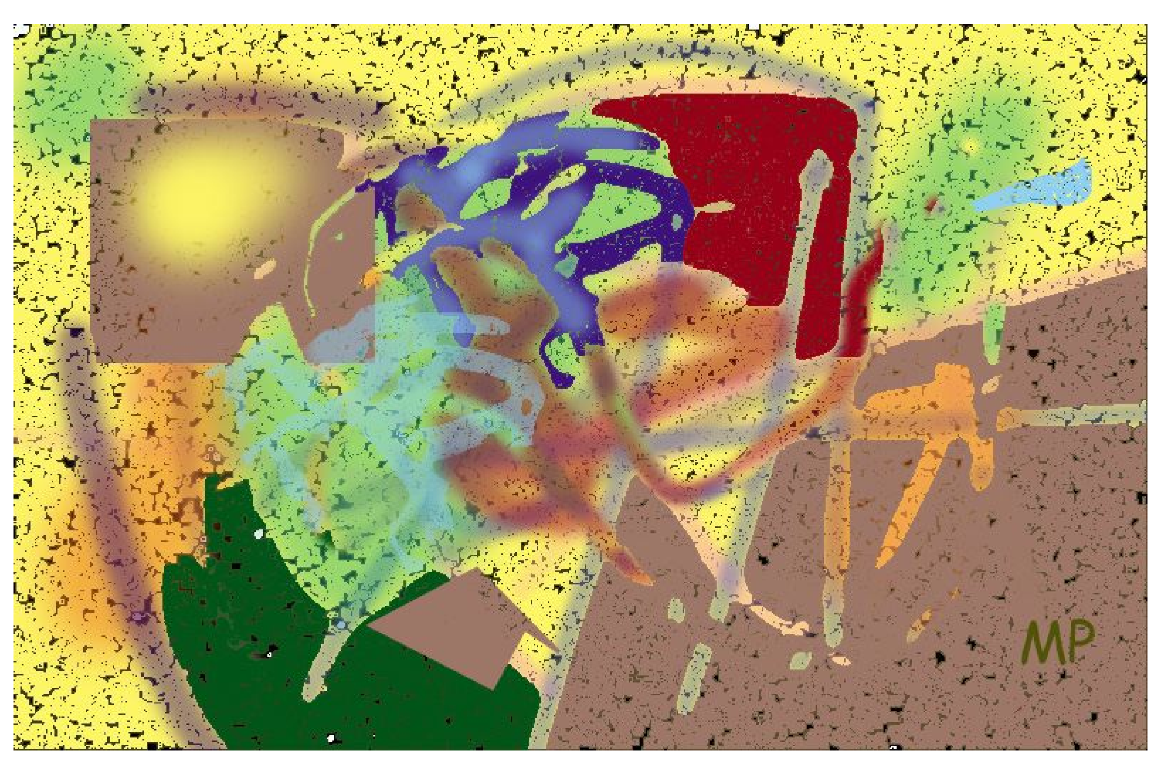
RÉSEAUX DE NEURONES

GIF-21140 et GIF-64326

par ***Marc Parizeau***



Automne 2004

Table des matières

1 Introduction **1**

1.1 Objectifs . . . 2

1.2 Histoire . . . 2

1.3 Applications . . . 4

2 Modèle de neurone et réseau **5**

2.1 Notations . . . 5

2.2 Modèle d'un neurone . . . 8

2.3 Fonctions de transfert . . . 10

2.4 Architecture de réseau . . . 12

3 Algèbre linéaire **17**

3.1 Définition d'un espace vectoriel . . . 17

3.1.1 Dépendance linéaire . . . 19

3.1.2 Bases et dimensions . . . 19

3.1.3 Produit scalaire . . . 20

3.1.4 Norme . . . 20

3.1.5 Orthogonalité . . . 21

3.2 Transformations linéaires . . . 22

3.2.1 Représentations matricielles . . . 22

3.2.2 Changement de base . . . 24

3.2.3 Valeurs et vecteurs propres . . . 25

4 Processus d'apprentissage **27**

4.1 Par correction d'erreur . . . 28

4.2 Par la règle de Hebb . . . 29

4.3 Compétitif . . . 32

4.4 Problème de l'affectation du crédit . . . 33

4.5 Supervisé . . . 33
4.6 Par renforcement . . . 34
4.7 Non-supervisé . . . 35
4.8 Tâches d'apprentissage . . . 36

5 Perceptron multicouche **39**
5.1 Perceptron simple . . . 39
5.2 Règle LMS . . . 44
5.3 Réseau multicouche . . . 47
5.3.1 Problème du «ou exclusif» . . . 48
5.3.2 Approximation de fonction . . . 49
5.3.3 Classification . . . 50
5.4 Rétropropagation des erreurs . . . 51
5.4.1 Calcul des sensibilités . . . 54
5.4.2 Algorithme d'entraînement . . . 55
5.4.3 Critères d'arrêt . . . 56
5.4.4 Phénomène de saturation . . . 58
5.4.5 Groupage . . . 59
5.4.6 Momentum . . . 59
5.4.7 Taux d'apprentissage variable . . . 60
5.4.8 Autres considérations pratiques . . . 60
5.5 Méthode de Newton . . . 62
5.6 Méthode du gradient conjugué . . . 63
5.6.1 Algorithme du gradient conjugué . . . 64
5.6.2 Recherche du minimum le long d'une droite . . . 65

6 Nuées dynamiques **69**
6.1 K-means . . . 70
6.2 Fuzzy K-means . . . 72
6.3 Possibilistic K-means . . . 74

7 Réseau de Kohonen **79**
7.1 Algorithme de Kohonen . . . 81
7.2 Propriétés . . . 82
7.3 Exemples . . . 84
7.4 Réseau LVQ . . . 87

8 Réseau GNG **89**
8.1 Algorithme . . . 89
8.2 Exemple . . . 91

9 Architectures ART **93**
9.1 Fuzzy ART . . . 94
9.2 Fuzzy ARTmap . . . 98

10 ACP et apprentissage hebbien **103**
10.1 Règle de Hebb . . . 106
10.2 Règle de Oja . . . 106
10.3 Règle de Sanger . . . 107
10.4 Apprentissage de Hebb supervisé . . . 108
10.4.1 Règle de la matrice pseudo-inverse . . . 109
10.4.2 Exemple d'auto-association . . . 110

11 Réseau RBF **113**
11.1 Entraînement d'un réseau RBF . . . 115

Table des figures

2.1 *Modèle d'un neurone artificiel.* 8
2.2 *Schéma d'un neurone biologique.* 9
2.3 *Représentation matricielle du modèle d'un neurone artificiel.* 10
2.4 *Fonction de transfert : (a) du neurone «seuil» ; (b) du neurone «linéaire», et (c) du neurone «sigmoïde».* 12
2.5 *Couche de S neurones.* 13
2.6 *Représentation matricielle d'une couche de S neurones.* 14
2.7 *Représentation matricielle d'un réseau de trois couches.* 14
2.8 *Élément de retard.* 15

3.1 *Différents sous-ensembles de $\Re^2$: (a) région rectangulaire ; (b) droite ; (c) plan.* . . . 18
3.2 *Illustration de la méthode de transformation orthogonale Gram-Schmidt.* 22
3.3 *Transformation de rotation : (a) rotation du vecteur $\mathbf{x}$; (b) rotation des vecteurs de base $\mathbf{v}_1$ et $\mathbf{v}_2$.* 24

4.1 *Trajectoire de la descente du gradient.* 29
4.2 *Représentation graphique de la règle «instar» lors d'une activité post-synaptique positive.* 32
4.3 *Schéma bloc de l'apprentissage supervisé.* 34

5.1 *Perceptron à une seule couche avec fonction seuil.* 40
5.2 *Frontière de décision pour un perceptron simple à 1 neurone et deux entrées.* . . . 41
5.3 *Exemple d'un problème à deux classes (points noirs vs points blancs).* 42
5.4 *Exemples de problèmes non linéairement séparables.* 43
5.5 *Réseau ADALINE.* 44
5.6 *Trajectoire de la descente du gradient pour différents taux d'apprentissage : (a) taux faible ; (b) taux moyen ; (c) taux (trop) élevé.* 47
5.7 *Réseau multicouche pour résoudre le problème du «ou exclusif».* 48

5.8 *Frontières de décision engendrées par le réseau de la figure 5.7 : (a) neurone 1 de la couche 1 ; (b) neurone 2 de la couche 1 ; (c) neurone 1 de la couche 2.* 49
5.9 *Frontière de décision engendrée par le neurone qui effectue une conjonction.* . . . 49
5.10 *Réseau multicouche permettant de faire de l'approximation de fonction.* 50
5.11 *Exemples de frontières de décision : (a) convexe ouverte ; (b) convexe fermée ; (c) concave ouverte ; et (d) concave fermée.* 50
5.12 *Représentation matricielle d'un réseau de trois couches (reproduction de la figure 2.7).* 51
5.13 *Illustration du phénomène de sur-apprentissage pour le cas simple d'une approximation de fonction.* 57
5.14 *Illustration de la validation croisée.* 57
5.15 *Exemple d'un neurone saturé.* 58
5.16 *Illustration de la méthode du gradient conjugué.* 65
5.17 *Étape de localisation d'un intervalle initial de recherche.* 66
5.18 *Étape de réduction de l'intervalle de recherche : (a) cas où $F_c < F_d$; (b) cas où $F_d < F_c$.* 67

6.1 *Couche compétitive de $S = K$ neurones.* 70
6.2 *Algorithme du «k-means».* 71
6.3 *Example d'une partition rigide en deux classes : (a) cas sans bruit ; (b) cas avec bruit.* 72
6.4 *Algorithme du «fuzzy k-means».* 73
6.5 *Exemple d'une partition floue à deux classes.* 74
6.6 *Algorithme du «possibilistic k-means».* 77

7.1 *Réseau de Kohonen avec carte rectangulaire de $S = 6 \times 7 = 42$ neurones.* 80
7.2 *Topologie de voisinage (quatre voisins) pour une carte à deux dimensions : (a) $\Lambda_{18} = 2$; (b) $\Lambda_{18} = 1$; et (c) $\Lambda_{18} = 0$.* 80
7.3 *Exemple de décroissance (a) du taux d'apprentissage et (b) de la fenêtre de voisinage en fonction du temps.* 81
7.4 *Algorithme de Kohonen.* 82
7.5 *Fonction de voisinage gaussienne.* 83
7.6 *Illustration de la relation entre la carte auto-organisée $\mathbf{\Phi}$ et le vecteur ${}_g\mathbf{w}$ du neurone gagnant pour le stimulus $\mathbf{p}$.* 84
7.7 *Exemple d'une carte auto-organisée à une dimension. Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un triangle.* 85
7.8 *Exemple d'une carte auto-organisée à deux dimensions. Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un carré.* 85

7.9 *Exemple d'une carte auto-organisée à deux dimensions (droite). Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un volume tridimensionnel en forme de cactus (gauche).* . . . 86
7.10 *Réseau LVQ* . . . 87

8.1 *Exemple d'un GNG entraîné sur des stimuli échantillonnés dans un volume en forme de prisme rectangulaire, sur une surface rectangulaire apposée perpendiculairement à l'une des faces du prisme et à une courbe en forme d'anneau au bout d'une tige apposée à l'une des extrémités de la surface.* . . . 92

9.1 *Architecture du réseau fuzzy ART.* . . . 94
9.2 *Représentation vectorielle d'un ensemble flou E défini sur un référentiel de deux éléments.* . . . 95
9.3 *Régions associées à ${}_i\mathbf{w} = [\mathbf{x}\ \mathbf{y}^c]^T$ (en trait plein) et à $\mathbf{a}^1 \cap {}_i\mathbf{w}$ (en trait pointillé).* . . . 97
9.4 *Architecture du réseau fuzzy ARTmap.* . . . 99

10.1 *Illustration des composantes principales pour un nuage de points en deux dimensions.* . . . 104
10.2 *Réseau permettant d'effectuer une analyse en S composantes principales.* . . . 107
10.3 *Réseau auto-associatif pour la reconnaissance de chiffres.* . . . 110
10.4 *Prototypes pour l'apprentissage auto-associatif des chiffres «0», «1» et «2».* . . . 111
10.5 *Exemples de réponses du réseau auto-associatif de la figure 10.3 pour des stimuli dégradés ou bruités : (a) chiffres 0 ; (b) chiffres 1 ; et (c) chiffres 2.* . . . 111

11.1 *Réseau RBF avec fonctions radiales gaussiennes.* . . . 114

Liste des tableaux

2.1 *Fonctions de transfert* $a = f(n)$. 11

9.1 *Valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte de classement.* 101

9.2 *Valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte d'approximation de fonction.* 101

Chapitre 1

Introduction

Les réseaux de neurones, fabriqués de structures cellulaires artificielles, constituent une approche permettant d'aborder sous des angles nouveaux les problèmes de perception, de mémoire, d'apprentissage et de raisonnement. Ils s'avèrent aussi des alternatives très prometteuses pour contourner certaines des limitations des ordinateurs classiques. Grâce à leur traitement parallèle de l'information et à leurs mécanismes inspirés des cellules nerveuses (neurones), ils infèrent des propriétés émergentes permettant de solutionner des problèmes jadis qualifiés de complexes.

Nous aborderons dans ce cours les principales architectures de réseaux de neurones que l'on retrouve dans la littérature. Il ne s'agit pas de les étudier toutes, car elles sont trop nombreuses, mais plutôt d'en comprendre les mécanismes internes fondamentaux et de savoir comment et quand les utiliser. En ce sens, nous mettrons autant l'emphase sur l'analyse mathématique de ces réseaux que sur la façon de les utiliser dans la pratique pour résoudre des problèmes concrets.

Nous aborderons également certaines notions relatives aux ensembles flous et à la logique dans la mesure où ces derniers sont incorporés dans certaines des architectures de réseaux de neurones que nous étudierons.

Le reste de ce chapitre élabore davantage sur les objectifs poursuivis par ce cours, puis présente un bref historique du domaine des réseaux de neurones avant de terminer par un survol de leurs différentes applications. Le chapitre 2 introduit ensuite le modèle mathématique du neurone artificiel et établit une notation cohérente qui sera suivie tout au long des chapitres subséquents. Le chapitre 3 effectue un certain nombre de rappels en algèbre linéaire, rappels qui seront forts utiles tout au long de cet ouvrage lorsque nous étudierons différentes architectures de réseaux de neurones. Le chapitre 4 présente quant à lui la problématique générale de l'apprentissage en décrivant les principales règles pouvant être utilisées ainsi que les différents types d'approche. Ainsi, nous aborderons les notions de correction d'erreur, d'apprentissage hebbien, compétitif, supervisé, non-supervisé et, finalement, d'apprentissage par renforcement. Par la suite, nous étudierons en détails différentes architectures de réseau de neurones dont le perceptron multicouche, le réseau de Kohonen, le «Growing Neural Gas» (GNG), certains membres de la famille des réseaux ART («Adaptive Resonance Theory»), le «Radial Basis Function» (RBF) et le «Support Vector Machine» (SVM). Nous traiterons aussi de l'algorithme du «K-means» qui s'apparente au réseau de Koho-

nen, ainsi que de l'analyse en composantes principales (ACP) via un apprentissage hebbien.

Ces notes de cours sont dérivées d'un certain nombre d'ouvrages dont les principaux sont énumérés en annexe.

1.1 Objectifs

Le cerveau humain contient environ 100 milliards de neurones. Ces neurones vous permettent, entre autre, de lire ce texte tout en maintenant une respiration régulière permettant d'oxygéner votre sang, en actionnant votre c ?ur qui assure une circulation efficace de ce sang pour nourrir vos cellules, etc. Ils vous permettent même, je l'espère, de comprendre les idées que je tente de vous transmettre !

Chacun de ces neurones est par ailleurs fort complexe. Essentiellement, il s'agit de tissu vivant et de chimie. Les spécialistes des neurones biologiques (ceux qui ?uvrent en neurophysiologie) commencent à peine à comprendre quelques uns de leurs mécanismes internes. On croit en général que leurs différentes fonctions neuronales, y compris celle de la mémoire, sont stockées au niveau des connexions (synapses) entre les neurones. C'est ce genre de théorie[1] qui a inspiré la plupart des architectures de réseaux de neurones artificiels[2] que nous aborderons dans ce cours. L'apprentissage consiste alors soit à établir de nouvelles connexions, soit à en modifier des existantes.

Ceci nous amène à poser une question fondamentale : en ce basant sur nos connaissances actuelles, peut-on construire des modèles approximatifs de neurones et les entraîner pour, éventuellement, réaliser des tâches utiles ? Eh bien, la réponse courte est «oui», même si les réseaux que nous allons développer ne possèdent qu'une **infime fraction** de la puissance du cerveau humain, et c'est l'objectif du cours de vous montrer comment on peut y arriver sans salir son linge !

Pour ce qui est de la réponse longue (plus détaillée), elle suit dans les chapitres subséquents. Mais avant d'y arriver, faisons un peu d'histoire...

1.2 Histoire

De nombreux ouvrages ont permis de documenter l'histoire des recherches en réseaux de neurones. En particulier, le livre intitulé «Neurocomputing : Foundations of Research» édité par John Anderson et Edward Rosenfeld est une compilation de 43 articles qui ont marqué le domaine sur le plan historique. Chacun d'entre eux est d'ailleurs précédé d'une introduction qui permet de situer l'article dans son contexte.

[1]Des théories récentes suggèrent au contraire que l'information pourrait être stockée au niveau de la morphologie des connexions (des dendrites) ; mais ceci est totalement en dehors du cadre de ce cours (ainsi que du domaine de compétence du professeur !).

[2]Ce cours traitant exclusivement des réseaux de neurones artificiels (par opposition à biologique), nous omettrons parfois d'ajouter le mot «artificiel» à la suite de «neurone» et de «réseau de neurones», sachant qu'il est toujours sous-entendu, sauf lorsque mention explicite du contraire.

Deux ingrédients sont à la base de tout avancement des connaissances. Premièrement, il importe de posséder un nouveau concept, ou un nouveau point de vue à propos d'un sujet, qui vient jeter une lumière là où il n'y avait qu'obscurité. Par exemple, considérons le c ?ur humain. À différentes époques on le considérait comme le centre de l'âme ou encore comme une source de chaleur. Quelque part au 17e siècle, les médecins ont commencé à le considérer comme une pompe et ont donc conçu des expériences pour tenter de comprendre son fonctionnement, ce qui a éventuellement permis une compréhension du système sanguin, etc. Sans le concept de pompe, une compréhension du c ?ur et du système sanguin en général était simplement hors d'atteinte.

Deuxièmement, il importe aussi de posséder des outils technologiques permettant de construire des systèmes concrets. Par exemple, on connaissait les théories physiques permettant d'envisager la conception d'une bombe atomique bien avant d'être capable de réaliser une telle bombe. On savait aussi mathématiquement reconstruire des images de radiographie en coupe (tomographie) bien avant de posséder les ordinateurs et les algorithmes capables d'effectuer efficacement les calculs requis dans un temps raisonnable.

L'histoire des réseaux de neurones est donc tissée à travers des découvertes conceptuelles et des développements technologiques survenus à diverses époques.

Brièvement, les premières recherches remontent à la fin du 19e et au début du 20e siècle. Ils consistent en de travaux multidisciplinaires en physique, en psychologie et en neurophysiologie par des scientifiques tels Hermann von Helmholtz, Ernst Mach et Ivan Pavlov. À cette époque, il s'agissait de théories plutôt générales sans modèle mathématique précis d'un neurone. On s'entend pour dire que la naissance du domaine des réseaux de neurones artificiels remonte aux années 1940 avec les travaux de Warren McCulloch et Walter Pitts qui ont montré qu'avec de tels réseaux, on pouvait, en principe, calculer n'importe quelle fonction arithmétique ou logique. Vers la fin des années 1940, Donald Hebb[3] a ensuite proposé une théorie fondamentale pour l'apprentissage. Nous y reviendrons d'ailleurs à plusieurs reprises dans les chapitres suivants.

La première application concrète des réseaux de neurones artificiels est survenue vers la fin des années 1950 avec l'invention du réseau dit «perceptron» par un dénommé Frank Rosenblatt. Rosenblatt et ses collègues ont construit un réseau et démontré ses habilités à reconnaître des formes. Malheureusement, il a été démontré par la suite que ce perceptron simple ne pouvait résoudre qu'une classe limitée de problème. Environ au même moment, Bernard Widrow et Ted Hoff ont proposé un nouvel algorithme d'apprentissage pour entraîner un réseau adaptatif de neurones linéaires, dont la structure et les capacités sont similaires au perceptron. Nous les étudierons tous les deux au chapitre 5.

Vers la fin des années 1960, un livre publié par Marvin Minsky et Seymour Papert est venu jeter beaucoup d'ombre sur le domaine des réseaux de neurones. Entre autres choses, ces deux auteurs ont démontré les limitations des réseaux développés par Rosenblatt et Widrow-Hoff. Beaucoup de gens ont été influencés par cette démonstration qu'ils ont généralement mal interprétée. Ils ont conclu à tort que le domaine des réseaux de neurones était un cul de sac et qu'il fallait cesser de s'y intéresser (et de financer la recherche dans ce domaine), d'autant plus qu'on ne disposait pas à l'époque d'ordinateurs suffisamment puissants pour effectuer des calculs complexes.

[3]Un canadien qui a passé la majorité de sa carrière académique à l'Université McGill.

Heureusement, certains chercheurs ont persévéré en développant de nouvelles architectures et de nouveaux algorithmes plus puissants. En 1972, Teuvo Kohonen et James Anderson ont développé indépendamment et simultanément de nouveaux réseaux pouvant servir de mémoires associatives (chapitre 7). Également, Stephen Grossberg a investigué ce qu'on appelle les réseaux auto-organisés (chapitre 9).

Dans les années 1980, une pierre d'achoppement a été levée par l'invention de l'algorithme de rétropropagation des erreurs (section 5.4). Cette algorithme est la réponse aux critiques de Minsky et Papert formulées à la fin des années 1960. C'est ce nouveau développement, généralement attribué à David Rumelhart et James McClelland, mais aussi découvert plus ou moins en même temps par Paul Werbos et par Yann LeCun, qui a littéralement ressuscité le domaine des réseaux de neurones. Depuis ce temps, c'est un domaine où bouillonne constamment de nouvelles théories, de nouvelles structures et de nouveaux algorithmes. Dans ce cours, nous allons tenter d'en survoler les principaux.

1.3 Applications

Les réseaux de neurones servent aujourd'hui à toutes sortes d'applications dans divers domaines. Par exemple, on a développé un auto-pilote pour avion, ou encore un système de guidage pour automobile, on a conçu des systèmes de lecture automatique de chèques bancaires et d'adresses postales, on produit des systèmes de traitement du signal pour différentes applications militaires, un système pour la synthèse de la parole, des réseaux sont aussi utilisés pour bâtir des systèmes de vision par ordinateur, pour faire des prévisions sur les marchés monétaires, pour évaluer le risque financier ou en assurance, pour différents processus manufacturiers, pour le diagnostic médical, pour l'exploration pétrolière ou gazière, en robotique, en télécommunication, et j'en passe ! Bref, les réseaux de neurones ont aujourd'hui un impact considérable et, il y a fort à parier, que leur importance ira grandissant dans le futur.

Chapitre 2

Modèle de neurone et réseau

Dans ce chapitre, nous présentons le modèle mathématique que nous emploierons dans les chapitres suivants pour décrire, d'une part, un neurone artificiel et, d'autre part, un réseau de neurones complet, c'est-à-dire un ensemble de neurones reliés en réseau. Le modèle que nous présentons dans ce chapitre est celui de base, commun à beaucoup d'architectures. Il n'est cependant pas universel, nous présenterons dans les chapitres subséquents les différentes variantes au fur et à mesure qu'il sera nécessaire de le faire.

2.1 Notations

Tout au long de cet ouvrage, nous tenterons d'adopter une notation mathématique cohérente. Les principales notations que nous adopterons sont énumérées ci-dessous. Il n'est pas nécessaire de tout mémoriser d'un seul coup, on pourra au besoin s'y rapporter plus tard.

Concepts de base

- Les scalaires seront désignés par des lettres minuscules italiques : p. ex. a, b, $c \ldots$
- Un «vecteur» désigne une colonne de nombres.
- Les vecteurs seront représentés par des minuscules grasses («bold») non italiques : p. ex. $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c} \ldots$
- Une «matrice» désigne un tableau de nombres ayant un certain nombre de lignes et de colonnes.
- Les matrices seront dénotées par des majuscules grasses («bold») non italiques : p. ex. $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C} \ldots$
- Un «vecteur-rangée» est une rangée d'une matrice utilisée comme un vecteur (donc une rangée transposée).

Poids d'une couche de neurones

- $\mathbf{W}^k(t)$ désigne la matrice des poids pour la couche k d'un réseau au temps t.
- $\mathbf{w}_j^k(t)$ désigne le vecteur correspondant à la colonne j de $\mathbf{W}^k(t)$.
- ${}_i\mathbf{w}^k(t)$ désigne le vecteur-rangée correspondant à la ligne i de $\mathbf{W}^k(t)$.
- $w_{i,j}^k(t)$ désigne l'élément (i, j) de $\mathbf{W}^k(t)$ (i désigne toujours une ligne et j une colonne).

Biais d'une couche de neurones

- $\mathbf{b}^k(t)$ désigne le vecteur des biais pour la couche k d'un réseau au temps t.
- $b_i^k(t)$ désigne l'élément i de $\mathbf{b}^k(t)$.

Stimulus d'un réseau

- $\mathbf{p}(t)$ désigne un vecteur stimulus présenté à l'entrée d'un réseau au temps t.
- $p_i(t)$ désigne l'élément i de $\mathbf{p}(t)$.

Niveaux d'activation d'une couche de neurones

- $\mathbf{n}^k(t)$ désigne le vecteur des niveaux d'activation pour la couche k d'un réseau au temps t.
- $n_i^k(t)$ désigne l'élément i de $\mathbf{n}^k(t)$.

Sorties d'une couche de neurones

- $\mathbf{a}^k(t)$ désigne un vecteur des sorties pour la couche k d'un réseau au temps t.
- $a_i^k(t)$ désigne l'élément i de $\mathbf{a}^k(t)$.

Cibles d'un réseau

- $\mathbf{d}(t)$ désigne un vecteur cible pour les sorties désirées d'un réseau au temps t.
- $d_i(t)$ désigne l'élément i de $\mathbf{d}(t)$.

Base d'apprentissage

- $\{(\mathbf{p}_1, \mathbf{d}_1), (\mathbf{p}_2, \mathbf{d}_2), \ldots, (\mathbf{p}_Q, \mathbf{d}_Q)\}$ désigne un ensemble de Q associations stimulus/cible pour l'apprentissage supervisé.

Signaux d'erreur

- $\mathbf{e}(t) = \mathbf{d}(t) - \mathbf{a}(t)$ désigne un vecteur mesurant l'erreur entre les sorties désirées (cible) et les sorties calculées d'un réseau au temps t.

– $e_i(t)$ désigne l'élément i de $\mathbf{e}(t)$.

Dimensions

– M désigne le nombre de couches d'un réseau.
– S^k désigne le nombre de neurones sur la couche k d'un réseau.
– Q désigne le nombre d'associations pour l'apprentissage.
– R désigne la dimension des stimulus d'entrée.

Fonctions de transfert d'une couche de neurones

– $\mathbf{f}^k(\mathbf{n}^k) = \mathbf{a}^k$ désigne le vecteur des sorties de la couche k, telles que calculées par la fonction de transfert f appliquée sur chacun des n_i^k, $i = 1, \ldots, S^k$.
– $f^k(n_i^k) = a_i^k$ désigne l'élément i de $\mathbf{f}^k(\mathbf{n}^k)$.
– $\dot{f}(n) = \frac{\partial}{\partial n} f(n)$ désigne la dérivée partielle de f par rapport à n.
– $\dot{\mathbf{F}}(\mathbf{n}) = \begin{bmatrix} \dot{f}(n_1) & 0 & \cdots & 0 \\ 0 & \dot{f}(n_2) & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}(n_S) \end{bmatrix}$.

Notations diverses

– $||\mathbf{x}||$ désigne la norme du vecteur $\mathbf{x}$.
– $F(\mathbf{x})$ désigne un indice de performance (une fonction) appliqué sur le vecteur $\mathbf{x}$.
– $\widehat{F}(\mathbf{x})$ désigne une approximation de $F(\mathbf{x})$.
– $\nabla F(\mathbf{x}) = \left[\frac{\partial F}{\partial x_1} \frac{\partial F}{\partial x_2} \cdots \frac{\partial F}{\partial x_n} \right]^T$ désigne le vecteur gradient de $F(\mathbf{x})$.
– $\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1 \partial x_1} F & \frac{\partial^2}{\partial x_1 \partial x_2} F & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} F \\ \frac{\partial^2}{\partial x_2 \partial x_1} F & \frac{\partial^2}{\partial x_2 \partial x_2} F & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} F \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F & \frac{\partial^2}{\partial x_n \partial x_2} F & \cdots & \frac{\partial^2}{\partial x_n \partial x_n} F \end{bmatrix}$ désigne la matrice «hessienne» de $F(\mathbf{x})$.
– λ_i désigne une valeur propre d'une matrice.
– $\mathbf{z}_i$ désigne un vecteur propre.

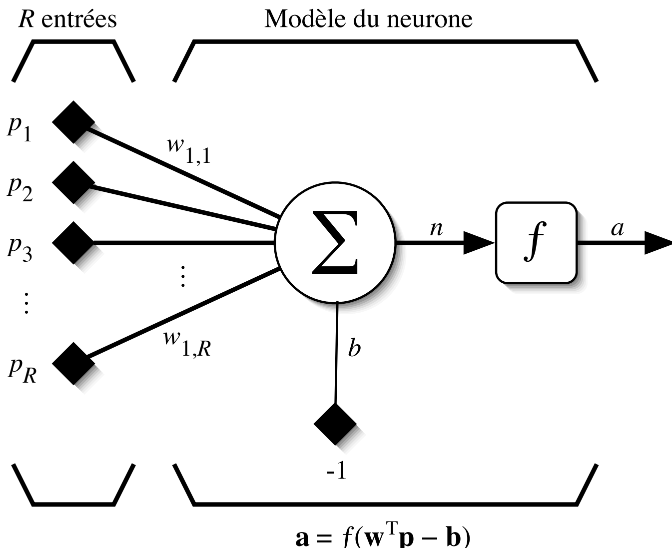


FIG. 2.1 – *Modèle d'un neurone artificiel.*

2.2 Modèle d'un neurone

Le modèle mathématique d'un neurone artificiel est illustré à la figure 2.1. Un neurone est essentiellement constitué d'un intégrateur qui effectue la somme pondérée de ses entrées. Le résultat n de cette somme est ensuite transformée par une fonction de transfert f qui produit la sortie a du neurone. En suivant les notations présentées à la section précédente, les R entrées du neurones correspondent au vecteur $\mathbf{p} = [p_1 p_2 \cdots p_R]^T$, alors que $\mathbf{w} = [w_{1,1} w_{1,2} \cdots w_{1,R}]^T$ représente le vecteur des poids du neurone. La sortie n de l'intégrateur est donnée par l'équation suivante :

$$\begin{aligned} n &= \sum_{j=1}^{R} w_{1,j}\, p_j - b \\ &= w_{1,1}\, p_1 + w_{1,2}\, p_2 + \cdots + w_{1,R}\, p_R - b, \end{aligned} \tag{2.1}$$

que l'on peut aussi écrire sous forme matricielle :

$$n = \mathbf{w}^T \mathbf{p} - b. \tag{2.2}$$

Cette sortie correspond à une somme pondérée des poids et des entrées moins ce qu'on nomme le biais b du neurone. Le résultat n de la somme pondérée s'appelle le niveau d'activation du neurone. Le biais b s'appelle aussi le seuil d'activation du neurone. Lorsque le niveau d'activation atteint ou dépasse le seuil b, alors l'argument de f devient positif (ou nul). Sinon, il est négatif.

On peut faire un parallèle entre ce modèle mathématique et certaines informations que l'on connait (ou que l'on croit connaître) à propos du neurone biologique. Ce dernier possède trois principales composantes : les dendrites, le corps cellulaire et l'axone (voir figure 2.2). Les dendrites forment un maillage de récepteurs nerveux qui permettent d'acheminer vers le corps du neurone des signaux électriques en provenance d'autres neurones. Celui-ci agit comme un espèce d'intégrateur en accumulant des charges électriques. Lorsque le neurone devient suffisamment excité (lorsque la charge accumulée dépasse un certain seuil), par un processus électrochimique,

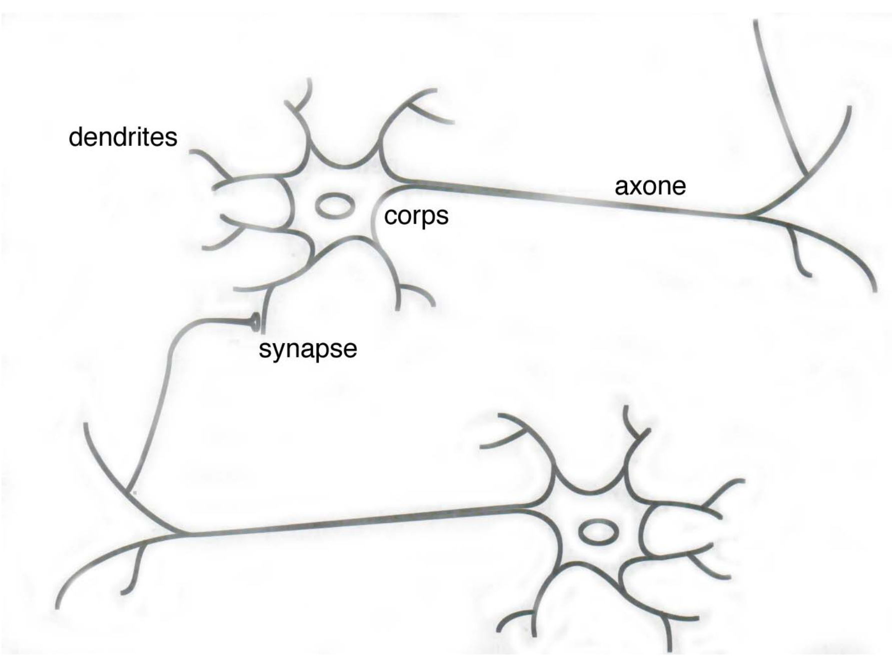


FIG. 2.2 – *Schéma d'un neurone biologique.*

il engendre un potentiel électrique qui se propage à travers son axone[1] pour éventuellement venir exciter d'autres neurones. Le point de contact entre l'axone d'un neurone et le dendrite d'un autre neurone s'appelle le synapse. Il semble que c'est l'arrangement spatial des neurones et de leur axone, ainsi que la qualité des connexions synaptiques individuelles qui détermine la fonction[2] précise d'un réseau de neurones biologique. C'est en se basant sur ces connaissances que le modèle mathématique décrit ci-dessus a été défini.

