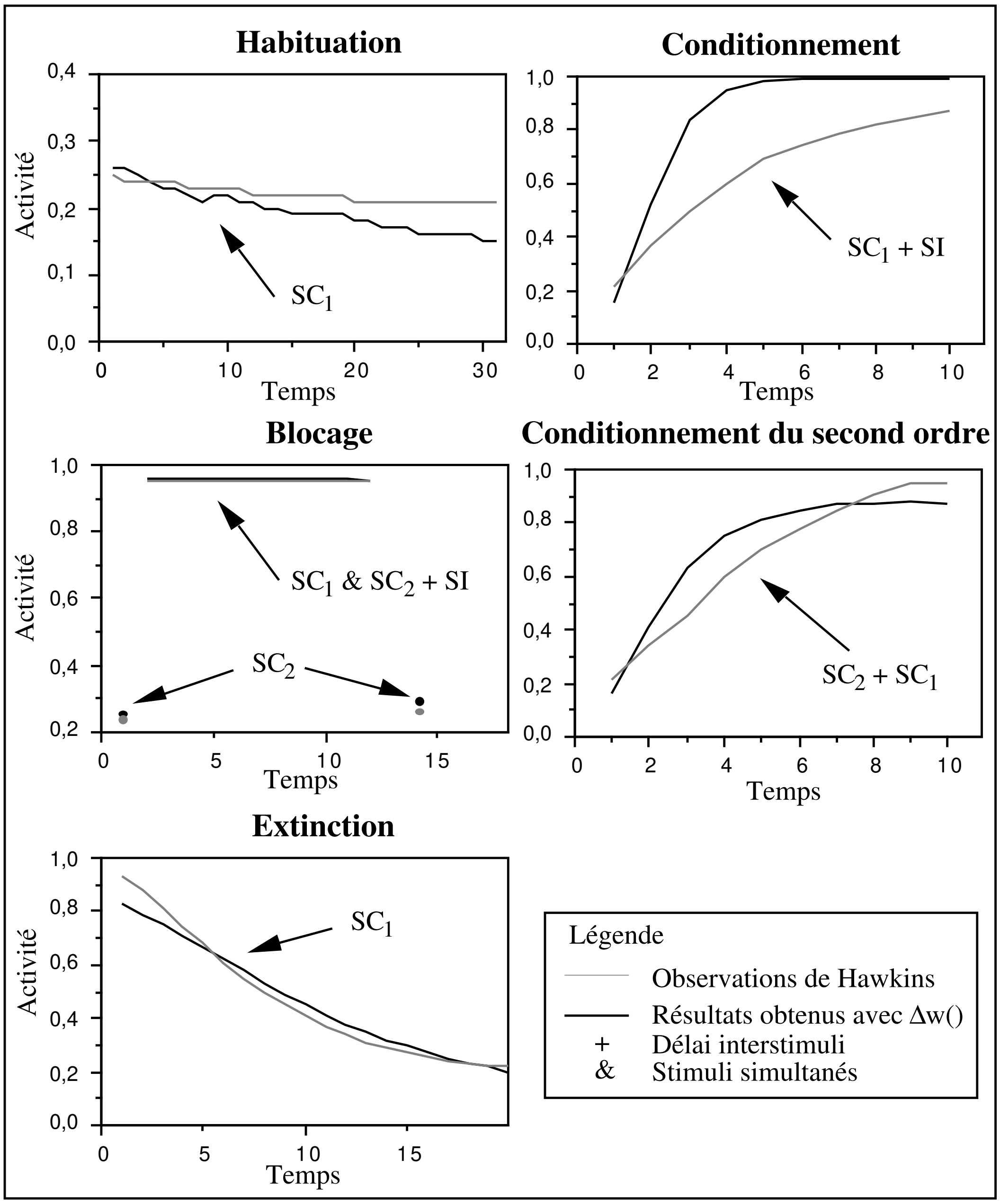

Figure 6.4: Comparaison des résultats obtenus par la règle d'apprentissage et du comportement désiré, basé sur des résultats qualitatifs fournis par Hawkins sur l'Aplysia. Dans chaque graphique, l'ordonnée représente l'activité du neurone de sortie NM, alors que l'abscisse représente la durée temporelle nécessaire pour observer le phénomène de conditionnement.

nous l'avons déjà mentionné, le principal est de tenir compte du fait que lorsque la réponse doit augmenter dans le temps, elle le fait, lorsqu'elle doit diminuer, elle diminue, et lorsqu'elle doit rester stationnaire (pour le blocage), elle le reste. C'est donc dans ses grandes lignes que notre règle d'apprentissage est capable d'apprendre les 5 phénomènes de conditionnement. Rappelons que d'un animal à l'autre, les séquences temporelles sont plus ou moins longues, ce qui peut expliquer que notre règle se comporte de façon légèrement différente des observations de Hawkins. Enfin, rappelons que la règle trouvée est biologiquement plausible (même si la méthode utilisée pour la trouver, à savoir la rétropropagation de l'erreur, ne l'est pas).

6.2 Tâches booléennes

La deuxième série d'expériences, sur des tâches (ou fonctions) booléennes à deux variables, a pour but de vérifier s'il est possible, à l'intérieur d'un cadre simplifié, de trouver une règle d'apprentissage pour réseaux de neurones avec couches cachées. On cherche ainsi une règle d'apprentissage qui aborde le problème du *credit assignment* consistant à attribuer aux connexions des couches inférieures leur part de l'erreur globale. Rappelons que la solution la plus connue à ce problème est la règle de la rétropropagation de l'erreur.

6.2.1 Description du problème

Il y a en tout 16 fonctions booléennes à deux variables différentes. De ce nombre, 14 sont linéairement séparables, et 2 sont linéairement non séparables. À cause de ces dernières, et suivant les résultats de Minsky et Papert ([45]), le réseau de neurones minimal capable de résoudre l'ensemble des 16 fonctions booléennes doit posséder au moins un neurone caché. La figure 6.5 montre l'architecture d'un tel réseau.

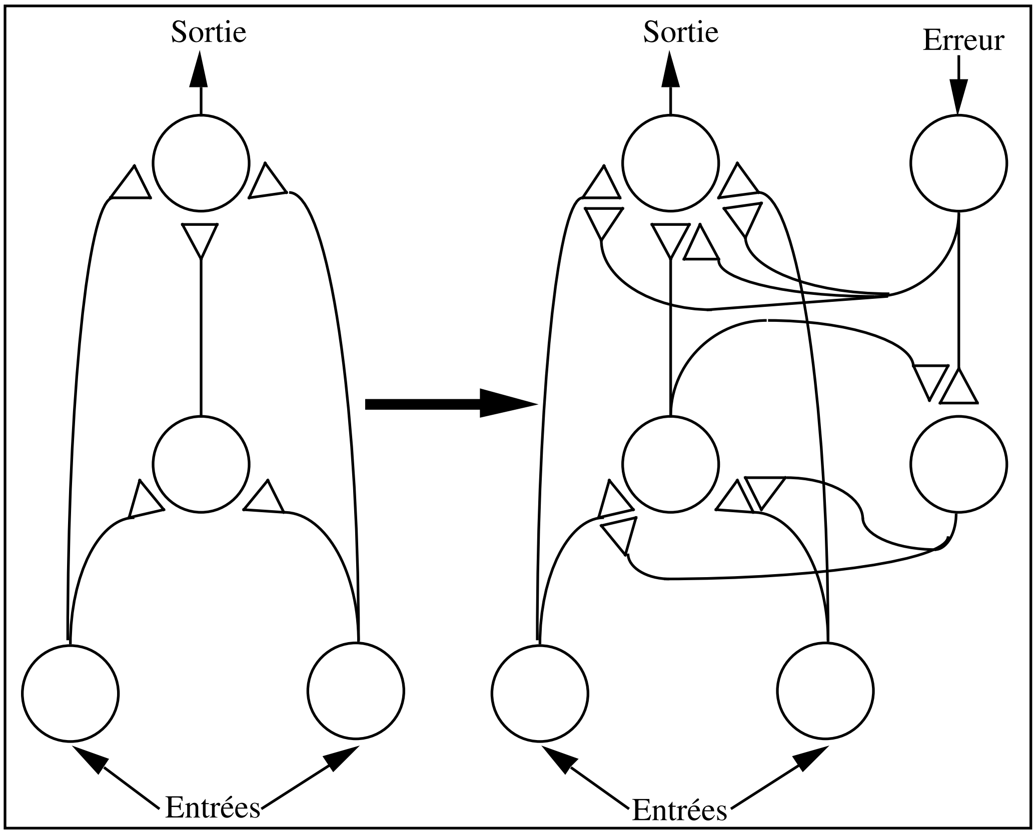

Figure 6.5: Architecture d'un réseau de neurones pouvant résoudre toutes les tâches booléennes, ainsi que sa transformation pour inclure une notion locale de l'erreur par le biais de neurones modulateurs.

Afin de n'utiliser que des informations locales à chaque synapse pour calculer le changement de poids, et comme le montre la figure 6.5, l'architecture est modifiée pour ajouter un chemin de neurones allant dans le sens inverse, et pouvant fournir à chaque connexion, par le biais de neurones modulateurs, une notion de l'erreur globale du réseau.

6.2.2 Forme de la règle d'apprentissage

La modification de la structure du réseau présentée à la figure 6.5 permet l'utilisation de la même forme de règle d'apprentissage que celle utilisée pour les expériences de conditionnement et décrite à l'équation (3.2). Le but de ces expériences est en effet de vérifier s'il est possible de résoudre des tâches relativement difficiles (nécessitant une couche cachée) en utilisant une règle d'apprentissage biologiquement plausible, et construite à l'aide de connaissances biologiques afin de réduire adéquatement la capacité de la règle.

6.2.3 Représentation des données

Toute fonction booléenne à deux variables peut s'exprimer à l'aide de quatre vecteurs. Ainsi, le tableau 6.1 donne un exemple de la fonction *OU*.

A	B	$A \vee B$
0	1	1
0	0	0
1	0	1
1	1	1

Tableau 6.1: Fonction booléenne *OU*: $A \vee B$.

On sait cependant que pour qu'un réseau de neurones *apprenne*, c'est-à-dire trouve les poids solutionnant les quatre vecteurs, il a besoin d'un ensemble d'apprentissage bien plus grand. Pour résoudre ce problème, on peut par exemple répéter plusieurs fois chacun des quatre vecteurs. Une autre solution employée dans la littérature, et permettant d'accélérer l'apprentissage et d'améliorer la généralisation consiste plutôt à introduire du bruit dans les vecteurs d'entrée. Ainsi, au lieu de présenter le vecteur d'entrée $(0, 1)$, on pourra présenter un grand nombre de vecteurs $(0 + X_0, 1 + X_1)$ où X_0 et X_1 sont des variables aléatoires normales centrées à 0 et dont la variance influence la quantité de bruit que l'on désire injecter dans le système. On s'assurera cependant que le bruit n'influence jamais le résultat de la fonction.

Pour déterminer la taille de l'ensemble d'apprentissage, nous avons utilisé le nombre de vecteurs nécessaires pour trouver des poids optimaux avec la règle de la rétropropagation de l'erreur. Bien que cette taille varie selon les paramètres d'apprentissage utilisée, elle s'établie en moyenne à 800. Notre ensemble d'apprentissage consistait donc, pour chaque fonction booléenne à apprendre, par la présentation de 800 vecteurs bruités représentant les 4 vecteurs de base de la fonction. De plus, le nombre de fonctions booléennes utilisées pour l'optimisation des paramètres de la règle peut varier et affecte comme nous allons le constater la qualité de la règle engendrée.