Un poids d'un neurone artificiel représente donc l'efficacité d'une connexion synaptique. Un poids négatif vient inhiber une entrée, alors qu'un poids positif vient l'accentuer. Il importe de retenir que ceci est une grossière approximation d'un véritable synapse qui résulte en fait d'un processus chimique très complexe et dépendant de nombreux facteurs extérieurs encore mal connus. Il faut bien comprendre que notre neurone artificiel est un modèle pragmatique qui, comme nous le verrons plus loin, nous permettra d'accomplir des tâches intéressantes. La vraisemblance biologique de ce modèle ne nous importe peu. Ce qui compte est le résultat que ce modèle nous permettra d'atteindre.

Un autre facteur limitatif dans le modèle que nous nous sommes donnés concerne son caractère discret. En effet, pour pouvoir simuler un réseau de neurones, nous allons rendre le temps discret dans nos équations. Autrement dit, nous allons supposer que tous les neurones sont synchrones, c'est-à-dire qu'à chaque temps t, ils vont simultanément calculer leur somme pondérée et produire une sortie $a(t) = f(n(t))$. Dans les réseaux biologiques, tous les neurones sont en fait asynchrones.

Revenons donc à notre modèle tel que formulé par l'équation 2.2 et ajoutons la fonction d'ac-

[1] Un axone peut être plus ou moins long selon le type de neurone.

[2] Notez bien, cependant, que des théories récentes remettent en cause cette hypothèse. Mais ceci sort du cadre du cours !

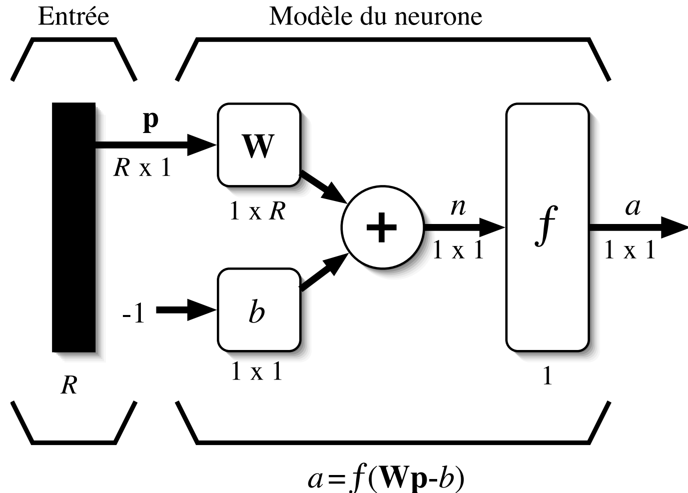


FIG. 2.3 – *Représentation matricielle du modèle d'un neurone artificiel.*

tivation f pour obtenir la sortie du neurone :

$$a = f(n) = f(\mathbf{w}^T\mathbf{p} - b). \tag{2.3}$$

En remplaçant $\mathbf{w}^T$ par une matrice $\mathbf{W} = \mathbf{w}^T$ d'une seule ligne, on obtient une forme générale que nous adopterons tout au long de cet ouvrage :

$$a = f(\mathbf{W}\mathbf{p} - b). \tag{2.4}$$

L'équation 2.4 nous amène à introduire un schéma de notre modèle plus compact que celui de la figure 2.1. La figure 2.3 illustre celui-ci. On y représente les R entrées comme un rectangle noir (le nombre d'entrées est indiqué sous le rectangle). De ce rectangle sort le vecteur $\mathbf{p}$ dont la dimension matricielle est $R \times 1$. Ce vecteur est multiplié par une matrice $\mathbf{W}$ qui contient les poids (synaptiques) du neurones. Dans le cas d'un neurone simple, cette matrice possède la dimension $1 \times R$. Le résultat de la multiplication correspond au niveau d'activation qui est ensuite comparé au seuil b (un scalaire) par soustraction. Finalement, la sortie du neurone est calculée par la fonction d'activation f. La sortie d'un neurone est toujours un scalaire.

2.3 Fonctions de transfert

Jusqu'à présent, nous n'avons pas spécifié la nature de la fonction d'activation de notre modèle. Il se trouve que plusieurs possibilités existent. Différentes fonctions de transfert pouvant être utilisées comme fonction d'activation du neurone sont énumérées au tableau 2.1. Les trois les plus utilisées sont les fonctions «seuil» (en anglais «hard limit»), «linéaire» et «sigmoïde».

Comme son nom l'indique, la fonction seuil applique un seuil sur son entrée. Plus précisément, une entrée négative ne passe pas le seuil, la fonction retourne alors la valeur 0 (on peut interpréter ce 0 comme signifiant *faux*), alors qu'une entrée positive ou nulle dépasse le seuil, et la fonction

TAB. 2.1 – *Fonctions de transfert* $a = f(n)$.

Nom de la fonction	Relation d'entrée/sortie	Icône	Nom Matlab
seuil	$a = 0 \quad \text{si } n < 0$ $a = 1 \quad \text{si } n \geq 0$		hardlim
seuil symétrique	$a = -1 \quad \text{si } n < 0$ $a = 1 \quad \text{si } n \geq 0$		hardlims
linéaire	$a = n$		purelin
linéaire saturée	$a = 0 \quad \text{si } n < 0$ $a = n \quad \text{si } 0 \leq n \leq 1$ $a = 1 \quad \text{si } n > 1$		satlin
linéaire saturée symétrique	$a = -1 \quad \text{si } n < -1$ $a = n \quad \text{si } -1 \leq n \leq 1$ $a = 1 \quad \text{si } n > 1$		satlins
linéaire positive	$a = 0 \quad \text{si } n < 0$ $a = n \quad \text{si } n \geq 0$		poslin
sigmoïde	$a = \frac{1}{1+\exp^{-n}}$		logsig
tangente hyperbolique	$a = \frac{e^n - e^{-n}}{e^n + e^{-n}}$		tansig
compétitive	$a = 1 \quad \text{si } n \text{ maximum}$ $a = 0 \quad \text{autrement}$	C	compet

retourne 1 (vrai). Utilisée dans le contexte d'un neurone, cette fonction est illustrée à la figure 2.4a. On remarque alors que le biais b dans l'expression de $a = \text{hardlim}(\mathbf{w}^T\mathbf{p} - b)$ (équation 2.4) détermine l'emplacement du seuil sur l'axe $\mathbf{w}^T\mathbf{p}$, où la fonction passe de 0 à 1. Nous verrons plus loin que cette fonction permet de prendre des décisions binaires.

La fonction linéaire est très simple, elle affecte directement son entrée à sa sortie :

$$a = n. \tag{2.5}$$

Appliquée dans le contexte d'un neurone, cette fonction est illustrée à la figure 2.4b. Dans ce cas, la sortie du neurone correspond à son niveau d'activation dont le passage à zéro se produit lorsque $\mathbf{w}^T\mathbf{p} = b$.

La fonction de transfert sigmoïde est quant à elle illustrée à la figure 2.4c. Son équation est donnée par :

$$a = \frac{1}{1 + \exp^{-n}}. \tag{2.6}$$

Elle ressemble soit à la fonction seuil, soit à la fonction linéaire, selon que l'on est loin ou près de b, respectivement. La fonction seuil est très non-linéaire car il y a une discontinuité lorsque $\mathbf{w}^T\mathbf{p} = b$. De son côté, la fonction linéaire est tout à fait linéaire. Elle ne comporte aucun changement de pente. La sigmoïde est un compromis intéressant entre les deux précédentes. Notons finalement, que la fonction «tangente hyperbolique» est une version symétrique de la sigmoïde.

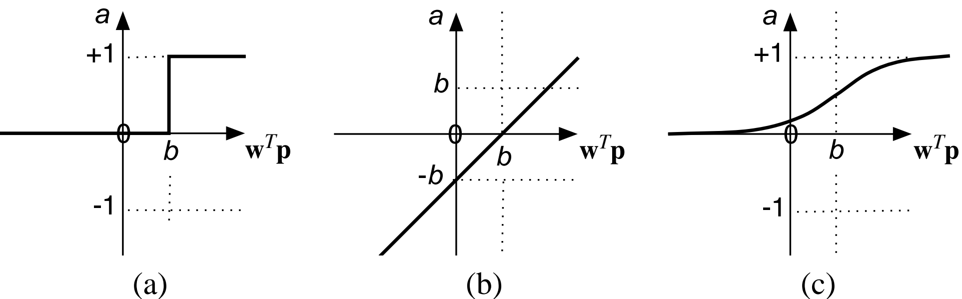


FIG. 2.4 – *Fonction de transfert : (a) du neurone «seuil»; (b) du neurone «linéaire», et (c) du neurone «sigmoïde».*

2.4 Architecture de réseau

Un réseau de neurones est un maillage de plusieurs neurones, généralement organisé en couches. Pour construire une couche de S neurones, il s'agit simplement de les assembler comme à la figure 2.5. Les S neurones d'une même couche sont tous branchés aux R entrées. On dit alors que la couche est totalement connectée. Un poids $w_{i,j}$ est associé à chacune des connexions. Nous noterons toujours le premier indice par i et le deuxième par j (jamais l'inverse). Le premier indice (rangée) désigne toujours le numéro de neurone sur la couche, alors que le deuxième indice (colonne) spécifie le numéro de l'entrée. Ainsi, $w_{i,j}$ désigne le poids de la connexion qui relie le neurone i à son entrée j. L'ensemble des poids d'une couche forme donc une matrice $\mathbf{W}$ de dimension $S \times R$:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} \tag{2.7}$$

Notez bien que $S \neq R$, dans le cas général (les nombres de neurones et d'entrées sont indépendants). Si l'on considère que les S neurones forment un vecteur de neurones, alors on peut créer les vecteurs $\mathbf{b} = [b_1 b_2 \cdots b_S]^T$, $\mathbf{n} = [n_1 n_2 \cdots n_S]^T$ et $\mathbf{a} = [a_1 a_2 \cdots a_S]^T$. Ceci nous amène à la représentation graphique simplifiée, illustrée à la figure 2.6. On y retrouve, comme à la figure 2.3, les mêmes vecteurs et matrice. La seule différence se situe au niveau de la taille, ou plus précisément du nombre de rangées (S), de $\mathbf{b}$, $\mathbf{n}$, $\mathbf{a}$ et $\mathbf{W}$.

Finalement, pour construire un réseau, il ne suffit plus que de combiner des couches comme à la figure 2.7. Cet exemple comporte R entrées et trois couches de neurones comptant respectivement S^1, S^2 et S^3 neurones. Dans le cas général, de nouveau, $S^1 \neq S^2 \neq S^3$. Chaque couche possède sa propre matrice de poids $\mathbf{W}^k$, où k désigne l'indice de couche. Dans le contexte des vecteurs et des matrices relatives à une couche, nous emploierons toujours un exposant pour désigner cet indice. Ainsi, les vecteurs $\mathbf{b}^k$, $\mathbf{n}^k$ et $\mathbf{a}^k$ sont aussi associés à la couche k.

Il importe de remarquer dans cet exemple que les couches qui suivent la première ont comme entrée la sortie de la couche précédente. Ainsi, on peut enfiler autant de couche que l'on veut,

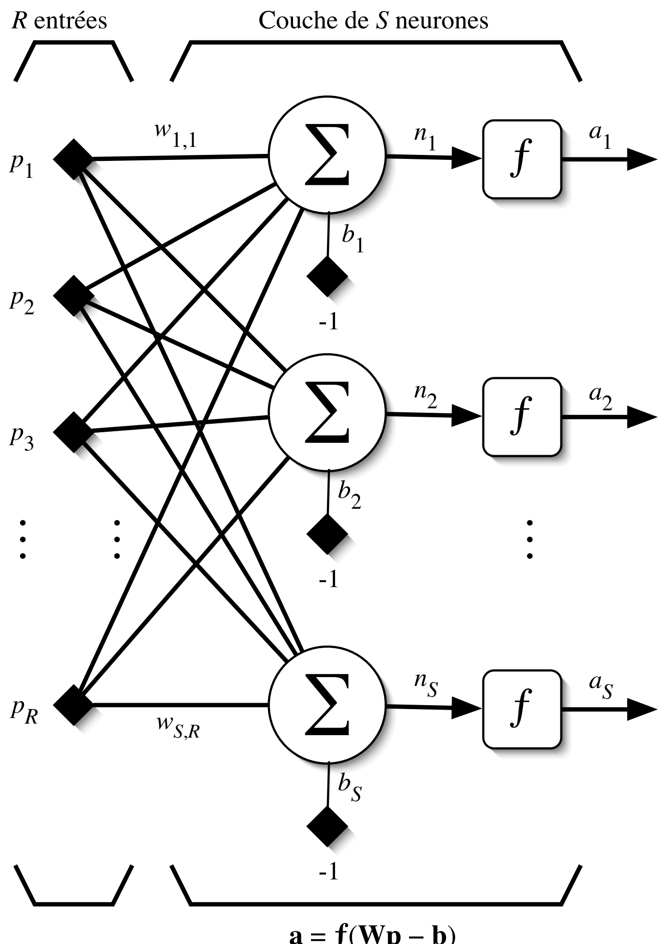


FIG. 2.5 – *Couche de S neurones.*

du moins en théorie. Nous pouvons aussi fixer un nombre quelconque de neurones sur chaque couche. En pratique, nous verrons plus tard qu'il n'est cependant pas souhaitable d'utiliser trop de neurones. Finalement, notez aussi que l'on peut changer de fonction de transfert d'une couche à l'autre. Ainsi, toujours dans le cas général, $\mathbf{f}^1 \neq \mathbf{f}^2 \neq \mathbf{f}^3$.

La dernière couche est nommée «couche de sortie». Les couches qui précèdent la couche de sortie sont nommées «couches cachées». Nous verrons un peu plus tard pourquoi. Le réseau de la figure 2.7 possède donc deux couches cachées et une couche de sortie.

Les réseaux multicouches sont beaucoup plus puissants que les réseaux simples à une seule couche. En utilisant deux couches (une couche cachée et une couche de sortie), à condition d'employer une fonction d'activation sigmoïde sur la couche cachée, on peut entraîner un réseau à produire une approximation de la plupart des fonctions, avec une précision arbitraire (cela peut cependant requérir un grand nombre de neurones sur la couche cachée). Sauf dans de rares cas, les réseaux de neurones artificiels exploitent deux ou trois couches.

Entraîner un réseau de neurones signifie modifier la valeur de ses poids et de ses biais pour qu'il réalise la fonction entrée/sortie désirée. Nous étudierons en détails, dans des chapitres subséquents,

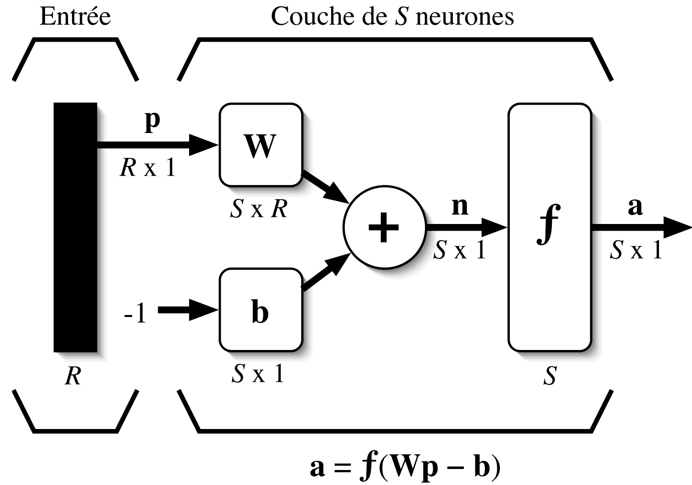


FIG. 2.6 – *Représentation matricielle d'une couche de S neurones.*

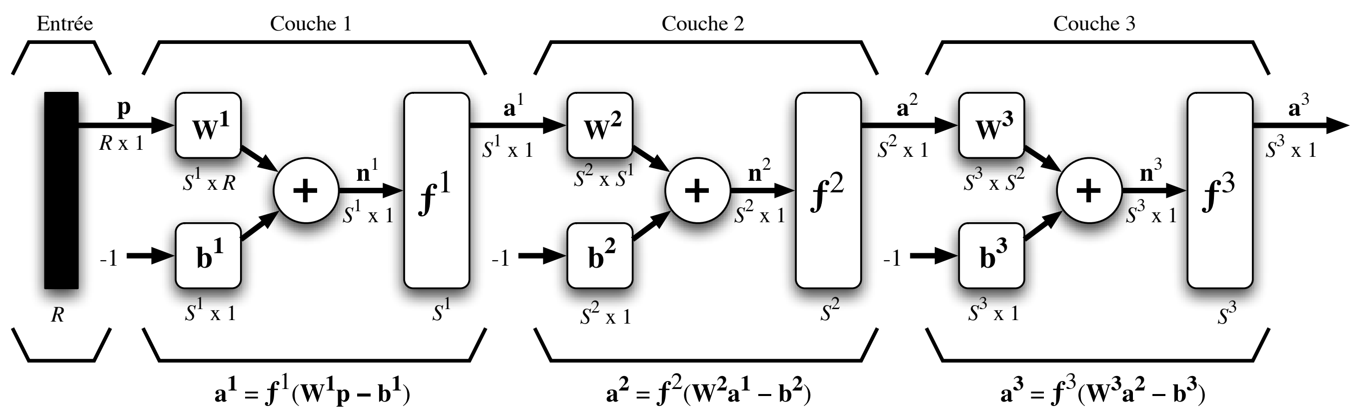


FIG. 2.7 – *Représentation matricielle d'un réseau de trois couches.*

différents algorithmes pour y parvenir dans différents contextes. Pour spécifier la structure du réseau, il faut aussi choisir le nombre de couches et le nombre de neurones sur chaque couche. Tout d'abord, rappelons que le nombre d'entrées du réseau (R), de même que le nombre de neurones sur la couche de sortie est fixé par les spécifications du problème que l'on veut résoudre avec ce réseau. Par exemple, si la donnée du problème comporte quatre variables en entrée et qu'elle exige de produire trois variables en sortie, alors nous aurons simplement $R = 4$ et $S^M = 3$, où M correspond à l'indice de la couche de sortie (ainsi qu'au nombre de couches). Ensuite, la nature du problème peut aussi nous guider dans le choix des fonctions de transfert. Par exemple, si l'on désire produire des sorties binaires 0 ou 1, alors on choisira probablement une fonction seuil (voir tableau 2.1, page 11) pour la couche de sortie. Il reste ensuite à choisir le nombre de couches cachées ainsi que le nombre de neurones sur ces couches, et leur fonction de transfert. Il faudra aussi fixer les différents paramètres de l'algorithme d'apprentissage. Mais nous y reviendrons en temps et lieu !

Finalement, la figure 2.8 illustre le dernier élément de construction que nous emploierons

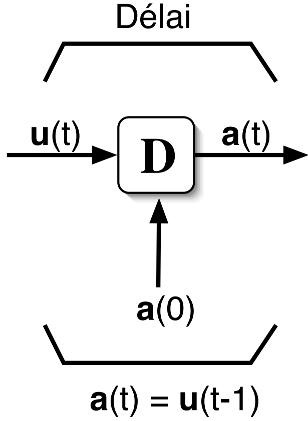


FIG. 2.8 – *Élément de retard.*

pour bâtir des réseaux dit «récurrents». Il s'agit d'un registre à décalage qui permet d'introduire un retard dans une donnée que l'on veut acheminer dans un réseau. La sortie retardée $\mathbf{a}(t)$ prend la valeur de l'entrée $\mathbf{u}$ au temps $t-1$. Cet élément de retard présuppose que l'on peut initialiser la sortie au temps $t=0$ avec la valeur $\mathbf{a}(0)$. Cette condition initiale est indiquée à la figure 2.8 par une flèche qui entre par le bas de l'élément.

Avant de passer à la décription des architectures concrètes et de leur algorithmes d'apprentissage, nous allons d'abord effectuer au chapitre suivant quelques rappels sur l'algèbre linéaire. En effet, comme le lecteur attentif l'aura deviné, nous allons avoir à manipuler beaucoup de matrices et d'opérations matricielles tout au long de ces chapitres. La notation matricielle peut être très puissante, parce que compacte, mais également obscure lorsqu'on ne la maîtrise pas bien, d'où l'importance de ces rappels.

Chapitre 3

Algèbre linéaire

Dans ce chapitre, nous faisons plusieurs rappels essentiels en algèbre linéaire. Au chapitre précédent, nous avons vu que les entrées et les sorties d'un réseau de neurones, ainsi que les rangées de ses matrices de poids forment des vecteurs. Il est donc important de bien comprendre ce qu'est un espace vectoriel en étudiant ses principales propriétés. Ensuite, nous aborderons des outils algébriques de base tels les transformations linéaires, les changements de base ainsi que les valeurs et vecteurs propres. Ces outils serviront par la suite tout au long des chapitres subséquents.

3.1 Définition d'un espace vectoriel

Lorsque nous définissons un vecteur $\mathbf{x} = [x_1 x_2 \cdots x_n]^T$, nous faisons habituellement référence à un espace euclidien de n dimensions, que nous notons $\Re^n$. Cependant, la notion d'espace vectoriel est beaucoup plus vaste que ce dernier qui ne représente qu'un cas particulier.

Définition. Un *espace vectoriel* linéaire $\mathcal{X}$ est un ensemble d'éléments (de vecteurs) défini sur un champ scalaire $\mathcal{F}$, et respectant les propriétés suivantes :

1. possède un opérateur d'addition tel que :
   (a) $\mathbf{x}, \mathbf{y} \in \mathcal{X}$ implique $\mathbf{x} + \mathbf{y} \in \mathcal{X}$;
   (b) $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$; (commutativité)
   (c) $(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})$; (associativité)
   (d) $\exists \mathbf{0} \in \mathcal{X}$ tel que $\mathbf{x} + \mathbf{0} = \mathbf{x}, \forall \mathbf{x} \in \mathcal{X}$; (élément neutre)
   (e) $\forall \mathbf{x} \in \mathcal{X}, \exists$-$\mathbf{x}$ tel que $\mathbf{x} + (\text{-}\mathbf{x}) = \mathbf{0}$; (élément inverse)
2. possède un opérateur de multiplication tel que :
   (a) $a \in \mathcal{F}$ et $\mathbf{x} \in \mathcal{X}$ implique $a\mathbf{x} \in \mathcal{X}$;
   (b) $\forall \mathbf{x} \in \mathcal{X}$ et le scalaire 1, $1\mathbf{x} = \mathbf{x}$; (élément neutre)

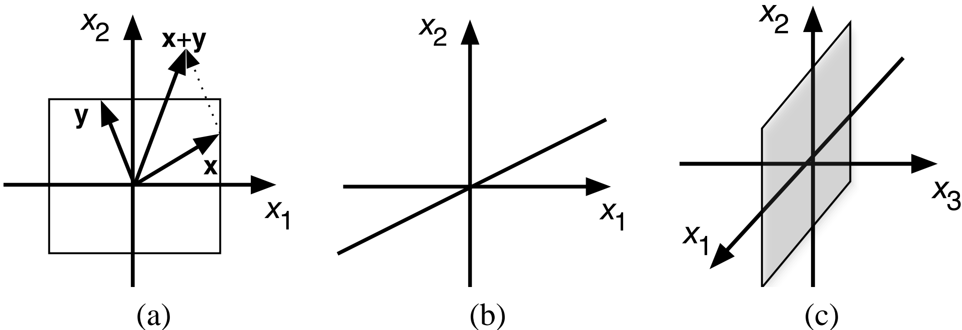


FIG. 3.1 – *Différents sous-ensembles de $\Re^2$: (a) région rectangulaire ; (b) droite ; (c) plan.*

(c) $\forall a, b \in \mathcal{F}$ et $\forall \mathbf{x} \in \mathcal{X}$, $a(b\mathbf{x}) = (ab)\mathbf{x}$; (associativité)

(d) $(a + b)\mathbf{x} = a\mathbf{x} + b\mathbf{x}$; (distributivité)

(e) $a(\mathbf{x} + \mathbf{y}) = a\mathbf{x} + a\mathbf{y}$; (distributivité)

Il est facile de démontrer que ces propriétés sont respectées pour $\Re^n$ et, par conséquent, $\Re^2$. On peut cependant se poser la question à propos de certains sous-ensembles de $\Re^2$. Par exemple, considérons la région rectangulaire illustrée à la figure 3.1a. Ce sous-ensemble de $\Re^2$ n'est pas un espace vectoriel car, entre autres, la propriété 1a n'est pas respectée. En effet, si l'on prend deux vecteurs à l'intérieur du rectangle et qu'on les additionne, il se peut que le résultat sorte du rectangle. Par contre, on peut montrer (et ceci est laissée en exercice) que la droite infinie illustrée à la figure 3.1b respecte toutes les propriétés énumérées ci-dessus et, par conséquent, défini un espace vectoriel. Notez bien, cependant, que cette droite se doit de passer par l'origine, sinon la propriété 1d ne serait pas respectée.

Un autre exemple d'un espace vectoriel est l'ensemble $\mathcal{P}^2$ des polynômes de degré 2 ou moins. Par exemple, deux éléments de cet espace sont :

$$\mathbf{x} = 3 + 2t + t^2, \quad (3.1)$$

$$\mathbf{y} = 5 - t. \quad (3.2)$$

Cet ensemble respecte les 10 propriétés d'un espace vectoriel. En effet, si l'on additionne deux polynômes de degré 2 ou moins, on obtient un autre polynôme de degré 2 ou moins. On peut aussi multiplier un polynôme par un scalaire sans changer l'ordre de celui-ci, etc. En notation vectorielle, on peut donc représenter les deux polynômes de l'exemple par $\mathbf{x} = [3\ 2\ 1]^T$ et $\mathbf{y} = [5\ \text{-}1\ 0]^T$.

Mentionnons qu'on peut aussi former des espaces vectoriels avec des ensembles de fonctions plus générales que des polynômes. Il importe seulement de respecter les 10 propriétés fondamentales d'un espace vectoriel ! Si nous prenons la peine de préciser cette définition formelle, c'est parce que la résolution d'un problème avec un réseau de neurones requiert toujours de pouvoir représenter ce problème à l'aide d'un espace vectoriel. C'est donc une notion tout à fait fondamentale à ce sujet d'étude.

3.1.1 Dépendance linéaire

Soient les n vecteurs $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$. Alors ces vecteurs sont linéairement dépendants s'il existe n scalaires $a_1, a_2, \ldots, a_n$ tels qu'au moins un d'eux est non nul et que :

$$a_1\mathbf{x}_1 + a_2\mathbf{x}_2 + \cdots + a_n\mathbf{x}_n = 0. \tag{3.3}$$

Et réciproquement, si $a_1\mathbf{x}_1 + a_1\mathbf{x}_1 + \cdots + a_1\mathbf{x}_1 = 0$ implique que $\forall i, a_i = 0$, alors les vecteurs sont (linéairement) indépendants.

Par exemple, les vecteurs suivants :

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, \ \mathbf{x}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \tag{3.4}$$

sont linéairement indépendants car si $a_1\mathbf{x}_1 + a_2\mathbf{x}2 = 0$, alors :

$$\begin{bmatrix} a_1 + a_2 \\ \text{-}a_1 + a_2 \\ \text{-}a_1 - a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \tag{3.5}$$

et $a_1 + a_2 = 0$ implique que $a_1 = -a_2$, et $-a_1 + a_2 = 0$ implique $a_1 = a_2$. Ainsi, il faut que $a_1 = a_2 = 0$.

Par contre, les polynômes $\mathbf{p}_1 = 1 + t + t^2$, $\mathbf{p}_2 = 2 + 2t + t^2$ et $\mathbf{p}_3 = 1 + t$ sont linéairement dépendants puisque $a_1\mathbf{p}_1 + a_2\mathbf{p}_2 + a_3\mathbf{p}_3 = 0$ pour $a_1 = 1$, $a_2 = -1$ et $a_3 = 1$.

3.1.2 Bases et dimensions

La dimension d'un espace vectoriel est déterminée par le nombre minimum de vecteurs de base requis pour couvrir l'espace vectoriel en entier. On dit d'un ensemble de vecteur $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ qu'il couvre un espace vectoriel $\mathcal{X}$ si et seulement si tous les $\mathbf{x} \in \mathcal{X}$ de cet espace peuvent être exprimés comme une combinaison linéaire des vecteurs de base :

$$\mathbf{x} = a_1\mathbf{u}_1 + a_2\mathbf{u}_2 + \cdots + a_n\mathbf{u}_n. \tag{3.6}$$

Par exemple, dans $\Re^2$, nous avons l'habitude de travailler avec les vecteurs de base $\mathbf{u}_1 = [1\ 0]$ et $\mathbf{u}_2 = [0\ 1]$, mais ce n'est pas la seule possibilité. Un autre choix serait $[0.5\ 0.5]$ et $[-0.5\ 0.5]$ ou encore $[2\ 0]$ et $[0\ 2]$. La seule chose qui importe est que les vecteurs de base soient linéairement indépendants.

Pour notre espace $\mathcal{P}^2$ des polynômes de degré 2 ou moins, on peut choisir autant $\{1, t, t^2\}$ que $\{1, 1 + t, 1 + t + t^2\}$, par exemple.

3.1.3 Produit scalaire

Le produit scalaire entre deux vecteurs $\mathbf{x}$ et $\mathbf{y}$, que nous noterons $<\mathbf{x}, \mathbf{y}>$, est une opération très importante pour les réseaux de neurones. N'importe quelle fonction scalaire prenant deux vecteurs comme argument et respectant les trois propriétés suivantes peut servir à définir un produit scalaire :

1. $<\mathbf{x}, \mathbf{y}> = <\mathbf{y}, \mathbf{x}>$;
2. $<\mathbf{x}, (a\mathbf{y}_1 + b\mathbf{y}_2)> = a<\mathbf{x}, \mathbf{y}_1> + b<\mathbf{x}, \mathbf{y}_2>$;
3. $<\mathbf{x}, \mathbf{x}> \geq 0$, avec $<\mathbf{x}, \mathbf{x}> = 0$ uniquement pour $\mathbf{x} = \mathbf{0}$;

La première propriété spécifie qu'un produit scalaire doit être symétrique. La deuxième précise que le produit d'un vecteur par une combinaison linéaire de deux vecteurs est égale à la combinaison linéaire des produits scalaires. Finalement, la troisième propriété restreint le produit scalaire d'un vecteur avec lui-même aux valeurs positives, sauf pour le vecteur nul qui doit donner zéro.

Le produit scalaire habituellement utilisé sur $\Re^n$ est défini par :

$$<\mathbf{x}, \mathbf{y}> = \mathbf{x}^T\mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_ny_n. \tag{3.7}$$

3.1.4 Norme

La norme d'un vecteur est une mesure de longueur. La fonction scalaire $||\mathbf{x}||$ s'appelle une norme si elle satisfait aux quatre propriétés suivantes :

1. $||\mathbf{x}|| \geq 0$;
2. $||\mathbf{x}|| = 0$ si, et seulement si, $\mathbf{x} = \mathbf{0}$;
3. $||a\mathbf{x}|| = |a|\,||\mathbf{x}||$;
4. $||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}||$;

La première propriété spécifie qu'une norme est toujours positive ou nulle. La deuxième précise qu'elle n'est nulle que pour le vecteur nul. La troisième impose que la norme d'un vecteur multiplié par un scalaire soit (linéairement) proportionnelle à ce scalaire. Finalement, la dernière propriété impose que la norme d'une somme de deux vecteurs soit inférieure ou égale à la somme des normes.

La norme la plus souvent utilisée, nommée l_2, est définie par $||\mathbf{x}|| = \sqrt{<\mathbf{x}, \mathbf{x}>}$, ce qui dans un espace euclidien $\Re^n$ correspond à la norme euclienne habituelle :

$$||\mathbf{x}|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \tag{3.8}$$

Beaucoup d'autres normes sont possibles telle que, par exemple, $||\mathbf{x}|| = |x_1| + |x_2| + \cdots + |x_n|$. Cette dernière s'appelle norme l_1 (on dit aussi «city block» ou encore «manhattan»). Dans le cas général, il existe les normes dites l_p telles que :

$$||\mathbf{x}|| = \sqrt[p]{|x_1|^p + |x_2|^p + \cdots + |x_n|^p}, \tag{3.9}$$

où $p \geq 1$. Dans le cas où $p \rightarrow \infty$, on obtient la norme l_∞ suivante :

$$||\mathbf{x}|| = \max_i |x_i|. \tag{3.10}$$

Ceci nous indique que plus p devient grand, plus on attache de l'importance aux grandes composantes de $\mathbf{x}$. À la limite, on ne tient compte que de la plus grande composante du vecteur.

Finalement, mentionnons qu'il importe parfois de «normaliser» nos vecteurs en les divisant par leur norme :

$$\left|\left|\frac{\mathbf{x}}{||\mathbf{x}||}\right|\right| = 1. \tag{3.11}$$

On obtient alors un vecteur qui pointe dans la même direction qu'auparavant mais dont la norme est unitaire.

Les concepts de produit scalaire et de norme permettent aussi d'introduire la notion d'angle θ entre deux vecteurs $\mathbf{x}$ et $\mathbf{y}$ via la fameuse loi des cosinus :

$$<\mathbf{x}, \mathbf{y}> = ||\mathbf{x}||\,||\mathbf{y}|| \cos\theta. \tag{3.12}$$

3.1.5 Orthogonalité

Deux vecteurs $\mathbf{x}$ et $\mathbf{y}$ sont dits orthogonaux si leur produit scalaire $<\mathbf{x}, \mathbf{y}>$ est nul ($\theta = 90°$).

Un vecteur $\mathbf{x} \in \mathcal{X}$ est aussi dit orthogonal à un sous-espace $\mathcal{X}' \subset \mathcal{X}$ lorsqu'il est orthogonal avec tous les vecteurs $\mathbf{x}'$ de ce sous-espace. Par exemple, un plan dans $\Re^3$ définit un sous-espace de dimension 2 pour lequel il existe un vecteur perpendiculaire (orthogonal) à ce plan (voir figure 3.1c).

Parfois, il importe de convertir un ensemble de n vecteurs indépendants $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ en n vecteurs orthogonaux $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$. On peut effectuer cette opération avec la méthode de Gram-Schmidt. Le premier vecteur orthogonal peut être quelconque, nous choisissons donc le premier vecteur indépendant :

$$\mathbf{v}_1 = \mathbf{x}_1. \tag{3.13}$$

Pour le second vecteur orthogonal, on utilise $\mathbf{x}_2$, mais après avoir soustrait de $\mathbf{x}_2$, la portion du vecteur qui est dans la direction de $\mathbf{v}_1$. On obtient :

$$\mathbf{v}_2 = \mathbf{x}_2 - a\mathbf{v}_1, \tag{3.14}$$

où a est choisi de manière à ce que $\mathbf{v}_2$ soit orthogonal à $\mathbf{v}_1$. Ceci implique que :

$$<\mathbf{v}_1, \mathbf{v}_2> = <\mathbf{v}_1, (\mathbf{x}_2 - a\mathbf{v}_1)> = <\mathbf{v}_1, \mathbf{x}_2> - a<\mathbf{v}_1, \mathbf{v}_1> = 0 \tag{3.15}$$

et :

$$a = \frac{<\mathbf{v}_1, \mathbf{x}_2>}{<\mathbf{v}_1, \mathbf{v}_1>}. \tag{3.16}$$

Ainsi, pour trouver la composante de $\mathbf{x}_2$ dans la direction de $\mathbf{v}_1$, c'est-à-dire $a\mathbf{v}_1$, il s'agit de calculer le produit scalaire entre les deux vecteurs. Ceci s'appelle le *projecteur* de $\mathbf{x}_2$ sur $\mathbf{v}_1$ (voir figure 3.2). Si l'on continue ce processus, le $k^{\text{ème}}$ vecteur orthogonal est obtenu par l'expression :

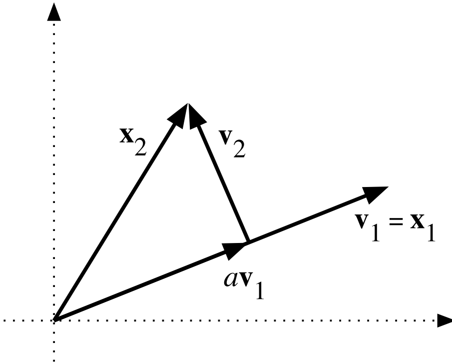


FIG. 3.2 – *Illustration de la méthode de transformation orthogonale Gram-Schmidt.*

$$\mathbf{v}_k = \mathbf{x}_k - \sum_{i=1}^{k-1} \frac{<\mathbf{v}_i, \mathbf{x}_k>}{<\mathbf{v}_i, \mathbf{v}_i>} \mathbf{v}_i. \quad (3.17)$$

3.2 Transformations linéaires

Une transformation linéaire $\mathcal{A}$ est une application d'un espace vectoriel $\mathcal{X}$ vers un espace vectoriel $\mathcal{Y}$ telle que :

1. $\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}, \mathcal{A}(\mathbf{x}_1 + \mathbf{x}_2) = \mathcal{A}(\mathbf{x}_1) + \mathcal{A}(\mathbf{x}_2)$;
2. $\forall \mathbf{x} \in \mathcal{X}, a \in \Re, \mathcal{A}(a\mathbf{x}) = a\mathcal{A}(\mathbf{x})$.

La première propriété spécifie que la transformée d'une somme de vecteurs doit être égale à la somme des transformées, pour qu'elle soit linéaire. La deuxième propriété précise que la transformée d'un vecteur auquel on a appliqué un facteur d'échelle doit aussi être égale à ce facteur appliqué sur la transformée du vecteur original. Si l'une ou l'autre de ces deux propriétés n'est pas respectée, la transformation n'est pas linéaire.

3.2.1 Représentations matricielles

Nous allons maintenant montrer que toute transformation linéaire peut être représentée par une matrice. Soient $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ les vecteurs de base pour $\mathcal{X}$ et $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m\}$ ceux de $\mathcal{Y}$. Avec ces bases, nous pouvons représenter n'importe quels vecteurs $\mathbf{x} \in \mathcal{X}$ et $\mathbf{y} \in \mathcal{Y}$ avec les combinaisons linéaires suivantes :

$$\mathbf{x} = \sum_{j=1}^{n} x_j \mathbf{v}_j \text{ et } \mathbf{y} = \sum_{i=1}^{m} y_i \mathbf{u}_i. \quad (3.18)$$

Soit la transformation linéaire $\mathcal{A}$ qui applique $\mathcal{X}$ sur $\mathcal{Y}$ ($\mathcal{A} : \mathcal{X} \longrightarrow \mathcal{Y}$). Donc :

$$\mathcal{A}(\mathbf{x}) = \mathbf{y}, \quad (3.19)$$

que l'on peut ré-écrire de la façon suivante :

$$\mathcal{A}\left(\sum_{j=1}^{n} x_j \mathbf{v}_j\right) = \sum_{i=1}^{m} y_i \mathbf{u}_i. \tag{3.20}$$

Mais puisque $\mathcal{A}$ est un opérateur linéaire, on peut aussi écrire :

$$\sum_{j=1}^{n} x_j \mathcal{A}(\mathbf{v}_j) = \sum_{i=1}^{m} y_i \mathbf{u}_i. \tag{3.21}$$

En considérant maintenant que les vecteurs $\mathcal{A}(\mathbf{v}_j)$ sont des éléments de $\mathcal{Y}$, on peut les ré-écrire en tant qu'une combinaison linéaire de ses vecteurs de base :

$$\mathcal{A}(\mathbf{v}_j) = \sum_{i=1}^{m} a_{ij} \mathbf{u}_i, \tag{3.22}$$

et en substituant l'équation 3.22 dans l'équation 3.21, on obtient :

$$\sum_{j=1}^{n} x_j \sum_{i=1}^{m} a_{ij} \mathbf{u}_i = \sum_{i=1}^{m} y_i \mathbf{u}_i. \tag{3.23}$$

En inversant l'ordre des sommations, on peut écrire :

$$\sum_{i=1}^{m} \mathbf{u}_i \sum_{j=1}^{n} a_{ij} x_j = \sum_{i=1}^{m} y_i \mathbf{u}_i, \tag{3.24}$$

et en réarrangeant cette dernière équation, on produit le résultat :

$$\sum_{i=1}^{m} \mathbf{u}_i \left(\sum_{j=1}^{n} a_{ij} x_j - y_i\right) = 0. \tag{3.25}$$

Finalement, en se rappelant que les vecteurs de base $\mathbf{u}_i$ doivent être indépendants, on peut conclure que leurs coefficients doivent forcément être nuls, donc :

$$\sum_{j=1}^{n} a_{ij} x_j = y_i. \tag{3.26}$$

Ce qui correspond au produit de matrice :

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ x_2 \\ \vdots \\ y_m \end{bmatrix}, \tag{3.27}$$

que l'on peut noter $\mathbf{A}\mathbf{x} = \mathbf{y}$.

Autrement dit, toute transformation linéaire peut être décrite par une matrice $\mathbf{A}$ qu'il s'agit de multiplier avec le vecteur que l'on veut transformer, pour obtenir le vecteur résultant de la transformation.

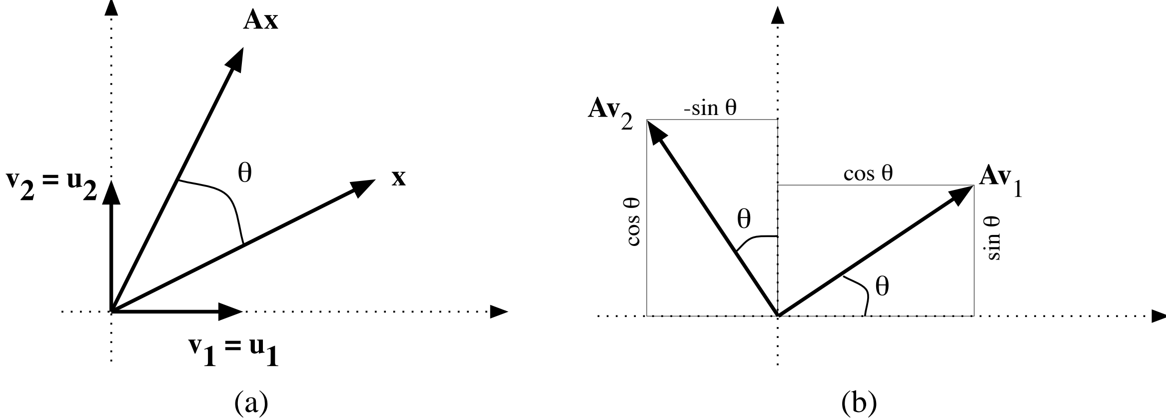


FIG. 3.3 – *Transformation de rotation : (a) rotation du vecteur* $\mathbf{x}$ *; (b) rotation des vecteurs de base* $\mathbf{v}_1$ *et* $\mathbf{v}_2$.

Comme exemple de transformation linéaire, considérons la rotation qui consiste à faire tourner un vecteur autour de l'origine. Pour simplifier, utilisons $\mathcal{X} = \mathcal{Y} = \Re^2$ et travaillons avec les vecteurs de base habituels, c'est-à-dire ceux du plan cartésien illustrés à la figure 3.3a. La clef ici est de transformer chaque vecteur de base comme à la figure 3.3b. Ceci s'accomplit grâce à l'équation 3.20 :

$$\mathcal{A}(\mathbf{v}_1) = \cos(\theta)\mathbf{v}_1 + \sin(\theta)\mathbf{v}_2 = a_{11}\mathbf{v}_1 + a_{21}\mathbf{v}_2, \tag{3.28}$$

$$\mathcal{A}(\mathbf{v}_2) = -\sin(\theta)\mathbf{v}_1 + \cos(\theta)\mathbf{v}_2 = a_{12}\mathbf{v}_1 + a_{22}\mathbf{v}_2. \tag{3.29}$$

Ce qui nous donne les deux colonnes d'une matrice de rotation $\mathbf{A}$ dans $\Re^2$:

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.30}$$

3.2.2 Changement de base

Il est important de noter que la représentation matricielle d'une transformation linéaire n'est pas unique car elle dépend des vecteurs de base. Dans cette sous-section, nous allons examiner ce qu'il advient d'une transformation lorsqu'on effectue un changement de base.

Soit la transformation linéaire $\mathcal{A} : \mathcal{X} \longrightarrow \mathcal{Y}$ et l'ensemble $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ des vecteurs de base de $\mathcal{X}$, et $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m\}$ ceux de $\mathcal{Y}$. Par conséquent :

$$\mathbf{A}\mathbf{x} = \mathbf{y}, \forall \mathbf{x} \in \mathcal{X}. \tag{3.31}$$

Supposons maintenant que l'on veuille changer de base pour $\mathcal{X}$ et $\mathcal{Y}$. Soient $\{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n\}$ et $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m\}$ les nouveaux ensembles de vecteurs de base. Avec ces bases, nous aurons $\mathbf{A}'\mathbf{x}' = \mathbf{y}', \forall \mathbf{x}' \in \mathcal{X}$, où $\mathbf{x} = \mathbf{B}_t\mathbf{x}'$, $\mathbf{y} = \mathbf{B}_w\mathbf{y}'$, $\mathbf{B}_t = [\mathbf{t}_1\mathbf{t}_2 \cdots \mathbf{t}_n]$ et $\mathbf{B}_w = [\mathbf{w}_1\mathbf{w}_2 \cdots \mathbf{w}_m]$. En substituant ces résultats dans l'équation 3.31, on obtient l'expression suivante :

$$\mathbf{A}\mathbf{B}_t\mathbf{x}' = \mathbf{B}_w\mathbf{y}' \tag{3.32}$$

puis en multipliant de part et d'autre par $\mathbf{B}_w^{-1}$:

$$(\mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t)\mathbf{x}' = \mathbf{y}', \tag{3.33}$$

ce qui implique que $\mathbf{A}' = \mathbf{B}_w^{-1}\mathbf{A}\mathbf{B}_t$.

On doit retenir qu'en changeant de base pour représenter nos vecteurs, nous changerons aussi la représentation matricielle de nos transformations. Le résultat sera le même car les deux transformations sont similaires. Seule la représentation change (les colonnes de nombres). L'intérêt d'un changement de base est que certaines représentations sont plus faciles à interpréter que d'autres, comme nous le verrons plus loin.

3.2.3 Valeurs et vecteurs propres

Nous terminons ce chapitre en abordant une autre notion fondamentale pour l'analyse des transformations linéaires en générale, et des réseaux de neurones en particulier : les valeurs et vecteurs propres.

Soit la transformation linéaire $\mathcal{A} : \mathcal{X} \longrightarrow \mathcal{X}$ (ici le domaine et l'image de la transformation sont les mêmes). Alors, les vecteurs $\mathbf{z} \in \mathcal{X}$ et les scalaires λ satisfaisant à la relation :

$$\mathcal{A}(\mathbf{z}) = \lambda\mathbf{z} \tag{3.34}$$

sont dits «vecteurs propres» ($\mathbf{z}$) et «valeurs propres» (λ), respectivement. Cette définition spécifie qu'un vecteur propre d'une transformation donnée représente une direction dans laquelle tous les vecteurs pointant dans cette direction continueront à pointer dans la même direction après la transformation, mais avec un facteur d'échelle λ correspondant à la valeur propre associée. Notez bien que cette interprétation n'est valide que lorsque les valeurs propres sont réelles et que les vecteurs propres existent ce qui n'est pas toujours le cas.

En posant un certain ensemble de vecteur de base, on peut reformuler l'équation 3.34 sous sa forme matricielle :

$$\mathbf{A}\mathbf{z} = \lambda\mathbf{z} \tag{3.35}$$

ou d'une manière équivalente :

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{z} = \mathbf{0} \tag{3.36}$$

où $\mathbf{I}$ représente la matrice identité. Cette dernière équation implique que les colonnes de $\mathbf{A} - \lambda\mathbf{I}$ sont dépendantes et, par conséquent, que son déterminant est nul :

$$|\mathbf{A} - \lambda\mathbf{I}| = 0 \tag{3.37}$$

Ce déterminant est un polynôme de degré n possédant exactement n racines, dont certaines peuvent être complexes et d'autres répétées.

Si l'on reprend notre exemple de la transformation de rotation :

$$\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.38}$$

On peut ré-écrire l'équation 3.36 de la façon suivante :

$$\begin{vmatrix} \cos\theta - \lambda & -\sin\theta \\ \sin\theta & \cos\theta - \lambda \end{vmatrix} = 0, \tag{3.39}$$

ce qui donne le polynôme suivant :

$$\lambda^2 - 2\lambda\cos\theta + (\cos^2\theta + \sin^2\theta) = \lambda^2 - 2\lambda\cos\theta + 1 = 0, \tag{3.40}$$

dont les racines $\lambda_1 = \cos\theta + j\sin\theta$ et $\lambda_2 = \cos\theta - j\sin\theta$ sont complexes. Ainsi, puisque qu'il n'y a pas de valeur propre réelle (sauf pour $\theta = 0°$ ou encore $\theta = 180°$), cela implique que tout vecteur réel transformé pointera dans une nouvelle direction (ce qui est l'effet recherché pour une rotation !).

Lorsqu'une matrice $\mathbf{A}$ de dimension $n \times n$ engendre n valeurs propres distinctes, alors il est possible d'engendrer n vecteurs propres indépendants qui correspondent à un ensemble de vecteurs de base pour la transformation que $\mathbf{A}$ représente. Dans ce cas, on peut diagonaliser la matrice de la transformation en effectuant un changement de base. Plus formellement, si $\mathbf{B} = [\mathbf{z}_1\mathbf{z}_2 \cdots \mathbf{z}_n]$, la matrice des n vecteurs propres, alors :

$$\mathbf{B}^{-1}\mathbf{A}\mathbf{B} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}, \tag{3.41}$$

où $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ sont les valeurs propres de $\mathcal{A}$. Ce résultat découle directement de la définition des vecteurs et valeurs propres de l'équation 3.35 :

$$\mathbf{A}\mathbf{B} = \mathbf{B}\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{3.42}$$

Il nous sera très utile dans les chapitres à venir, lorsque nous analyserons la performance de divers algorithmes d'apprentissage pour les réseaux de neurones.

Chapitre 4

Processus d'apprentissage

Parmi les propriétés désirables pour un réseau de neurones, la plus fondamentale est sûrement la capacité d'apprendre de son environnement, d'améliorer sa performance à travers un processus d'apprentissage. Mais qu'est-ce donc que l'apprentissage ? Malheureusement, il n'existe pas de définition générale, universellement acceptée, car ce concept touche à trop de notions distinctes qui dépendent du point de vue que l'on adopte.

Dans le contexte des réseaux de neurones artificiels, nous adopterons un point de vue pragmatique en proposant la définition suivante :

> *L'apprentissage est un processus dynamique et itératif permettant de modifier les paramètres d'un réseau en réaction avec les stimuli qu'il reçoit de son environnement. Le type d'apprentissage est déterminé par la manière dont les changements de paramètre surviennent.*

Cette définition implique qu'un réseau se doit d'être stimulé par un environnement, qu'il subisse des changements en réaction avec cette stimulation, et que ceux-ci provoquent dans le futur une réponse nouvelle vis-à-vis de l'environnement. Ainsi, le réseau peut s'améliorer avec le temps.

Dans la plupart des architectures que nous étudierons, l'apprentissage se traduit par une modification de l'efficacité synaptique, c'est-à-dire par un changement dans la valeur des poids qui relient les neurones d'une couche à l'autre. Soit le poids $w_{i,j}$ reliant le neurone i à son entrée j. Au temps t, un changement $\Delta w_{i,j}(t)$ de poids peut s'exprimer simplement de la façon suivante :

$$\Delta w_{i,j}(t) = w_{i,j}(t+1) - w_{i,j}(t), \tag{4.1}$$

et, par conséquent, $w_{i,j}(t+1) = w_{i,j}(t) + \Delta w_{i,j}(t)$, avec $w_{i,j}(t+1)$ et $w_{i,j}(t)$ représentant respectivement les nouvelle et ancienne valeurs du poids $w_{i,j}$.

Un ensemble de règles bien définies permettant de réaliser un tel processus d'adaptation des poids constitue ce qu'on appelle l'algorithme[1] d'apprentissage du réseau.

[1] Le mot «algorithme» provient du nom de famille d'un mathématicien perse nommé Mohammed Al-Khwarizmi qui a vécu au 9è siècle de notre ère. C'est à celui-ci que l'on attribue l'invention de règles pas-à-pas pour l'addition, la soustraction, la multiplication ainsi que la division de nombres décimaux. En latin, son nom fut traduit par Algorismus, qui par la suite se transforma en algorithme.

Dans la suite de ce chapitre, nous allons passer en revue différents types de règles ainsi que différents principes pouvant guider l'apprentissage d'un réseau de neurone.

4.1 Par correction d'erreur

La première règle que l'on peut utiliser est fondée sur la correction de l'erreur observée en sortie. Soit $a_i(t)$ la sortie que l'on obtient pour le neurone i au temps t. Cette sortie résulte d'un stimulus $\mathbf{p}(t)$ que l'on applique aux entrées du réseau dont un des neurones correspond au neurone i. Soit $d_i(t)$ la sortie que l'on désire obtenir pour ce même neurone i au temps t. Alors, $a_i(t)$ et $d_i(t)$ seront généralement différents et il est naturel de calculer l'erreur $e_i(t)$ entre ce qu'on obtient et ce qu'on voudrait obtenir :

$$e_i(t) = d_i(t) - a_i(t), \tag{4.2}$$

et de chercher un moyen de réduire autant que possible cette erreur. Sous forme vectorielle, on obtient :

$$\mathbf{e}(t) = \mathbf{d}(t) - \mathbf{a}(t), \tag{4.3}$$

avec $\mathbf{e}(t) = [e_1(t)e_2(t)\cdots e_i(t)\cdots e_S(t)]$ qui désigne le vecteur des erreurs observées sur les S neurones de sortie du réseau. L'apprentissage par correction des erreurs consiste à minimiser un indice de performance F basé sur les signaux d'erreur $e_i(t)$, dans le but de faire converger les sorties du réseau avec ce qu'on voudrait qu'elles soient. Un critère très populaire est la somme des erreurs quadratiques :

$$F(\mathbf{e}(t)) = \sum_{i=1}^{S} e_i^2(t) = \mathbf{e}(t)^T\mathbf{e}(t). \tag{4.4}$$

Maintenant, il importe de remarquer que les paramètres libres d'un réseau sont ses poids. Prenons l'ensemble de ces poids et assemblons les sous la forme d'un vecteur $\mathbf{w}(t)$ au temps t. Pour minimiser $F(\mathbf{e}(t)) = F(\mathbf{w}(t)) = F(t)$, nous allons commencer par choisir des poids initiaux ($t = 0$) au hasard, puis nous allons modifier ces poids de la manière suivante :

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\mathbf{x}(t), \tag{4.5}$$

où le vecteur $\mathbf{x}(t)$ désigne la direction dans laquelle nous allons chercher le minimum et η est une constante positive déterminant l'amplitude du pas dans cette direction (la vitesse d'apprentissage). L'objectif est de faire en sorte que $F(t+1) < F(t)$. Mais comment peut-on choisir la direction $\mathbf{x}$ pour que la condition précédente soit respectée ? Considérons la série de Taylor de 1er ordre autour de $\mathbf{w}(t)$:

$$F(t+1) = F(t) + \nabla F(t)^T \Delta\mathbf{w}(t), \tag{4.6}$$

où $\nabla F(t)$ désigne le gradient de F par rapport à ses paramètres libres (les poids $\mathbf{w}$) au temps t, et $\Delta\mathbf{w}(t) = \mathbf{w}(t+1) - \mathbf{w}(t)$. Or, pour que $F(t+1) < F(t)$, il faut que la condition suivante soit respectée :

$$\nabla F(t)^T \Delta\mathbf{w}(t) = \eta\nabla F(t)^T\mathbf{x}(t) < 0. \tag{4.7}$$

N'importe quel vecteur $\mathbf{x}(t)$ qui respecte l'inégalité de l'équation 4.7 pointe donc dans une direction qui diminue F. On parle alors d'une direction de «descente». Pour obtenir une descente

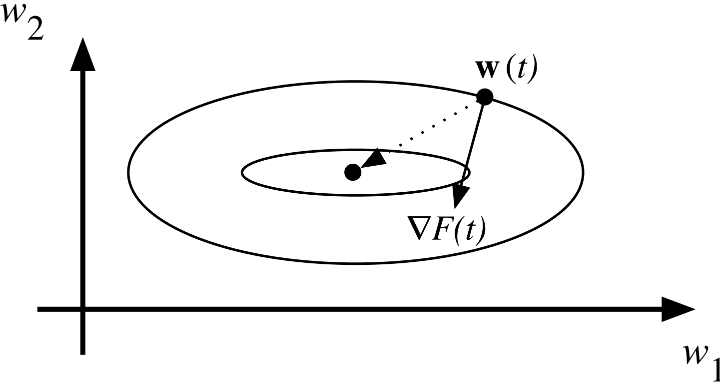


FIG. 4.1 – *Trajectoire de la descente du gradient.*

maximum, étant donnée $\eta > 0$, il faut que le vecteur $\mathbf{x}(t)$ pointe dans le sens opposé au gradient car c'est dans ce cas que le produit scalaire sera minimum :

$$\mathbf{x}(t) = -\nabla F(t) \tag{4.8}$$

Ce qui engendre la règle dite de «descente du gradient» :

$$\Delta\mathbf{w}(t) = -\eta\nabla F(t) \tag{4.9}$$

illustrée à la figure 4.1. Dans l'espace des poids, cette figure montre les courbes de niveau de F représentées par des ellipses hypothétiques. La flèche en pointillés montre la direction optimale pour atteindre le minimum de F. La flèche pleine montre la direction du gradient qui est perpendiculaire à la courbe de niveau en $\mathbf{w}(t)$.

L'expression exacte du gradient dépend de la fonction d'activation utilisée pour les neurones. Nous reviendrons en détails sur la méthode de la descente du gradient au chapitre 5, lorsque nous traiterons du perceptron multicouche.

La règle de la correction des erreurs est utilisée pour beaucoup de réseaux de neurones artificiels, bien qu'elle ne soit pas plausible biologiquement. En effet, comment le cerveau pourrait-il connaître a priori les sorties qu'il doit produire ? Cette règle ne peut être utilisée que dans un contexte d'apprentissage supervisé sur lequel nous reviendrons bientôt.

4.2 Par la règle de Hebb

Dans cette section nous abordons une règle qui s'inspire des travaux du neurophysiologiste Donald Hebb :

> *«When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.»*

Dans un contexte neurobiologique, Hebb cherchait à établir une forme d'apprentissage associatif au niveau cellulaire. Dans le contexte des réseaux artificiels, on peut reformuler l'énoncé de Hebb sous la forme d'une règle d'apprentissage en deux parties :

1. Si deux neurones de part et d'autre d'un synapse (connexion) sont activés simultanément (d'une manière synchrone), alors la force de ce synapse doit être augmentée ;
2. Si les mêmes deux neurones sont activés d'une manière asynchrone, alors le synapse correspondant doit être affaibli ou carrément éliminé.

Un tel synapse est dit «synapse hebbien». Il utilise un mécanisme interactif, dépendant du temps et de l'espace, pour augmenter l'efficacité synaptique d'une manière proportionnelle à la corrélation des activités pré- et post-synaptiques. De cette définition ressort les propriétés suivantes :

1. *Dépendance temporelle*. Les modifications d'un synapse hebbien dépendent du moment exact des activités pré- et post-synaptiques ;
2. *Dépendance spatiale*. Étant donné la nature même du synapse qui constitue un lieu de transmission d'information, l'apprentissage hebbien se doit de posséder une contiguïté spatiale. C'est cette propriété qui, entre autres, permet l'apprentissage dit non-supervisé sur lequel nous reviendrons bientôt ;
3. *Interaction*. L'apprentissage hebbien dépend d'une interaction entre les activités de part et d'autre du synapse.
4. *Conjonction ou corrélation*. Une interprétation de l'énoncé de Hebb est que la condition permettant un changement dans l'efficacité synaptique est une conjonction des activités pré et post-synaptiques. C'est la co-occurrence des activités de part et d'autre du synapse qui engendre une modification de celui-ci. Une interprétation plus statistique réfère à la corrélation de ces activités. Deux activités positives simultanées (corrélation positive) engendrent une augmentation de l'efficacité synaptique, alors que l'absence d'une telle corrélation engendre une baisse de cette efficacité.

Mathématiquement, on peut exprimer la règle de Hebb sous sa forme la plus simple par la formule suivante :

$$\Delta w_j(t-1) = \eta p_j(t)a(t), \tag{4.10}$$

où η est une constante positive qui détermine la vitesse de l'apprentissage, $p_j(t)$ correspond à l'activité pré-synaptique (l'entrée j du neurone) au temps t, et $a(t)$ à l'activité post-synaptique (sortie du neurone) à ce même temps t. Cette formule fait ressortir explicitement la corrélation entre le signal qui entre et celui qui sort. Sous une forme vectorielle, on écrit :

$$\Delta \mathbf{w}(t-1) = \eta \mathbf{p}(t)a(t). \tag{4.11}$$

Un problème immédiat avec la règle de l'équation 4.11 est que les changements de poids $\Delta w_j(t)$ peuvent croître de façon exponentielle si, par exemple, l'entrée et la sortie demeurent constantes dans le temps. Pour pallier à cette croissance exponentielle qui provoquerait invariablement une saturation du poids, on ajoute parfois un facteur d'oubli qui retranche de la variation de poids, une fraction α du poids actuel. On obtient ainsi :

$$\Delta w_j(t-1) = \eta p_j(t)a(t) - \alpha w_j(t-1), \tag{4.12}$$

où $0 \leq \alpha \leq 1$ est une nouvelle constante. Sous forme vectorielle, on écrit :

$$\Delta \mathbf{w}(t-1) = \eta \mathbf{p}(t) a(t) - \alpha \mathbf{w}(t-1). \tag{4.13}$$

La règle de Hebb avec oubli, énoncée à l'équation 4.13, contourne efficacement le problème des poids qui croissent (ou décroissent) sans limite. Supposons que $p_j(t) = a(t) = 1$ et que nous ayons atteint le régime permanent où $\Delta w_j = 0$. Alors, la valeur maximale $w_j^{\max}$ que peut atteindre le poids $w_j(t)$ est donnée par :

$$\begin{aligned} w_j^{\max} &= (1-\alpha) w_j^{\max} + \eta & (4.14) \\ &= \frac{\eta}{\alpha}. & (4.15) \end{aligned}$$

Mais cette règle ne résout pas tous les problèmes. À cause du terme d'oubli, il est primordial que les stimuli soient répétés régulièrement, sinon les associations apprises grâce à la règle de l'équation 4.13 seront éventuellement perdues car complètement oubliées. Une autre variante de la règle de Hebb s'exprime donc de la manière suivante :

$$\Delta w_j(t-1) = \eta p_j(t) a(t) - \alpha a(t) w_j(t-1). \tag{4.16}$$

Et si l'on fixe $\alpha = \eta$ pour simplifier (on pose un rythme d'apprentissage égale à celui de l'oubli), on obtient la règle dite «instar» :

$$\Delta w_j(t-1) = \eta a(t) \left[p_j(t) - w_j(t-1) \right], \tag{4.17}$$

que l'on peut ré-écrire sous sa forme vectorielle de la façon suivante :

$$\Delta \mathbf{w}(t-1) = \eta a(t) \left[\mathbf{p}(t) - \mathbf{w}(t-1) \right]. \tag{4.18}$$

Une façon d'aborder cette règle, est de regarder ce qui se passe lorsque $a(t) = 1$:

$$\begin{aligned} \mathbf{w}(t) &= \mathbf{w}(t-1) + \eta \left[\mathbf{p}(t) - \mathbf{w}(t-1) \right] & (4.19) \\ &= (1-\eta) \mathbf{w}(t-1) + \eta \mathbf{p}(t). & (4.20) \end{aligned}$$

Alors, on constate qu'en présence d'une activité post-synaptique positive, le vecteur de poids est déplacé dans la direction du vecteur d'entrée $\mathbf{p}(t)$, le long du segment qui relie l'ancien vecteur de poids avec le vecteur d'entrée, tel qu'illustré à la figure 4.2. Lorsque $\eta = 0$, le nouveau vecteur de poids est égal à l'ancien (aucun changement). Lorsque $\eta = 1$, le nouveau vecteur de poids est égal au vecteur d'entrée. Finalement, lorsque $\eta = \frac{1}{2}$, le nouveau vecteur est à mi-chemin entre l'ancien vecteur de poids et le vecteur d'entrée.

Une propriété particulièrement intéressante de la règle instar est qu'avec des entrées normalisées, suite au processus d'apprentissage, les poids $\mathbf{w}$ convergeront également vers des vecteurs normalisés. Mais nous y reviendrons lorsque nous traiterons du réseau «instar».

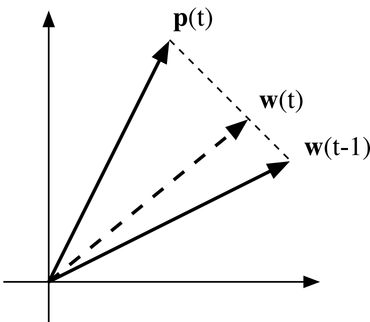


FIG. 4.2 – *Représentation graphique de la règle «instar» lors d'une activité post-synaptique positive.*

4.3 Compétitif

L'apprentissage compétitif, comme son nom l'indique, consiste à faire compétitionner les neurones d'un réseau pour déterminer lequel sera actif à un instant donné. Contrairement aux autres types d'apprentissage où, généralement, tous les neurones peuvent apprendre simultanément et de la même manière, l'apprentissage compétitif produit un «vainqueur» ainsi que, parfois, un ensemble de neurones «voisins» du vainqueur, et seuls ce vainqueur et, potentiellement, son voisinage bénéficient d'une adaptation de leur poids. On dit alors que l'apprentissage est local car limité à un sous-ensemble des neurones du réseau.

Une règle d'apprentissage compétitif comporte les éléments suivants :

- Un ensemble de neurones identiques (même type) sauf pour les valeurs de leurs poids synaptiques ;
- Une limite imposée à la «force» d'un neurone ;
- Un mécanisme permettant aux neurones de compétitionner pour le droit de répondre à un certain sous-ensemble des stimuli d'entrée, de manière à ce qu'un seul neurone de sortie soit actif à la fois.

Ainsi, les neurones individuels peuvent apprendre à se spécialiser sur des sous-ensembles de stimuli similaires pour devenir des détecteurs de caractéristiques.

Dans leur forme la plus simple, les réseaux de neurones qui utilisent l'apprentissage compétitif sont souvent constitués d'une seule couche de neurones de sortie, totalement connectée sur les entrées. Un neurone vainqueur modifiera ses poids synaptiques en les rapprochant (géométriquement) d'un stimulus d'entrée $\mathbf{p}$ pour lequel il a battu tous les autres neurones lors de la compétition :

$$\Delta \mathbf{w} = \begin{cases} \eta(\mathbf{p} - \mathbf{w}) & \text{si le neurone est vainqueur} \\ 0 & \text{autrement} \end{cases}, \tag{4.21}$$

où $0 < \eta < 1$ correspond à un taux d'apprentissage. Un neurone qui ne gagne pas la compétition ne modifiera aucunement ses poids. Il ne sera donc pas affecté par le stimulus en question. Parfois, on définit également un voisinage autour du neurone gagnant et on applique une règle similaire sur

les voisins, mais avec un taux d'apprentissage différent :

$$\Delta \mathbf{w} = \begin{cases} \eta_1(\mathbf{p} - \mathbf{w}) & \text{si le neurone est vainqueur} \\ \eta_2(\mathbf{p} - \mathbf{w}) & \text{si le neurone est voisin du vainqueur} \\ 0 & \text{autrement} \end{cases}, \quad (4.22)$$

avec $\eta_2 \leq \eta_1$.

Comme nous le verrons plus loin dans ce chapitre, l'apprentissage compétitif est surtout utilisé dans le contexte d'un apprentissage dit non-supervisé, c'est-à-dire lorsqu'on ne connait pas les valeurs désirées pour les sorties du réseau.

4.4 Problème de l'affectation du crédit

Dans le domaine général de l'apprentissage, il existe un problème qui tourne autour de la notion de «affectation du crédit»[2]. Essentiellement, il s'agit d'affecter le crédit d'un résultat global, par exemple l'adéquation des sorties d'un réseau face à un certain stimulus d'entrée, à l'ensemble des décisions internes prises par le système (le réseau) et ayant conduit à ce résultat global. Dans le cas de l'exemple d'un réseau, les décisions internes correspondent aux sorties des neurones situés sur les couches qui précèdent la couche de sortie. Ces couches sont habituellement qualifiées de «couches cachées» car on ne dispose pas, a priori, d'information sur l'adéquation de leurs sorties.