6.2.4 Méthodes d'optimisation utilisées

Deux méthodes d'optimisation ont été utilisées pour trouver une règle d'apprentissage capable de résoudre tous les problèmes booléens avec l'architecture décrite à la figure 6.5: la descente du gradient et le recuit simulé. De plus, pour une meilleure comparaison des résultats obtenus et pour vérifier que des méthodes d'optimisation étaient vraiment nécessaires pour trouver une bonne solution, une simple recherche aléatoire dans l'espace des solutions a aussi été tentée.

Pour chaque méthode, le coût à minimiser est le critère des moindres carrés, calculé sur l'ensemble d'entraînement de chaque réseau de neurones:

$$E_{bool} = \frac{1}{2} \sum_{i \in fonctions} \sum_{j \in vecteurs(i)} (d_{i,j} - y_{i,j})^2 \tag{6.5}$$

où $d_{i,j}$ représente la réponse désirée pour le vecteur d'entraînement j du réseau de neurones chargé d'apprendre la fonction booléenne i, et $y_{i,j}$ est la réponse effectivement obtenue par le réseau.

6.2.5 Résultats

Plusieurs expériences ont été réalisées, variant tantôt la méthode d'optimisation, tantôt le nombre de fonctions booléennes utilisées pendant la phase d'optimisation, ainsi que la nature des fonctions.

Pour chaque expérience, on choisit une méthode d'optimisation (dans ce cas le recuit simulé ou la descente du gradient) et les fonctions booléennes utilisées pour l'optimisation des paramètres de la règle d'apprentissage. On choisit ensuite aléatoirement dans l'intervalle $[-1, 1]$ un ensemble de paramètres initiaux de la règle d'apprentissage, et on lance la procédure d'optimisation.

Chaque itération de la procédure d'optimisation suit alors les étapes suivantes:

1. l'initialisation aléatoire des poids des réseaux de neurones dans l'intervalle $[-1, 1]$,
2. la présentation, pour chaque réseau de neurones de l'ensemble des 800 vecteurs d'apprentissage correspondant à la fonction devant être apprise par le réseau,
3. la modification des paramètres de la règle selon la méthode d'optimisation utilisée.

Lorsque plusieurs tâches booléennes sont utilisées pour l'optimisation de la règle, les étapes 1 et 2 sont alors répétées pour chaque tâche.

Le tableau 6.2 décrit les résultats obtenus pour l'ensemble des expériences réalisées. La colonne *nombre d'étapes nécessaires* représente le nombre d'étapes d'optimisation nécessaires pour trouver une règle d'apprentissage capable de résoudre l'ensemble des tâches soumises pendant la phase d'optimisation. La colonne *généralisation* stipule quant à elle si la règle ainsi trouvée peut résoudre l'ensemble des tâches linéairement séparables (LS) ainsi que l'ensemble des tâches linéairement non séparables (LNS).

nombre de tâches	type de tâches	nombre d'étapes nécessaires	généralisation (nouvelles tâches) LS	LNS	méthode utilisée	sensible à l'initialisation
1	LS	3	oui	non	descente du gradient	oui
1	LNS	15	oui	non		
4	LS	5	oui	non		
5	4LS, 1LNS	100	oui	oui		
1	LS	100	oui	non	recuit simulé	non
1	LNS	1000	oui	non		
5	4LS, 1LNS	24000	oui	oui		

Tableau 6.2: Sommaire des résultats obtenus avec les expériences sur des fonctions booléennes. Dans ce tableau, LS signifie une tâche linéairement séparable, alors que LNS signifie une tâche linéairement non séparable.

6.2.6 Discussion

Première constatation en regardant le tableau 6.2: il est effectivement possible de trouver une règle d'apprentissage capable de résoudre des tâches linéairement non séparables, règle donc meilleure que les premières règles d'apprentissage des années 60 critiquées par Minsky et Papert. La meilleure règle trouvée (par recuit simulé) est la suivante:

$$\begin{aligned}\Delta w(i,j) &= -0.998 + 0.866\ y(i) + 0.493\ x(j) + 0.801\ y(mod(j)) + \\ &\quad 1.464\ y(j)\ y(mod(j)) - 0.197\ y(i)\ x(j) + 0.143\ y(i)\ w(i,j)\end{aligned} \tag{6.6}$$

On remarque notamment que le terme modulant $y(j)\ y(mod(j))$ est plus élevé que les autres. Une analyse plus complète pourrait être réalisée dans des travaux futurs.

De plus, on remarque que plus le nombre de tâches utilisées pour l'optimisation de la règle augmente, plus le nombre d'étapes pour trouver les bons paramètres augmente, et meilleure est la généralisation de la règle sur les autres tâches de même complexité ou de complexité inférieure. Aussi, la règle ne généralise pas sur des fonctions plus complexes que celles utilisées pendant l'apprentissage[3].

Enfin, une première comparaison entre deux méthodes d'optimisation nous montre premièrement que les deux méthodes sont capables de trouver des règles d'apprentissage satisfaisantes, mais que la descente du gradient est bien plus rapide que le recuit simulé[4]. Cependant, il a fallu reprendre plusieurs fois les expériences utilisant la descente du gradient, cette dernière étant très sensible au minima locaux, et donc à l'initialisation des paramètres de la règle d'apprentissage.

Pour vérifier qu'il était nécessaire d'utiliser une méthode d'optimisation pour trouver un ensemble satisfaisant de paramètres pour une règle d'apprentissage, nous avons aussi essayé une recherche aléatoire dans l'espace des solutions. Après 50000 tentatives aléatoires dans $[-1,1]^7$, la meilleure solution trouvée par cette méthode n'était capable de résoudre aucune tâche booléenne à deux variables et linéairement non séparable. Il en ressort donc que l'espace des règles d'apprentissage est trop large pour être exploré par une

[3]Des expériences plus poussées sur la généralisation des règles d'apprentissage paramétriques sont présentées à la section 6.3.

[4]Les expériences avec la descente du gradient s'effectuaient en moins d'une heure alors que celles avec le recuit simulé prenaient souvent près d'une journée de temps calcul sur un Sparc 2.

simple recherche aléatoire, alors que le nombre de règles pouvant donner lieu à un apprentissage est probablement très petit. Il est donc nécessaire d'utiliser des méthodes d'optimisation sophistiquées. De plus comme nous le verrons dans la prochaine section, cet espace semble non lisse et truffé de minima locaux.

6.3 Tâches de classification

La troisième série d'expériences, réalisée sur un ensemble de tâches de classification bi-dimensionnelle, a pour buts d'une part de comparer diverses méthodes d'optimisation appliquées au problème de la recherche d'une règle d'apprentissage, et d'autre part de vérifier expérimentalement certains aspects de la théorie sur la capacité des règles d'apprentissage paramétriques, telle que présentée au chapitre 4.

6.3.1 Description du problème

Le problème général de classification ([19]) consiste à établir une correspondance entre un ensemble de vecteurs $\mathcal{V}$ et un ensemble de classes $\mathcal{C}$. On cherche donc une fonction $f : \mathcal{V} \to \mathcal{C}$, où $\mathcal{V} \subseteq \mathbb{R}^n$ est l'ensemble des vecteurs à n dimensions à classifier, et $\mathcal{C}$ est l'ensemble des classes possibles. Plusieurs problèmes peuvent être exprimés sous cette forme. Ainsi, celui de la reconnaissance de caractères consiste à déterminer à quelle lettre (la classe) de l'alphabet ($\mathcal{C}$) correspond un ensemble de points sur une matrice (un vecteur de $\mathcal{V}$).

En simplifiant le problème pour des vecteurs de deux dimensions à séparer en deux classes distinctes, on peut l'exprimer formellement comme suit. Soit l'ensemble des classes $\mathcal{C} = \{C_0, C_1\}$ et soient les deux ensembles de vecteurs $\mathcal{V}_0 = \{v \in \mathbb{R}^2 | v \text{ est un vecteur de la classe } C_0\}$ et $\mathcal{V}_1 = \{v \in \mathbb{R}^2 | v \text{ est un vecteur de la classe } C_1\}$. Alors, la tâche de classification consiste, après apprentissage, à déterminer adéquatement la classe $C_i \in \mathcal{C}$ (où $i \in \{0, 1\}$) d'un vecteur quelconque $v \in (\mathcal{V}_0 \cup \mathcal{V}_1)$[5].

On peut résoudre ce problème en entraînant un réseau de neurones à l'aide d'une règle d'apprentissage, par la présentation successive et supervisée de vecteurs $v \in (\mathcal{V}_0 \cup \mathcal{V}_1)$.

Les figures 6.6 et 6.7 montrent deux exemples de tâches de classification bi-dimensionnelle. Le premier est un exemple de tâche linéairement séparable (qui peut donc être résolu avec un réseau de neurones sans couche cachée), et le second un exemple de tâche linéairement non séparable (et donc plus difficile que le premier et nécessitant au moins une couche cachée de neurones).

6.3.2 Méthodes d'optimisation utilisées

Un des buts de cette série d'expériences étant de comparer différentes méthodes d'optimisation entre elles, toutes les méthodes décrites au chapitre 5 furent essayées: la descente du gradient, le recuit simulé, les algorithmes génétiques ainsi que la programmation génétique.

[5]Il est possible qu'un vecteur donné v appartienne à la fois à C_0 et C_1. En d'autres termes, il est possible que $C_0 \cap C_1 \neq \emptyset$. En ce cas, il devient évidemment impossible de résoudre la tâche exactement. Il s'agit alors plutôt de minimiser l'erreur.

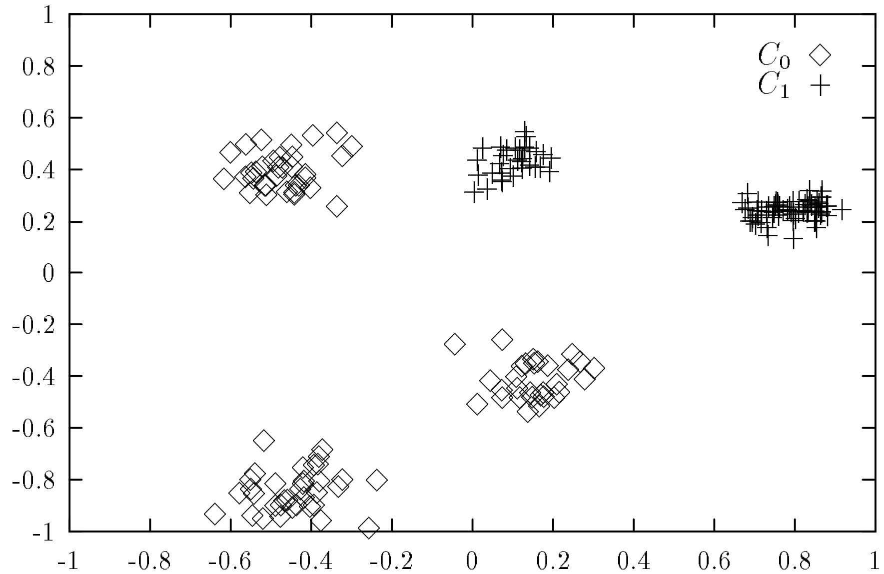

Figure 6.6: Exemple de tâche de classification linéairement séparable. On peut tracer une ligne entre les $\diamond$ et les $+$.