Le problème de l'affectation du crédit est donc bien présent dans l'apprentissage des réseaux de neurones. Par exemple, si l'on adopte une règle basée sur la correction des erreurs, comment fera-t-on pour calculer cette erreur sur les couches cachées, si l'on ne possède pas l'information à propos de leurs sorties désirées ? De même, que fera-t-on si l'on dispose uniquement d'une appréciation générale de performance du réseau face à chaque stimulus, et non des sorties désirées pour chaque neurone de la couche de sortie ? Nous apporterons certains éléments de réponse à ces questions dans les sous-sections suivantes, puis dans les chapitres subséquents au fur et à mesure que nous aborderons des algorithmes concrets d'apprentissage.

4.5 Supervisé

L'apprentissage dit supervisé est caractérisé par la présence d'un «professeur» qui possède une connaissance approfondie de l'environnement dans lequel évolue le réseau de neurones. En pratique, les connaissances de ce professeur prennent la forme d'un ensemble de Q couples de vecteurs d'entrée et de sortie que nous noterons $\{(\mathbf{p}_1, \mathbf{d}_1), (\mathbf{p}_2, \mathbf{d}_2), \ldots, (\mathbf{p}_Q, \mathbf{d}_Q)\}$, où $\mathbf{p}_i$ désigne un stimulus (entrée) et $\mathbf{d}_i$ la cible pour ce stimulus, c'est-à-dire les sorties désirées du réseau. Chaque couple $(\mathbf{p}_i, \mathbf{d}_i)$ correspond donc à un cas d'espèce de ce que le réseau devrait produire (la cible) pour un stimulus donné. Pour cette raison, l'apprentissage supervisé est aussi qualifié d'apprentissage par des exemples.

[2]Traduction littérale de «Credit assignment».

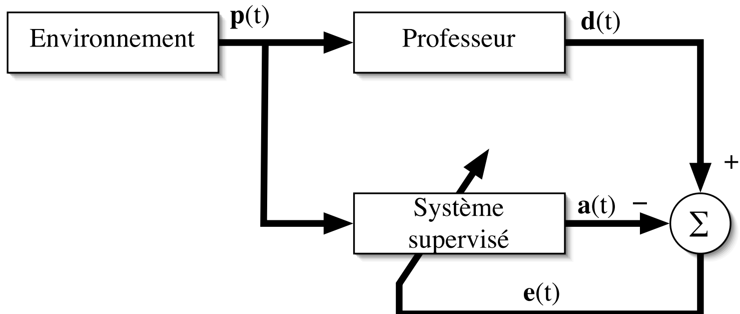


FIG. 4.3 – *Schéma bloc de l'apprentissage supervisé.*

L'apprentissage supervisé est illustré d'une manière conceptuelle à la figure 4.3. L'environnement est inconnu du réseau. Celui-ci produit un stimulus $\mathbf{p}$ qui est acheminé à la fois au professeur et au réseau. Grâce à ses connaissances intrinsèques, le professeur produit une sortie désirée $\mathbf{d}(t)$ pour ce stimulus. On suppose que cette réponse est optimale. Elle est ensuite comparée (par soustration) avec la sortie du réseau pour produire un signal d'erreur $\mathbf{e}(t)$ qui est ré-injecté dans le réseau pour modifier son comportement via une procédure itérative qui, éventuellement, lui permet de simuler la réponse du professeur. Autrement dit, la connaissance de l'environnement par le professeur est graduellement transférée vers le réseau jusqu'à l'atteinte d'un certain critère d'arrêt. Par la suite, on peut éliminer le professeur et laisser le réseau fonctionner de façon autonome.

Le lecteur attentif aura remarqué qu'un apprentissage supervisé n'est rien d'autre qu'un synonyme de l'apprentissage par correction des erreurs (voir section 4.1). Il possède donc les mêmes limitations, à savoir que sans professeur pour fournir les valeurs cibles, il ne peut d'aucune façon apprendre de nouvelles stratégies pour de nouvelles situations qui ne sont pas couvertes par les exemples d'apprentissage.

4.6 Par renforcement

L'apprentissage par renforcement permet de contourner certaines des limitations de l'apprentissage supervisé. Il consiste en un espèce d'apprentissage supervisé, mais avec un indice de satisfaction scalaire au lieu d'un signal d'erreur vectoriel. Ce type d'apprentissage est inspiré des travaux en psychologie expérimentale de Thorndike (1911) :

> *«Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur ; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.»*

Même si cet énoncé ne peut expliquer à lui seul le comportement animal au niveau biologique, sa simplicité et son pragmatisme peut nous permettre de composer des règles d'apprentissage utiles.

Dans le contexte des réseaux de neurones artificiels, nous pouvons reformuler l'énoncé de Thorndike de la façon suivante :

> Lorsqu'une action (décision) prise par le réseau engendre un indice de satisfaction positif, alors la tendance du réseau à prendre cette action doit être renforcée. Autrement, la tendance à prendre cette action doit être diminuée.

En pratique, l'usage de l'apprentissage par renforcement est complexe à mettre en ?uvre, de sorte que nous n'aborderons aucun réseau qui l'emploie. Il importe cependant de bien comprendre la différence entre ce type d'apprentissage et l'apprentissage supervisé que nous étudierons en détails au chapitre 5.

L'apprentissage supervisé dispose d'un signal d'erreur qui non seulement permet de calculer un indice de satisfaction (p.ex. l'erreur quadratique moyenne), mais permet aussi d'estimer le gradient local qui indique une direction pour l'adaptation des poids synaptiques. C'est cette information fournie par le professeur qui fait toute la différence. Dans l'apprentissage par renforcement, l'absence de signal d'erreur rend le calcul de ce gradient impossible. Pour estimer le gradient, le réseau est obligé de tenter des actions et d'observer le résultat, pour éventuellement inférer une direction de changement pour les poids synaptiques. Pour ce faire, il s'agit alors d'implanter un processus d'essais et d'erreurs tout en retardant la récompense offerte par l'indice de satisfaction. Ainsi, on introduit deux étapes distinctes : une d'exploration où l'on essaie des directions aléatoires de changement, et une d'exploitation où l'on prend une décision. Ce processus en deux étapes peut ralentir considérablement l'apprentissage. De plus, il introduit un dilemme entre le désir d'utiliser l'information déjà apprise à propos du mérite des différentes actions, et celui d'acquérir de nouvelles connaissances sur les conséquences de ces décisions pour, éventuellement, mieux les choisir dans le futur.

4.7 Non-supervisé

La dernière forme d'apprentissage que nous abordons est dite «non-supervisée» ou encore «auto-organisée». Elle est caractérisée par l'absence complète de professeur, c'est-à-dire qu'on ne dispose ni d'un signal d'erreur, comme dans le cas supervisé, ni d'un indice de satisfaction, comme dans le cas par renforcement. Nous ne disposons donc que d'un environnement qui fournit des stimuli, et d'un réseau qui doit apprendre sans intervention externe. En assimilant les stimuli de l'environnement à une description de son état interne, la tâche du réseau est alors de modéliser cet état le mieux possible. Pour y arriver, il importe d'abord de définir une mesure de la qualité pour ce modèle, et de s'en servir par la suite pour optimiser les paramètres libres du réseau, c'est-à-dire ses poids synaptiques. À la fin de l'apprentissage, le réseau a développé une habilité à former des représentations internes des stimuli de l'environnement permettant d'encoder les caractéristiques de ceux-ci et, par conséquent, de créer automatiquement des classes de stimuli similaires.

L'apprentissage non-supervisé s'appuie généralement sur un processus compétitif (voir section 4.3) permettant d'engendrer un modèle où les poids synaptiques des neurones représentent des prototypes de stimuli. La qualité du modèle résultant doit s'évaluer à l'aide d'une métrique permettant de mesurer la distance entre les stimuli et leurs prototypes. Souvent, cette métrique est basée

sur la norme l_2 (voir section 3.1.4). C'est le processus de compétition qui permet de sélectionner le prototype associé à chaque stimulus en recherchant le neurone dont le vecteur de poids synaptiques est le plus proche (au sens de la métrique choisie) du stimulus en question.

4.8 Tâches d'apprentissage

Nous terminons ce chapitre en énumérant différentes catégories de tâches que l'on peut vouloir réaliser avec un réseau de neurones :

1. *Approximation*. Soit la fonction g telle que :

$$d = g(\mathbf{p}), \tag{4.23}$$

   où $\mathbf{p}$ est l'argument de la fonction (un vecteur) et d la valeur (un scalaire) de cette fonction évaluée en $\mathbf{p}$. Supposons maintenant que la fonction $g(\cdot)$ est inconnue. La tâche d'approximation consiste alors à concevoir un réseau de neurones capable d'associer les éléments des couples entrée-sortie : $\{(\mathbf{p}_1, d_1), (\mathbf{p}_2, d_2), \ldots, (\mathbf{p}_Q, d_Q)\}$. Ce problème peut être résolu à l'aide d'un apprentissage supervisé sur les Q exemples, avec les $\mathbf{p}_i$ représentant les stimuli, et les d_i représentant les sorties désirées pour chacun de ces stimuli, avec $i = 1, 2, \ldots, Q$. Ou inversement, on peut aussi dire que l'apprentissage supervisé est un problème d'approximation de fonction ;

2. *Association*. Il en existe deux types : l'auto-association et l'hétéro-association. Le problème de l'auto-association consiste à mémoriser un ensemble de patrons (vecteurs) en les présentant successivement au réseau. Par la suite, on présente au réseau une version partielle ou déformée d'un patron original, et la tâche consiste à produire en sortie le patron original correspondant. Le problème de l'hétéro-association consiste quant à lui à associer des paires de patrons : un patron d'entrée et un patron de sortie. L'auto-association implique un apprentissage non supervisé, alors que l'hétéro-association requiert plutôt un apprentissage supervisé.

3. *Classement*. Pour cette tâche, il existe un nombre fixe de catégories (classes) de stimuli d'entrée que le réseau doit apprendre à reconnaître. Dans un premier temps, le réseau doit entreprendre une phase d'apprentissage supervisé durant laquelle les stimuli sont présentés en entrée et les catégories sont utilisées pour former les sorties désirées, généralement en utilisant une sortie par catégorie. Ainsi, la sortie 1 est associée à la catégorie 1, la sortie 2 à la catégorie 2, etc. Pour un problème comportant Q catégories, on peut par exemple fixer les sorties désirées $\mathbf{d} = [d_1, d_2, \ldots, d_Q]^T$ à l'aide de l'expression suivante :

$$d_i = \begin{cases} 1 & \text{si le stimulus appartient à la catégorie } i \\ 0 & \text{autrement} \end{cases}, i = 1, \ldots, Q. \tag{4.24}$$

   Par la suite, dans une phase de reconnaissance, il suffira de présenter au réseau n'importe quel stimulus inconnu pour pouvoir procéder au classement de celui-ci dans l'une ou l'autre des catégories. Une règle simple de classement consiste, par exemple, à choisir la catégorie associée avec la sortie maximale.

4. *Prédiction*. La notion de prédiction est l'une des plus fondamentales en apprentissage. Il s'agit d'un problème de traitement temporel de signal. En supposant que nous possédons M échantillons passés d'un signal, $x(t-1), x(t-2), \ldots, x(t-M)$, échantillonnés à intervalle de temps fixe, la tâche consiste à prédire la valeur de x au temps t. Ce problème de prédiction peut être résolu grâce à un apprentissage par correction des erreurs, mais d'une manière non supervisé (sans professeur), étant donné que les valeurs de sortie désirée peuvent être inférées directement de la série chronologique. Plus précisément, l'échantillon de $x(t)$ peut servir de valeur désirée et le signal d'erreur pour l'adaptation des poids se calcule simplement par l'équation suivante :

$$e(t) = x(t) - \widehat{x}(t \mid t-1, t-2, \ldots, t-M), \tag{4.25}$$

   où $x(t)$ désigne la sortie désirée et $\widehat{x}(t \mid t-1, t-2, \ldots, t-M)$ représente la sortie observée du réseau étant donné les M échantillons précédents. La prédiction s'apparente à la construction d'un modèle physique de la série chronologique. Dans la mesure où le réseau possède des neurones dont la fonction de transfert est non-linéaire, le modèle pourra lui-aussi être non-linéaire.

5. *Commande*. La commande d'un processus est une autre tâche d'apprentissage que l'on peut aborder à l'aide d'un réseau de neurones. Considérons un système dynamique non-linéaire $\{u(t), y(t)\}$ où $u(t)$ désigne l'entrée du système et $y(t)$ correspond à la réponse de celui-ci. Dans le cas général, on désire commander ce système de manière à ce qu'il se comporte selon un modèle de référence, souvent un modèle linéaire, $\{r(t), d(t)\}$, où pour tout temps $t \geq 0$, on arrive à produire une commande $u(t)$ telle que :

$$\lim_{t \to \infty} |d(t) - y(t)| = 0, \tag{4.26}$$

   de manière à ce que la sortie du système suivent de près celle du modèle de référence. Ceci peut se réaliser grâce à certains types de réseaux supervisés.

Dans les chapitres qui suivent, nous allons aborder des réseaux spécifiques en commençant par l'un des plus connus et des plus utilisés : le perceptron multicouches et son algorithme de rétropropagation des erreurs.

Chapitre 5

Perceptron multicouche

Le premier réseau de neurones que nous allons étudier s'appelle le «perceptron multicouche[1]» (PMC). Ce type de réseau est dans la famille générale des réseaux à «propagation vers l'avant[2]», c'est-à-dire qu'en mode normal d'utilisation, l'information se propage dans un sens unique, des entrées vers les sorties sans aucune rétroaction. Son apprentissage est de type supervisé, par correction des erreurs (chapitre 4). Dans ce cas uniquement, le signal d'erreur est «rétropropagé» vers les entrées pour mettre à jour les poids des neurones.

Le perceptron multicouche est un des réseaux de neurones les plus utilisés pour des problèmes d'approximation, de classification et de prédiction. Il est habituellement constitué de deux ou trois couches de neurones totalement connectés. Avant d'en étudier le fonctionnement global, nous allons nous attarder à divers cas particuliers plus simples. En particulier, nous allons aborder le cas du perceptron simple, c'est-à-dire le perceptron à une seule couche de neurones dont les fonctions d'activation sont de type seuils (section 2.3). Nous allons ensuite considérer différentes règles d'apprentissage pour la correction des erreurs. Nous traiterons le cas de la règle LMS[3], de l'algorithme de rétropropagation (en anglais «backpropagation»), de la méthode de Newton et de la méthode du gradient conjugué.

5.1 Perceptron simple

Le perceptron simple est illustré à la figure 5.1. En suivant la notation schématique établie au chapitre 2, il s'agit d'une seule couche de S neurones totalement connectée sur un vecteur $\mathbf{p}$ de R entrées. La matrice $\mathbf{W} = [{}_1\mathbf{w}\,{}_2\mathbf{w} \cdots {}_S\mathbf{w}]^T$ de dimension $S \times R$ représente l'ensemble des poids de la couche, avec les vecteur-rangées ${}_i\mathbf{w}$ (dimension $R \times 1$) représentant les R poids des connexions reliant le neurone i avec ses entrées. Le vecteur $\mathbf{b}$ (dimension $S \times 1$) désigne l'ensemble des S biais de la couche. Les niveaux d'activation $\mathbf{n} = \mathbf{W}\mathbf{p} - \mathbf{b} = [n_1 n_2 \cdots n_S]^T$ des neurones de la couche servent d'argument à la fonction d'activation qui applique un seuil au niveau 0 (section 2.3) pour

[1]En anglais «multilayer perceptron» ou MLP.
[2]En anglais «feedforward networks».
[3]En anglais «Least Mean Square».

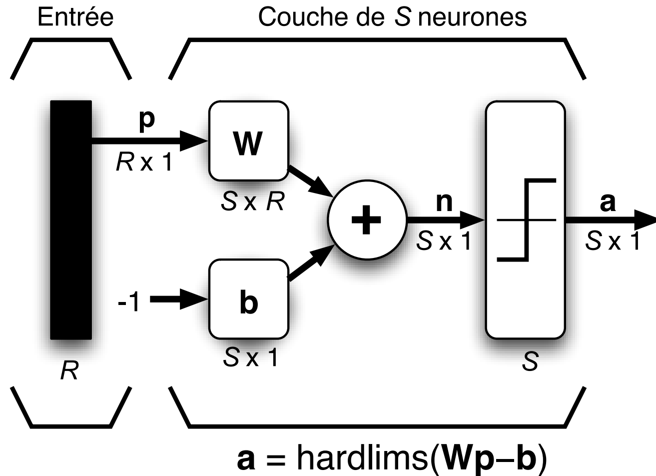


FIG. 5.1 – *Perceptron à une seule couche avec fonction seuil.*

produire le vecteur des sorties $\mathbf{a} = [a_1\, a_2 \cdots a_S]^T$, où :

$$a_i = \begin{cases} +1 & \text{si } n_i \geq 0 \\ -1 & \text{autrement} \end{cases} \tag{5.1}$$

Considérons maintenant le cas non-trivial le plus simple, à savoir lorsque $R = 2$ et $S = 1$, c'est-à-dire lorsque la couche n'est formée que d'un seul neurone relié à deux entrées. Dans ce cas, nous aurons $\mathbf{p} = [p_1\, p_2]^T$, $\mathbf{W} = [{}_1\mathbf{w}]^T = [w_{1,1}\, w_{1,2}]$, $\mathbf{b} = [b_1]$ et $\mathbf{a} = [a_1]$, où :

$$a_1 = \begin{cases} +1 & \text{si } w_{1,1}\, p_1 + w_{1,2}\, p_2 \geq b_1 \\ -1 & \text{autrement} \end{cases} \tag{5.2}$$

Cette dernière équation nous indique clairement que la sortie du réseau (neurone) peut prendre seulement deux valeurs distinctes selon le niveau d'activation du neurone : -1 lorsque ce dernier est strictement inférieur à 0 ; +1 dans le cas contraire. Il existe donc dans l'espace des entrées une frontière délimitant deux régions correspondantes. Cette frontière est définie par la condition $w_{1,1}\, p_1 + w_{1,2}\, p_2 = b_1$ de l'équation 5.2 qui correspond à l'expression générale d'une droite, telle qu'illustrée à la figure 5.2. Étant donné un certain vecteur de poids $\mathbf{w} = [w_{1,1}\, w_{1,2}]^T$, il est aisé de montrer que ce vecteur doit être perpendiculaire à cette droite. En effet, pour tous les points $\mathbf{p}$ de la droite, nous avons la relation $\mathbf{w}^T\mathbf{p} = b$, où $b = b_1$. Or le terme $\mathbf{w}^T\mathbf{p}$ correspond à un produit scalaire (section 3.1.3) et l'on sait que $<\mathbf{x}, \mathbf{y}> = ||\mathbf{x}||\; ||\mathbf{y}|| \cos\theta$ (équation 3.12), où θ représente l'angle entre les vecteurs $\mathbf{x}$ et $\mathbf{y}$. Nous avons donc :

$$<\mathbf{w}, \mathbf{p}> = ||\mathbf{w}||\; ||\mathbf{p}|| \cos\theta = b \tag{5.3}$$

pour tous les points $\mathbf{p}$ qui appartiennent à la droite, et le produit scalaire doit rester constant. Mais s'il reste constant alors que la norme de $\mathbf{p}$ change, c'est parce que l'angle entre les vecteurs doit aussi changer. Soit $\mathbf{p}_\perp$, le point de la droite dont le vecteur correspondant possède la plus petite norme. Ce vecteur est perpendiculaire à la droite et sa norme correspond à la distance perpendiculaire entre la droite et l'origine. Maintenant, si sa norme est minimale, c'est que $\cos\theta$ est maximal

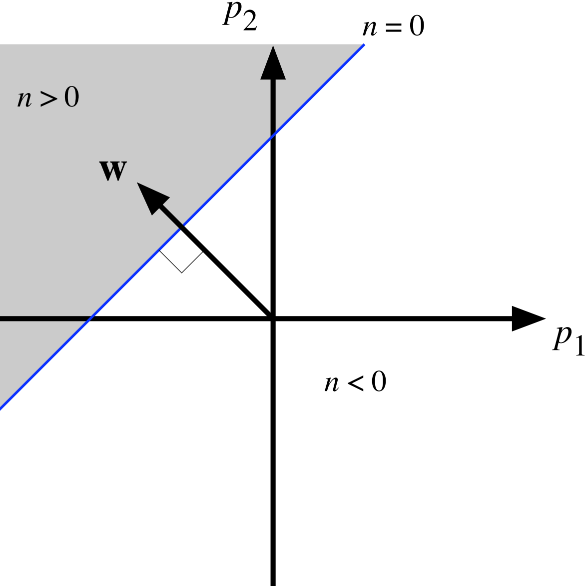


FIG. 5.2 – *Frontière de décision pour un perceptron simple à 1 neurone et deux entrées.*

et, par conséquent, que l'angle θ entre $\mathbf{p}_\perp$ et $\mathbf{w}$ est nul. Ainsi, $\mathbf{w}$ pointe dans la même direction que $\mathbf{p}_\perp$ et :

$$||\mathbf{p}_\perp|| = \frac{b}{||\mathbf{w}||} \tag{5.4}$$

Nous pouvons également déduire que l'origine appartiendra à la région grisée ($n > 0$) si, et seulement si, $b < 0$. Autrement, comme à la figure 5.2, l'origine appartiendra à la région $n < 0$. Si $b = 0$, alors la frontière de décision passera par l'origine.

Si l'on considère maintenant le cas où $S > 1$, alors chaque neurone i possédera son propre vecteur de poids ${}_i\mathbf{w}$ et son propre biais b_i, et nous nous retrouverons avec S frontières de décision distinctes. Toutes ces frontières de décision seront linéaires. Elles permettront chacune de découper l'espace d'entrée en deux régions infinies, de part et d'autre d'une droite. Chaque neurone d'un perceptron simple permet donc de résoudre parfaitement un problème de classification (voir section 4.8) à deux classes, à condition que celles-ci soient linéairement séparables. Il ne reste plus qu'à trouver une règle d'apprentissage pour pouvoir déterminer les poids et les biais du réseau permettant de classer au mieux Q couples d'apprentissage :

$$\{(\mathbf{p}_1, \mathbf{d}_1), (\mathbf{p}_2, \mathbf{d}_2), \ldots, (\mathbf{p}_Q, \mathbf{d}_Q)\} \tag{5.5}$$

Pour fixer les idées, considérons le problème particulier, illustré à la figure 5.3, consistant à discriminer entre le point noir ($\mathbf{p}_1$) et les points blancs ($\mathbf{p}_2$ et $\mathbf{p}_3$) définis par :

$$\left\{ \left(\mathbf{p}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, d_1 = +1 \right), \left(\mathbf{p}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, d_2 = -1 \right), \left(\mathbf{p}_3 = \begin{bmatrix} 0 \\ -2 \end{bmatrix}, d_3 = -1 \right) \right\} \tag{5.6}$$

et fixons $S = 1$ (un seul neurone). Il s'agit de trouver un vecteur de poids $\mathbf{w}$ correspondant à l'une ou l'autre des frontières de décision illustrées à la figure 5.3a. Pour simplifier davantage,

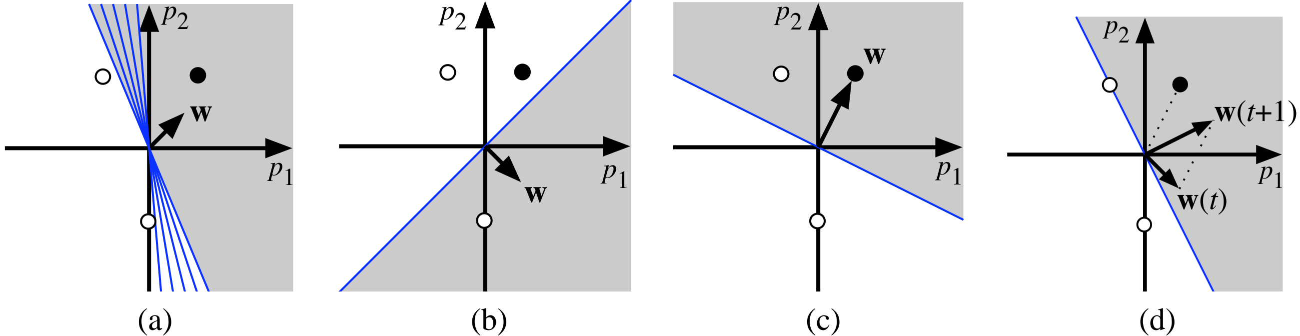


FIG. 5.3 – *Exemple d'un problème à deux classes (points noirs vs points blancs).*

nous supposons pour cet exemple que $b = 0$, de sorte que les frontières de décision induites par $\mathbf{w}$ passent toutes par l'origine. Le problème, bien sûr, est que nous ne connaissons pas a priori la bonne orientation pour $\mathbf{w}$. Nous allons donc l'initialiser aléatoirement, par exemple $\mathbf{w} = [1\ {-1}]^T$ (voir figure 5.3b).

Considérons le point $\mathbf{p}_1$ (point noir). La sortie du réseau pour ce point est donnée par :

$$a = \text{hardlims}(\mathbf{w}^T\mathbf{p}_1) = \text{hardlims}\left([1\ {-1}]\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right) = \text{hardlims}(-1) = -1 \tag{5.7}$$

Or, la sortie désirée pour ce point est +1 (les zones grises à la figure 5.3 produisent une sortie +1). Le réseau n'a donc pas le comportement désiré, il faudra modifier le vecteur $\mathbf{w}$. On peut remarquer que dans le cas particulier de ce problème simplifié, la norme de $\mathbf{w}$ ne compte pas car le biais est nul, seule son orientation importe.

Comment pourrions-nous modifier $\mathbf{w}$ pour que le réseau puisse classifier adéquatement le point $\mathbf{p}_1$? Une solution consisterait à fixer $\mathbf{w} = \mathbf{p}_1$, tel qu'illustré à la figure 5.3c. De cette manière, le point $\mathbf{p}_1$ serait parfaitement classer. Mais le problème avec cette approche est que la frontière de décision bondirait d'un stimulus à l'autre au fil de l'apprentissage ce qui pourrait engendrer des oscillations et empêcher la convergence dans certains cas. La solution consiste donc à prendre une position intermédiaire en approchant la direction de $\mathbf{w}$ de celle de $\mathbf{p}_1$:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{p}_1 \tag{5.8}$$

tel qu'illustré à la figure 5.3d. Cette règle fonctionne bien pour la catégorie de stimulus où l'on désire obtenir une sortie +1. Dans la situation inverse, il faut au contraire éloigner $\mathbf{w}$ de $\mathbf{p}_1$. Définissons un signal d'erreur $e = \frac{d-a}{2}$ où $e \in \{-1, 0, +1\}$. Alors, nous avons l'ensemble suivant de règles :

$$\Delta\mathbf{w} = \begin{cases} \mathbf{p} & \text{si } e = +1 \\ 0 & \text{si } e = 0 \\ -\mathbf{p} & \text{si } e = -1 \end{cases} \tag{5.9}$$

où $\Delta\mathbf{w} = \mathbf{w}(t+1) - \mathbf{w}(t)$ et $\mathbf{p}$ est le stimulus que l'on cherche à apprendre. Dans le cas où $b \neq 0$, on peut aussi mettre à jour le biais en observant simplement que celui-ci n'est rien d'autre qu'un poids comme les autres, mais dont l'entrée est fixée à -1. Ainsi :

$$\Delta b = -e \tag{5.10}$$

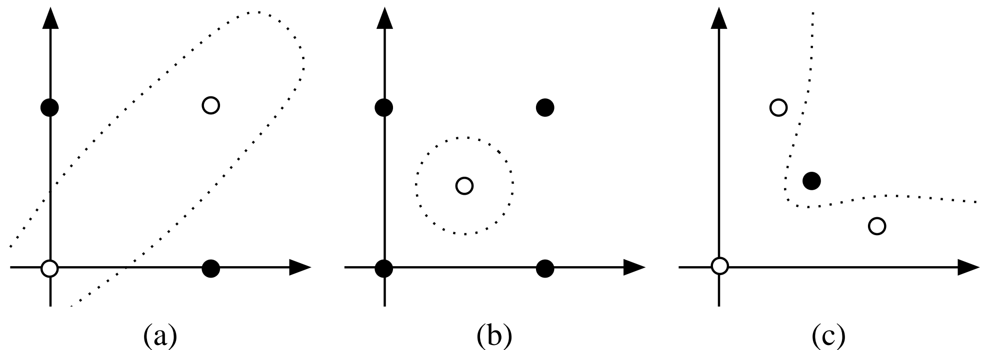


FIG. 5.4 – *Exemples de problèmes non linéairement séparables.*

Et dans le cas général où l'on dispose de S neurones, on peut réécrire l'équation 5.9 sous forme matricielle de la façon suivante :

$$\Delta \mathbf{W} = \mathbf{e}\mathbf{p}^T \quad (5.11)$$

$$\Delta \mathbf{b} = -\mathbf{e} \quad (5.12)$$

où $\mathbf{e} = [e_1 e_2 \cdots e_S]^T = \mathbf{d} - \mathbf{a}$ est le vecteur des erreurs que l'on observe en sortie pour le stimulus $\mathbf{p}$.

Malgré sa relative simplicité, la règle du perceptron s'avère très puissante. Vous pouvez facilement expérimenter avec cette règle grâce à la «Neural Network toolbox» de Matlab, programme de démonstration `nnd4pr`.

Nous ne démontrerons pas ici qu'elle converge toujours vers une solution en un nombre fini d'itérations, mais sachez qu'une telle preuve existe. Il importe cependant de connaître les hypothèses sous-jacentes à cette preuve :

1. Le problème doit être linéairement séparable ;
2. Les poids ne sont mis à jour que lorsqu'un stimulus d'entrée est classé incorrectement ;
3. Il existe une borne supérieure sur la norme des vecteurs de poids.

La première hypothèse va de soit car s'il n'existe aucune solution linéaire au problème, on ne peut pas s'attendre à ce qu'un réseau qui ne peut produire que des solutions linéaires puisse converger ! La deuxième hypothèse est implicite dans l'équation 5.11. Lorsque le signal d'erreur $\mathbf{e}$ est nul, le changement de poids $\Delta\mathbf{W}$ est également nul. La troisième hypothèse est plus subtile mais non limitative. Si l'on s'arrange pour conserver le ratio $\frac{||\mathbf{w}||}{b}$ constant, sans changer l'orientation de $\mathbf{w}$ pour un neurone donné, on ne change aucunement la frontière de décision que ce neurone engendre. Sans perte de généralité, on peut donc réduire la norme des poids lorsque celle-ci devient trop grande.

Mais qu'entend-on par un problème à deux classes «linéairement séparables» ? Et bien simplement un problème de classification dont la frontière de décision permettant de séparer les deux classes peut s'exprimer sous la forme d'un hyperplan (plan dans un espace à n dimensions). Par exemple, les problèmes de la figure 5.4 ne sont pas séparables en deux dimensions (par de simples droites). Des frontières possibles sont dessinées en pointillés. Elles sont toutes non linéaires.

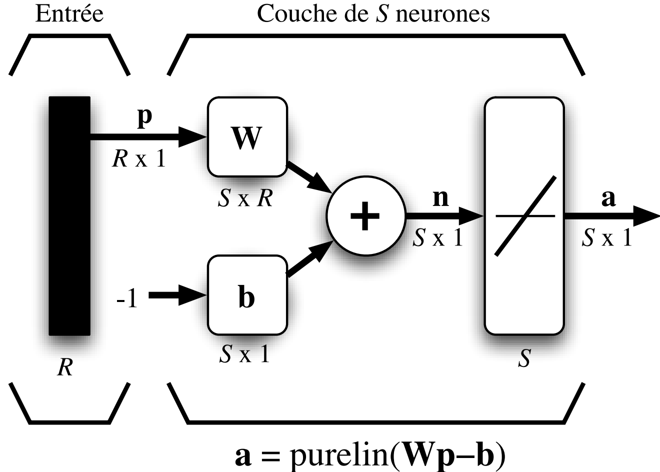


FIG. 5.5 – *Réseau ADALINE.*

5.2 Règle LMS

À la section précédente, nous avons traité le cas du perceptron simple où les neurones utilisent une fonction de transfert de type «seuil». Nous allons maintenant considérer la même architecture de réseau à une seule couche mais avec cette fois-ci une fonction de transfert linéaire comme à la figure 5.5. Ce réseau s'appelle «ADALINE» (en anglais «ADAptive LInear NEuron») à cause de sa fonction de transfert linéaire. Il souffre des mêmes limitations que le perceptron simple : il ne peut résoudre que des problèmes linéairement séparables. Cependant, son algorithme d'apprentissage, la règle du «Least Mean Square», est beaucoup plus puissante que la règle du perceptron original, car bien que cette dernière soit assurée de converger vers une solution, si celle-ci existe, le réseau résultant est parfois sensible au bruit puisque la frontière de décision se retrouve souvent trop proche des patrons d'apprentissage (l'algorithme s'arrête dès que tous les patrons sont biens classés). En revanche, la règle LMS minimise l'erreur quadratique moyenne, de sorte que la frontière de décision a tendance à se retrouver aussi loin que possible des prototypes.

En pratique, la règle du LMS a débouché vers de nombreuses applications dont une des plus fameuses est l'annulation de l'écho pour les communications téléphoniques. Lorsque que vous faites un appel inter-urbain ou outre-mer, vous vous trouvez peut-être, sans le savoir, à utiliser un réseau ADALINE !

Comme à la section 4.1 où nous avons développé le concept d'un apprentissage par correction des erreurs, et comme son nom l'indique, la règle LMS consiste à tenter de minimiser un indice de performance F basé sur l'erreur quadratique moyenne. Possédant un ensemble d'apprentissage de Q associations stimulus/cible $\{(\mathbf{p}_q, \mathbf{d}_q)\}$, $q = 1, \ldots, Q$, où $\mathbf{p}_q$ représente un vecteur stimulus (entrées) et $\mathbf{d}_q$ un vecteur cible (sorties désirées), à chaque instant t, on peut propager vers l'avant un stimulus différent $\mathbf{p}(t)$ à travers le réseau de la figure 5.5 pour obtenir un vecteur de sorties $\mathbf{a}(t)$. Ceci nous permet de calculer l'erreur $\mathbf{e}(t)$ entre ce que le réseau produit en sortie pour ce stimulus et la cible $\mathbf{d}(t)$ qui lui est associée :

$$\mathbf{e}(t) = \mathbf{d}(t) - \mathbf{a}(t). \tag{5.13}$$

Sachant que tous les neurones d'une même couche sont indépendants les uns des autres, et pour simplifier les équations, nous allons développer la règle LMS pour $S = 1$, c'est-à-dire le cas d'un seul neurone. Ensuite, nous pourrons facilement l'étendre au cas général de S neurones. Nous allons aussi regrouper tous les paramètres libres du neurone en un seul vecteur $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}. \tag{5.14}$$

De même, nous allons regrouper en un vecteur $\mathbf{y}$ le stimulus $\mathbf{p}$ et l'entrée virtuelle -1 associée au biais du neurone :

$$\mathbf{y} = \begin{bmatrix} \mathbf{p} \\ -1 \end{bmatrix}. \tag{5.15}$$

Ce qui nous permettra d'écrire la sortie a du neurone sous une forme simplifiée :

$$a = \mathbf{w}^T\mathbf{p} - b = \mathbf{x}^T\mathbf{y}. \tag{5.16}$$

Nous allons donc travailler avec le signal d'erreur scalaire $e(t) = d(t) - a(t)$ et construire notre indice de performance F en fonction du vecteur $\mathbf{x}$ des paramètres libres du neurone :

$$F(\mathbf{x}) = E\left[e^2(t)\right], \tag{5.17}$$

où $E[\cdot]$ désigne l'espérance mathématique. Le problème avec cette équation est que l'on ne peut pas facilement calculer cette espérance mathématique puisqu'on ne connaît pas les lois de probabilité de $\mathbf{x}$. On pourrait faire la moyenne des erreurs pour les Q associations d'apprentissage mais ce serait long. Une idée plus intéressante, et plus performante en pratique, consiste simplement à estimer l'erreur quadratique moyenne par l'erreur quadratique instantanée pour chaque association d'apprentissage :

$$\hat{F}(\mathbf{x}) = e^2(t). \tag{5.18}$$

Alors, à chaque itération de l'algorithme, on peut calculer le vecteur gradient de cet estimé :

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(t), \tag{5.19}$$

où les R premiers éléments de $\nabla e^2(t)$ correspondent aux dérivés partielles par rapport aux R poids du neurone, et le dernier élément correspond à la dérivé partielle par rapport à son biais. Ainsi :

$$\left[\nabla e^2(t)\right]_j = \frac{\partial e^2(t)}{\partial w_{1,j}} = 2e(t)\frac{\partial e(t)}{\partial w_{1,j}}, \quad j = 1, \ldots, R, \tag{5.20}$$

et :

$$\left[\nabla e^2(t)\right]_{R+1} = \frac{\partial e^2(t)}{\partial b} = 2e(t)\frac{\partial e(t)}{\partial b}. \tag{5.21}$$

Il s'agit maintenant de calculer les deux dérivés partielles de $e(t)$ par rapport à $w_{1,j}$:

$$\begin{aligned} \frac{\partial e(t)}{\partial w_{1,j}} &= \frac{\partial [d(t) - a(t)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[d(t) - ({}_1\mathbf{w}^T\mathbf{p}(t) - b_1)\right] \\ &= \frac{\partial}{\partial w_{1,j}}\left[d(t) - \left(\sum_{k=1}^{R} w_{1,k}\, p_k(t) - b_1\right)\right] \\ &= -p_j(t), \end{aligned} \tag{5.22}$$

et b :

$$\frac{\partial e(t)}{\partial b} = 1. \quad (5.23)$$

Notez bien que les termes $p_j(t)$ et -1 sont les éléments de $\mathbf{y}$, de sorte qu'on peut écrire :

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(t) = -2e(t)\mathbf{y}(t). \quad (5.24)$$

Ce résultat nous permet aussi d'apprécier la simplicité qu'engendre l'idée d'utiliser l'erreur instantanée plutôt que l'erreur moyenne. Pour calculer le gradient estimé de notre indice de performance, il suffit de multiplier l'erreur instantanée par le stimulus d'entrée !

L'équation 5.24 va nous permettre d'appliquer la méthode de la descente du gradient décrite par l'équation 4.7 (voir section 4.1, page 28) pour modifier les paramètres du neurone dans le sens d'une diminution de F :

$$\Delta\mathbf{x}(t) = \mathbf{x}(t+1) - \mathbf{x}(t) = -\eta \nabla F(\mathbf{x}) \Big|_{\mathbf{x}=\mathbf{x}(t)} . \quad (5.25)$$

En substituant $\nabla F(\mathbf{x})$ par $\hat{\nabla} F(\mathbf{x})$, on obtient :

$$\Delta\mathbf{x}(t) = 2\eta e(t)\mathbf{y}(t), \quad (5.26)$$

ce qui équivaut à :

$$\Delta\mathbf{w}(t) = 2\eta e(t)\mathbf{p}(t), \quad (5.27)$$
$$\Delta b(t) = -2\eta e(t). \quad (5.28)$$

Les équations 5.27 et 5.28 définissent la règle LMS de base. On la nomme également règle de Widrow-Hoff, du nom de ses auteurs. Dans le cas d'une couche de S neurones, nous pourrons mettre à jour chaque rangée i de la matrice de poids ainsi que chaque élément i du vecteur de biais à l'aide des équations suivantes :

$$\Delta_i\mathbf{w}(t) = 2\eta e_i(t)\mathbf{p}(t), \quad (5.29)$$
$$\Delta b_i(t) = -2\eta e_i(t). \quad (5.30)$$

Ce qui nous permet de réécrire le tout sous la forme matricielle :

$$\Delta\mathbf{W}(t) = 2\eta \mathbf{e}(t)\mathbf{p}^T(t), \quad (5.31)$$
$$\Delta\mathbf{b}(t) = -2\eta \mathbf{e}(t). \quad (5.32)$$

Même si nous ne démontrerons pas ici la convergence de l'algorithme LMS, il importe de retenir que pour les indices de performance quadratiques (comme dans le cas ADALINE), la méthode de la descente du gradient est garantie de converger vers un minimum global, à condition de restreindre la valeur du taux d'apprentissage. En pratique, nous sommes intéressés à fixer η le plus grand possible pour converger le plus rapidement possible (par de grands pas). Mais il existe un seuil à partir duquel un trop grand η peut faire diverger l'algorithme. Le gradient étant toujours perpendiculaire aux lignes de contour de $F(\mathbf{x})$, un petit η permettra de suivre ces lignes de contour vers le bas jusqu'à ce qu'on rencontre le minimum global. En voulant aller trop vite, l'algorithme

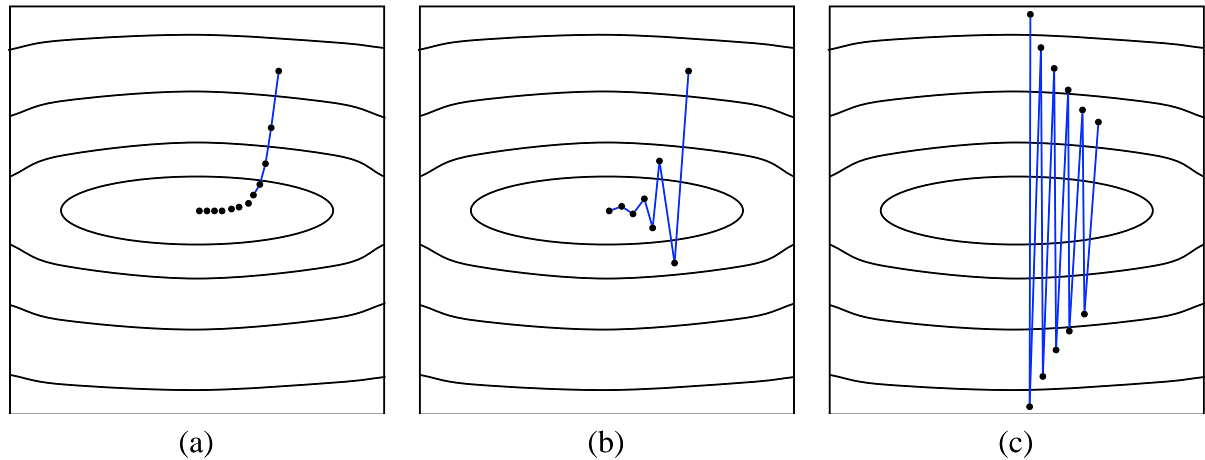


FIG. 5.6 – *Trajectoire de la descente du gradient pour différents taux d'apprentissage : (a) taux faible ; (b) taux moyen ; (c) taux (trop) élevé.*

peut sauter par dessus un contour et se mettre à osciller. Dans le cas quadratique, les lignes de contour ont une forme elliptique comme à la figure 5.6. Lorsque le taux est faible, la trajectoire est continue mais peut converger lentement vers l'optimum. Avec un taux plus élevé (moyen), les pas sont plus grands mais peuvent avoir tendance à osciller. On atteint normalement l'optimum plus rapidement. Lorsque le taux est trop élevé, l'algorithme peut diverger.

On peut montrer que pour garantir la convergence de l'algorithme LMS avec le réseau ADALINE, il faut que $0 < \eta < \frac{1}{\lambda_{max}}$ où λ_{max} est la plus grande valeur propre de la matrice $E\left[\mathbf{y}\,\mathbf{y}^T\right]$. Pour initialiser l'algorithme, il s'agit simplement de fixer tous les poids et biais du réseau à zéro. Puis, pour réaliser l'apprentissage, il s'agit de lui présenter toutes les associations stimulus/cible disponibles, à tour de rôle, et de mettre les poids à jour à chaque fois en utilisant les équations 5.31 et 5.32. Une période d'entraînement correspond à appliquer ces équations une fois pour chaque couple $(\mathbf{p}_i, \mathbf{d}_i)$, $i = 1, \ldots, Q$. Notez qu'il peut être avantageux de permuter l'ordre de présentation à chaque période. L'algorithme itère ainsi jusqu'à un nombre maximum (fixé a priori) de périodes ou encore jusqu'à ce que la somme des erreurs quadratiques en sortie soit inférieure à un certain seuil.

5.3 Réseau multicouche

Jusqu'à présent, nous n'avons traité que des réseaux à une seule couche de neurones. Nous avons aussi vu que ces réseaux ne pouvaient résoudre que des problèmes de classification linéairement séparables. Les réseaux multicouches permettent de lever cette limitation. On peut même démontrer qu'avec un réseau de trois couches (deux couches cachées + une couche de sortie), comme celui de la figure 2.7 (voir page 14), on peut construire des frontières de décision de complexité quelconque, ouvertes ou fermées, concaves ou convexes, à condition d'employer une fonction de transfert non linéaire et de disposer de suffisamment de neurones sur les couches cachées.

Un réseau multicouche n'est rien d'autre qu'un assemblage de couches concaténées les unes

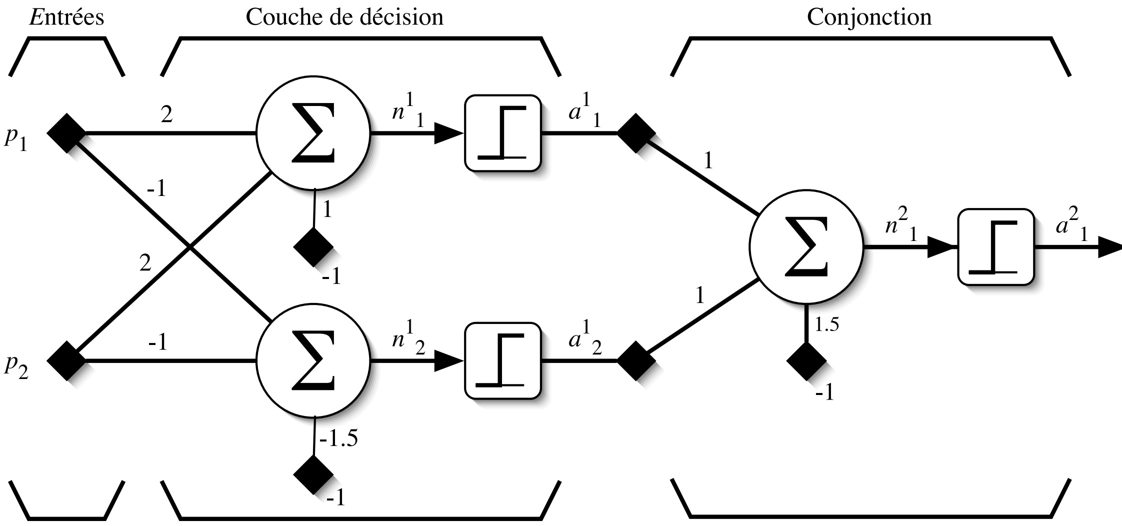


FIG. 5.7 – *Réseau multicouche pour résoudre le problème du «ou exclusif».*

aux autres, de la gauche vers la droite, en prenant les sorties d'une couche et en les injectant comme les entrées de la couche suivante. À la section suivante, nous allons développer l'algorithme dit de «rétropropagation des erreurs» qui permet d'entraîner un réseau multicouche. Mais pour l'instant nous allons tenter d'illustrer à quoi servent les couches supplémentaires. Une chose que l'on peut déjà remarquer est qu'il ne sert à rien d'assembler plusieurs couches ADALINE car la combinaison de plusieurs couches linéaires peut toujours se ramener à une seule couche linéaire équivalente. C'est pourquoi, pour être utile, un réseau multicouche doit toujours posséder des neurones avec fonctions de transfert non-linéaires sur ses couches cachées. Sur sa couche de sortie, selon le type d'application, il pourra comporter des neurones linéaires ou non-linéaires.

5.3.1 Problème du «ou exclusif»

À la figure 5.4a, nous avons illustré un problème de classification non séparable linéairement. Il s'agit du problème classique du «ou exclusif» (xor) que l'on ne peut pas résoudre ni avec un perceptron simple, ni avec un réseau ADALINE, car les points noirs ne peuvent pas être séparés des blancs à l'aide d'une seule frontière de décision linéaire. Dans ce problème, les points noirs représentent le vrai (valeur 1) et les points blancs le faux (valeur 0). Le «ou exclusif», pour être vrai, exige qu'une seule de ses entrées soit vraie, sinon il est faux. On peut résoudre facilement ce problème à l'aide du réseau multicouche illustré à la figure 5.7. Ce réseau à deux couches utilise des fonctions de transfert seuil. Sur la première couche, chaque neurone engendre les frontières de décision illustrées aux figures 5.8a et 5.8b. Les zones grisées représentent la région de l'espace d'entrée du réseau pour laquelle le neurone correspondant produit une réponse vrai. Le rôle du neurone sur la couche de sortie, illustré à la figure 5.8c, consiste à effectuer la conjonction des deux régions produites par les neurones de la première couche. Notez bien que les entrées de la deuxième couche sont les sorties de la première couche. La figure 5.8 représente toutes les frontières de décision dans l'espace des entrées. La frontière de décision engendrée par le neurone

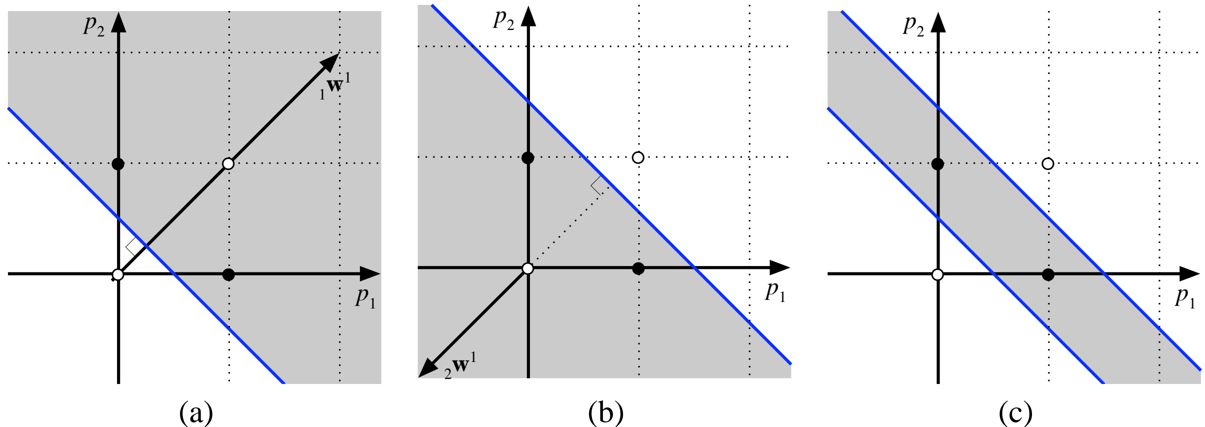


FIG. 5.8 – *Frontières de décision engendrées par le réseau de la figure 5.7 : (a) neurone 1 de la couche 1 ; (b) neurone 2 de la couche 1 ; (c) neurone 1 de la couche 2.*

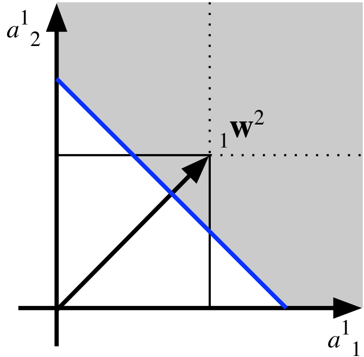


FIG. 5.9 – *Frontière de décision engendrée par le neurone qui effectue une conjonction.*

de la couche de sortie est aussi illustrée dans son propre espace d'entrée à la figure 5.9. Il importe de remarquer que la sortie des fonctions seuils employées étant limitée aux valeurs $\{0, 1\}$ (que l'on interprète comme étant respectivement faux et vrai), seuls les coins du carré illustré à la figure sont pertinents. Pour réaliser une conjonction (un «et logique»), le neurone effectue donc la somme de ses deux entrées et fixe un seuil à 1.5. Si la somme est inférieur à 1.5, alors il produit vrai en sortie, sinon il produit faux. Dans ce cas, seul le coin supérieur droite du carré produit vrai en sortie.

Mentionnons finalement que le réseau de la figure 5.7 n'est pas le seul à pouvoir résoudre ce problème du «ou exclusif». D'autres combinaisons de poids et de biais pourraient produire le même résultat (pouvez-vous en trouver d'autres ?).

5.3.2 Approximation de fonction

Pour faire de l'approximation de fonction (section 4.8), on peut montrer qu'un réseau multicouche comme celui de la figure 5.10, avec une seule couche cachée de neurones sigmoïdes et une couche de sortie avec des neurones linéaires permet d'approximer n'importe quelle fonction d'intérêt avec une précision arbitraire, à condition de disposer de suffisamment de neurones sur la

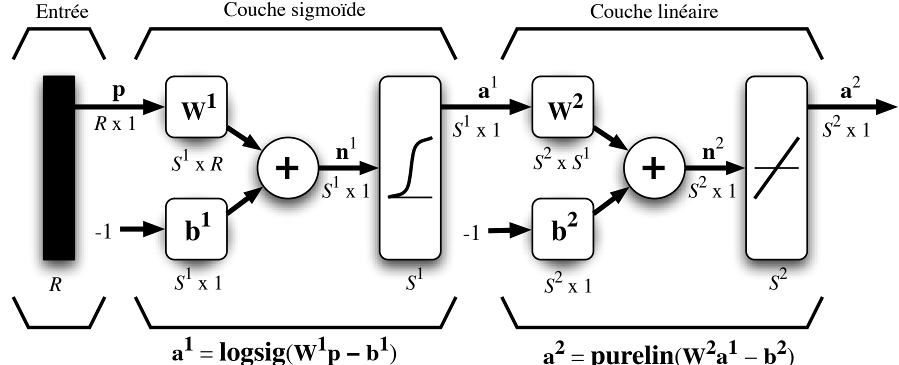


FIG. 5.10 – *Réseau multicouche permettant de faire de l'approximation de fonction.*

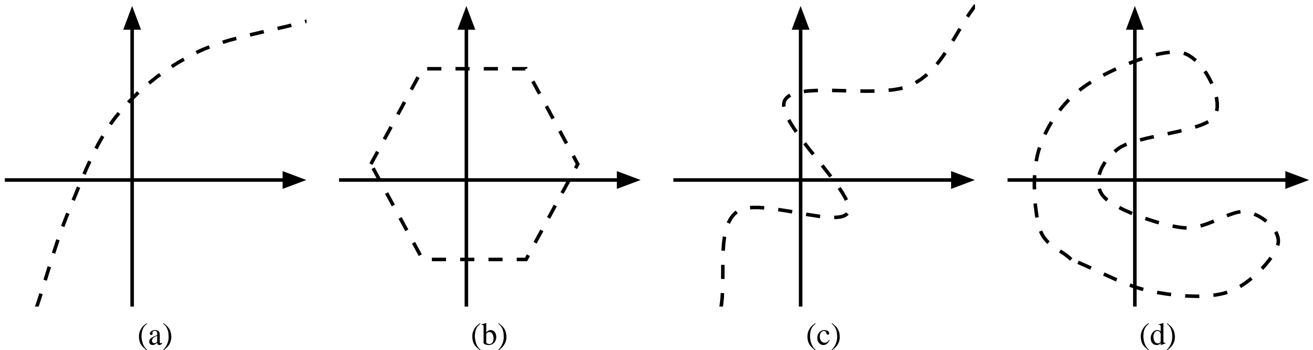


FIG. 5.11 – *Exemples de frontières de décision : (a) convexe ouverte ; (b) convexe fermée ; (c) concave ouverte ; et (d) concave fermée.*

couche cachée. Intuitivement, un peu à la façon des séries de Fourier qui utilisent des sinus et cosinus, cette preuve passe par la démonstration que l'on peut approximer n'importe quelle fonction d'intérêt par une combinaison linéaire de sigmoïdes.

5.3.3 Classification

Pour faire de la classification, on utilisera des réseaux soit à deux, soit à trois couches de neurones sigmoïdes. On peut montrer qu'une seule couche cachée suffit à engendrer des frontières de décision[4] convexes, ouvertes ou fermées, de complexité arbitraire, alors que deux couches cachées permettent de créer des frontières de décision concaves[5] ou convexes, ouvertes ou fermées, de complexité arbitraire. La figure 5.11 montre en deux dimensions différents types de frontières de décision. Intuitivement, on veut voir que la première couche cachée d'un tel réseau sert à découper l'espace d'entrée à l'aide de frontières de décision linéaires, comme on l'a vu pour le perceptron simple, la deuxième couche sert à assembler des frontières de décision non-linéaires[6] convexes en

[4]Notez bien qu'une frontière de décision n'est pas nécessairement une fonction !

[5]Une courbe (surface) convexe ne comporte aucun changement dans le signe de la courbure, alors qu'une courbe concave implique un point d'inflexion.

[6]Les non-linéarités proviennent des sigmoïdes !

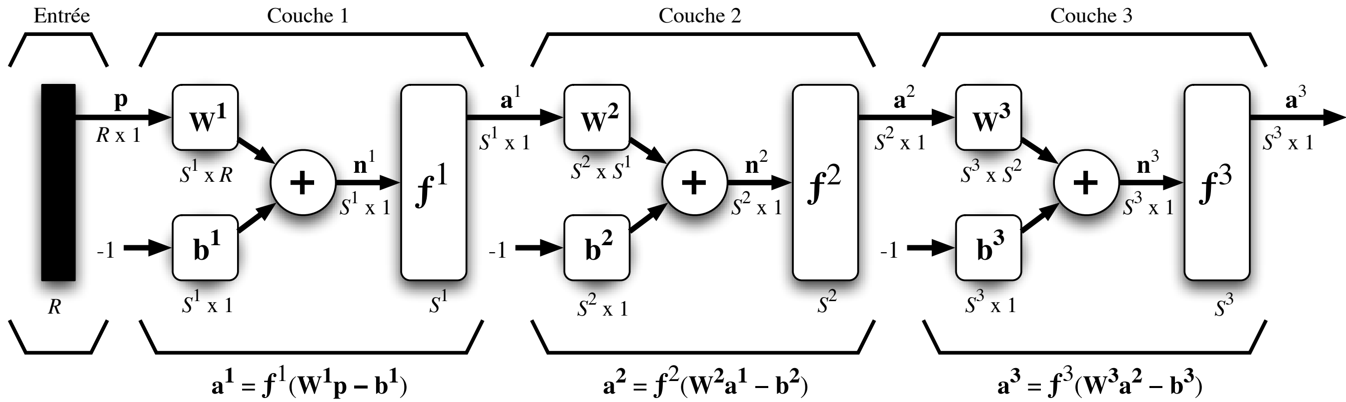


FIG. 5.12 – *Représentation matricielle d'un réseau de trois couches (reproduction de la figure 2.7).*

sélectionnant ou en retranchant des régions engendrées par la couche précédente et, de même, la couche de sortie permet d'assembler des frontières de décision concaves en sélectionnant ou en retranchant des régions convexes engendrées par la couche précédente.

Avant de passer à l'algorithme de rétropropagation qui nous permettra d'entraîner un réseau multicouche, que nous nommerons dorénavant *perceptron multicouche* ou PMC, mentionnons que ce n'est pas par hasard que nous avons remplacé la fonction de transfert seuil par la fonction sigmoïde, mais bien pour pouvoir procéder à un apprentissage automatique. Par exemple, même si nous avons pu construire à la main, avec la fonction seuil, le réseau de la figure 5.7 pour résoudre le problème du «ou exclusif», nous ne saurions pas comment apprendre automatiquement à générer les bons poids et les bons biais de ce réseau. Le problème avec la fonction seuil est que sa dérivée est toujours nulle sauf en un point ou elle n'est même pas définie ! On ne peut donc pas l'utiliser avec la méthode de la descente du gradient qui nous a si bien servi pour le réseau ADALINE.

5.4 Rétropropagation des erreurs

Pour développer les équations de l'algorithme de rétropropagation des erreur (en anglais «backpropagation»), nous aurons besoin de toute la puissance des notations introduites à la section 2.1 (voir page 5) et illustrées à la figure 2.7 que nous reproduisons à la figure 5.12.

L'équation qui décrit les sorties d'une couche k dans un perceptron multicouche est donnée par :

$$\mathbf{a}^k = \mathbf{f}^k\left(\mathbf{W}^k\mathbf{a}^{k-1} - \mathbf{b}^k\right), \text{ pour } k = 1, \ldots, M, \tag{5.33}$$

où M est le nombre total de couches et $\mathbf{a}^0 = \mathbf{p}$ définit le cas de base de cette formule de récurrence. Les sorties du réseau correspondent alors à $\mathbf{a}^M$. L'algorithme de rétropropagation est une généralisation de la règle LMS. Tous deux utilisent comme indice de performance l'erreur quadratique moyenne, et tous deux permettent un apprentissage de type supervisé avec un ensemble d'association stimulus/cible $\{(\mathbf{p}_q, \mathbf{d}_q)\}$, $q = 1, \ldots, Q$, où $\mathbf{p}_q$ représente un vecteur stimulus (entrées) et $\mathbf{d}_q$ un vecteur cible (sorties désirées). À chaque instant t, on peut propager vers

l'avant un stimulus différent $\mathbf{p}(t)$ à travers le réseau de la figure 5.12 pour obtenir un vecteur de sorties $\mathbf{a}(t)$. Ceci nous permet de calculer l'erreur $\mathbf{e}(t)$ entre ce que le réseau produit en sortie pour ce stimulus et la cible $\mathbf{d}(t)$ qui lui est associée :

$$\mathbf{e}(t) = \mathbf{d}(t) - \mathbf{a}(t). \tag{5.34}$$

L'indice de performance F permet de minimiser l'erreur quadratique moyenne :

$$F(\mathbf{x}) = E\left[\mathbf{e}^T(t)\mathbf{e}(t)\right] \tag{5.35}$$

où $E[.]$ désigne l'espérance mathématique et le vecteur $\mathbf{x}$ regroupe l'ensemble des poids et des biais du réseau. Tout comme pour la règle LMS, nous allons approximer cet indice par l'erreur instantanée :

$$\hat{F}(\mathbf{x}) = \mathbf{e}^T(t)\mathbf{e}(t) \tag{5.36}$$

et nous allons utiliser la méthode de la descente du gradient pour optimiser $\mathbf{x}$:

$$\Delta w^k_{i,j}(t) = -\eta \frac{\partial \hat{F}}{\partial w^k_{i,j}} \tag{5.37}$$

$$\Delta b^k_i(t) = -\eta \frac{\partial \hat{F}}{\partial b^k_i} \tag{5.38}$$

où η désigne le taux d'apprentissage.

La procédure d'optimisation est donc très semblable à celle de la règle LMS. Cependant, il faut faire face à deux difficultés supplémentaires. Premièrement, les fonctions de transfert des neurones ne sont plus nécessairement linéaires. Leur dérivé partielle ne sera donc plus constante. Deuxièmement, nous ne possédons les sorties désirées (les cibles) que pour les neurones de la couche de sortie. C'est surtout cette deuxième observation qui va nous poser problème.

Pour calculer la dérivé partielle de $\hat{F}$, il faudra faire appel à la règle de chaînage des dérivés :

$$\frac{df[n(w)]}{dw} = \frac{df[n]}{dn} \times \frac{dn(w)}{dw}. \tag{5.39}$$

Par exemple, si $f[n] = e^n$ et $n = 2w$, donc $f[w] = e^{2w}$, alors :

$$\frac{df[n(w)]}{dw} = \left(\frac{de^n}{dn}\right) \times \left(\frac{d2w}{dw}\right) = (e^n)(2) = 2e^{2w}. \tag{5.40}$$

Nous allons nous servir de cette règle pour calculer les dérivés partielles des équations 5.37 et 5.38 :

$$\frac{\partial \hat{F}}{\partial w^k_{i,j}} = \frac{\partial \hat{F}}{\partial n^k_i} \times \frac{\partial n^k_i}{\partial w^k_{i,j}}, \tag{5.41}$$

$$\frac{\partial \hat{F}}{\partial b^k_i} = \frac{\partial \hat{F}}{\partial n^k_i} \times \frac{\partial n^k_i}{\partial b^k_i}. \tag{5.42}$$

Le deuxième terme de ces équations est facile à calculer car les niveaux d'activation n_i^k de la couche k dépendent directement des poids et des biais sur cette couche :

$$n_i^k = \sum_{j=1}^{S^{k-1}} w_{i,j}^k \, a_j^{k-1} - b_i^k. \tag{5.43}$$

Par conséquent :

$$\frac{\partial n_i^k}{\partial w_{i,j}^k} = a_j^{k-1}, \quad \frac{\partial n_i^k}{\partial b_i^k} = -1. \tag{5.44}$$

On remarque que cette partie de la dérivée partielle de $\hat{F}$ par rapport à un poids (ou un biais) est toujours égale à l'entrée de la connexion correspondante.

Maintenant, pour le premier terme des équations 5.41 et 5.42, définissons la sensibilité s_i^k de $\hat{F}$ aux changements dans le niveau d'activation n_i^k du neurone i de la couche k :

$$s_i^k \equiv \frac{\partial \hat{F}}{\partial n_i^k}. \tag{5.45}$$

On peut alors réécrire les équations 5.41 et 5.42 de la façon suivante :

$$\frac{\partial \hat{F}}{\partial w_{i,j}^k} = s_i^k a_j^{k-1}, \tag{5.46}$$

$$\frac{\partial \hat{F}}{\partial b_i^k} = -s_i^k, \tag{5.47}$$

et les expressions des équations 5.37 et 5.38 de la façon suivante :

$$\Delta w_{i,j}^k(t) = -\eta s_i^k(t) a_j^{k-1}(t), \tag{5.48}$$

$$\Delta b_i^k(t) = \eta s_i^k(t), \tag{5.49}$$

ce qui donne en notation matricielle :

$$\Delta \mathbf{W}^k(t) = -\eta \mathbf{s}^k(t) (\mathbf{a}^{k-1})^T(t), \tag{5.50}$$

$$\Delta \mathbf{b}^k(t) = \eta \mathbf{s}^k(t), \tag{5.51}$$

avec :

$$\mathbf{s}^k \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^k} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^k} \\ \frac{\partial \hat{F}}{\partial n_2^k} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^k}^k} \end{bmatrix}. \tag{5.52}$$

Par rapport à la règle LMS, il est intéressant de noter la ressemblance des équations ci-dessus avec les équations 5.31 et 5.32. On remarque que le terme $2\mathbf{e}(t)$ est simplement remplacé par $\mathbf{s}^M(t)$.

5.4.1 Calcul des sensibilités

Il reste maintenant à calculer les sensibilités $\mathbf{s}^k$, ce qui requerra une nouvelle application de la règle de chaînage des dérivés. Dans ce cas, nous obtiendrons une formule de récurrence où la sensibilité des couches en amont (entrées) dépendra de la sensibilité des couches en aval (sorties). C'est de là que provient l'expression «rétropropagation», car le sens de propagation de l'information est inversé par rapport à celui de l'équation 5.33.