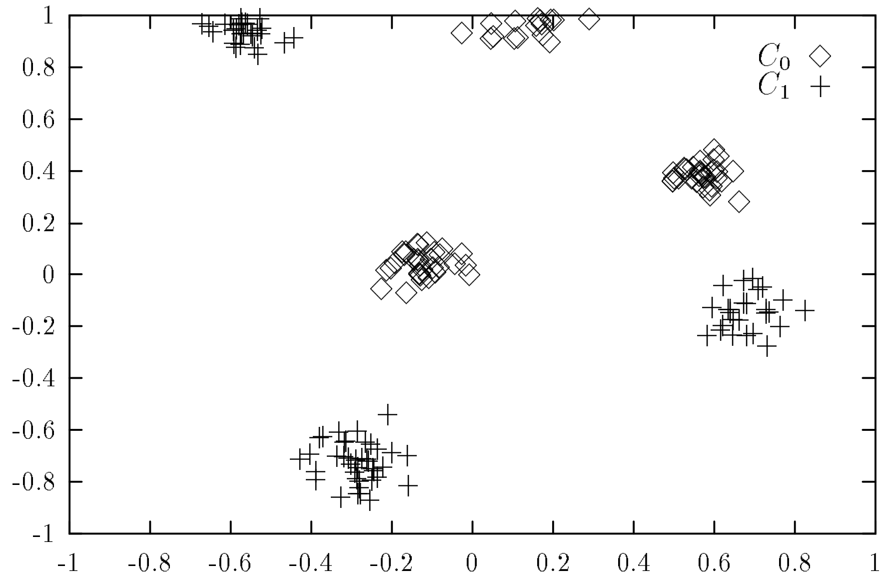

Figure 6.7: Exemple de tâche de classification linéairement non séparable. On ne peut tracer de ligne séparant les $\diamond$ des $+$.

Pour chaque méthode, le nombre maximum d'itérations[6] fut fixé à 50000, sauf pour la programmation génétique qui, prenant plus de temps pour l'éva-

[6]Une itération équivaut à l'évaluation d'une solution. Ainsi, avec les méthodes génétiques, une itération correspond à un individu et non à une génération.

luation d'un individu[7], fut limitée à 10000 itérations. Les solutions initiales étaient toujours choisies aléatoirement.

Pour les algorithmes génétiques, une solution (un ensemble de paramètres) est codé comme une séquence de bits, et les opérateurs de mutation et croisement utilisés sont ceux décrits à la section 5.3. Pour la programmation génétique, une solution est codée sous la forme d'un arbre représentant la fonction, ou les noeud sont des opérateurs (dans ce cas-ci, l'addition et la multiplication), et les feuilles sont des paramètres (nombre réel) ou des variables. De plus, la profondeur des arbres générés est limitée afin d'empêcher la création d'arbres trop grands. Pour les deux méthodes génétiques, aucune mise au point particulière n'a été effectuée.

Le coût E à minimiser par ces méthodes représente le pourcentage moyen de vecteurs mal classifiés sur l'ensemble des tâches:

$$E = \frac{\displaystyle\sum_{i \in \mathcal{T}} \left(\frac{\displaystyle\sum_{j \in v(i)} \delta_{s_{i,j}}^{d_{i,j}}}{|v(i)|} \right)}{|\mathcal{T}|} \tag{6.7}$$

où $\mathcal{T}$ est l'ensemble des tâches sur lequel on calcule l'erreur, $v(i)$ est l'ensemble des vecteurs définissant la tâche i, $s_{i,j}$ est la classe (C_0 ou C_1) obtenue par la règle sur le vecteur j de la tâche i, alors que $d_{i,j}$ est la classe désirée pour le même vecteur. Enfin, $\delta_{s_{i,j}}^{d_{i,j}}$ vaut 1 lorsque $s_{i,j} = d_{i,j}$, et vaut 0 autrement[8].

Ce coût fut préféré à celui des moindres carrés parce qu'il donne une

[7] L'implantation en LISP plutôt qu'en C augmente le temps de calcul.

[8] En pratique, $s_{i,j}$ et $d_{i,j}$ étant deux nombres réels, on ne vérifie pas l'égalité stricte mais simplement si la différence est plus petite qu'un seuil donné.

meilleure idée de la performance de la règle lorsqu'appliqué à des tâches de classification (par opposition aux tâches de régression où l'on doit utiliser les moindres carrés)[9].

6.3.3 Forme de la règle d'apprentissage

Les deux règles d'apprentissage paramétriques utilisées sont celles décrites au chapitre 3 par les équations (3.2) et (3.3).

Pour les expériences utilisant la programmation génétique comme méthode d'optimisation, il n'est pas nécessaire de fournir une forme fixe de règle d'apprentissage; il suffit de choisir les variables locales pouvant affecter le changement de poids, ainsi que les opérateurs pouvant être utilisés par la règle. Nous avons choisi les mêmes variables que pour les règles décrites par les équations (3.2) et (3.3), et les seuls opérateurs utilisés furent l'addition et la multiplication. La seule contrainte supplémentaire fut de borner supérieurement la capacité du système, c'est-à-dire le nombre maximum de paramètres permis pour la construction de nouvelles règles. Cette borne fut fixée à 40, mais comme nous le verrons plus loin, les meilleures règles trouvées utilisaient moins de 10 paramètres.

[9]Cependant, cette fonction de coût n'étant pas différentiable, elle ne peut être utilisée avec la descente du gradient. Dans ce cas, on utilise alors le le critère des moindres carrés. Cependant, comme nous le verrons ultérieurement, les expériences utilisant la descente du gradient n'ayant donné aucun résultat concluant, on s'en tiendra exclusivement au coût décrit par l'équation (6.7).

6.3.4 Résultats

Plusieurs expériences ont été réalisées et sont présentées dans leur intégralité à l'annexe C. Dans cette section nous illustrons plutôt, à l'aide de quelques uns des résultats obtenus, les conséquences les plus importantes. Nous vérifions principalement si les expériences se conforment à l'équation (4.1) du chapitre 4.

La première expérience consistait à vérifier si le nombre de tâches utilisées pour l'optimisation des paramètres influençait réellement l'erreur de généralisation de la règle. La figure 6.8 résume plusieurs expériences utilisant la même méthode d'optimisation et le même type de tâches; seul le nombre de tâches varie. On peut y voir qu'effectivement, pour une capacité donnée et fixe, plus le nombre de tâches augmente, plus l'erreur de généralisation diminue, tel que le prévoit la théorie.

La deuxième expérience consistait à vérifier si le type de tâches utilisées pour l'optimisation des paramètres influence la généralisation de la règle. La figure 6.9 illustre les résultats obtenus. On y remarque que lorsque la règle est optimisée avec des tâches linéairement séparables (LS), l'erreur de généralisation sur des tâches linéairement séparables et non séparables reste assez élevée indépendamment du nombre de tâches utilisées pour l'optimisation des paramètres. Cependant, si l'on utilise plutôt des tâches linéairement non séparables (et donc plus difficiles) pour l'optimisation des paramètres, on remarque que l'erreur de généralisation diminue avec l'augmentation du nombre de tâches, tel que le spécifie l'équation (4.1). Cela suggère d'*utiliser pour l'optimisation de la règle des tâches représentatives de l'ensemble des tâches sur laquelle elle sera utilisée*, de façon à ne pas spécialiser la règle sur des tâches trop simples.

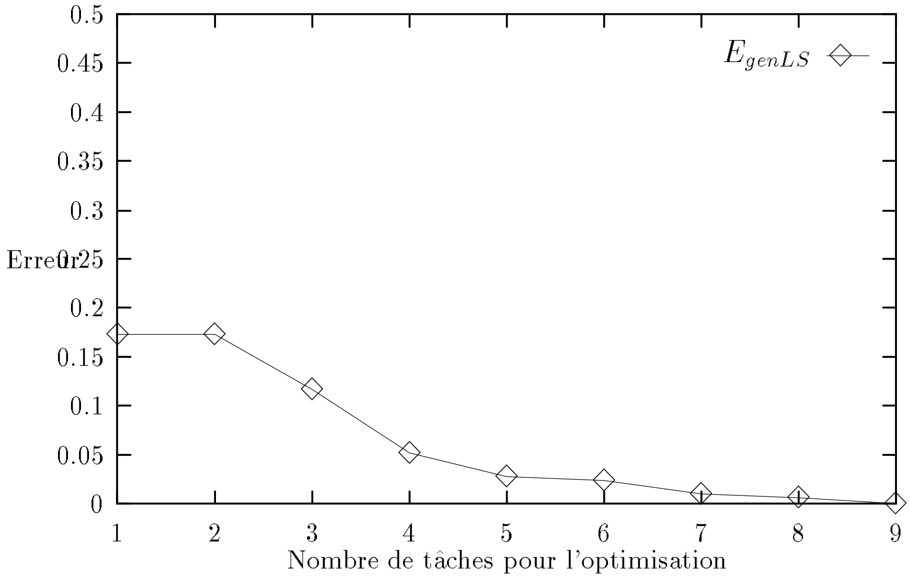

Figure 6.8: Exemple de l'évolution de l'erreur de généralisation (E_{genLS}) en fonction du nombre de tâches utilisées pendant l'optimisation de la règle. Dans cet exemple, on a utilisé les **algorithmes génétiques** comme méthode d'optimisation avec une règle ayant **7 paramètres**. Les tâches sont **linéairement séparables**.

Dans la troisième expérience (figure 6.10), on tente de vérifier si la capacité de la règle d'apprentissage paramétrique influence son erreur de généralisation. Pour cette expérience, deux règles sont comparées; il s'agit des règles décrites par les équations (3.2) et (3.3), qui ont respectivement 7 et 16 paramètres (on peut considérer que la capacité varie en fonction de ce nombre de paramètres). Nous constatons que lorsque le nombre de tâches utilisées pour l'optimisation est trop petit, la règle ayant la plus petite capacité est meilleure que l'autre, mais cet avantage diminue avec l'augmentation du nombre de tâches. Ceci est une fois de plus en accord avec l'équation (4.1). On atteint ensuite un plateau critique, tel que le stipule l'équation (4.1).

Enfin, il peut être intéressant de comparer différentes méthodes d'optimisation entre elles, lorsqu'utilisée pour l'optimisation d'une règle d'apprentissage. La figure 6.11 compare trois méthodes d'optimisation: les algorithmes

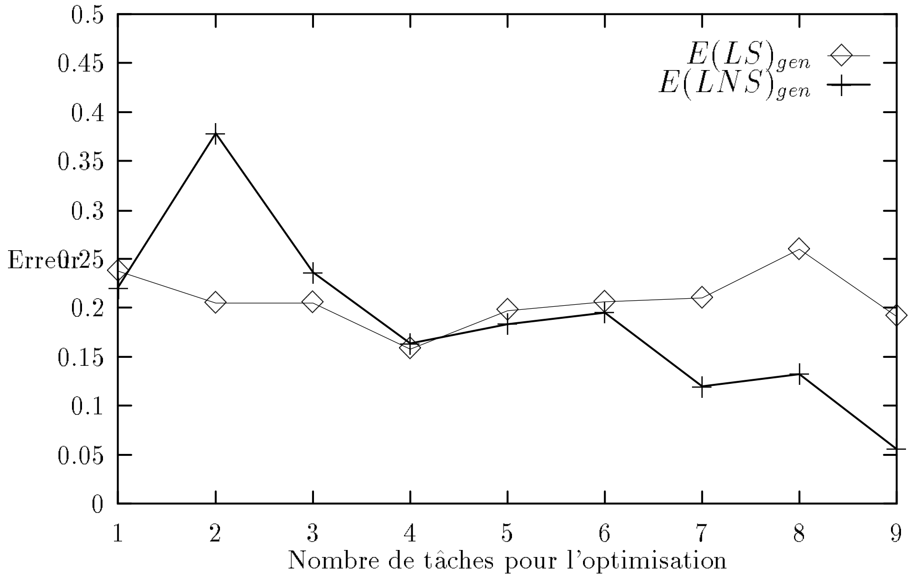

Figure 6.9: Exemple de l'évolution de l'erreur de généralisation en fonction de la difficulté des tâches utilisées pendant l'optimisation de la règle. Dans cet exemple, on a utilisé **les algorithmes génétiques** comme méthode d'optimisation et la règle avait **7 paramètres**. $E(LS)_{gen}$ représente l'erreur de généralisation lorsque la règle est optimisée sur des tâches linéairement séparables, alors que $E(LNS)_{gen}$ représente l'erreur de généralisation lorsque la règle est optimisée sur des tâches linéairement non séparables.

génétiques (**ag**), le recuit simulé (**rs**) et la programmation génétique (**pg**)[10].