Pour dériver la formule de récurrence des sensibilités, nous allons commencer par calculer la matrice suivante :

$$\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k} = \begin{bmatrix} \frac{\partial n_1^{k+1}}{\partial n_1^k} & \frac{\partial n_1^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_1^{k+1}}{\partial n_{S^k}^k} \\ \frac{\partial n_2^{k+1}}{\partial n_1^k} & \frac{\partial n_2^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_2^{k+1}}{\partial n_{S^k}^k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial n_{S^{k+1}}^{k+1}}{\partial n_1^k} & \frac{\partial n_{S^{k+1}}^{k+1}}{\partial n_2^k} & \cdots & \frac{\partial n_{S^{k+1}}^{k+1}}{\partial n_{S^k}^k} \end{bmatrix}. \tag{5.53}$$

Cette matrice énumère toutes les sensibilités des niveaux d'activation d'une couche par rapport à ceux de la couche précédente. Considérons chaque élément (i, j) de cette matrice :

$$\begin{aligned} \frac{\partial n_i^{k+1}}{\partial n_j^k} &= \frac{\partial}{\partial n_j^k}\left(\sum_{l=1}^{S^k} w_{i,l}^{k+1} a_l^k - b_i^{k+1}\right) = w_{i,j}^{k+1}\frac{\partial a_j^k}{\partial n_j^k} \\ &= w_{i,j}^{k+1}\frac{\partial f^k(n_j^k)}{\partial n_j^k} = w_{i,j}^{k+1}\dot{f}^k(n_j^k), \end{aligned} \tag{5.54}$$

avec :

$$\dot{f}^k(n_j^k) = \frac{\partial f^k(n_j^k)}{\partial n_j^k}. \tag{5.55}$$

Par conséquent, la matrice de l'équation 5.53 peut s'écrire de la façon suivante :

$$\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k} = \mathbf{W}^{k+1}\dot{\mathbf{F}}^k(\mathbf{n}^k), \tag{5.56}$$

où :

$$\dot{\mathbf{F}}^k(\mathbf{n}^k) = \begin{bmatrix} \dot{f}^k(n_1^k) & 0 & \cdots & 0 \\ 0 & \dot{f}^k(n_2^k) & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^k(n_{S^k}^k) \end{bmatrix}. \tag{5.57}$$

Ceci nous permet maintenant d'écrire la relation de récurrence pour les sensibilités :

$$\begin{aligned} \mathbf{s}^k &= \frac{\partial \hat{F}}{\partial \mathbf{n}^k} = \left(\frac{\partial \mathbf{n}^{k+1}}{\partial \mathbf{n}^k}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{k+1}} = \dot{\mathbf{F}}^k\left(\mathbf{n}^k\right)\left(\mathbf{W}^{k+1}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{k+1}} \\ &= \dot{\mathbf{F}}^k\left(\mathbf{n}^k\right)\left(\mathbf{W}^{k+1}\right)^T \mathbf{s}^{k+1}. \end{aligned} \tag{5.58}$$

Cette équation nous permet de calculer s^1 à partir de s^2, qui lui-même est calculé à partir de s^3, etc., jusqu'à s^M. Ainsi les sensibilités sont rétropropagées de la couche de sortie jusqu'à la couche d'entrée :

$$\mathbf{s}^M \to \mathbf{s}^{M-1} \to \cdots \mathbf{s}^2 \to \mathbf{s}^1. \tag{5.59}$$

Il ne nous reste plus qu'à trouver le cas de base, s^M, permettant de mettre fin à la récurrence :

$$\begin{aligned} s_i^M &= \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial \left(\mathbf{d} - \mathbf{a}^M\right)^T \left(\mathbf{d} - \mathbf{a}^M\right)}{\partial n_i^M} = \frac{\partial}{\partial n_i^M}\left(\sum_{l=1}^{S^M}(d_l - a_l^M)^2\right) \\ &= -2\left(d_i - a_i^M\right)\frac{\partial a_i^M}{\partial n_i^M} \\ &= -2\left(d_i - a_i^M\right)\dot{f}^M\!\left(n_i^M\right). \end{aligned} \tag{5.60}$$

En notation matricielle, on écrit :

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M\!\left(\mathbf{n}^M\right)\left(\mathbf{d} - \mathbf{a}^M\right). \tag{5.61}$$

5.4.2 Algorithme d'entraînement

Voici donc un résumé de la démarche à suivre pour entraîner un perceptron multicouche :

1. Initialiser tous les poids du réseau à de petites valeurs aléatoires.
2. Pour chaque association $(\mathbf{p}_q, \mathbf{d}_q)$ dans la base d'apprentissage :
   (a) Propager les entrées $\mathbf{p}_q$ vers l'avant à travers les couches du réseau :

$$\mathbf{a}^0 = \mathbf{p}_q, \tag{5.62}$$

$$\mathbf{a}^k = \mathbf{f}^k\left(\mathbf{W}^k\mathbf{a}^{k-1} - \mathbf{b}^k\right), \text{ pour } k = 1, \ldots, M. \tag{5.63}$$

   (b) Rétropropager les sensibilités vers l'arrière à travers les couches du réseau :

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M\!\left(\mathbf{n}^M\right)\left(\mathbf{d}_q - \mathbf{a}^M\right), \tag{5.64}$$

$$\mathbf{s}^k = \dot{\mathbf{F}}^k\!\left(\mathbf{n}^k\right)\left(\mathbf{W}^{k+1}\right)^T\mathbf{s}^{k+1}, \text{ pour } k = M-1, \ldots, 1. \tag{5.65}$$

   (c) Mettre à jour les poids et biais :

$$\Delta\mathbf{W}^k = -\eta\mathbf{s}^k\left(\mathbf{a}^{k-1}\right)^T, \text{ pour } k = 1, \ldots, M, \tag{5.66}$$

$$\Delta\mathbf{b}^k = \eta\mathbf{s}^k, \text{ pour } k = 1, \ldots, M. \tag{5.67}$$

3. Si le critère d'arrêt est atteint, alors **stop**.
4. Sinon, permuter l'ordre de présentation des associations de la base d'apprentissage.
5. Recommencer à l'étape 2.

5.4.3 Critères d'arrêt

Plusieurs critères d'arrêts peuvent être utilisés avec l'algorithme de rétropropagation des erreurs. Le plus commun consiste à fixer un nombre maximum de périodes d'entraînement, ce qui fixe effectivement une limite supérieure sur la durée de l'apprentissage. Ce critère est important car la rétropropagation des erreurs n'offre aucune garantie quant à la convergence de l'algorithme. Il peut arriver, par exemple, que le processus d'optimisation reste pris dans un minimum local. Sans un tel critère, l'algorithme pourrait ne jamais se terminer.

Un deuxième critère commun consiste à fixer une borne inférieure sur l'erreur quadratique moyenne, ou encore sur la racine[7] carrée de cette erreur. Dépendant de l'application, il est parfois possible de fixer a priori un objectif à atteindre. Lorsque l'indice de performance choisi diminue en dessous de cet objectif, on considère simplement que le PMC a suffisamment bien appris ses données et on arrête l'apprentissage.

Les deux critères précédents sont utiles mais ils comportent aussi des limitations. Le critère relatif au nombre maximum de périodes d'entraînement n'est aucunement lié à la performance du réseau. Le critère relatif à l'erreur minimale obtenue mesure quant à lui un indice de performance mais ce dernier peut engendrer un phénomène dit de sur-apprentissage qui n'est pas désirable dans la pratique, surtout si l'on ne possède pas une grande quantité de données d'apprentissage, ou si ces dernières ne sont pas de bonne qualité.

Un processus d'apprentissage par correction des erreurs, comme celui de la rétropropagation, vise à réduire autant que possible l'erreur que commet le réseau. Mais cette erreur est mesurée sur un ensemble de données d'apprentissage. Si les données sont bonnes, c'est-à-dire quelles représentent bien le processus physique sous-jacent que l'on tente d'apprendre ou de modéliser, et que l'algorithme a convergé sur un optimum global, alors il devrait bien performer sur d'autres données issues du même processus physique. Cependant, si les données d'apprentissage sont partiellement corrompues par du bruit ou par des erreurs de mesure, alors il n'est pas évident que la performance optimale du réseau sera atteinte en minimisant l'erreur, lorsqu'on la testera sur un jeu de données différent de celui qui a servi à l'entraînement. On parle alors de la capacité du réseau à généraliser, c'est-à-dire de bien performer avec des données qu'il n'a jamais vu auparavant.

Par exemple, la figure 5.13 illustre le problème du sur-apprentissage dans le contexte d'une tâche d'approximation de fonction (voir section 4.8). La droite en pointillés montre une fonction linéaire que l'on voudrait approximer en ne connaissant que les points noirs. La courbe en trait plein montre ce qu'un réseau hypothétique pourrait apprendre. On constate que la courbe passe par tous les points d'entraînement et donc que l'erreur est nulle. De toute évidence, ce réseau ne généralisera pas bien si l'on échantillonne d'autres points sur la droite !

Une solution à ce problème consiste à utiliser un autre critère d'arrêt basé sur une technique dite de validation croisée (en anglais «cross-validation»). Cette technique consiste à utiliser deux ensembles indépendants[8] de données pour entraîner notre réseau : un pour l'apprentissage (l'ajustement des poids) et l'autre pour la validation, c'est-à-dire le calcul d'un indice de performance (une

[7] On parle alors de la racine de l'erreur quadratique moyenne. En anglais, on dit «Root Mean Square» ou RMS.

[8] En pratique cela consiste à partitionner les données disponibles en deux ensembles distincts.

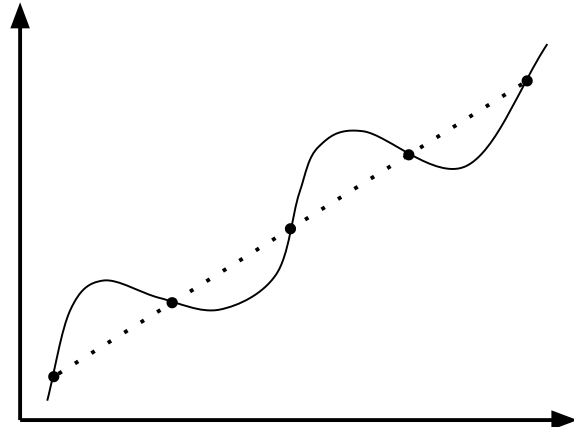


FIG. 5.13 – *Illustration du phénomène de sur-apprentissage pour le cas simple d'une approximation de fonction.*

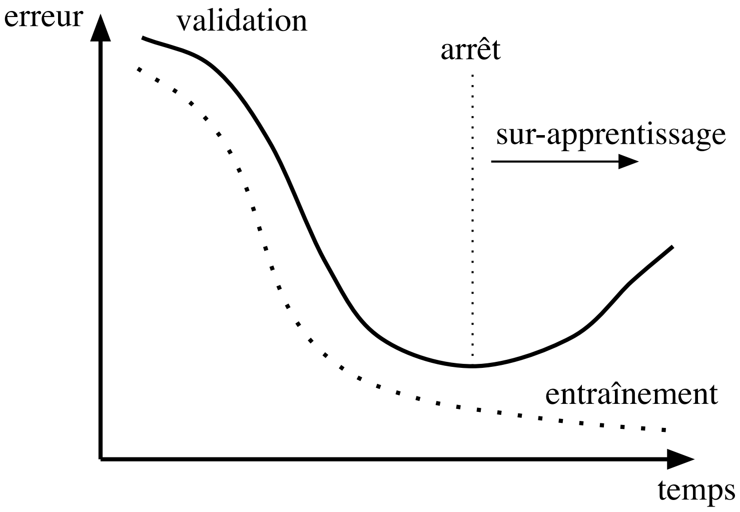


FIG. 5.14 – *Illustration de la validation croisée.*

erreur, un taux de reconnaissance ou tout autre mesure pertinente à l'application). Le critère d'arrêt consiste alors à stopper l'apprentissage lorsque l'indice de performance calculé sur les données de validation cesse de s'améliorer pendant plusieurs périodes d'entraînement. La figure 5.14 illustre le critère de la validation croisée dans le cas d'un indice de performance que l'on cherche à minimiser. La courbe en pointillés de ce graphique représente l'indice de performance d'un réseau hypothétique[9] calculé sur les données d'apprentissage, alors que la courbe en trait plein montre le même indice mais calculé sur les données de validation. On voit qu'il peut exister un moment au cours de l'apprentissage où l'indice en validation se détériore alors que le même indice continue à s'améliorer pour les données d'entraînement. C'est alors le début du «sur-apprentissage».

[9]Des courbes semblables s'observent couramment dans la pratique.

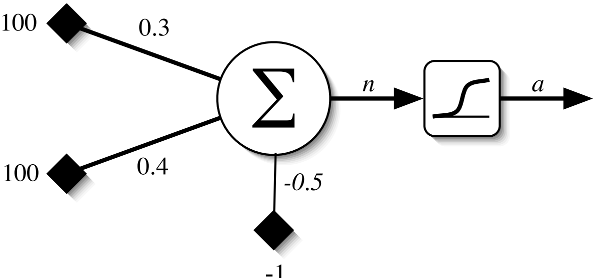


FIG. 5.15 – *Exemple d'un neurone saturé.*

5.4.4 Phénomène de saturation

Une autre considération pratique dont on doit tenir compte lorsqu'on entraîne un PMC concerne le phénomène de saturation des neurones où, sous certaines conditions, les neurones peuvent à toute fin pratique cesser d'apprendre tellement leur convergence devient lente. Considérons par exemple le réseau de la figure 5.15, constitué d'un seul neurone à deux entrées avec $p_1 = p_2 = 100$. Si l'on calcule son niveau d'activation n, on obtient :

$$n = 100 \times 0.3 + 100 \times 0.4 + 0.5 = 70.5 \tag{5.68}$$

On peut déjà remarquer que l'effet du biais est négligeable devant celui des deux poids d'entrée, malgré le fait qu'ils soient tous les trois du même ordre de grandeur, à cause de l'amplitude des entrées. Si l'on calcule la sortie du neurone, on obtient :

$$a = \text{logsig}(n) = \frac{1}{1+\exp(-n)} = \frac{1}{1+\exp(-70.5)} \approx 1. \tag{5.69}$$

En effet, $\exp(-70.5) = 2.4 \times 10^{-31}$. On dit alors que le neurone est saturé. Le problème avec un tel neurone est qu'il ne peut presque plus apprendre car la dérivée de sa fonction d'activation est pratiquement nulle :

$$\begin{aligned}\dot{a} = \frac{da}{dn} &= \frac{d}{dn}\left[\frac{1}{1+\exp(-n)}\right] = \frac{(-1)\frac{d}{dn}(1+\exp(-n))}{(1+\exp(-n))^2} = \frac{\exp(-n)}{(1+\exp(-n))^2} \\ &= a \times \frac{\exp(-n)}{1+\exp(-n)} = a \times \frac{1+\exp(-n)-1}{1+\exp(-n)} \\ &= a(1-a)\end{aligned} \tag{5.70}$$

Avec $a \approx 1$, on obtient :

$$\dot{a} \approx 1 \times (1-1) = 0 \tag{5.71}$$

Or, comme les variations de poids dans l'algorithme de rétropropagation des erreurs, définies aux équations 5.66 et 5.67, dépendent linéairement des sensibilités (voir équations 5.64 et 5.65) qui elles-mêmes dépendent de la dérivée de la fonction d'activation, on voit immédiatement qu'elles tendent vers zéro lorsque le neurone est saturé et que la convergence, même si elle est toujours possible, requerra beaucoup de périodes d'apprentissage.

Par conséquent, à cause de ce phénomène de saturation, il importe de normaliser les données à l'entrée d'un PMC, c'est-à-dire de les transformer de manière à éviter tout risque de saturation. Une autre façon de procéder est d'initialiser les poids sur la première couche en choisissant un intervalle de valeurs aléatoires ajusté aux stimuli d'apprentissage. Par exemple, pour l'entrée j d'un réseau à R entrées, on pourrait choisir l'intervalle suivant :

$$\left[\frac{1}{-\max_q |p_j^q|}, \frac{1}{\max_q |p_j^q|}\right], j = 1, \ldots, R, \tag{5.72}$$

où $\{q\}$ désigne l'ensemble des stimuli d'apprentissage.

Une autre alternative serait de fixer tous les poids à zéro. Bien que ceci réglerait certes le problème de la saturation des neurones, ce n'est malheureusement pas une alternative viable. En effet, il se trouve que l'origine de l'espace des poids correspond souvent à un lieu d'instabilité de la fonction d'erreur du réseau. Et ceci peut facilement entraîner la divergence de l'algorithme de rétropropagation.

5.4.5 Groupage

Au lieu de mettre à jour les poids pour chaque donnée d'entraînement, une alternative consiste à accumuler les variations de poids sur une période d'apprentissage complète et de mettre à jour les poids en une seule fois avec la moyenne de ces variations. On parle alors d'apprentissage «hors-ligne» ou par groupage (en anglais «batching»). L'idée est la suivante : l'estimation du gradient qu'engendre chaque donnée d'entraînement est peu précise, la moyenne de ces estimations devrait être plus près du gradient réel. En fait, si les données d'entraînement couvrent adéquatement l'espace des entrées, alors la moyenne de ces estimations sera exacte.

Mais le groupage n'est pas une panacée car cela peut aussi ralentir considérablement la convergence, puisque les poids changent moins souvent. Autrement dit, si l'estimation du gradient basée sur une seule donnée d'entraînement a tendance à être bonne, alors on pourrait converger jusqu'à Q fois plus lentement si l'on procède par groupage. Par contre, lorsque cette estimation est plutôt mauvaise, le groupage sert à éviter de partir dans une mauvaise direction qui, autrement, augmenterait nos chances de rester pris dans un minimum local inadéquat.

5.4.6 Momentum

Une façon d'améliorer l'algorithme de rétropropagation est de rajouter un terme d'inertie dont le rôle est de filtrer les oscillations dans la trajectoire de la descente du gradient :

$$\Delta\mathbf{W}^k(t) = \alpha\Delta\mathbf{W}^k(t-1) - (1-\alpha)\eta\mathbf{s}^k\left(\mathbf{a}^{k-1}\right)^T, \text{ pour } k = 1, \ldots, M, \tag{5.73}$$

$$\Delta\mathbf{b}^k(t) = \alpha\Delta\mathbf{b}^k(t-1) + (1-\alpha)\eta\mathbf{s}^k, \text{ pour } k = 1, \ldots, M. \tag{5.74}$$

où $0 \leq \alpha < 1$ s'appelle le momentum. Lorsque $\alpha = 0$, les équations 5.73 et 5.74 sont équivalentes aux équations 5.66 et 5.67, respectivement. Lorsque $\alpha = 1$, les $\Delta\mathbf{W}^k(t)$ et $\Delta\mathbf{b}^k(t)$ ne dépendent

plus des équations de rétropropagation des erreurs, mais uniquement des $\Delta\mathbf{W}^k(t-1)$ et $\Delta\mathbf{b}^k(t-1)$, c'est-à-dire des changements de poids à l'étape précédente.

Le terme du momentum produit deux effets distincts selon la situation. Premièrement, lorsque la trajectoire du gradient a tendance à osciller (comme à la figure 5.6c), il contribue à la stabiliser en ralentissant les changements de direction. Par exemple, avec $\alpha = 0.8$, cela correspond d'emblée à ajouter 80% du changement précédent au changement courant. Deuxièmement, lorsque le gradient courant pointe dans la même direction que le gradient précédent, le terme d'inertie contribue à augmenter l'ampleur du pas dans cette direction et donc à accélérer la convergence.

5.4.7 Taux d'apprentissage variable

Une autre façon d'améliorer la vitesse de convergence pour la rétropropagation des erreurs serait de modifier le taux d'apprentissage dynamiquement tout au long de l'entraînement. Plusieurs approches peuvent être considérées. Par exemple, on peut adopter la stratégie suivante :

1. Si l'erreur quadratique totale, calculée pour toutes les associations de la base d'apprentissage, augmente d'une période à l'autre par plus d'un certain pourcentage β (typiquement de 1 à 5%) à la suite d'une mise à jour des poids, alors cette mise à jour doit être abandonnée et le taux d'apprentissage doit être multiplié par un facteur $0 < \rho < 1$, et le momentum doit être fixé à zéro ;
2. Si l'erreur quadratique totale diminue à la suite d'une mise à jour des poids, alors celle-ci est conservée et le taux d'apprentissage est multiplié par un facteur $\gamma > 1$; si le momentum avait précédemment été fixé à zéro, alors on lui redonne sa valeur originale ;
3. Si l'erreur quadratique totale augmente par moins de β, alors la mise à jour des poids est acceptée et le taux d'apprentissage reste inchangé ; Si le momentum avait précédemment été fixé à zéro, alors on lui redonne sa valeur originale ;

Cette approche suppose que l'apprentissage fonctionne par groupage, c'est-à-dire que les mises à jour des poids sont accumulées sur l'ensemble des associations de la base d'apprentissage et appliquées une fois à la fin de chaque période (section 5.4.5). Dans certains cas cela peut accélérer grandement la convergence. Dans d'autres cas, cette approche peut aussi nuire à la convergence. Il faut comprendre que ce genre de technique ajoute des paramètres[10] (β, ρ et γ) qu'il faut fixer a priori. Pour un problème donné, certaines combinaisons de paramètres peuvent être bénéfiques et d'autres non. Parfois, l'emploi d'une telle méthode peut même entraîner une divergence rapide là où la rétropropagation des erreurs avec momentum produisait une convergence lente.

5.4.8 Autres considérations pratiques

Nous énumérons ci-dessous d'autres considérations pratiques pour l'entraînement du PMC. Selon les circonstances, celles-ci peuvent aussi avoir un effet appréciable sur la performance de l'algorithme de rétropropagation des erreurs.

[10] L'approche décrite ici, proposée par Vogl et al. en 1988, est relativement simple parmi l'ensemble des variantes qui ont été explorées dans la littérature pour faire varier dynamiquement le taux d'apprentissage.

1. Lorsqu'on utilise une couche de sortie non-linéaire, c'est-à-dire une couche dont les neurones possèdent des fonctions d'activation non linéaires telles que la sigmoïde ou la tangente hyperbolique (voir section 2.3), il importe de ne pas chercher à saturer les neurones en fixant des sorties désirées qui tendent vers l'assymptote de la fonction. Dans le cas de la sigmoïde, par exemple, au lieu de fixer des sorties désirées à 0 ou à 1, on peut les fixer à 0.05 et 0.95. Ainsi, la rétropropagation des erreurs ne cherchera pas à entraîner les poids dans une direction qui pourrait rendre le neurone incapable de s'adapter.
2. Les sensibilités des neurones sur les dernières couches ont tendance à être plus grandes que sur les premières couches ; le taux d'apprentissage sur ces dernières devrait donc être plus grand que sur ces premières si l'on veut que les différentes couches apprennent approximativement au même rythme.
3. À chaque période d'entraînement, il importe de permuter l'ordre de présentation des stimuli pour réduire la probabilité qu'une séquence de données pathologique nous garde prisonnier d'un piètre minimum local. En effet, la performance de la méthode de la descente du gradient peut dépendre grandement de cet ordre de présentation qui engendre des trajectoires différentes dans l'espace des paramètres, et des trajectoires différentes peuvent nous amener à des minimums locaux différents. Même s'il existe des séquences pathologiques, le fait de permuter les données à chaque période nous garantit que l'on ne tombera pas systématiquement sur les mêmes.
4. Dans le contexte d'un problème de classification à n classes, on associe généralement un neurone de sortie distinct à chacune d'elles ($S^M = n$). Ainsi, on interprétera chaque neurone sur la couche de sortie comme indiquant si oui ou non le stimulus d'entrée appartient à la classe correspondante. On construira les vecteurs $\mathbf{d}$ de sorties désirées avec deux valeurs possibles pour chaque composante : une valeur pour le «oui» et une valeur pour le «non». Si l'on choisit la fonction d'activation logistique, on pourra coder le «oui» avec une valeur proche de 1 et le «non» avec une valeur proche de 0 (voir item 1 ci-dessus). En mode de reconnaissance, on pourra classer un stimulus inconnu dans la catégorie associée au neurone ayant produit la sortie maximale.
5. Dans le contexte d'un problème d'approximation de fonction, on choisit généralement des neurones linéaires pour la couche de sortie. Autrement, cela force le réseau à apprendre l'inverse de la fonction d'activation utilisée, en plus de la fonction que l'on veut vraiment qu'il apprenne.
6. Effectuer l'apprentissage d'un réseau quelconque revient à estimer les bonnes valeurs pour chacun de ses poids. Pour pouvoir estimer les paramètres d'un système quelconque possédant un certain nombre de degrés de liberté (paramètres indépendants), il est nécessaire de posséder au moins un nombre équivalent de données. Mais dans la pratique il en faut bien plus ! Une règle heuristique nous indique que pour pouvoir espérer estimer correctement les n poids d'un réseau de neurones, $10n$ données d'entraînement sont requises.
7. La performance d'un réseau lorsqu'évaluée avec ses données d'entraînement est presque toujours sur-estimée. Il faut bien comprendre que le réseau ne comporte aucune intelligence réelle. Il ne fait qu'apprendre les associations qu'on lui fournit. À la limite, il peut les apprendre «par c ?ur». Nous avons discuté à la section 5.4.3 du phénomène de «sur-apprentissage». Nous avons vu qu'une procédure de validation-croisée peut augmenter la

capacité de généralisation d'un réseau. Si l'on veut évaluer correctement la performance d'un réseau, il faut le faire avec des données qui n'ont aucunement servi au processus d'apprentissage, ni pour la rétropropagation des erreurs, ni pour la validation croisée. En pratique, ceci implique de diviser les données d'entraînement en trois sous-ensembles distincts : les données d'entraînement, de validation et de test. La proportion relative de ces ensembles peut évidemment varier selon l'application, mais une bonne proportion se situe aux alentours de 50-20-30%, respectivement.

5.5 Méthode de Newton

La méthode de Newton est une autre procédure d'optimisation, parfois plus efficace que la descente du gradient. Alors que cette dernière est basée sur une approximation par la série de Taylor de premier ordre (qui n'utilise que la dérivé première), la méthode de Newton est basée sur la série de Taylor de deuxième ordre, où l'on tient compte non seulement de la dérivé première, mais aussi de la dérivé seconde :

$$F(\mathbf{x}') = F(\mathbf{x} + \Delta\mathbf{x}) \approx F(\mathbf{x}) + \nabla F(\mathbf{x})^T \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T \nabla^2 F(\mathbf{x}) \Delta\mathbf{x} = \hat{F}(\mathbf{x}), \tag{5.75}$$

où $\mathbf{x}'$ est un point dans le voisinage de $\mathbf{x}$, $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$, $\nabla F(\mathbf{x})$ est le vecteur gradient de $F(\mathbf{x})$ et $\nabla^2 F(\mathbf{x})$ est la matrice hessienne de $F(\mathbf{x})$. L'idée consiste à rechercher un plateau dans l'expression quadratique de $\hat{F}(\mathbf{x})$. En dérivant l'expression de $\hat{F}(\mathbf{x})$ par rapport à $\Delta\mathbf{x}$ et en fixant le tout à zéro, on obtient :

$$\nabla F(\mathbf{x}) + \nabla^2 F(\mathbf{x}) \Delta\mathbf{x} = 0. \tag{5.76}$$

et :

$$\Delta\mathbf{x} = -\left(\nabla^2 F(\mathbf{x})\right)^{-1} \nabla F(\mathbf{x}) \tag{5.77}$$

La méthode de Newton propose donc d'explorer l'espace des paramètres libres dans la direction $\Delta\mathbf{x}$ qui pointe vers un plateau de $\hat{F}(\mathbf{x})$, c'est-à-dire soit un maximum, soit un minimum.

Cette méthode converge en une seule étape lorsque F est une fonction quadratique, puisqu'on approxime alors une fonction quadratique par une autre fonction quadratique équivalente. Lorsque F n'est pas quadratique, cette méthode converge généralement très rapidement à condition d'avoir un bon point de départ ($\mathbf{x}$ n'est pas trop loin de l'optimum), car la plupart des fonctions analytiques s'approximent bien avec une fonction quadratique à l'intérieur d'un petit voisinage autour d'un optimum. Cependant, même dans ce cas, on ne saura pas si cet optimum est minimum ou maximum. On ne saura pas non plus si l'optimum est local ou global. De plus, lorsqu'on s'éloigne de ce voisinage, la méthode de Newton peut donner des résultats imprévisibles. Finalement, cette méthode comporte aussi le gros désavantage de nécessiter le calcul et le stockage de la matrice hessienne, d'une part, et de son inverse, d'autre part. Dans le cas d'un perceptron multicouche comportant plusieurs centaines ou même plusieurs milliers de poids, cela s'avère totalement impossible en pratique[11].

[11] Il faut se rappeler que si le vecteur gradient croît linéairement avec le nombre de paramètres libres de la fonction, la taille (en nombre d'éléments) de la matrice hessienne, elle, croît avec le carré de ce nombre. De plus, le meilleur algorithme connu pour l'inversion d'une matrice $n \times n$ possède une complexité $O(n^{2.376})$. Quant à la méthode classique d'élimination de Gauss, elle requiert un temps $O(n^3)$.

Nous avons présenté dans cette section la méthode de Newton non pas pour l'utiliser avec le perceptron multicouche, mais bien pour mettre en relief les mérites de la rétropropagation, et aussi pour faire le tour des différentes alternatives disponibles. Ceci nous amène à discuter d'un compromis où il ne sera pas nécessaire de calculer ni d'inverser la matrice hessienne.

5.6 Méthode du gradient conjugué

La méthode du gradient possède la plupart des avantages de la méthode de Newton mais sans l'inconvénient d'avoir à calculer et à inverser la matrice hessienne. Elle est basée sur le concept des vecteurs conjugués, d'une part, ainsi que sur la recherche d'un minimum le long d'une droite, d'autre part.

Les vecteurs d'un ensemble $\{\mathbf{p}_k\}$ sont mutuellement conjugués par rapport à une matrice $\mathbf{A}$ positive définie (dont les valeurs propres sont toutes strictement positives) si, et seulement si,

$$\mathbf{p}_k^T \mathbf{A} \mathbf{p}_j = 0, \quad k \neq j. \tag{5.78}$$

Comme pour les vecteurs orthogonaux, il existe une infinité d'ensembles de vecteurs conjugués qui couvrent un espace vectoriel de dimension m. Un de ceux-là est formé des vecteurs propres de $\mathbf{A}$, $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m\}$, associés aux valeurs propres $\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$. Pour le montrer, il suffit de remplacer les $\mathbf{p}_k$ par des $\mathbf{z}_k$ dans l'équation précédente :

$$\mathbf{z}_k^T \mathbf{A} \mathbf{z}_j = \mathbf{z}_k^T \lambda_j \mathbf{z}_j = \lambda_j \mathbf{z}_k^T \mathbf{z}_j = 0, \quad k \neq j, \tag{5.79}$$

où la dernière égalité découle du fait que les vecteurs propres d'une matrice positive définie sont toujours orthogonaux. Par conséquent, les vecteurs propres d'une telle matrice sont à la fois orthogonaux et conjugués. Cependant, cette observation ne nous aide pas beaucoup si la matrice $\mathbf{A}$ en question correspond à la matrice hessienne $\nabla^2 F$ que l'on veut éviter de calculer dans la méthode de Newton, notre objectif étant de trouver un algorithme efficace utilisant la dérivée seconde mais sans être obligé de la calculer explicitement.

Pour une fonction F quadratique possédant m paramètres libres, on peut montrer qu'il est toujours possible d'atteindre son minimum en effectuant tout au plus m recherches linéaires le long de droites orientées dans des directions conjuguées $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$. La question qui demeure est comment construire ces directions conjuguées sans faire référence à la matrice hessienne de F ? L'expression générale d'une fonction quadratique est donnée par :

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c \tag{5.80}$$

où le gradient ∇F est donné par :

$$\nabla F(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{d} \tag{5.81}$$

et la matrice hessienne par $\nabla^2 F(\mathbf{x}) = \mathbf{A}$. En posant $\mathbf{g}_t \equiv \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_t}$ et en combinant ces équations, on peut trouver le changement de gradient $\Delta \mathbf{g}_t$ à l'itération t :

$$\Delta \mathbf{g}_t = \mathbf{g}_{t+1} - \mathbf{g}_t = (\mathbf{A} \mathbf{x}_{t+1} + \mathbf{d}) - (\mathbf{A} \mathbf{x}_t + \mathbf{d}) = \mathbf{A} \Delta \mathbf{x}_t \tag{5.82}$$

où la variation des paramètres libres $\Delta \mathbf{x}_t$ au temps t est donnée par :

$$\Delta \mathbf{x}_t = \mathbf{x}_{t+1} - \mathbf{x}_t = \alpha_t \mathbf{p}_t \tag{5.83}$$

avec un α_t choisi de manière à minimiser $F(\mathbf{x}_t)$ dans la direction de $\mathbf{p}_t$.

On peut maintenant réécrire la condition des vecteurs conjugués de l'équation 5.78 de la manière suivante :

$$\alpha_t \mathbf{p}_t^T \mathbf{A} \mathbf{p}_j = \Delta \mathbf{x}_t^T \mathbf{A} \mathbf{p}_j = \Delta \mathbf{g}_t^T \mathbf{p}_j = 0, \quad t \neq j. \tag{5.84}$$

On remarque immédiatement qu'en considérant le changement de gradient à chaque itération t de l'algorithme, on peut faire disparaître la matrice hessienne de l'équation qui définit la condition des vecteurs conjugués. La direction de recherche $\mathbf{p}_j$ sera alors conjuguée à condition d'être orthogonale à la variation du gradient !

À chaque itération t de l'algorithme des gradients conjugués, il s'agit donc de construire une direction de recherche $\mathbf{p}_t$ qui est orthogonale à $\{\Delta \mathbf{g}_0, \Delta \mathbf{g}_1, \ldots, \Delta \mathbf{g}_{t-1}\}$ en utilisant une procédure semblable à la méthode de Gram-Schmidt (section 3.1.5, page 21), qui peut se simplifier à l'expression suivante :

$$\mathbf{p}_t = -\mathbf{g}_t + \beta_t \mathbf{p}_{t-1} \tag{5.85}$$

où les scalaires β_t peuvent se calculer de trois manières équivalentes :

$$\beta_t = \frac{\Delta \mathbf{g}_{t-1}^T \mathbf{g}_t}{\Delta \mathbf{g}_{t-1}^T \mathbf{p}_{t-1}}, \quad \beta_t = \frac{\mathbf{g}_t^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}}, \quad \beta_t = \frac{\Delta \mathbf{g}_{t-1}^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}} \tag{5.86}$$

5.6.1 Algorithme du gradient conjugué

Pour entraîner un réseau perceptron multicouche avec la méthode du gradient conjugué, il importe tout d'abord de procéder par groupage. En effet, puisqu'on cherche à exploiter l'information contenue dans la variation du gradient (une forme de dérivée seconde), il est primordial de calculer celle-ci à partir de gradients estimés avec un maximum de précision. Sinon, on se retrouverait à surtout exploiter du bruit qui entraînerait à coup sûr la divergence de l'algorithme. Ensuite, il s'agit de remarquer que l'indice de performance d'un perceptron multicouche n'est pas une fonction quadratique, de sorte qu'on ne doit pas s'attendre à converger en m itérations comme ce serait le cas pour une fonction quadratique. Ceci implique qu'il faudra ré-initialiser la méthode à toutes les m itérations, à chaque fois que nous passerons à travers m directions[12] conjuguées successives. Pour ce faire, nous pourrons simplement employer la méthode de descente du gradient. Voici donc les principales étapes de l'algorithme :

1. $t = 0$;
2. Choisir la prochaine direction conjuguée de recherche :

$$\mathbf{p}_t = \begin{cases} -\mathbf{g}_t + \beta_t \mathbf{p}_{t-1} & \text{si } t \bmod m \neq 0 \\ -\mathbf{g}_t & \text{si } t \bmod m = 0 \end{cases} \tag{5.87}$$

[12]Il y a aura autant de directions conjuguées qu'il y a de poids dans le réseau.

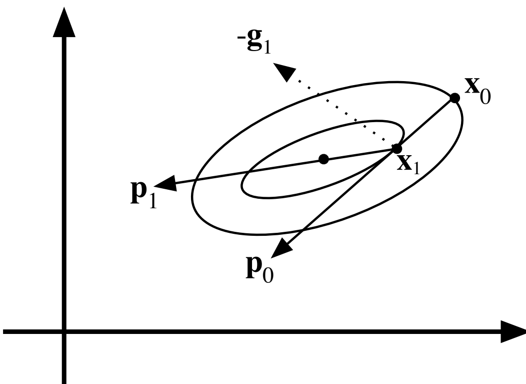


FIG. 5.16 – *Illustration de la méthode du gradient conjugué.*

avec $\mathbf{g}_t \equiv \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_t}$ et β_t est calculé comme à l'équation 5.86 :

$$\beta_t = \frac{\Delta \mathbf{g}_{t-1}^T \mathbf{g}_t}{\Delta \mathbf{g}_{t-1}^T \mathbf{p}_{t-1}}, \quad \beta_t = \frac{\mathbf{g}_t^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}}, \quad \beta_t = \frac{\Delta \mathbf{g}_{t-1}^T \mathbf{g}_t}{\mathbf{g}_{t-1}^T \mathbf{g}_{t-1}} \tag{5.88}$$

3. Faire un pas comme à l'équation 5.83, en choisissant α_t de manière à minimiser la fonction F dans la direction de recherche $\mathbf{p}_t$:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \alpha_t \mathbf{p}_t; \tag{5.89}$$

4. $t = t + 1$;
5. Si le critère d'arrêt n'est pas atteint, alors recommencer à l'étape 2.

Cet algorithme est illustré à la figure 5.16 dans le cas d'une fonction F quadratique à deux variables. Dans ce cas, on converge en deux itérations. La direction initiale $\mathbf{p}_0$ est choisie dans le sens inverse du gradient. En faisant une recherche du minimum dans cette direction, on obtient un point $\mathbf{x}_1$ sur une autre courbe de niveau. Au lieu de recommencer dans la direction inverse du gradient, perpendiculaire à la courbe de niveau, on choisit plutôt une direction conjuguée $\mathbf{p}_1$ qui pointe alors dans la direction du minimum global, puisque F est quadratique et ne possède que deux paramètres libres.

5.6.2 Recherche du minimum le long d'une droite

Pour compléter la description de la méthode du gradient conjugué, il ne nous reste plus qu'à résoudre le problème de la recherche du minimum d'une fonction le long d'une droite. Soit $F(\mathbf{x}_0)$ la valeur initiale de la fonction à minimiser et $\mathbf{p}$ la direction dans laquelle on veut faire cette minimisation. Alors, on commence par calculer :

$$F_i = F(\mathbf{x}_0 + 2^i \varepsilon \mathbf{p}), \quad i = 0, 1, \ldots, I \tag{5.90}$$

jusqu'à ce qu'on trouve un $F_I > F_{I-1}$, avec ε représentant une distance élémentaire pour notre recherche d'un intervalle initial contenant le minimum. Ce dernier doit donc se trouver quelque

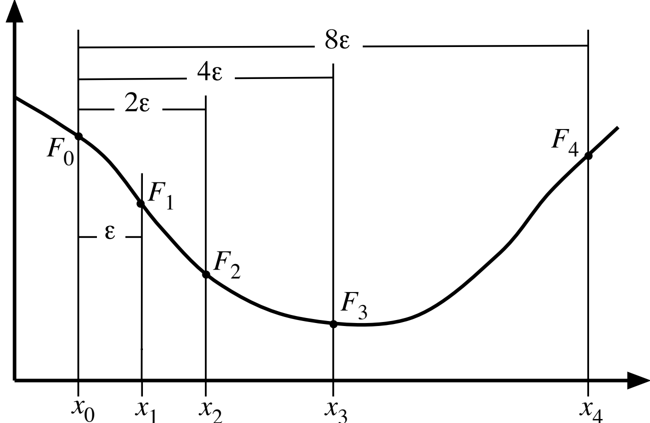


FIG. 5.17 – *Étape de localisation d'un intervalle initial de recherche.*

part entre F_{I-2} et F_I et l'intervalle de recherche est maintenant réduit à $\mathbf{a}_1 = \mathbf{x}_0 + 2^{I-2}\varepsilon\mathbf{p}$ et $\mathbf{b}_1 = \mathbf{x}_0 + 2^I\varepsilon\mathbf{p}$ (voir figure 5.17). Pour réduire davantage cet intervalle, on peut appliquer l'algorithme suivant appelé «Golden Section search» :

1. $\alpha = 0.618$;
2. Calculer :

$$\begin{aligned} \mathbf{c}_1 &= \mathbf{a}_1 + (1-\alpha)(\mathbf{b}_1 - \mathbf{a}_1) \\ \mathbf{d}_1 &= \mathbf{b}_1 - (1-\alpha)(\mathbf{b}_1 - \mathbf{a}_1) \\ F_c &= F(\mathbf{c}_1) \\ F_d &= F(\mathbf{d}_1) \end{aligned}$$

3. $k = 1$;
4. Si $F_c < F_d$, alors calculer (voir figure 5.18a) :

$$\begin{aligned} \mathbf{a}_{k+1} &= \mathbf{a}_k \\ \mathbf{b}_{k+1} &= \mathbf{d}_k \\ \mathbf{c}_{k+1} &= \mathbf{a}_{k+1} + (1-\alpha)(\mathbf{b}_{k+1} - \mathbf{a}_{k+1}) \\ \mathbf{d}_{k+1} &= \mathbf{c}_k \\ F_d &= F_c \\ F_c &= F(\mathbf{c}_{k+1}) \end{aligned}$$

   Autrement calculer (voir figure 5.18b) :

$$\begin{aligned} \mathbf{a}_{k+1} &= \mathbf{c}_k \\ \mathbf{b}_{k+1} &= \mathbf{b}_k \\ \mathbf{c}_{k+1} &= \mathbf{d}_k \\ \mathbf{d}_{k+1} &= \mathbf{b}_{k+1} - (1-\alpha)(\mathbf{b}_{k+1} - \mathbf{a}_{k+1}) \\ F_c &= F_d \\ F_d &= F(\mathbf{d}_{k+1}) \end{aligned}$$

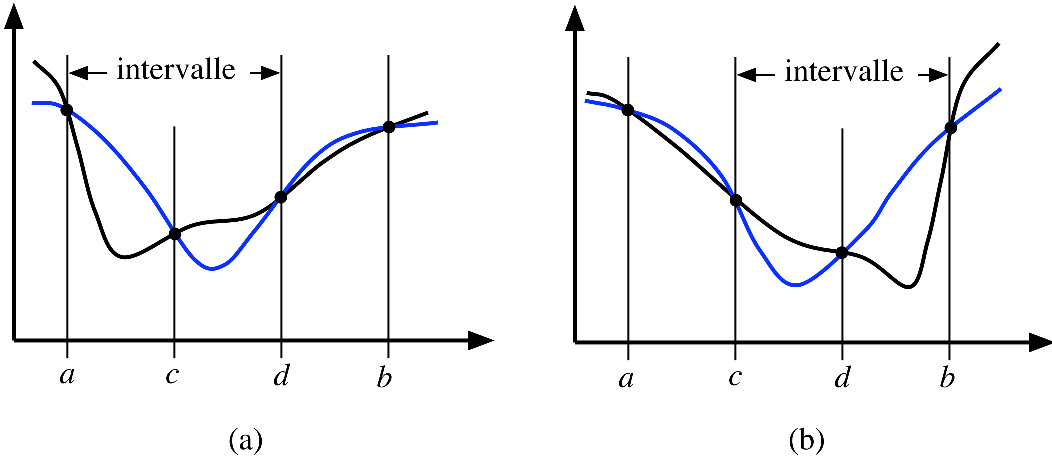


FIG. 5.18 – *Étape de réduction de l'intervalle de recherche : (a) cas où $F_c < F_d$; (b) cas où $F_d < F_c$.*

5. $k = k + 1$;
6. Si $||\mathbf{b}_k - \mathbf{a}_k|| > \tau$, alors recommencer à l'étape 4.

où τ est un paramètre de tolérance spécifiant la précision désirée pour la recherche du minimum. Notez bien que cet algorithme suppose qu'il n'existe qu'un seul minimum dans l'intervalle initial de recherche.

Chapitre 6

Nuées dynamiques

Dans ce chapitre, nous allons étudier trois variantes d'un algorithme nommé «nuées dynamiques» et permettant d'effectuer une classification non-supervisée d'un ensemble de Q stimuli $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}$. L'objectif est double : produire une partition en K classes de cet ensemble, d'une part, et trouver K prototypes $\mathbf{W} = \{{}_1\mathbf{w}, {}_2\mathbf{w}, \ldots, {}_K\mathbf{w}\}^T$ permettant de représenter au mieux les «centres» de ces classes. Bien qu'historiquement cet algorithme n'appartienne pas au domaine des réseaux de neurones, plusieurs architectures neuronales, dont celles décrites dans les deux chapitres suivants (Kohonen et GNG), s'en inspirent en effectuant des traitements semblables. Nous abordons donc cet algorithme en guise d'introduction aux réseaux non supervisés, basés sur l'apprentissage compétitif (voir chapitre 4).

On peut visualiser les prototypes ${}_i\mathbf{w}, i = 1, \ldots, K$, comme les poids de K neurones compétitifs alignés sur une seule couche, tel qu'illustré à la figure 6.1. Le niveau d'activation d'un neurone «compétitif» est déterminé par la distance entre son vecteur de poids et le stimulus d'entrée, contrairement au neurone de type «perceptron» où l'on mesurait plutôt une corrélation entre ces deux vecteurs. Ensuite, la fonction d'activation compétitive (`compet`) retourne un 1 pour le neurone ayant la plus grande sortie (le gagnant), et un 0 pour tous les autres :

$$a_i = \begin{cases} 1 & \text{si } n_i = \max_k(n_k), \quad k = 1, \ldots, K \\ 0 & \text{autrement} \end{cases} \tag{6.1}$$

En cas d'égalité pour la première place, on fait gagner arbitrairement le neurone dont l'indice est le plus petit.

C'est la norme $||\mathbf{x}-\mathbf{y}||$ qui définit la distance entre deux vecteurs $\mathbf{x}$ et $\mathbf{y}$ et donc leur manque de ressemblance. En calculant la négation de cette norme, on obtiendra une mesure de similarité qui nous permettra de regrouper les stimuli d'apprentissage en catégories (classes). Habituellement, on utilisera une norme basée sur le produit scalaire classique mais pouvant incorporer une matrice positive définie $\mathbf{A}$ telle que :

$$||\mathbf{x} - \mathbf{y}||_{\mathbf{A}} = \sqrt{(\mathbf{x} - \mathbf{y})^T \mathbf{A} (\mathbf{x} - \mathbf{y})}, \tag{6.2}$$

Lorsque $\mathbf{A}$ est la matrice identité, on parle alors de distance euclidienne entre $\mathbf{x}$ et $\mathbf{y}$. Dans le cas

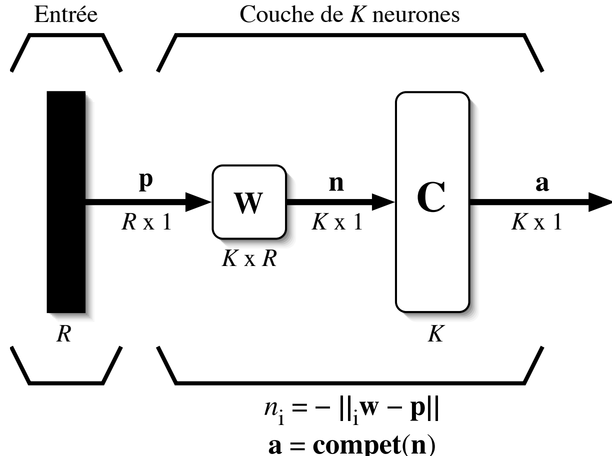


FIG. 6.1 – *Couche compétitive de $S = K$ neurones.*

où $\mathbf{A}$ correspond à l'inverse de la matrice de covariance des stimuli d'entraînement, on parle alors de la distance de Mahalanobis.

6.1 K-means

L'algorithme dit du «k-means» permet de partitionner l'ensemble des stimuli en K classes $\{C_1, C_2, \ldots, C_K\}$. Il s'agit ici d'une partition rigide, c'est-à-dire d'une collection de K sous-ensembles où chaque stimulus d'entrée appartient à une et une seule classe de la partition $\mathbf{U}$:

$$\mathbf{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,Q} \\ u_{2,1} & u_{2,2} & \ldots & u_{2,Q} \\ \vdots & \vdots & \ddots & \vdots \\ u_{K,1} & u_{K,2} & \cdots & u_{K,Q} \end{bmatrix} \tag{6.3}$$

avec $u_{i,j} \in \{0, 1\}$ désignant l'appartenance du stimulus $\mathbf{p}_j$ à la classe C_i :

$$u_{i,j} = \begin{cases} 1 & \text{si } \mathbf{p}_j \in C_i \\ 0 & \text{autrement} \end{cases} . \tag{6.4}$$

De plus, on impose les deux contraintes suivantes sur cette partition :

$$\sum_{i=1}^{K} u_{i,j} = 1, \quad j = 1, \ldots, Q, \tag{6.5}$$

$$\sum_{j=1}^{Q} u_{i,j} > 0, \quad i = 1, \ldots, K. \tag{6.6}$$

La première spécifie que tout stimulus doit appartenir à une et une seule classe de la partition, alors que la deuxième précise qu'une classe doit posséder au moins un stimulus.

1. Initialiser les centres $\mathbf{W}(0)$ en choisissant aléatoirement K stimuli parmi les Q données d'apprentissage ;
2. Calculer la partition initiale $\mathbf{U}(0)$ à l'aide de l'équation :

$$u_{i,j} = \begin{cases} 1 & \text{si } ||\mathbf{p}_j - {}_i\mathbf{w}|| = \min_k ||\mathbf{p}_j - {}_k\mathbf{w}|| \\ 0 & \text{autrement} \end{cases}, \quad i = 1, \ldots, K, \; j = 1, \ldots, Q, \quad (6.8)$$

   en brisant (arbitrairement) les égalités, s'il y a lieu, en choisissant la classe dont l'indice est minimum ;
3. $t = 1$;
4. **Répéter** :
   (a) Calculer les nouveaux centres $\mathbf{W}(t)$ en calculant les centroïdes des classes :

$$_i\mathbf{w} = \frac{\sum_{j=1}^{Q} u_{i,j}\, \mathbf{p}_j}{\sum_{j=1}^{Q} u_{i,j}}, \quad i = 1, \ldots, K. \quad (6.9)$$

   (b) Calculer la nouvelle partition $\mathbf{U}(t)$ à l'aide de l'équation 6.8 ;
   (c) $t = t + 1$;
5. **Tant que** $\mathbf{U}(t) \neq \mathbf{U}(t-1)$ et $t \leq t_{\max}$.

FIG. 6.2 – *Algorithme du «k-means».*

Connaissant les centres $\mathbf{W} = [{}_1\mathbf{w}\; {}_2\mathbf{w} \cdots {}_K\mathbf{w}]^T$ des classes, l'indice de performance $F(\mathbf{U}, \mathbf{W})$ que l'on désire minimiser peut s'exprimer de la façon suivante :

$$F(\mathbf{U}, \mathbf{W}) = \sum_{j=1}^{Q} \sum_{i=1}^{K} (u_{i,j})\, ||\mathbf{p}_j - {}_i\mathbf{w}||^2 \quad (6.7)$$

où l'on cherche à trouver la partition qui minimise la distance entre les centres des classes et les stimuli. L'algorithme itératif permettant d'optimiser cette fonction objectif est résumé à la figure 6.2. Il se poursuit tant que la mise à jour des centres engendre une modification de la partition, ou jusqu'à l'atteinte d'un nombre maximum d'itérations $t_{\max}$.

Un problème avec cet algorithme est que l'on force une partition rigide des stimuli d'entraînement ce qui, en présence de bruit, peut provoquer une certaine instabilité. Considérons par exemple les stimuli d'entraînement représentés à la figure 6.3a. Dans ce cas, nous avons deux classes bien séparées pour lesquelles l'algorithme du k-means, avec $K = 2$, convergera assurément vers la partition indiquée par les points noirs et les points blancs. Mais si l'on ajoute un stimulus éloigné des autres, par exemple une erreur de mesure, alors la partition engendrée par les k-means peut devenir instable comme à la figure 6.3b, étant donné que tous les stimuli ont la même importance dans la partition rigide. Pour limiter ce phénomène, une solution consiste à faire appel aux

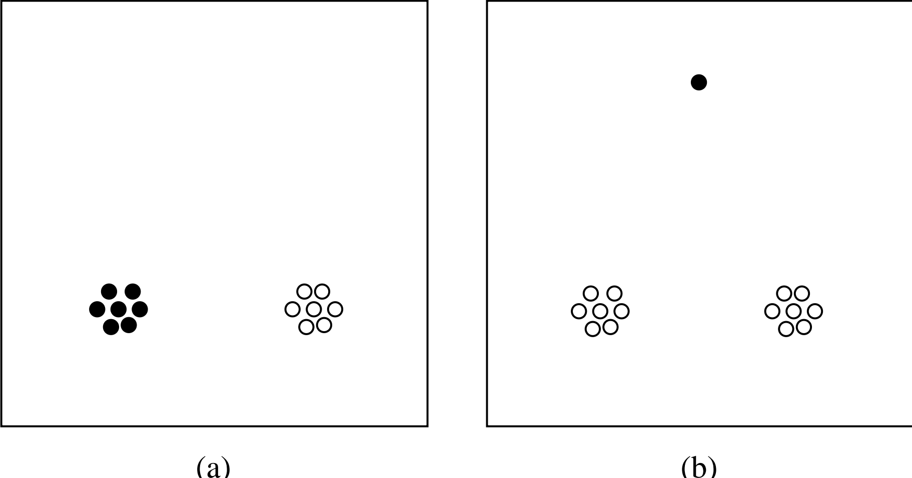


FIG. 6.3 – *Example d'une partition rigide en deux classes : (a) cas sans bruit ; (b) cas avec bruit.*

notions d'ensemble flou et de partition floue.

6.2 Fuzzy K-means

L'algorithme dit du «fuzzy k-means»[1] (le k-means flou) est semblable au k-means, sauf que la partition engendrée par les centres est floue, c'est-à-dire que le degré d'appartenance $u_{i,j}$ du stimulus $\mathbf{p}_j$ à la classe C_i varie dans l'intervalle $[0, 1]$ au lieu d'être élément de $\{0, 1\}$, comme précédemment. L'indice de performance que l'on désire minimiser s'exprime maintenant par l'expression :

$$F_m(\mathbf{U}, \mathbf{W}) = \sum_{j=1}^{Q} \sum_{i=1}^{K} (u_{i,j})^m \, ||\mathbf{p}_j - {}_i\mathbf{w}||^2, \tag{6.10}$$

où $m > 1$ est un exposant qui fixe le niveau de flou de l'algorithme dont les étapes sont résumées à la figure 6.4. L'algorithme se poursuit ainsi tant que la mise à jour des centres engendre une modification non négligeable de la partition floue, ou jusqu'à un nombre maximum d'itérations $t_{\max}$. Généralement, on juge de la convergence en fixant un seuil ε sur l'élément de la matrice $\mathbf{U}$ qui a changé le plus. Si ce changement est inférieur au seuil fixé, on considère alors que l'algorithme a convergé.

Le paramètre m de l'algorithme détermine le niveau de partage des degrés d'appartenance de la matrice $\mathbf{U}$. On fixe généralement $m = 2$. Plus m augmente, plus on se trouve à partager les degrés d'appartenance entre les différentes classes. Il importe de se rappeler que l'algorithme du fuzzy k-means, tout comme celui du k-means, impose la contrainte de l'équation 6.5, à savoir que l'appartenance global d'un stimulus à l'ensemble des classes est toujours égal à 1. Lorsque m s'approche de 1, on tend alors vers le k-means puisque la partition floue devient de plus en plus

[1]J.C. Dunn, «A Fuzzy Relative of the ISODATA Process and its Use in Detecting Compact Well-Separated Clusters», J. Cybernetics, vol. 3, no. 3, p. 32-57, 1973.

1. Initialiser les centres $\mathbf{W}(0)$ en choisissant aléatoirement K stimuli parmi les Q données d'apprentissage ;
2. Fixer les paramètres m et ε ;
3. Calculer la partition initiale $\mathbf{U}(0)$ à l'aide de l'équation :

$$u_{i,j} = \frac{1}{\displaystyle\sum_{k=1}^{K} \left(\frac{||\mathbf{p}_j - {}_i\mathbf{w}||}{||\mathbf{p}_j - {}_k\mathbf{w}||} \right)^{\frac{2}{m-1}}}, \quad i = 1, \ldots, K, \ j = 1, \ldots, Q; \tag{6.11}$$

4. $t = 1$;
5. **Répéter** :
   (a) Calculer les nouveaux centres $\mathbf{W}(t)$ à l'aide de l'équation :

$$_i\mathbf{w} = \frac{\displaystyle\sum_{j=1}^{Q} (u_{i,j})^m \mathbf{p}_j}{\displaystyle\sum_{j=1}^{Q} (u_{i,j})^m}, \quad i = 1, \ldots, K. \tag{6.12}$$

   (b) Calculer la nouvelle partition floue $\mathbf{U}(t)$ en utilisant l'équation 6.11 ;
   (c) $t = t + 1$;
6. **Tant que** $\max\limits_{i,j} |u_{i,j}(t) - u_{i,j}(t-1)| > \varepsilon$ et $t \leq t_{\max}$;

FIG. 6.4 – *Algorithme du «fuzzy k-means».*

rigide. Ceci devient un peu plus explicite en réécrivant l'équation 6.11 de la manière suivante :

$$u_{i,j}(t) = \frac{\left(\frac{1}{||\mathbf{p}_j - {}_i\mathbf{w}||} \right)^{\frac{2}{m-1}}}{\displaystyle\sum_{k=1}^{K} \left(\frac{1}{||\mathbf{p}_j - {}_k\mathbf{w}||} \right)^{\frac{2}{m-1}}} \tag{6.13}$$

Lorsque $m \to 1$, l'exposant $\frac{2}{m-1}$ tend vers l'infini et, par conséquent, le terme de la somme du dénominateur qui correspond au centre le plus proche du stimulus $\mathbf{p}_j$ devient infiniment dominant, de sorte que le degré d'appartenance à la classe correspondante tendra vers 1 (et les autres vers 0).

L'algorithme du fuzzy k-means généralise donc celui du k-means en étant beaucoup moins sensible au bruit dans les stimuli grâce au partage de l'appartenance entre les différentes classes. Ceci permet dans bien des situations d'éviter de rester pris dans des minimums locaux de la fonction objectif F_m. L'algorithme est donc également beaucoup moins sensible à une mauvaise initialisation des centres. Cependant, il demeure un problème important, illustré à la figure 6.5. On voit sur cette figure deux nuages de points bien séparés ainsi que deux points A et B qui semblent

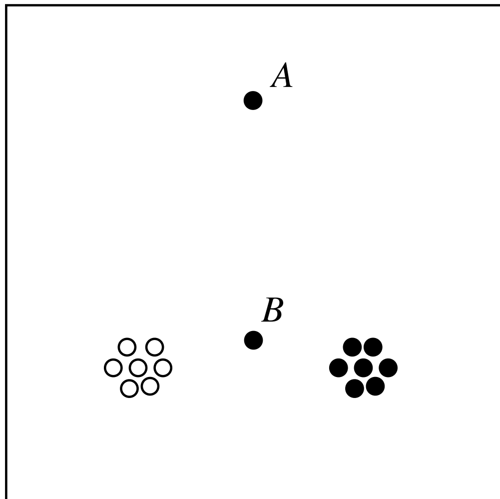


FIG. 6.5 – *Exemple d'une partition floue à deux classes.*

correspondre à du bruit. Typiquement, l'algorithme du fuzzy k-means affectera ces deux points à l'une ou l'autre des deux classes, en fonction de l'emplacement exact de ces points ainsi qu'en fonction de la position initiale des centres. Le problème est que peu importe cette affectation finale, les degrés d'appartenance de A et B aux deux classes en question seront tous les deux 0.5 puisqu'il sont approximativement à égale distance des centres. Ceci résulte de la contrainte de l'équation 6.5 qui impose un partage de l'appartenance avec une somme des degrés égale à 1. Autrement dit, un stimulus situé à égale distance de deux centres obtiendra toujours des degrés égaux d'appartenance aux classes correspondantes, peu importe la distance mesurée. Dans le cas de la figure 6.5, le point A obtient un degré d'appartenance de 0.5, tout comme le point B, même si le point B semble a priori beaucoup plus plausible que le point A, étant situé beaucoup plus près des deux centres. Ceci a comme conséquence d'accorder la même importance au point A qu'au point B. Dans le cas de données bruitées, cela peut empêcher les centres de converger vers des régions denses de l'espace d'entrée.

6.3 Possibilistic K-means

L'algorithme dit du «Possibilistic k-means»[2] cherche à pallier aux limitations du fuzzy k-means en levant la contrainte de l'équation 6.5 et en modifiant la fonction objectif de manière à ne pas converger vers une solution triviale où tous les degrés d'appartenance seraient nuls. Les éléments $u_{i,j}$ de la partition floue $\mathbf{U}$ doivent alors respecter les trois contraintes suivantes :

$$u_{i,j} \in [0,1], \quad i = 1, \ldots, K, \; j = 1, \ldots, Q, \tag{6.14}$$

$$\sum_{j=1}^{Q} u_{i,j} > 0, \quad i = 1, \ldots, K, \tag{6.15}$$

[2]R. Krishnapuram, J.M. Keller, «A Possibilistic Approach to Clustering», IEEE Transactions on Fuzzy Systems, vol. 1, no. 2, p. 98-110, mai 1993.

$$\sum_{i}^{K} u_{i,j} > 0, \quad j = 1, \ldots, Q, \tag{6.16}$$

où les deux premières contraintes spécifient respectivement que les degrés d'appartenance des stimuli aux classes sont compris entre 0 et 1, et que toute classe doit posséder au moins un stimulus avec un degré d'appartenance non nul. Ces deux contraintes sont identiques à celles de l'algorithme du fuzzy k-means. La troisième contrainte, celle de l'équation 6.16, se distingue cependant à la fois du k-means et du fuzzy k-means qui impose à chaque stimulus d'avoir une somme des degrés d'appartenance égale à 1 (équation 6.5). Le possibilistic k-means impose plutôt que chaque stimulus appartienne à au moins une classe avec un degré d'appartenance non nul. Ceci implique que la somme des degrés d'appartenance d'un stimulus à l'ensemble des classes peut maintenant être soit plus petite, soit plus grande que 1, d'une part, et qu'un stimulus puisse appartenir à une seule classe avec un degré d'appartenance inférieur à 1, d'autre part. Dans ce dernier cas, un stimulus bruité très éloigné de tous les centres pourra n'appartenir qu'à une seule classe, avec un degré d'appartenance arbitrairement petit.

Pour réaliser ces contraintes, cependant, on ne peut pas optimiser la même fonction objectif que pour le fuzzy k-means (équation 6.10), car on tendrait systématiquement vers des degrées d'appartenance arbitrairement petits. Il s'agit alors d'ajouter un deuxième terme à cette fonction objectif pour stimuler l'émergence de $u_{i,j}$ les plus grands possibles. On utilise l'expression suivante :

$$F_m(\mathbf{U}, \mathbf{W}) = \sum_{j=1}^{Q} \sum_{i=1}^{K} (u_{i,j})^m \, ||\mathbf{p}_j - {}_i\mathbf{w}||^2 + \sum_{i=1}^{K} \eta_i \sum_{j=1}^{Q} (1 - u_{i,j})^m, \tag{6.17}$$

où les η_i sont des valeurs positives adéquates représentant l'étendue du nuage associé à la classe C_i. Le premier terme de cette équation cherche à minimiser les distances entre les stimuli et les centres des classes, alors que le deuxième force les $u_{i,j}$ à être maximum en évitant donc la solution triviale. Les éléments clés dans cette fonction objectif sont les η_i qui viennent pondérer l'importance relative de ces deux critères.

Pour optimiser cette fonction objectif, il importe de remarquer que les contraintes imposées par le possibilistic k-means (équations 6.14 à 6.16) rendent les lignes et les colonnes de la matrice $\mathbf{U}$ indépendantes. En effet, les $u_{i,j}$ sont maintenant libres de changer indépendamment les uns des autres car la contrainte de l'équation 6.6 a été levée. Ceci nous permet donc de minimiser F_m par rapport à $\mathbf{U}$ en minimisant individuellement chaque $u_{i,j}$. On obtient alors les termes $F_m^{i,j}$ de F_m :

$$F_m^{i,j}(u_{i,j}, {}_i\mathbf{w}) = (u_{i,j})^m \, ||\mathbf{p}_j - {}_i\mathbf{w}||^2 + \eta_i (1 - u_{i,j})^m \, . \tag{6.18}$$

que l'on peut dériver par rapport à $u_{i,j}$ et, en affectant le résultat à 0, on obtient le résultat suivant :

$$u_{i,j} = \frac{1}{1 + \left(\frac{||\mathbf{p}_j - {}_i\mathbf{w}||^2}{\eta_i} \right)^{\frac{1}{m-1}}} \, . \tag{6.19}$$

qui nous précise la façon dont il faudra estimer la partition floue à chaque itération de l'algorithme, étant donné les stimuli d'apprentissage et les positions des centres. L'équation 6.19 spécifie que le degré d'appartenance à une classe ne dépend que de la distance entre le stimulus et le centre de la

classe. C'est exactement le résultat que nous cherchions. Il ne reste plus qu'à déterminer la façon de calculer les η_i qui viennent pondérer cette distance.

Le paramètre m dans l'équation 6.19 représente le niveau de flou de l'algorithme, comme pour le fuzzy k-means, mais son interprétation est différente. Lorsque $m \to 1$, l'exposant $\frac{1}{m-1}$ tend vers l'infini et les degrés d'appartenance deviennent binaires : 1 si la distance du stimulus au centre est inférieur à η_i, 0 autrement. Au contraire, lorsque $m \to \infty$, l'exposant $\frac{1}{m-1} \to 0$ et tous les degrés d'appartenance deviennent égaux à 0.5 peu importe la distance entre les stimuli et les centres ; on obtient alors le flou maximum puisque tous les stimuli appartiennent à toutes les classes avec 50% de possibilité. Pour le fuzzy k-means on conseille habituellement $m = 2$ qui donne des résultats satisfaisants dans la plupart des situations. Pour le possibilistic k-means, cette valeur est parfois trop élevée, on conseille plutôt $1.5 \leq m < 2$.

La valeur de η_i détermine la distance à partir de laquelle le degré d'appartenance d'un stimulus à la classe devient 50%. C'est en quelque sorte la zone d'influence de la classe C_i, à l'intérieur de laquelle la possibilité d'appartenance est supérieure à 50%. De façon générale, il importe donc que sa valeur soit corrélée avec l'étendu du nuage des stimuli associés à la classe. En pratique, la définition suivante fonctionne bien :

$$\eta_i = \frac{\sum_{j=1}^{Q} (u_{i,j})^m \, ||\mathbf{p}_j - {}_i\mathbf{w}||^2}{\sum_{j=1}^{Q} (u_{i,j})^m}. \tag{6.20}$$

Cette expression rend η_i proportionnel à la moyenne pondérée des distances intra-classe. Une autre façon de procéder est de considérer seulement dans l'équation ci-dessus les $u_{i,j} \geq \alpha$, on parle alors d'une coupe α de la partition. Dans ce cas, η_i représente la moyenne pondérée des distances intra-classe pour les «bons» stimuli, c'est-à-dire ceux dont le degré d'appartenance à la classe est supérieur à α.

Les valeurs de η_i peuvent être fixées a priori, ou encore être ajustées à chaque itération de l'algorithme. Dans ce dernier cas, cependant, il importe de prendre des précautions pour éviter les instabilités pouvant résulter de tels changements en continu. En pratique, l'algorithme du possibilistic k-means est assez robuste à de larges variations de η_i, à condition d'avoir bien initialiser les centres. Ainsi, on peut calculer les η_i à partir de la partition floue initiale, puis les ré-estimer lorsque l'algorithme a convergé une première fois. Ensuite, on peut recommencer la procédure si l'on désire obtenir une meilleure estimation de la partition floue. Cette deuxième étape converge généralement en quelques itérations seulement puisque les centres sont déjà correctement positionnés. Dans le cas de stimuli bruités, elle permet de déterminer les degrés d'appartenance aux classes avec presque la même précision que dans le cas d'un environnement non bruité. Des valeurs de α entre 0.1 et 0.4 produisent généralement des résultats satisfaisants.

L'algorithme du possibilistic k-means est résumé à la figure 6.6. Il comporte deux parties dont la deuxième est optionnelle. Dans les deux cas, nous adoptons le même critère d'arrêt que pour le fuzzy k-means. Pour le choix des centres initiaux, on ne peut pas procéder comme précédemment en choisissant aléatoirement des stimuli à l'intérieur de la base d'apprentissage car les centres de l'algorithme du possibilistic k-means ne sont pas aussi mobiles que ceux des deux algorithmes

1. Initialiser les centres $\mathbf{W}(0)$ ainsi que la partition floue $\mathbf{U}(0)$ en utilisant l'algorithme du fuzzy k-means ;
2. Estimer les η_i en utilisant l'équation 6.20 avec une coupe $\alpha = 0$;
3. Fixer les valeurs de m et ε ;
4. $t = 1$;
5. **Répéter** :
   (a) Calculer les nouveaux centres $\mathbf{W}(t)$ en utilisant l'équation 6.12 ;
   (b) Calculer la nouvelle partition floue $\mathbf{U}(t)$ en utilisant l'équation 6.19 ;
   (c) $t = t + 1$;
6. **Tant que** $\max_{i,j} |u_{i,j}(t) - u_{i,j}(t-1)| > \varepsilon$ et $t \leq t_{\max}$;
7. Ré-estimer les η_i en utilisant l'équation 6.20 avec une coupe $0.1 \leq \alpha \leq 0.4$;
8. $t = 1$;
9. **Répéter** :
   (a) Calculer les nouveaux centres $\mathbf{W}(t)$ en utilisant l'équation 6.12 ;
   (b) Calculer la nouvelle partition floue $\mathbf{U}(t)$ en utilisant l'équation 6.19 ;
   (c) $t = t + 1$;
10. **Tant que** $\max_{i,j} |u_{i,j}(t) - u_{i,j}(t-1)| > \varepsilon$ et $t \leq t_{\max}$;

FIG. 6.6 – *Algorithme du «possibilistic k-means».*

précédents. En effet, à cause du paramètre η_i qui restreint le rayon d'action des centres, un mauvais choix de leur position initiale peut limiter grandement la performance de l'algorithme, et même empêcher sa convergence. Pour cette raison, on commence habituellement avec un fuzzy k-means qui, lui, est beaucoup plus robuste à une mauvaise initialisation des centres.