On remarque que la méthode d'optimisation n'influence finalement que très peu le résultat final. Notons tout de même que la programmation génétique offre de meilleurs résultats dans l'ensemble (pourtant, cette méthode, implantée en LISP et donc plus lente pour l'évaluation d'un individu, bénéficiait de moins d'itérations pour optimiser les paramètres).

Finalement, la figure 6.12 montre un exemple de l'évolution de l'erreur d'optimisation (et non l'erreur de généralisation) pendant l'optimisation d'une règle d'apprentissage. On y voit qu'au début du processus, l'erreur obtenue sur l'apprentissage des tâches choisies est élevée, mais qu'elle diminue

[10]La descente du gradient fut trop sensible aux minima locaux pour fournir des renseignements suffisamment intéressant pour être consignés.

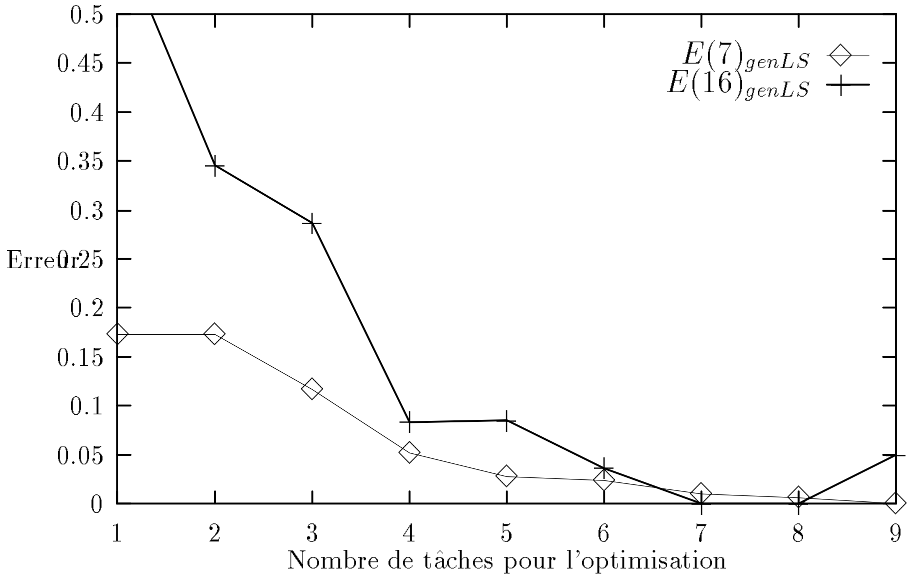

Figure 6.10: Exemple de l'évolution de l'erreur de généralisation en fonction de la capacité de la règle utilisée pour l'optimisation. Dans cet exemple, on a utilisé **les algorithmes génétiques** comme méthode d'optimisation et les tâches étaient **linéairement séparables**. $E(7)_{genLS}$ est l'erreur de généralisation d'une règle ayant 7 paramètres alors que $E(16)_{genLS}$ est l'erreur de généralisation d'une règle ayant 16 paramètres.

rapidement avec le nombre d'itérations.

6.3.5 Inclusion de la rétropropagation de l'erreur dans l'espace des règles

Dans cette section, nous décrivons une nouvelle série d'expériences (aussi détaillée à l'annexe C) qui consistait à modifier l'espace des règles d'apprentissage paramétrique pour y inclure la règle de la rétropropagation de l'erreur. Rappelons que pour toutes les expériences décrites précédemment, nous avions contraint l'espace des règles à un sous-espace de règles biologiquement plausibles (voir la figure 3.2). Cette contrainte éliminait la règle de la rétropropagation de l'erreur de l'espace des règles envisageables par

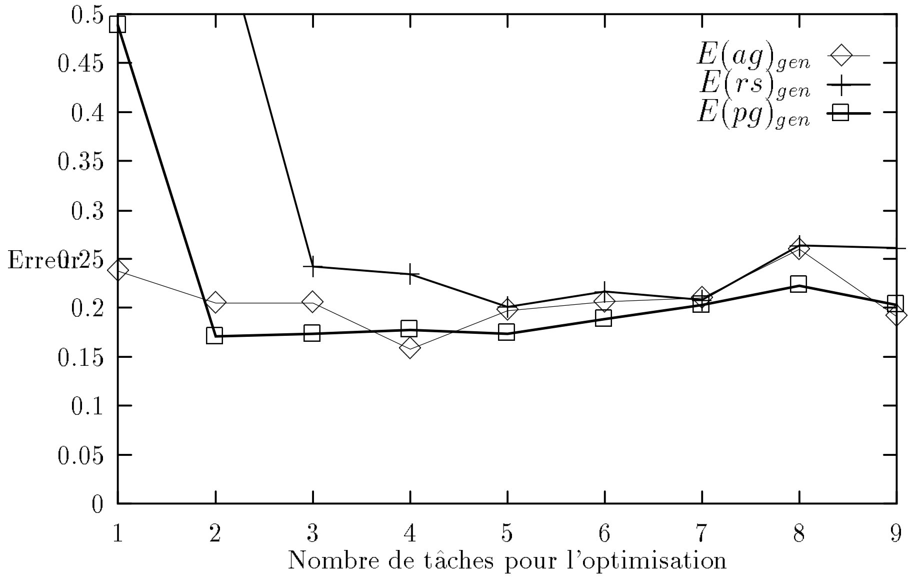

Figure 6.11: Exemple de l'évolution de l'erreur de généralisation en fonction de la méthode d'optimisation utilisée. Dans cet exemple, les tâches sont **linéairement séparables** et la règle utilisée a **7 paramètres**. Ici, $E(ag)_{gen}$ représente l'erreur de généralisation atteinte avec les algorithmes génétiques, $E(rs)_{gen}$ l'erreur de généralisation atteinte avec le recuit simulé, et $E(pg)_{gen}$ l'erreur de généralisation atteinte avec la programmation génétique.

optimisation.

L'annexe B décrit une méthode permettant de construire une classe de règles d'apprentissage qui inclue la règle de la rétropropagation de l'erreur[11].

Le but de ces nouvelles expériences consiste à vérifier si la règle de la rétropropagation de l'erreur est, comme on pourrait le penser, un minimum global dans cet espace de règles d'apprentissage, c'est-à-dire que c'est effectivement *la meilleure de ces règles*, indépendamment du type de tâches et des contraintes de temps.

Nous avons donc comparé les meilleures règles d'apprentissage trouvées par chaque méthode d'optimisation à la règle de la rétropropagation de l'er-

[11]Il existe plusieurs méthodes permettant d'arriver à cette solution; l'annexe B décrit l'une d'elles.

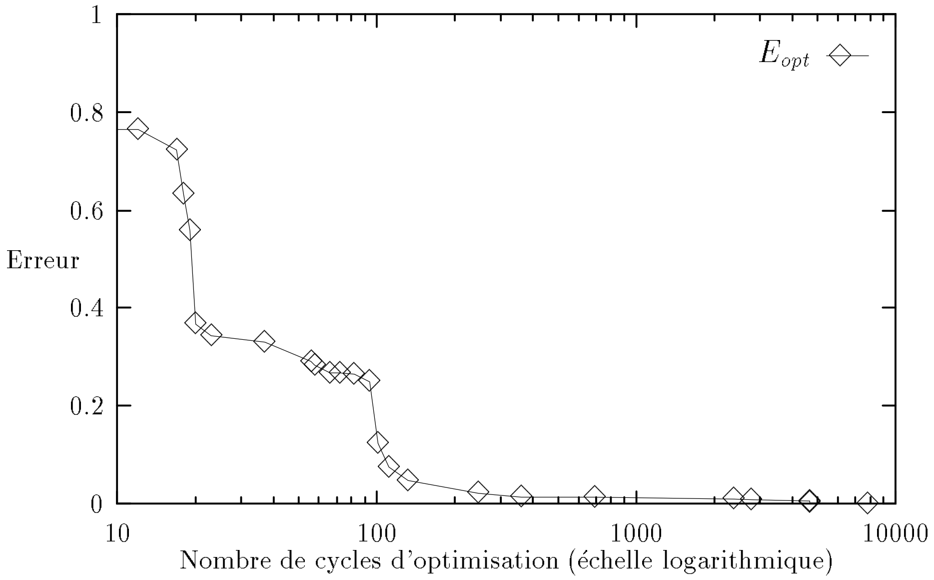

Figure 6.12: Exemple de l'évolution de l'erreur d'optimisation en fonction du nombre d'itérations pendant l'optimisation d'une règle d'apprentissage. Dans cet exemple, la méthode d'optimisation utilisée est le **recuit simulé** et la règle a **7 paramètres**.

reur. Sur la figure 6.13, on montre l'erreur moyenne obtenue sur l'ensemble des tâches (linéairement séparables et non séparables) par la meilleure règle trouvée par chaque combinaison (méthode d'optimisation, forme de la règle). On voit que la règle de la rétropropagation de l'erreur fournit bien sûr un résultat constant (qui ne varie pas en fonction du nombre de tâches utilisées pour l'optimisation de la règle puisqu'elle n'est pas optimisée mais bien fixe), mais elle n'est pas toujours la meilleure. Au contraire, la règle trouvée par programmation génétique dans un espace de règles qui incluait la règle de la rétropropagation de l'erreur est légèrement meilleure.

Cette règle est décrite par l'équation suivante:

$$\Delta w(i,j) = y(i) \cdot y(mod(j)) \cdot [f'(x(j))]^3 \tag{6.8}$$

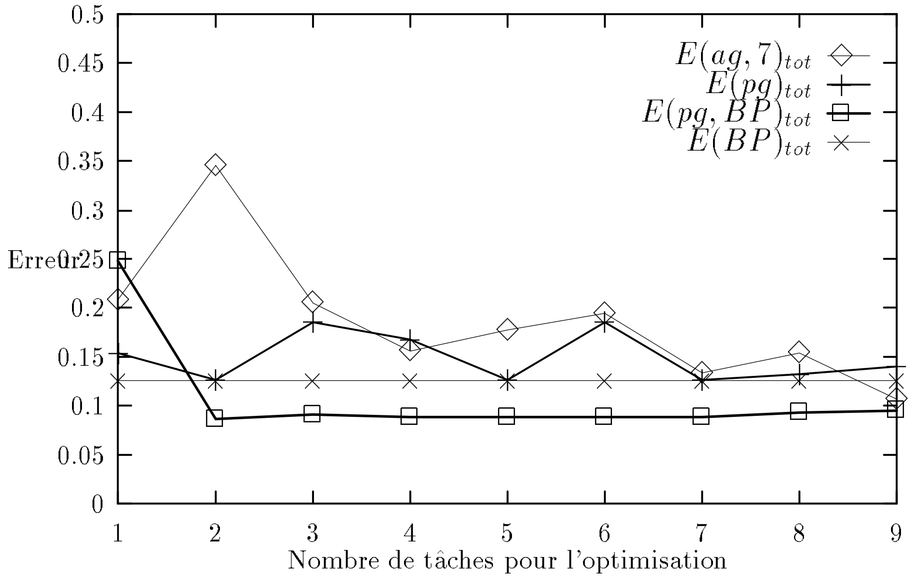

Figure 6.13: Comparaison des meilleures règles d'apprentissage trouvées avec différentes méthodes d'optimisation et rapprochement avec la méthode de la rétropropagation de l'erreur. $E(ag, 7)$ est l'erreur obtenue avec les algorithmes génétiques sur une règle de 7 paramètres, $E(pg)$ est l'erreur obtenue avec la programmation génétique, $E(pg, BP)$ est l'erreur obtenue avec la programmation génétique en incluant la règle de la rétropropagation de l'erreur dans l'espace des règles, et $E(BP)$ est l'erreur obtenue par la règle de la rétropropagation de l'erreur.

alors que la règle de la rétropropagation de l'erreur pourrait être décrite, dans le contexte actuel[12], par:

$$\Delta w(i,j) = y(i) \cdot y(mod(j)) \cdot f'(x(j)) \tag{6.9}$$

Dans les deux cas, $y(i)$ représente l'activité du neurone présynaptique, $y(mod(j))$ représente l'activité du neurone modulant la connexion, et $f'(x(j))$ représente la dérivée de la fonction d'activation du neurone j. La seule différence entre les deux règles tient donc au fait que le terme $f'(x(j))$ soit plus important (puisque mis au cube) dans l'équation (6.8) que dans

[12]Voir l'annexe B pour une explication.

l'équation (6.9).

Par comparaison, la règle obtenue avec les algorithmes génétiques sur une règle à 8 paramètres incluant la règle de la rétropropagation de l'erreur dans l'espace des règles (et qui donne aussi de meilleurs résultats que la règle de la rétropropagation de l'erreur lorsque le nombre de tâches est suffisant) est la suivante:

$$\begin{aligned}\Delta w(i,j) \;=\;& -0.547 + 0.181\; y(i) - 0.605\; x(j) - 0.633\; y(mod(j)) + \\ & 0.311\; y(i)\; y(mod(j)) + 0.032\; y(i)\; x(j) + 0.039\; y(i)\; w(i,j) + \\ & 0.995\; y(i)\; y(mod(j))\; f'(x(j)) \qquad (6.10)\end{aligned}$$

Notons dans cette règle la forte prépondérance du dernier paramètre, modulant le groupe représentant la règle de la rétropropagation de l'erreur. On remarque aussi que certains paramètres sont beaucoup moins influents que d'autres et il est probable que l'on pourrait réduire encore la capacité de notre règle sans trop détériorer sa performance en éliminant par exemple les deux avant-derniers paramètres. Cependant, une interprétation plus précise de la valeur des paramètres semble difficile et pourrait faire l'objet d'un travail subséquent.

Finalement, à titre de vérification supplémentaire, nous avons testé la règle de l'équation (6.8) sur deux autres types de problèmes: des tâches booléennes, et une tâche de reconnaissance de caractères. Pour les tâches booléennes, nous avons utilisé la même architecture de réseau de neurones (voir figure 6.5). La nouvelle règle fut effectivement capable de résoudre toutes les fonctions booléennes, linéairement séparables et non séparables. Ce résultat n'est pas surprenant car l'ensemble des fonctions booléennes est un sous-ensemble des tâches de classification bi-dimensionnelle.

La tâche de reconnaissance de caractères (se limitant aux chiffres seulement) consistait en un problème de classification plus complexe dont l'espace d'entrée avait 7 dimensions et représentait la présence ou l'absence des 7 segments composant l'affichage indiqué à la figure 6.14. L'espace de sortie a quant à lui 10 dimensions, une pour chaque chiffre. Le but du problème consiste donc à déterminer, pour un affichage donné, à quel chiffre il correspond. Le réseau de neurones permettant de résoudre ce problème possède donc 7 neurones d'entrée, 10 neurones de sortie, et un certain nombre de neurones cachés (dans notre cas, nous avons choisi d'en mettre 10). Ainsi, nous avons pu tester notre nouvelle règle d'apprentissage sur un réseau de neurones de taille différente et plus grande que celui utilisé pour l'optimisation des paramètres. L'apprentissage consistait en la présentation de 1000 vecteurs d'entrées, et l'ensemble de test comportait 100 autres vecteurs. Les vecteurs étaient bruités pour faciliter la généralisation. Le résultat est étonnant: la nouvelle règle d'apprentissage a pu résoudre le problème sans aucune erreur de généralisation.

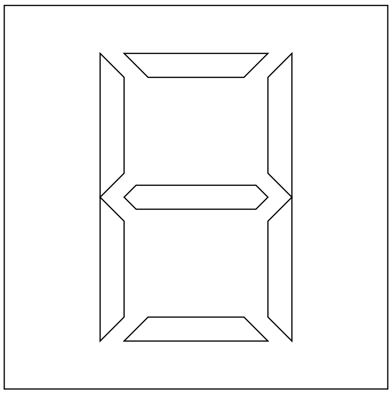

Figure 6.14: Illustration d'un afficheur à 7 segments.

6.3.6 Discussion

À la lumière des résultats obtenus à la section précédente, il est intéressant de souligner les points suivants:

- Les expériences se conforment qualitativement à la théorie sur la capacité des règles d'apprentissage paramétriques.
- L'espace des règles d'apprentissage semble être truffé de minima locaux car il fut impossible d'obtenir des résultats intéressants sur les tâches considérées avec une méthode d'optimisation locale comme la descente du gradient. Pour s'en convaincre, nous avons décidé d'observer graphiquement une partie de l'espace des règles d'apprentissage. Naturellement, nous avons dû choisir, pour ce faire, de nous limiter à observer seulement l'effet de la variation de deux paramètres θ, exprimés en fonction de l'erreur de généralisation (les autres paramètres sont restés fixes aux valeurs prises dans l'équation 6.10). La figure 6.15 montre cet espace. On remarque aisément qu'il est effectivement non lisse et rempli de minima locaux. Dans ces conditions, il n'est donc pas étonnant qu'une méthode d'optimisation locale comme la descente du gradient ne puisse trouver de solution intéressante.
- Il est possible de trouver par optimisation des règles d'apprentissage meilleures que la règle de la rétropropagation de l'erreur, dans un contexte donné et restreint. Cette dernière n'est donc pas toujours le minimum global dans l'espace des règles d'apprentissage.
- Mieux encore, la règle trouvée peut aussi servir pour résoudre des tâches de dimension supérieure (telles que l'affichage à 7 segments). Il sera intéressant dans des travaux futurs d'examiner les limites précises de la nouvelle règle.

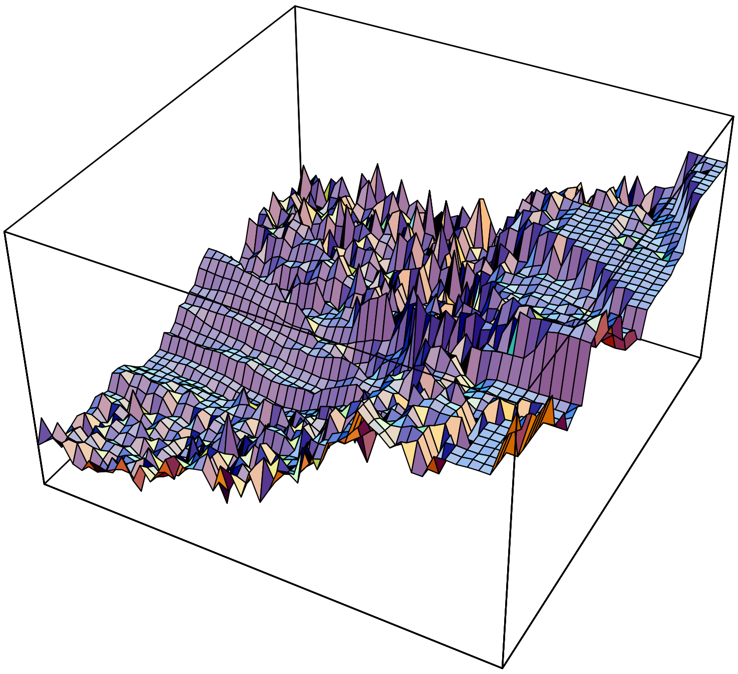

Figure 6.15: Espace des règles d'apprentissage. Dans cette figure, on montre l'erreur de généralisation (axe vertical) en fonction de la valeur de deux des paramètres de la règle décrite à l'équation (6.10). Ces deux paramètres (θ_2 et θ_4) prennent des valeurs comprises entre -1 et 1. Les autres paramètres prennent les valeurs de l'équation (6.10).

- De plus, une analyse plus poussée de l'espace des paramètres des règles d'apprentissage paramétriques devrait nous permettre par exemple de déterminer l'importance relative de chaque paramètre et vérifier si les performance d'une règle donnée sont sensibles à la valeur exacte de tous les paramètres (ce qui semble le cas pour certains). Cela nous mène a penser qu'afin d'éliminer des paramètres inutiles, il serait intéressant d'ajouter une contrainte sur l'envergure de chaque paramètre pendant la phase d'optimisation.

Chapitre 7

Conclusion

Dans cette thèse, nous avons proposé une nouvelle méthode pour la recherche et l'optimisation de la règle d'apprentissage des réseaux de neurones artificiels.

Pour faciliter la recherche, nous avons montré qu'il était préférable de chercher une règle d'apprentissage spécialisée pour la résolution d'une classe restreinte et donnée de tâches, plutôt qu'une règle générale, capable de résoudre n'importe quelle tâche.

De plus, nous avons montré comment la théorie sur la capacité et la généralisation des systèmes d'apprentissage pouvait être étendue et appliquée au problème du design de règles d'apprentissage paramétriques. Cette théorie permet ainsi de relier l'erreur de généralisation que la règle pourra commettre en pire cas sur de nouvelles tâches, en fonction du nombre de tâches utilisées pour l'optimisation de ses paramètres et de la complexité de la règle.

Les expériences réalisées au chapitre 6 indiquent qu'il est effectivement possible de trouver, par optimisation des paramètres, des règles d'apprentissage capables de résoudre des problèmes non-triviaux, tels que la séparation non linéaire.

Une comparaison avec la règle de la rétropropagation de l'erreur, actuellement la règle la plus populaire et la plus performante, a montré qu'il existe néanmoins de meilleures règles d'apprentissage lorsque le contexte (tels que le nombre d'itérations et la classe des tâches) est fixé. Plus encore, une règle trouvée pour un contexte donné s'est avérée aussi efficace pour des tâches plus complexes. Une analyse plus profonde des règles trouvées reste donc à faire.

Cependant, il faut noter que pour découvrir des règles d'apprentissage pouvant résoudre des problèmes de grande taille, le temps de calcul devient rapidement trop grand, dû principalement à la lenteur des méthodes d'optimisation utilisées.

De plus, l'espace des paramètres des règles d'apprentissage étant non lisse, il est important d'utiliser des méthodes d'optimisation qui soient peu sensibles aux minima locaux. Or, la méthode la plus rapide, la descente du gradient, est justement une méthode locale et donc trop sensible aux minima locaux. Il pourrait être intéressant d'utiliser des méthodes du second degré pour contraindre adéquatement l'espace des paramètres [41], et donc accélérer l'apprentissage. La méthode reste cependant sensible aux minima locaux (mais étant plus rapide, on peut envisager de recommencer l'apprentissage plusieurs fois pour se sortir des minima locaux).

Pour contourner le problème de la lenteur des méthodes d'optimisation classique, on peut aussi par exemple réduire la classe des tâches que la règle pourra résoudre (ce qui a pour effet de réduire le nombre de tâches nécessaires à la généralisation de la règle); dans le même ordre d'idée, on peut envisager d'optimiser la règle en utilisant des tâches réduites en terme du nombre d'itérations ou du nombre d'exemples représentant la tâche.

L'expérience a de plus montré que l'utilisation de connaissances a-priori dans le design de la règle d'apprentissage paramétrique était un facteur important. Il sera intéressant ainsi de chercher à développer des règles d'apprentissage spécialisées dans des classes de tâches ou les règles actuelles éprouvent encore de la difficulté, en utilisant le plus de connaissance possible sur ces tâches pour le design de la règle. Enfin, on pourrait aussi envisager une optimisation conjointe de la règle d'apprentissage et de l'architecture du réseau de neurones pour un problème donné. Les mécanismes évolutifs nous suggèrent d'ailleurs un système qui modifierait dans le temps ses propres mécanismes d'apprentissage et sa propre architecture en fonction de l'environnement (c'est-à-dire des problèmes à résoudre).

Finalement, le but de cette thèse n'était pas tellement de *trouver* nécessairement de nouvelles règles d'apprentissage meilleures que les règles actuelles, mais plutôt de suggérer une méthode pour *explorer* l'espace des règles d'apprentissage. De plus, nous avons aussi soulevé l'idée qu'il sera plus facile de trouver une règle d'apprentissage spécialisée pour un ensemble de tâches donné plutôt qu'une règle générale capable d'apprendre n'importe quoi, ce qui devrait, nous l'espérons, guider la recherche vers des solutions spécialisées pour des problèmes particuliers.

Bibliographie

[1] D. H. Ackley, G. E. Hinton, et T. J. Sejnowski, *A learning algorithm for boltzmann machines*, Cognitive Science, 9 (1985), pp. 147–169.

[2] J. A. Anderson, J. W. Silverstein, S. A. Ritz, et R. S. Jones, *Distinctive features, categorical perception, and probability learning: Some applications of a neural model*, Psychological Review, 84 (1977), pp. 413–451.

[3] E. R. Baum, *A proposal for more powerful learning algorithms*, Neural Computation, 1 (1989), pp. 201–207.

[4] E. R. Baum et D. Haussler, *What size network give valid generalization*, Neural Computation, 1 (1989), pp. 151–160.

[5] S. Becker et Y. Le Cun, *Improving the convergence of backpropagation learning with second-order methods*, dans Proceedings of the 1988 Connectionist Model Summer School, D. S. Touretzky, G. E. Hinton, et T. Sejnowski, eds., San Mateo, USA, 1989, Morgan Kaufmann, pp. 29–37.

[6] R. K. Belew, J. McInerney, et N. N. Schraudolph, *Evolving networks: using genetic algorithm with connectionist learning*, Rapp.

Tech. CS90-174, Computer Science and Eng. Department (C-014). University of California at San Diego, La Jolla, CA 92093, USA, 1991.

[7] S. Bengio, Y. Bengio, J. Cloutier, et J. Gecsei, *Aspects théoriques de l'optimisation d'une règle d'apprentissage*, dans Actes de la conférence Neuro-Nimes 1992, Nimes, France, 1992.

[8] ——, *On the optimization of a synaptic learning rule*, dans Conference on Optimality in Biological and Artificial Networks, Dallas, USA, 1992.

[9] ——, *Generalization of a parametric learning rule*, dans ICANN '93: Proceedings of the International Conference on Artificial Neural Networks, Amsterdam, Nederlands, 1993.

[10] Y. Bengio, S. Bengio, et J. Cloutier, *Learning a synaptic learning rule*, dans Proceedings of the International Joint Conference on Neural Networks, Seattle, USA, 1991.

[11] ——, *Learning synaptic learning rules*, dans Neural Networks for Computing, Snowbird, Utah, USA, 1991.

[12] Y. Bengio, R. De Mori, G. Flammia, et R. Kompe, *Global optimization of a neural network-hidden markov model hybrid*, IEEE Transactions on Neural Networks, 3 (1992), pp. 252–259.

[13] A. Blumer, A. Ehrenfeucht, D. Haussler, et M. Warmuth, *Learnability and the vapnik-chervonenkis dimension*, Journal of the Association of Computer Machinery, 36 (1989), pp. 929–965.

[14] L. Bottou, *Une approche théorique de l'apprentissage connexioniste; application à la reconnaissance de la parole*, PhD thesis, Université de Paris Sud, 1991.

[15] J. H. Byrne, *Cellular analysis of associative learning*, Physiological Review, 67 (1987), pp. 329–439.

[16] D. Chalmers, *The evolution of learning: An experiment in genetic connectionism*, dans Proceedings of the 1990 Connectionist Models Summer School, D. Touretzky, J. Elman, T. Sejnowski, et G. Hinton, eds., San Mateo, CA, USA, 1990, Morgan Kaufmann.

[17] G. Cybenko, *Approximation by superposition of sigmoidal functions*, Mathematics of Control, Signal and Systems, 2 (1989), pp. 303–314.

[18] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, NY, USA, 1991.

[19] R. O. Duda et P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley-Interscience, 1973.

[20] S. E. Fahlman et C. Lebiere, *The cascade-correlation learning architecture*, dans Advances in Neural Information Processing Systems 2, D. S. Touretzky, ed., San Mateo, USA, 1990, Morgan Kaufmann, pp. 524–532.

[21] K. I. Funahashi, *On the appropriate realization of continuous mappings by neural networks*, Neural Networks, 2 (1989), pp. 183–192.

[22] D. Gardner, *Synaptic diversity characterizes biological neural networks*, dans Proceedings of the IEEE First International Conference on Neural Networks, vol. IV, San Diego, CA, USA, 1987, pp. 17–22.

[23] M. R. Garey et D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, USA, 1979.

[24] S. Geman et D. Geman, *Stochastic relaxation, gibbs distributions, and the bayesian restoration of images*, dans IEEE Proceedings on Pattern Analysis and Machine Intelligence 6, 1984, pp. 721–741.

[25] M. A. Gluck et R. F. Thompson, *Modeling the neural substrate of associative learning and memory: a computational approach*, Psychological Review, 94 (1987), p. 176.

[26] D. Goldberg, *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, Reading, MA, USA, 1989.

[27] R. D. Hawkins, *A biologically based computational model for several simple forms of learning*, dans Computational Models of Learning in Simple Neural Systems, R. D. Hawkins et G. H. Bower, eds., Academic Press, 1989, pp. 65–108.

[28] R. D. Hawkins, T. W. Abrams, T. J. Carew, et E. R. Kandel, *A cellular mechanism of classical conditioning in aplysia: Activity-dependent amplification of presynaptic facilitation*, Science, 219 (1983), pp. 400–404.

[29] D. O. Hebb, *The Organization of Behavior*, Willey, New York, NY, USA, 1949.

[30] J. Hertz, A. Kroght, et R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wessley, 1991.

[31] G. E. Hinton, *Connectionist learning procedures*, Artificial Intelligence, 40 (1989), pp. 185–234.

[32] J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[33] K. Hornik, M. Stinchcombe, et H. White, *Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks*, Neural Networks, 2 (1989), pp. 359–368.

[34] R. A. Jacobs, *Increased rates of convergence through learning rate adaptation*, Neural Networks, 1 (1988), pp. 295–307.

[35] S. JUDD, *On the complexity of loading shallow neural network*, Journal of Complexity, 4 (1988).

[36] J. R. KOZA, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Bradford Book, MIT Press, Cambridge, MA, USA, 1992.

[37] P. LAARHOVEN ET E. AARTS, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Compagny, 1987.

[38] Y. LE CUN, *A theoretical framework for back-propagation*, dans Proceedings of the 1988 Connectionist Models Summer School, D. Touretzky, G. E. Hinton, et T. Sejnowski, eds., San Mateo, USA, 1988, Morgan Kaufmann.

[39] ——, *Generalization and network design strategies*, dans Connectionism in Perspective, R. Pfeifer, Z. Schreter, F. Fogelman-Soulié, et L. Steels, eds., North-Holland, 1989, pp. 143–156.

[40] Y. LE CUN, B. BOSER, J. S. DENKER, D. HENDERSON, R. E. HOWARD, W. HUBBARD, ET L. D. JACKEL, *Backpropagation applied to handwritten zip code recognition*, Neural Computation, 1 (1989), pp. 541–551.

[41] Y. LE CUN, J. S. DENKER, ET S. A. SOLLA, *Optimal brain damage*, dans Advances in Neural Information Processing Systems 2, D. S. Touretzky, ed., San Mateo, USA, 1990, Morgan Kaufmann, pp. 598–605.

[42] G. LEGENDRE, Y. MIYATA, ET P. SMOLENSKY, *Distributed recursive structure processing*, dans Advances in Neural Information Processing Systems 3, R. P. Lippmann, J. E. Moody, et D. S. Touretzky, eds., San Mateo, USA, 1991, Morgan Kaufmann, pp. 591–599.

[43] D. B. Malkoff, *A neural network for real time signal processing*, dans Advances in Neural Information Processing Systems 2, D. S. Touretzky, ed., San Mateo, USA, 1990, Morgan Kaufmann.

[44] W. McCulloch et W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics, 5 (1943), pp. 115–133.

[45] M. Minsky et S. Papert, *Perceptrons*, MIT Press, Cambridge, MA, USA, 1969.

[46] P. Orponen, *Neural networks and complexity theory*, rapp. tech., Department of Computer Science, University of Helsinki., Helsinki, Finland, 1992.

[47] I. P. Pavlov, *Les Réflexes Conditionels*, Alcan, Paris, 1932.

[48] S. Pinker et A. Prince, *On language and connectionism: Analysis of a parallel distributed processing model of language acquisition*, Cognition, 28 (1988), pp. 73–193.

[49] D. A. Pomerleau, *Alvinn: an autonomous land vehicle in a neural network*, dans Advances in Neural Information Processing Systems 1, D. S. Touretzky, ed., San Mateo, USA, 1989, Morgan Kaufmann, pp. 305–313.

[50] F. Rosenblatt, *Principles of Neurodynamics*, Spartan, New York, 1962.

[51] D. E. Rumelhart, G. E. Hinton, et R. J. Williams, *Learning internal representations by error propagation*, dans Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 1: Foundations, D. E. Rumelhart et J. L. McClelland, eds., Bradford Book, MIT Press, 1986.

[52] D. E. Rumerlhart, G. E. Hinton, et M. J. L., *A general framework for parallel distributed processing*, dans Parallel Distributed Processing: Explorations in the Microstructure of Cognition - Volume 1: Foundations, D. E. Rumerlhart et J. L. McClelland, eds., Bradford Book, MIT Press, 1986.

[53] K. S., J. Gellat, et M. P. Vecchi, *Optimization by simulated annealing*, Science, 20 (1983), pp. 671–680.

[54] P. Smyth et J. Mellstrom, *Fault diagnosis of antenna pointing systems using hybrid neural network and signal processing models*, dans Advances in Neural Information Processing Systems 4, J. E. Moody, S. J. Hanson, et R. P. Lippmann, eds., San Mateo, USA, 1992, Morgan Kaufmann, pp. 667–675.

[55] R. S. Sutton, *Temporal Credit Assignment in Reinforcement Learning*, PhD thesis, University of Massachusetts, Amherst, MA, USA, 1984.

[56] G. Tesauro, *Practical issues in temporal difference learning*, Machine Learning, 8 (1992).

[57] G. Tesauro et T. J. Sejnowski, *A parallel network that learns to play backgammon*, Artificial Intelligence, 39 (1989), pp. 357–390.

[58] G. Towell et J. W. Shavlik, *Interpretation of artificial neural networks: Mapping knowledge-based neural networks into rules*, dans Advances in Neural Information Processing Systems 4, J. E. Moody, S. J. Hanson, et R. P. Lippmann, eds., San Mateo, USA, 1992, Morgan Kaufmann, pp. 977–985.

[59] V. N. Vapnik, *Estimation of Dependencies Based on Empirical Data*, Springer-Verlag, New-York, NY, USA, 1982.

[60] V. N. Vapnik et A. Y. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory of Probability and its Applications, 16 (1971), pp. 264–280.

[61] H. White, *An overview of representation and convergence results for multilayer feedforward networks*, dans Advances in Neural Information Processing systems 3, R. P. Lippman, J. E. Moody, et D. S. Touretzky, eds., San Mateo, USA, 1991, Morgan Kaufmann.

[62] D. Whitley, T. Starkweather, et C. bogart, *Genetic algorithms and neural networks: optimizing connections and connectivity*, Parallel Computing, 14 (1990), pp. 347–361.

[63] X. Yao, *A review of evolutionary artificial neural networks*, International Journal of Intelligent Systems, 8 (1993), pp. 539–567.

[64] M. Zeidenberg, *Neural Networks in Artificial Intelligence*, Ellis Horwood, 1990.

Annexe A

Dérivation de la règle de la rétropropagation de l'erreur

Dans cette annexe, nous montrons comment dériver les formules de la section 2.4 qui dictent le changement de poids d'un réseau de neurones entraîné par la règle de la rétropropagation de l'erreur.

A.1 Notation

Notons $x_e(j)$ l'état d'activation du neurone j à la présentation du vecteur d'entrée e. De la même façon, $y_e(j)$ est la sortie du neurone j, $w_e(i,j)$ le poids de la connexion entre les neurones i et j, et $d_e(j)$ la valeur désirée du neurone de sortie j.

De plus, notons $source(j)$ l'ensemble des neurones connectés au neurone j,

$dest(j)$ l'ensemble des neurones auxquels j se connecte, et *sortie* l'ensemble des neurones de sortie.

A.2 Fonctionnement du système

Rappelons tout d'abord les équations de base régissant le fonctionnement du réseau. La règle de propagation du réseau est la suivante:

$$x_e(j) = \sum_{i \in source(j)} w_e(i,j) \cdot y_e(i) \tag{A.1}$$

La fonction de sortie permet de calculer $y_e(j)$ en fonction de $x_e(j)$:

$$y_e(j) = f\left(x_e(j)\right) \tag{A.2}$$

Cette fonction doit être continue, croissante, et dérivable. On utilise souvent une fonction sigmoïde telle que:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{A.3}$$

A.3 Dérivation du gradient

Comme tout système d'apprentissage supervisé, nous cherchons à faire évoluer le réseau de façon à minimiser un coût donné (E_e) pour chaque vecteur d'entrée e. On utilise souvent le critère des moindres carrés:

$$E_e = \frac{1}{2} \sum_{j \in sortie} (d_e(j) - y_e(j))^2 \tag{A.4}$$

On cherche alors à découvrir le gradient de E_e par rapport aux poids du réseau: $\frac{\partial E_e}{w(i,j)}$. Pour plus de clarté, nous laisserons tomber les indices (e) dans les équations qui viennent. Il suffit de se rappeler que le véritable gradient est la somme des gradients obtenus pour chaque vecteur d'entrée e.

Par la règle de dérivation chaînée, on sait que:

$$\frac{\partial E}{\partial w(i,j)} = \frac{\partial E}{\partial y(j)} \cdot \frac{\partial y(j)}{\partial x(j)} \cdot \frac{\partial x(j)}{\partial w(i,j)} \tag{A.5}$$

Or, par l'équation (A.1), on déduit que

$$\frac{\partial x(j)}{\partial w(i,j)} = y(i) \tag{A.6}$$

et par l'équation (A.2), on déduit que

$$\frac{\partial y(j)}{\partial x(j)} = f'(x(j)) \tag{A.7}$$

Si de plus, f est définie comme dans l'équation (A.3), alors, l'équation précédente devient:

$$\frac{\partial y(j)}{\partial x(j)} = f\left(x(j)\right) \cdot \left(1 - f\left(x(j)\right)\right) \tag{A.8}$$

Il ne reste alors qu'à calculer $\frac{\partial E}{\partial y(j)}$. Lorsque j est un neurone de sortie, on le trouve aisément à l'aide de l'équation (A.4):

$$\frac{\partial E}{\partial y(j)} = y(j) - d(j) \tag{A.9}$$

et lorsque j est un neurone caché, on calcule $\frac{\partial E}{\partial y(j)}$ de façon récursive:

$$\frac{\partial E}{\partial y(j)} = \sum_{k \in dest(j)} \frac{\partial E}{\partial y(k)} \cdot w(j,k) \tag{A.10}$$

En combinant les équations précédentes, on trouve le gradient: (a) pour les neurones j de sortie:

$$\frac{\partial E}{\partial w(i,j)} = f\left(x(j)\right) \cdot \left(1 - f\left(x(j)\right)\right) \cdot y(i) \cdot \left(y(j) - d(j)\right) \tag{A.11}$$

et (b) pour les neurones cachés:

$$\frac{\partial E}{\partial w(i,j)} = f\left(x(j)\right) \cdot \left(1 - f\left(x(j)\right)\right) \cdot y(i) \cdot \sum_{k \in dest(j)} \frac{\partial E}{\partial y(k)} \cdot w(j,k) \tag{A.12}$$

avec $\frac{\partial E}{\partial y(k)}$ calculé avec les équations (A.9) et (A.10).

Finalement, pour effectuer la descente du gradient, il faut se déplacer dans la direction opposée au gradient:

$$\Delta w(i,j) = -\epsilon \frac{\partial E}{\partial w(i,j)} \tag{A.13}$$

où ϵ est le pas de déplacement, aussi appelé le taux d'apprentissage.

Annexe B

Une règle paramétrique qui englobe la règle de la rétropropagation de l'erreur

Dans cette annexe, nous décrivons une règle d'apprentissage paramétrique utilisée notamment pour les expériences de classification du chapitre 6 qui peut, en choisissant correctement les paramètres et en tenant compte de contraintes strictes dans le choix de l'architecture du réseau de neurones avec lequel la règle est utilisée, simuler le comportement de la règle de la rétropropagation de l'erreur (décrite à l'annexe A).

Soit un réseau de neurones artificiels utilisant des neurones modulateurs pour chaque connexion tel qu'illustré à la figure B.1 (ainsi la connexion entre les neurones i et j est modulée par le neurone k et de façon similaire, la connexion entre les neurones k et l est modulée par le neurone i), nous présentons ici une façon de *simuler* la règle de la rétropropagation de l'erreur en utilisant les neurones modulateurs.

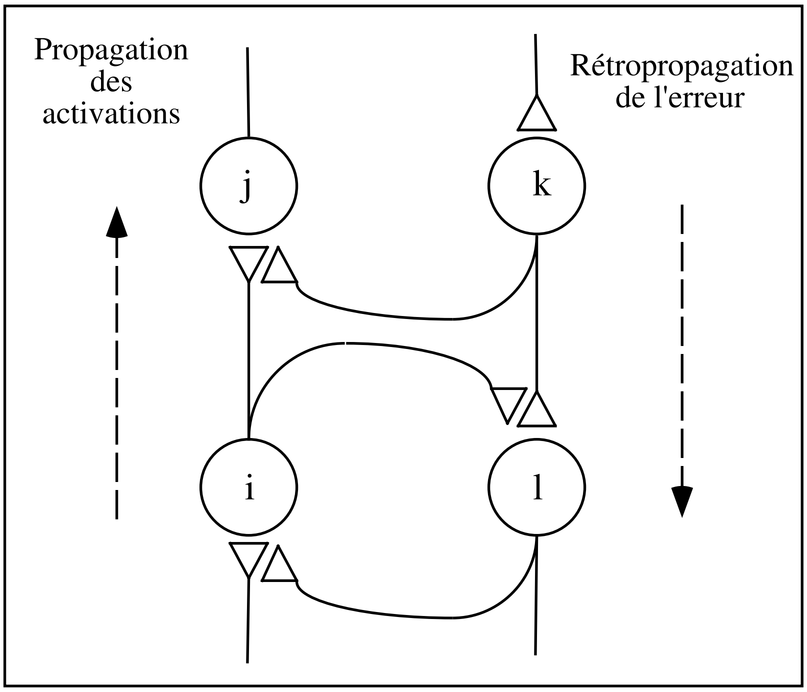

Figure B.1: Partie d'un réseau de neurone artificiel utilisant des neurones modulateurs. Le neurone k module la connexion $i \rightarrow j$ et le neurone i module la connexion $k \rightarrow l$.

Rappelons tout d'abord que le changement de poids tel que dicté par la règle de la rétropropagation de l'erreur est:

$$\Delta w(i,j) = -\epsilon \cdot y(i) \cdot f'(x(j)) \cdot \frac{\partial E}{\partial y(j)} \tag{B.1}$$

où $\Delta w(i,j)$ est le changement de poids apporté à la connexion entre les neurones i et j, ϵ est une constante, $y(i)$ est la valeur de sortie du neurone i, $f'(x(j))$ est la dérivée de la fonction d'activation utilisée au neurone j, E est le coût que l'on veut minimiser (les moindres carrés) et $\frac{\partial E}{\partial y(j)}$ la dérivée de ce coût par rapport à la valeur de sortie du neurone j. L'annexe A explique plus en détails la provenance de cette formule.

De plus, on sait que $\frac{\partial E}{\partial y(j)}$ varie selon que j soit un neurone de sortie ou un neurone caché. Si c'est un neurone de sortie,

$$\frac{\partial E}{\partial y(j)} = y(j) - d(j) \tag{B.2}$$

où $d(j)$ est la sortie désirée au neurone j. Si c'est un neurone caché, alors,

$$\frac{\partial E}{\partial y(j)} = \sum_{u \in dest(j)} \frac{\partial E}{\partial y(u)}\, w(j,u) \tag{B.3}$$

où $dest(j)$ est l'ensemble des neurones auxquels j se connecte, c'est-à-dire que $\frac{\partial E}{\partial y(j)}$ est calculé récursivement en fonction des neurones situés *en aval* de lui.

Or, si on regarde la valeur des neurones modulateurs selon l'architecture proposée (figures 6.5 et B.1), on voit par exemple que

$$y(k) = y(mod(j)) = d(j) - y(j) \tag{B.4}$$

si j est un neurone de sortie, et

$$\begin{aligned} y(l) = y(mod(i)) &= \sum_{u \in source(l)} y(u)\, w(u,l) \\ &= y(k)\, w(k,l) \end{aligned} \tag{B.5}$$

où $source(l)$ est l'ensemble des neurones se connectant au neurone l. Ainsi, on peut affirmer que pour le neurone de sortie j,

$$y(mod(j)) = -\frac{\partial E}{\partial y(j)} \tag{B.6}$$

Pour les neurones cachés, il faut d'une part s'assurer que les poids des connexions *avant* soient identiques en tout temps aux poids des connexions *arrière*, c'est-à-dire par exemple que

$$w(k,l) = w(i,j) \tag{B.7}$$

Pour ce faire, on peut par exemple initialiser les deux poids de façon identique puis effectuer à chaque itération le même changement de poids.

Il faut de plus modifier la fonction d'activation des neurones modulateurs des couches cachées pour y ajouter un facteur. Elle devient alors

$$x(l) = \sum_{u \in source(l)} y(u) \cdot w(u,l) \cdot f'(x(mod(u))) \tag{B.8}$$

Grâce à ces changement, pour effectuer un changement de poids équivalent à la règle de la rétropropagation de l'erreur, on peut maintenant faire

$$\Delta w(i,j) = \epsilon \; y(i) \; f'(x(j)) \; y(mod(j)) \tag{B.9}$$

où ϵ, la constante de changement de poids, peut être remplacée par un paramètre si l'on inclut ce *module* dans une règle paramétrique. Notons que ces modifications rendent cependant l'architecture du réseau non biologiquement plausible.

Annexe C

Résultats complets des expériences de classification

Nous présentons dans cette annexe *tous* les résultats obtenus dans le cadre des expériences de classifications présentées à la section 6.3.

Les résultats sont présentés sous forme de tableaux. Chaque tableau synthétise différentes expériences ayant plusieurs paramètres en commun: la méthode d'optimisation ainsi que la forme de la règle d'apprentissage paramétrique[1].

Les méthodes d'optimisation utilisées sont:

- la descente du gradient,
- le recuit simulé,

[1]ne s'applique pas pour la méthode de programmation génétique.

- les algorithmes génétiques,
- la programmation génétique.

Les formes de règles d'apprentissage paramétriques utilisées (pour les méthodes autres que la programmation génétique) sont:

- une règle fortement contrainte, à 7 paramètres:

$$\begin{aligned}\Delta w(i,j) &= \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) + \theta_4\, y(i)\, y(mod(j)) \\ &\quad + \theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, w(i,j) \qquad \text{(C.1)}\end{aligned}$$

- une règle moins contrainte, à 16 paramètres (bien qu'encore biologiquement plausible):

$$\begin{aligned}\Delta w(i,j) &= \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) + \theta_4\, w(i,j) \\ &\quad + \theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, y(mod(j)) + \theta_7\, y(i)\, w(i,j) \\ &\quad + \theta_8\, x(j)\, y(mod(j)) + \theta_9\, x(i)\, w(i,j) \\ &\quad + \theta_{10}\, y(mod(j))\, w(i,j) + \theta_{11}\, y(i)\, x(j)\, y(mod(j)) \\ &\quad + \theta_{12}\, y(i)\, x(j)\, w(i,j) + \theta_{13}\, y(i)\, y(mod(j))\, w(i,j) \\ &\quad + \theta_{14}\, x(j)\, y(mod(j))\, w(i,j) \\ &\quad + \theta_{15}\, y(i)\, x(j)\, y(mod(j))\, w(i,j) \qquad \text{(C.2)}\end{aligned}$$

- une règle fortement contrainte, à 8 paramètres, incluant notamment dans l'espace des solutions celle de la rétropropagation de l'erreur[2]:

$$\Delta w(i,j) = \theta_0 + \theta_1\, y(i) + \theta_2\, x(j) + \theta_3\, y(mod(j)) + \theta_4\, y(i)\, y(mod(j))$$

[2] Il faut de plus s'assurer que les connexions arrières du réseau aient les mêmes poids initiaux que les connexions avant, ce qui rends la règle d'apprentissage non biologiquement plausible (voir l'annexe B pour plus de détails).

$$+\theta_5\, y(i)\, x(j) + \theta_6\, y(i)\, w(i,j)$$
$$+\theta_7\, y(i)\, y(mod(j))\, f'(x(j)) \tag{C.3}$$

Les tâches utilisées pour ces expériences sont décrites à la section 6.3. Il s'agit de tâches de classification bi-dimensionnelle ne contenant que 2 classes distinctes. On peut cependant trouver, à l'intérieur de cette classe restreinte de tâches, deux types différents de tâches (principalement quant à leur complexité):

- les tâches linéairement séparables (voir figure 6.6), et
- les tâches linéairement non séparables (voir figure 6.7).

Chaque tâche a été créée de la façon suivante:

1. Choisir aléatoirement un nombre donné de points dans l'espace bi-dimensionnel.
2. Attribuer aléatoirement une classe à chaque point.
3. Générer autour de chaque point un nuage de points selon une distribution normale. Chaque point ainsi généré correspond à un exemple de la tâche, et sa classe est celle du point central du nuage.

Ensuite, les tâches générées étaient séparées en deux catégories: les tâches linéairement séparables et les tâches linéairement non séparables.

Finalement, chaque optimisation de la règle a été réalisée avec un nombre de tâches variant de 1 à 9. Les tâches utilisées pour l'optimisation de la règle étaient choisies aléatoirement dans un ensemble de 20 tâches (séparées en deux groupes: 10 tâches linéairement séparables et 10 tâches linéairement non séparables), et les tâches qui n'étaient pas utilisées pour l'optimisation de la règle permettaient de vérifier sa capacité de généralisation.

Chaque tableau montre l'erreur de généralisation sur des tâches linéairement séparables (LS) et linéairement non séparables (LNS) ainsi que l'erreur globale sur l'ensemble des tâches (LS et LNS), et ce, en variant le nombre de tâches utilisées pour l'optimisation ainsi que le type de ces tâches (LS ou LNS).

Dans tous les cas, soit C_0 et C_1 les deux classes possibles, l'erreur E est calculée de la façon suivante:

$$E = \frac{\sum_{i \in \mathcal{T}} \left(\frac{\sum_{j \in v(i)} \delta_{s_{i,j}}^{d_{i,j}}}{|v(i)|} \right)}{|\mathcal{T}|} \tag{C.4}$$

où $\mathcal{T}$ est l'ensemble des tâches sur lequel on calcule l'erreur, $v(i)$ est l'ensemble des vecteurs définissant la tâche i, $s_{i,j}$ est la classe (C_0 ou C_1) obtenue par la règle sur le vecteur j de la tâche i, alors que $d_{i,j}$ est la classe désirée pour le même vecteur. Enfin, $\delta_{s_{i,j}}^{d_{i,j}}$ vaut 1 lorsque $s_{i,j} = d_{i,j}$, et vaut 0 autrement. En somme, l'erreur E représente le *pourcentage moyen de vecteurs mal classifiés sur l'ensemble des tâches*[3]

[3]ce critère a été préféré à celui des moindres carrés parce qu'il donne une meilleure idée de la performance de la règle lorsqu'appliqué à des tâches de classification (par opposition aux tâches de régression où l'on doit utiliser les moindres carrés).

C.1 Présentation des résultats

Pour chaque tableau, $E(LS)$ représente l'erreur de généralisation sur des tâches linéairement séparables, $E(LNS)$ l'erreur de généralisation sur des tâches linéairement non séparables, et $E(total)$ l'erreur de généralisation sur l'ensemble des tâches n'ayant pas servies à l'optimisation de la règle. Notons de plus que chaque expérience fut réalisée sur un Sparc 2 et prit entre quelques minutes et près de 10 heures (selon la complexité de l'expérience, le nombre de tâches, etc) de temps calcul.

De plus, on pourra retrouver à la section 6.3.4 une analyse plus approfondie des résultats de ces expériences.

Nombre	Apprentissage tâches LS			Apprentissage tâches LNS		
de tâches	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.59	0.50	0.51	0.17	0.32	0.23
2	0.62	0.60	0.54	0.28	0.39	0.30
3	0.18	0.28	0.20	0.69	0.74	0.65
4	0.21	0.25	0.18	0.34	0.38	0.31
5	0.01	0.29	0.15	0.42	0.24	0.28
6	0.10	0.26	0.15	0.07	0.20	0.10
7	0.00	0.25	0.13	0.19	0.19	0.17
8	0.00	0.29	0.15	0.30	0.14	0.21
9	0.25	0.26	0.14	0.13	0.30	0.15

Tableau C.1: Tableau des résultats obtenus avec une règle de **7 paramètres** en utilisant le **recuit simulé**.

Nombre	Apprentissage tâches LS			Apprentissage tâches LNS		
de tâches	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.39	0.36	0.35	0.63	0.58	0.58
2	0.27	0.38	0.30	0.80	0.57	0.63
3	0.21	0.33	0.24	0.39	0.49	0.36
4	0.04	0.24	0.13	0.41	0.44	0.37
5	0.14	0.41	0.24	0.63	0.53	0.48
6	0.48	0.25	0.22	0.15	0.46	0.20
7	0.00	0.75	0.38	0.16	0.33	0.18
8	0.00	0.22	0.11	0.08	0.15	0.12
9	0.01	0.28	0.14	0.10	0.36	0.14

Tableau C.2: Tableau des résultats obtenus avec une règle de **16 paramètres** en utilisant le **recuit simulé**.

Nombre de tâches	Apprentissage tâches LS			Apprentissage tâches LNS		
	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.17	0.30	0.22	0.14	0.31	0.21
2	0.17	0.23	0.18	0.36	0.41	0.35
3	0.12	0.26	0.17	0.11	0.43	0.21
4	0.05	0.22	0.12	0.14	0.21	0.16
5	0.03	0.27	0.14	0.16	0.23	0.18
6	0.02	0.27	0.14	0.19	0.21	0.19
7	0.01	0.26	0.13	0.11	0.17	0.13
8	0.01	0.28	0.15	0.13	0.14	0.15
9	0.00	0.21	0.11	0.05	0.15	0.11

Tableau C.3: Tableau des résultats obtenus avec une règle de **7 paramètres** en utilisant les **algorithmes génétiques**.

Nombre de tâches	Apprentissage tâches LS			Apprentissage tâches LNS		
	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.58	0.48	0.50	0.48	0.52	0.47
2	0.35	0.53	0.40	0.48	0.43	0.42
3	0.29	0.37	0.23	0.47	0.45	0.41
4	0.08	0.28	0.16	0.30	0.32	0.27
5	0.09	0.30	0.19	0.44	0.32	0.31
6	0.04	0.28	0.16	0.31	0.43	0.28
7	0.00	0.27	0.15	0.12	0.37	0.20
8	0.00	0.26	0.13	0.21	0.37	0.21
9	0.05	0.23	0.13	0.49	0.18	0.31

Tableau C.4: Tableau des résultats obtenus avec une règle de **16 paramètres** en utilisant les **algorithmes génétiques**.

Nombre de tâches	Apprentissage tâches LS			Apprentissage tâches LNS		
	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.45	0.53	0.46	0.08	0.24	0.15
2	0.09	0.23	0.15	0.05	0.19	0.13
3	0.08	0.23	0.15	0.17	0.23	0.19
4	0.08	0.23	0.15	0.09	0.23	0.17
5	0.01	0.25	0.14	0.04	0.14	0.13
6	0.01	0.25	0.14	0.17	0.20	0.19
7	0.00	0.25	0.15	0.04	0.12	0.13
8	0.00	0.25	0.13	0.04	0.10	0.13
9	0.00	0.20	0.13	0.06	0.17	0.14

Tableau C.5: Tableau des résultats obtenus en utilisant la **programmation génétique**.

Nombre de tâches	Apprentissage tâches LS			Apprentissage tâches LNS		
	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.40	0.48	0.41	0.14	0.38	0.25
2	0.01	0.20	0.11	0.01	0.16	0.09
3	0.01	0.16	0.09	0.02	0.15	0.09
4	0.00	0.16	0.09	0.01	0.15	0.09
5	0.00	0.16	0.09	0.01	0.15	0.09
6	0.00	0.16	0.09	0.01	0.14	0.09
7	0.00	0.20	0.11	0.01	0.14	0.09
8	0.00	0.20	0.11	0.01	0.14	0.09
9	0.00	0.16	0.09	0.02	0.14	0.10

Tableau C.6: Tableau des résultats obtenus en utilisant la **programmation génétique** et en **incluant la règle de la rétropropagation de l'erreur dans l'espace des solutions envisageables**.

Nombre de tâches	Apprentissage tâches LS			Apprentissage tâches LNS		
	$E(LS)$	$E(LNS)$	$E(total)$	$E(LS)$	$E(LNS)$	$E(total)$
1	0.52	0.48	0.47	0.26	0.37	0.31
2	0.09	0.32	0.19	0.21	0.39	0.26
3	0.05	0.30	0.17	0.16	0.32	0.19
4	0.03	0.29	0.16	0.40	0.42	0.34
5	0.02	0.28	0.15	0.34	0.43	0.30
6	0.01	0.27	0.13	0.13	0.30	0.18
7	0.03	0.27	0.14	0.31	0.57	0.27
8	0.07	0.28	0.15	0.12	0.14	0.15
9	0.00	0.27	0.14	0.14	0.16	0.14

Tableau C.7: Tableau des résultats obtenus avec une règle à **8 paramètres** en utilisant les **algorithmes génétiques** et en **incluant la règle de la rétropropagation de l'erreur dans l'espace des solutions envisageables**.

$E(LS)$	$E(LNS)$	$E(total)$
0.05	0.20	0.13

Tableau C.8: Tableau des résultats obtenus avec la **règle de la rétropropagation de l'erreur**.