Mentionnons finalement qu'en changeant la norme utilisée pour calculer la distance entre un stimulus et un centre, on peut construire différentes variantes du possibilistic k-means. Par exemple, en calculant la matrice de covariance floue des stimulus :

$$\mathbf{F}_i = \frac{\sum_{j=1}^{Q} (u_{i,j})^m (\mathbf{p}_j - {}_i\mathbf{w})(\mathbf{p}_j - {}_i\mathbf{w})^T}{\sum_{j=1}^{Q} (u_{i,j})^m}, \tag{6.21}$$

et en l'utilisant pour calculer la norme suivante :

$$||\mathbf{p}_j - {}_i\mathbf{w}||^2 = \sqrt[R]{|\mathbf{F}_i|}\,(\mathbf{p}_j - {}_i\mathbf{w})^T F_i^{-1} (\mathbf{p}_j - {}_i\mathbf{w}), \tag{6.22}$$

on obtient l'algorithme dit du «possibilistic Gustafson-Kessel».

Chapitre 7

Réseau de Kohonen

Dans ce chapitre, nous allons étudier un réseau de neurones dit réseau de Kohonen[1], ou encore carte auto-organisée de Kohonen. Il s'agit d'un réseau non supervisé avec un apprentissage compétitif où l'on apprend non seulement à modéliser l'espace des entrées avec des prototypes, comme avec les nuées dynamiques (chapitre 6), mais également à construire une carte à une ou deux dimensions permettant de structurer cet espace.

Un réseau de Kohonen est illustré à la figure 7.1. Les neurones de ce réseau correspondent aux prototypes (figure 7.1a). Ils sont constitués d'un vecteur de poids dans l'espace des entrées (d'une façon semblable aux centres des nuées dynamiques). La carte des neurones (figure 7.1b) définit quant à elle des relations de voisinage entre les neurones. Par exemple, la figure 7.2 montre la forme «carrée» de voisinage qui est la plus souvent utilisée (les neurones y sont représentés par des cercles vides). On voit sur cette figure que les neurones adjacents sont liés entre eux par des arêtes (on pourrait aussi avoir des arêtes diagonales). Ce sont les voisins immédiats spécifiés par le voisinage $\Lambda_i = k$ du neurone i, c'est-à-dire l'ensemble des neurones liés au neurone i par des chemins dans le graphe de la carte contenant au plus k arêtes.

L'algorithme d'apprentissage du réseau de Kohonen est de type compétitif (section 4.3). La mise à jour des poids $\Delta_i\mathbf{w}$ du neurone i au temps t s'exprime de la façon suivante :

$$\Delta_i\mathbf{w}(t) = \begin{cases} \eta(t)[\mathbf{p}(t) - {}_i\mathbf{w}(t)] & \text{si } i \in \Lambda_g(t) \\ 0 & \text{autrement} \end{cases} \tag{7.1}$$

où $\mathbf{p}(t)$ désigne le stimulus d'apprentissage au temps t et $\Lambda_g(t)$ représente le voisinage au temps t du neurone gagnant g. Ce dernier est déterminé simplement en retenant l'indice du neurone pour lequel la distance avec le stimulus $\mathbf{p}$ est minimum :

$$g(\mathbf{p}) = \arg\min_i ||\mathbf{p} - {}_i\mathbf{w}||, \quad i = 1, 2, \ldots, S \tag{7.2}$$

où S correspond au nombre de neurones du réseau.

[1] Teuvo Kohonen, «The Self-Organizing Map», *Proceedings of the IEEE*, p. 1464-1480, 1990 (la version originale de cet article remonte à 1982).

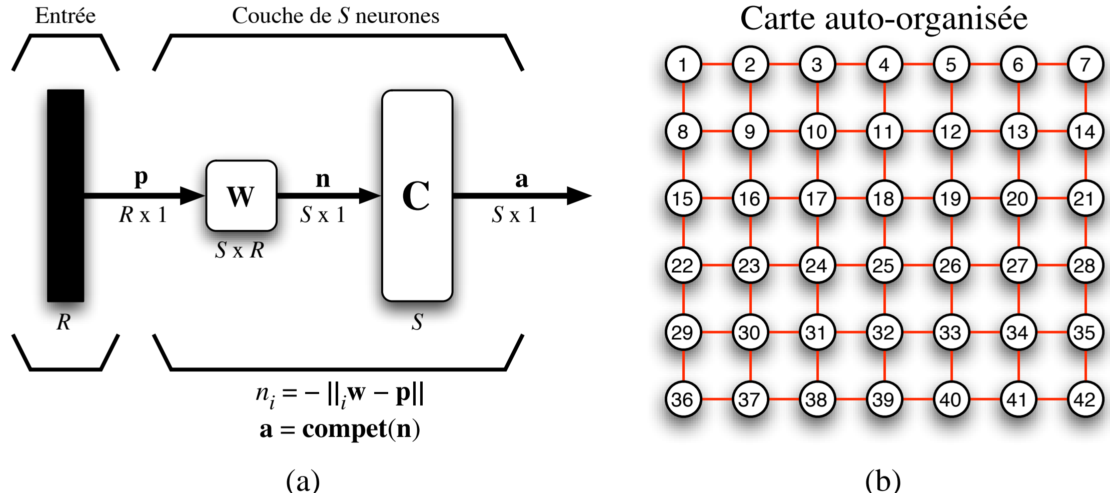


FIG. 7.1 – *Réseau de Kohonen avec carte rectangulaire de $S = 6 \times 7 = 42$ neurones.*

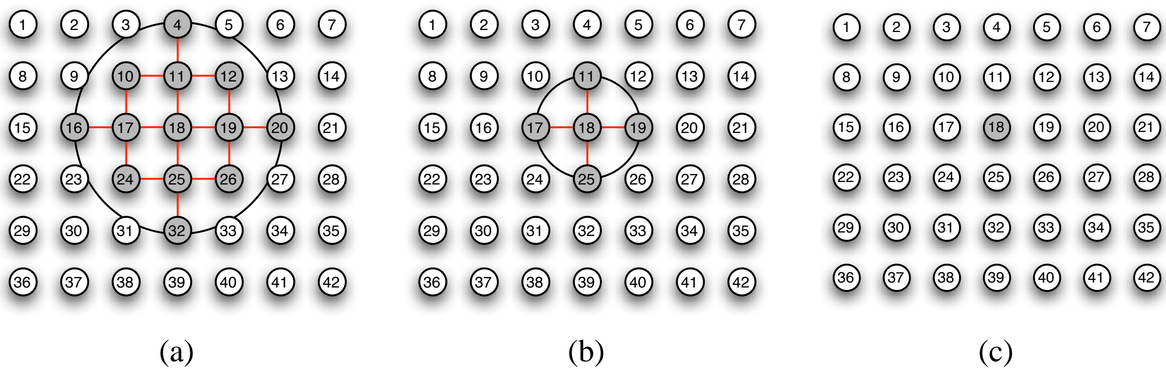


FIG. 7.2 – *Topologie de voisinage (quatre voisins) pour une carte à deux dimensions : (a) $\Lambda_{18} = 2$; (b) $\Lambda_{18} = 1$; et (c) $\Lambda_{18} = 0$.*

À l'équation 7.1, il importe de remarquer que le taux d'apprentissage η et le voisinage du neurone gagnant Λ_g dépendent tous deux du temps. L'idée étant d'employer au départ un grand taux d'apprentissage ainsi qu'un grand voisinage, et de réduire ceux-ci au fur et à mesure que le temps (donc l'apprentissage) progresse. On utilise souvent une décroissance linéaire pour le taux d'apprentissage :

$$\eta(t) = \begin{cases} \eta_0 - \left(\frac{\eta_0 - \eta_\tau}{\tau}\right) t & \text{si } t < \tau \\ \eta_\tau & \text{autrement} \end{cases} \tag{7.3}$$

où η_0 est le taux d'apprentissage initial, η_τ est le taux d'apprentissage final, et où τ délimite la frontière entre deux phases d'apprentissage. De même, pour le voisinage, on peut aussi utiliser une décroissance linéaire :

$$\Lambda(t) = \begin{cases} \Lambda_0 \left(1 - \frac{t}{\tau}\right) & \text{si } t < \tau \\ 0 & \text{autrement} \end{cases} \tag{7.4}$$

Ces fonctions linéaires décroissantes sont illustrées à la figure 7.3. Le paramètre τ marque la fin d'une période d'organisation et le début d'une période de convergence. La phase d'organisation, grâce à un taux d'apprentissage élevé ainsi qu'un voisinage étendu, permet de déployer la carte de

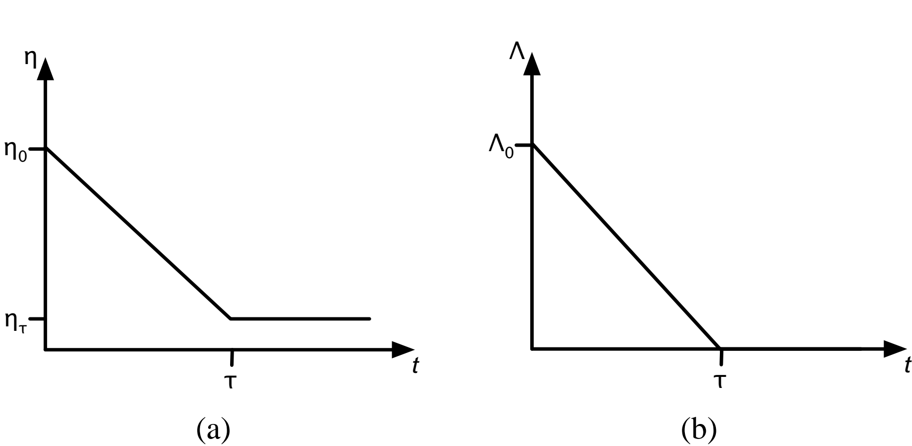


FIG. 7.3 – *Exemple de décroissance (a) du taux d'apprentissage et (b) de la fenêtre de voisinage en fonction du temps.*

neurones là où les données sont concentrées. Elle permet aussi, grâce à une décroissance progressive du taux d'apprentissage et du voisinage, de déplier la carte de manière à ce que sa topologie corresponde au mieux à la topologie des stimuli. Ainsi, les stimuli adjacents dans l'espace des entrées seront associés à des neurones adjacents dans la carte du réseau. La phase de convergence, grâce à un taux d'apprentissage faible et un voisinage nul, permet quant à elle de raffiner la position des neurones de manière à les centrer sur des nuages denses de stimuli.

7.1 Algorithme de Kohonen

L'algorithme de base de Kohonen est résumé à la figure 7.4. Il consiste à échantillonner les stimuli d'apprentissage jusqu'à l'atteinte d'un certain critère d'arrêt. Celui-ci est le plus souvent spécifié par un nombre maximum d'itération $t_{\max}$ mais peut aussi tenir compte de la stabilité des poids. Par exemple, lorsque $\max_i ||\Delta_i \mathbf{w}|| < \varepsilon$ durant plusieurs itérations, on peut décider arbitrairement de stopper l'apprentissage puisque le réseau modélise suffisamment bien l'espace des entrées. À chaque itération, on détermine le neurone gagnant (le plus proche) et on déplace celui-ci, ainsi que son voisinage actuel, dans la direction du stimulus courant en utilisant le taux d'apprentissage courant.

Une variante que l'on peut aussi rencontrer consiste à remplacer l'équation 7.1 par l'expression suivante :

$$\Delta_i w(t) = \pi_{i,g}(t)\eta(t)[\mathbf{p}(t) - {}_i\mathbf{w}(t)] \tag{7.5}$$

où :

$$\pi_{i,g}(t) = \exp\left(-\frac{d_{i,g}^2}{2\sigma^2(t)}\right) \tag{7.6}$$

vient remplacer le voisinage Λ_g par une fonction $\pi_{i,g}$ qui dépend de la distance topologique $d_{i,g}$ (en nombre d'arêtes dans la carte) entre le neurone i et le neurone gagnant g. La fonction $\pi_{i,g}$ a une forme gaussienne telle qu'illustrée à la figure 7.5. Elle se trouve à réduire le taux d'apprentissage

1. Initialiser les poids ${}_i\mathbf{w}(0)$ avec de petites valeurs aléatoires ;
2. Fixer η_0, η_τ, τ et Λ_0 ;
3. $t = 1$;
4. **Répéter tant que** $t \leq t_{\max}$:
   (a) Choisir aléatoirement un stimulus $\mathbf{p}(t)$ parmi l'ensemble d'apprentissage ;
   (b) Déterminer le neurone gagnant $g(\mathbf{p})$ à l'aide de l'équation 7.2 :
   $$g(\mathbf{p}) = \arg\min_i ||\mathbf{p}(t) - {}_i\mathbf{w}(t)||, \quad i = 1, 2, \ldots, S$$
   (c) Mettre à jour les poids à l'aide de l'équation 7.1 :
   $$\Delta_i\mathbf{w}(t) = \begin{cases} \eta(t)[\mathbf{p}(t) - {}_i\mathbf{w}(t)] & \text{si } i \in \Lambda_g(t) \\ 0 & \text{autrement} \end{cases}$$
   où $\eta(t)$ correspond au taux d'apprentissage et $\Lambda_g(t)$ à un voisinage autour du neurone gagnant g ; $\eta(t)$ et $\Lambda_g(t)$ sont toutes deux des fonctions décroissantes dans le temps.
   (d) $t = t + 1$;

FIG. 7.4 – *Algorithme de Kohonen.*

effectif du neurone i par rapport à son éloignement topologique du neurone gagnant. Tout comme pour le voisinage Λ_g, il importe que le rayon d'action de $\pi_{i,g}$ décroisse dans le temps de manière à ce que vers la fin de l'apprentissage, seul le neurone gagnant subisse une mise à jour significative de son vecteur ${}_g\mathbf{w}$. Pour ce faire, par exemple, on peut faire décroître exponentiellement la variance σ de la gaussienne en fonction du temps :

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau}\right) \tag{7.7}$$

où σ_0 définit la variance au temps initial. Pour l'interprétation de $\pi_{i,g}$, il importe de se rappeler que la distance topologique est discrète : $d_{i,g} \in I\!N$. Par exemple, pour $\sigma_0 = d_{i,g} = 5$ et $\tau = 1000$ on obtient $\sigma(1) = 4.995$ et $\pi_{i,g}(1) = \exp(-25/49.9) = 0.606$, $\pi_{i,g}(500) = 0.257$ et $\pi_{i,g}(1000) = 0.025$. Pour $\sigma_0 = 3$, $d_{i,g} = 5$ et $\tau = 1000$, on obtiendra $\pi_{i,g}(1) = 0.249$, $\pi_{i,g}(500) = 0.023$ et $\pi_{i,g}(1000) = 3.5 \times 10^{-5}$

7.2 Propriétés

Cette section résume les principales propriétés statistiques de l'algorithme de Kohonen par rapport à son espace continu d'entrée $\mathcal{X}$, pour lequel la topologie est définie par une métrique de distance, et son espace discret de sortie $\mathcal{Y}$ dont la topologie est spécifiée par le treillis de la carte

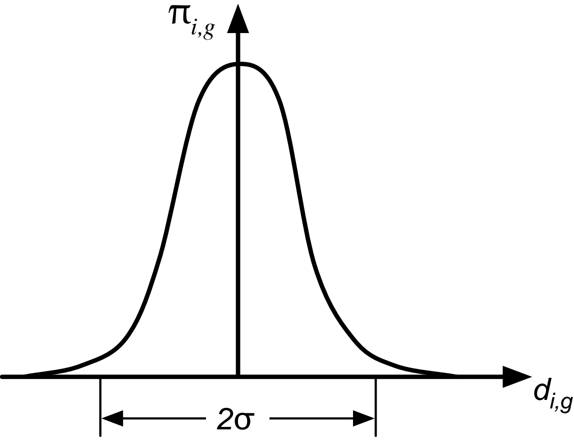


FIG. 7.5 – *Fonction de voisinage gaussienne.*

de neurones. Le réseau réalise donc une transformation non-linéaire $\mathbf{\Phi}$:

$$\mathbf{\Phi} : \mathcal{X} \rightarrow \mathcal{Y} \tag{7.8}$$

Cette transformation peut être vue comme une abstraction de l'équation 7.2 qui détermine l'identité du neurone gagnant en présence d'une stimulus $\mathbf{p} \in \mathcal{X}$.

Propriété 1. Approximation de l'espace d'entrée
La carte auto-organisée de Kohonen, représentée par un ensemble de vecteurs poids synaptiques $\{_i\mathbf{w}|i = 1, 2, \ldots, S\}$, construit dans l'espace de sortie $\mathcal{Y}$, une approximation de l'espace d'entrée $\mathcal{X}$, tel qu'illustré à la figure 7.6.

Propriété 2. Ordre topologique
La carte $\mathbf{\Phi}$ obtenue par l'algorithme de Kohonen est ordonnée topologiquement dans le sens où l'emplacement des neurones dans l'espace des sorties (le treillis) correspond à une région particulière de l'espace des entrées. Deux régions adjacentes dans l'espace des entrées seront aussi adjacentes dans l'espace des sorties. Ceci est une conséquence directe de l'équation 7.1 qui force le vecteurs des poids synaptiques $_g\mathbf{w}$ du neurone gagnant, ainsi qu'un certain nombre de poids voisins $_i\mathbf{w} \in \Lambda_g$ de ce neurone gagnant, à bouger dans la direction des stimuli $\mathbf{p}$.

Propriété 3. Appariement des fonctions de densité
La carte $\mathbf{\Phi}$ reflète les variations statistiques des distributions de points dans l'espace des entrées : les régions de $\mathcal{X}$ contenant beaucoup de stimuli seront appariées avec davantage de neurones que les régions éparses. Ainsi, elles seront mieux modélisées.

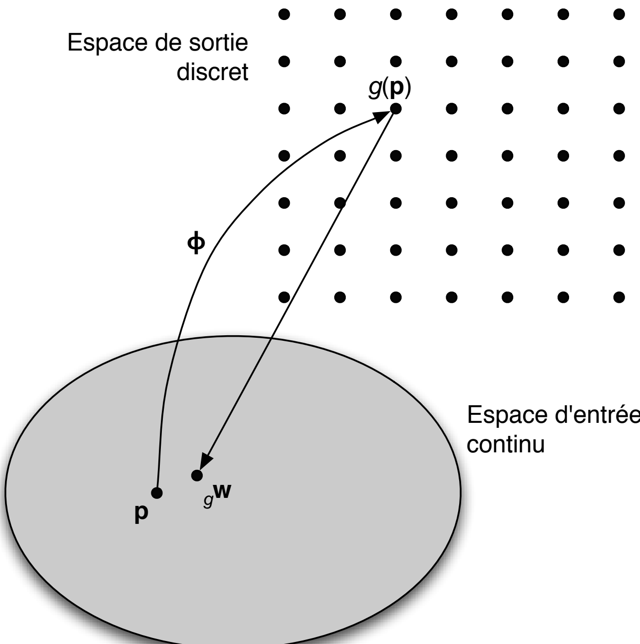


FIG. 7.6 – *Illustration de la relation entre la carte auto-organisée* $\mathbf{\Phi}$ *et le vecteur* ${}_g\mathbf{w}$ *du neurone gagnant pour le stimulus* $\mathbf{p}$.

7.3 Exemples

Cette section présente trois exemples tirés de l'article de Kohonen. Le premier, illustré à la figure 7.7, présente le cas d'une carte à une dimension, c'est-à-dire une carte constituée d'une chaîne unidimensionnelle de neurones. Les stimuli d'apprentissage pour cet exemple sont échantillonnés uniformément à l'intérieur d'un triangle. La première image de la figure ($t = 0$) montre l'initialisation aléatoire du réseau où la chaîne de neurones est totalement repliée sur elle-même. Dès les premières itérations de l'algorithme ($t = 20$), on commence à voir apparaître la chaîne qui se déplie graduellement au fil des itérations en prenant de l'expansion à l'intérieur du triangle où sont situées les stimuli. Après $25\,000$ itérations, on constate que la chaîne s'est complètement déployée pour occuper tout l'espace disponible.

La figure 7.8 présente un exemple avec cette fois-ci une carte à deux dimensions. Les stimuli d'apprentissage pour cet exemple sont échantillonnés uniformément à l'intérieur d'un carré. De nouveau, dès les premières itérations, la carte initialisée avec de petits vecteurs de poids aléatoires se déploie en se dépliant et, après quelques dizaines de milliers d'itérations, finit par occuper tout l'espace disponible.

Le dernier exemple présente un cas plus compliqué où les stimuli sont tridimensionnels, d'une part, et n'ont pas une topologie rectangulaire, d'autre part. Ils sont échantillonnés dans un volume synthétique dont la forme évoque vaguement celle d'un cactus illustré à la figure 7.9(a). Le résultat final de l'apprentissage, donné à la figure 7.9(b), illustre le principal défaut de l'approche de Kohonen. Bien que la carte obtenue approxime raisonnablement bien le volume des stimuli, on observe

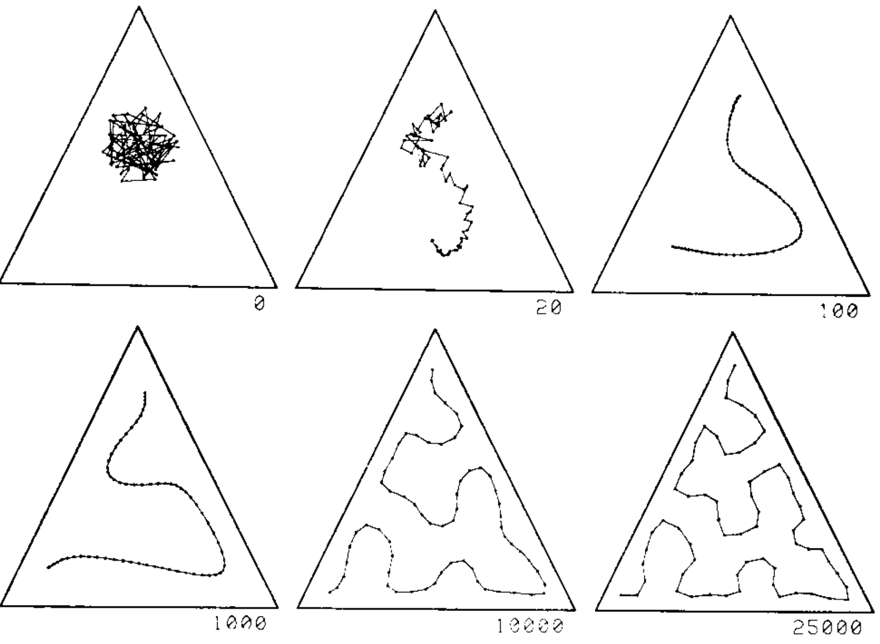


FIG. 7.7 – *Exemple d'une carte auto-organisée à une dimension. Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un triangle.*

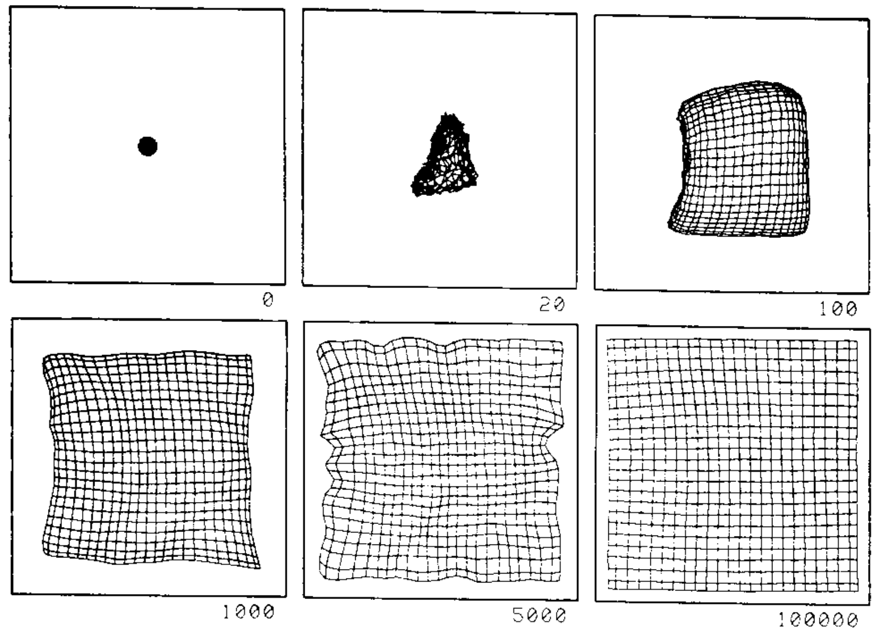


FIG. 7.8 – *Exemple d'une carte auto-organisée à deux dimensions. Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un carré.*

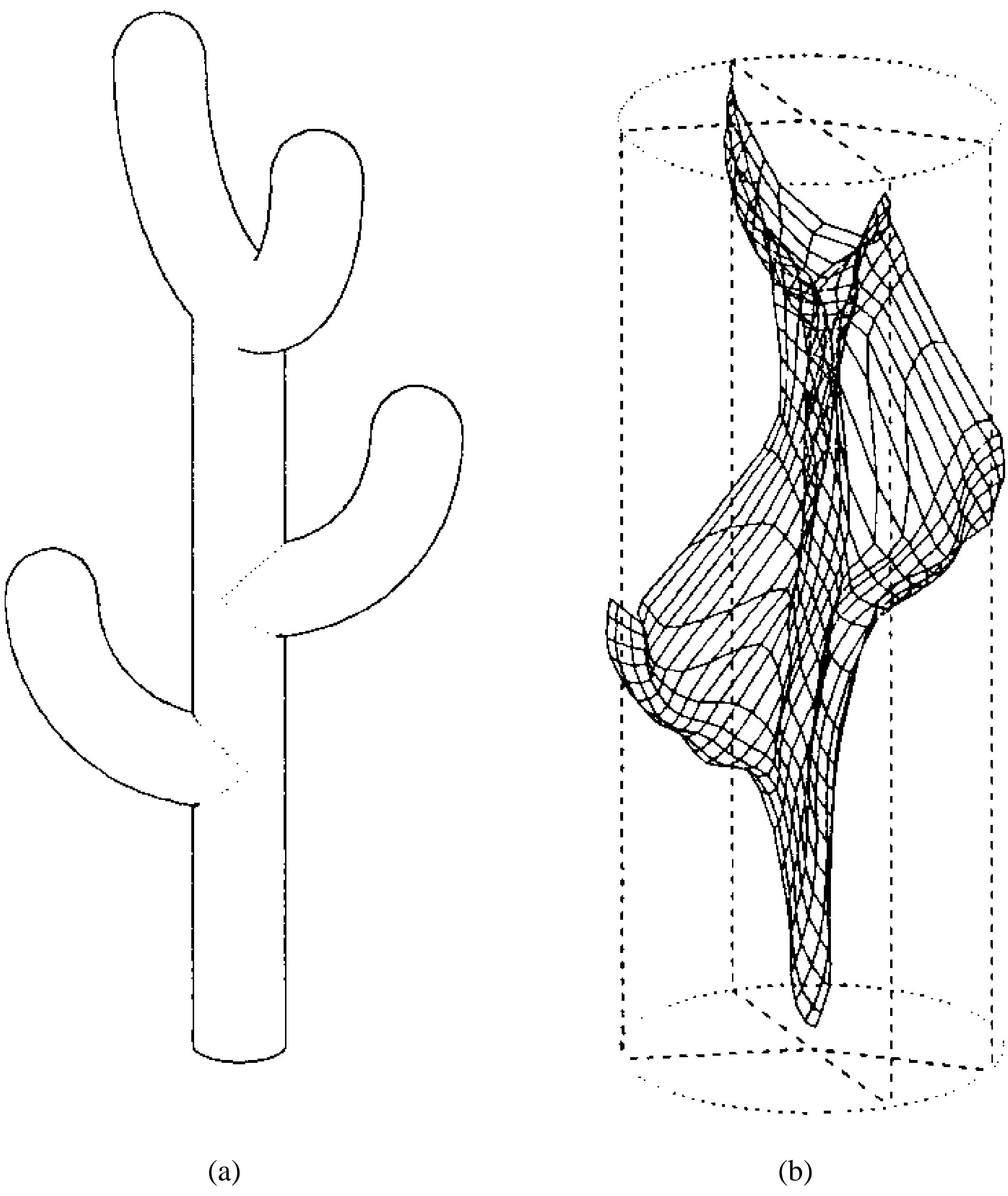


FIG. 7.9 – *Exemple d'une carte auto-organisée à deux dimensions (droite). Les stimuli d'apprentissage sont distribuées uniformément à l'intérieur d'un volume tridimensionnel en forme de cactus (gauche).*

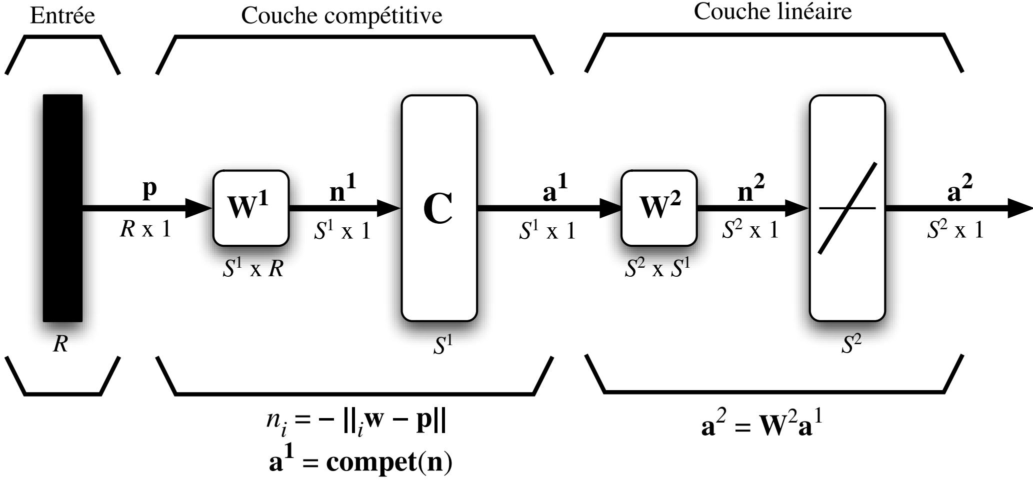


FIG. 7.10 – *Réseau LVQ*

aussi que sa topologie plus ou moins compatible avec celle des données engendre certaines aberrations comme, par exemple, les neurones qui se sont positionnés dans l'espace des entrées, entre les branches du cactus, à des endroits où il n'existe aucun stimulus d'apprentissage. La topologie fixée à priori des cartes de Kohonen constitue donc un handicap que nous tenterons de lever avec l'architecture présentée au chapitre suivant : le réseau GNG.

7.4 Réseau LVQ

En utilisant les principes de l'apprentissage compétitif on peut aussi construire un réseau supervisée nommée LVQ[2]. Ce réseau hybride est illustré à la figure 7.10. Sa première couche compétitive permet de modéliser l'espace des entrées, alors que la seconde linéaire permet de prendre des décisions. Chaque neurone de la première couche est associé a priori à une classe, avec généralement plusieurs neurones par classe ($S^1 >> S^2$). Ceci permet de réunir des frontières de décision convexes dans l'espace des entrées. Tout comme pour le Kohonen ou les nuées dynamiques, les neurones compétitifs de la première couche apprennent à positionner un vecteur prototype dans l'espace des entrées et celui-ci permet de classifier une région de ce dernier, région que l'on peut interpréter comme une sous-classe de la classe à laquelle chaque neurone est associé.

C'est la deuxième couche de neurones qui prend les décisions grâce aux poids $w^2_{i,j}$ de la matrice $\mathbf{W}^2$:

$$w^2_{i,j} = \begin{cases} 1 & \text{si le neurone } j \text{ est associé au neurone } i \text{ (classe } i\text{)} \\ 0 & \text{autrement} \end{cases} \tag{7.9}$$

La matrice $\mathbf{W}^2$ est donc fixée une fois pour toute avant même de commencer l'apprentissage. Les neurones de la première couche sont généralement répartis uniformément entre les classes, à

[2]En anglais : «Linear Vector Quantization».

moins que l'on dispose d'informations sur leur probabilité a priori, auquel cas on pourra affecter plus de neurones à certaines classes et moins à d'autres. L'apprentissage du réseau LVQ consiste simplement à positionner au mieux les prototypes en employant la règle de Kohonen. Lorsqu'un stimulus est correctement classé par le réseau, alors le vecteur de poids ${}_g\mathbf{w}$ du neurone gagnant est rapproché de ce stimulus avec un taux d'apprentissage η. Autrement, c'est que le mauvais neurone a gagné et il s'agit simplement d'éloigné son vecteur de poids du stimulus, au lieu de le rapprocher, avec le même taux d'apprentissage, dans l'espoir qu'un autre neurone associé à la bonne classe puisse gagner la prochaine fois :

$$\Delta_g \mathbf{w}(t) = \begin{cases} \eta\,[\mathbf{p}(t) - {}_g\mathbf{w}(t)] & \text{si } a_g^2 = d_g = 1 \\ -\eta\,[\mathbf{p}(t) - {}_g\mathbf{w}(t)] & \text{si } a_g^2 = 1 \neq d_g = 0 \end{cases} \tag{7.10}$$

Cette approche comporte néanmoins une faille importante, dépendant de l'initialisation des neurones, lorsque pour atteindre une zone associée à sa classe, un neurone doit traverser une zone associée à une autre classe. Parce que le vecteur de poids d'un tel neurone sera repoussé par les stimuli présents dans cette région de transit, il risque fort, en pratique, de ne jamais pouvoir traverser cette zone, et ainsi de ne jamais pouvoir servir à classifier une région associée à la bonne classe.

Pour contourner ce problème, une solution souvent utilisée consiste dans ce cas à non seulement ajuster le vecteur du neurone gagnant, en l'éloignant du stimulus, mais aussi à choisir le neurone le plus proche qui classifie correctement le stimulus pour le rapprocher de celui-ci. De cette façon, deux neurones sont ajustés simultanément : le mauvais gagnant est éloigné du stimulus et un éventuel futur bon gagnant est rapproché de ce dernier.

Chapitre 8

Réseau GNG

Le réseau «Growing Neural Gas»[1] (GNG) est un réseau constructif qui ne pose a priori aucune hypothèse sur la topologie de l'espace des entrées. Un réseau minimal est initialement créé et de nouveaux neurones ainsi que de nouvelles connexions entre les neurones sont ajoutés au fil de l'apprentissage non supervisé.

La topologie du réseau est représentée par un graphe $G = [V(t), E(t)]$ où $V(t)$ désigne l'ensemble des sommets du graphe au temps t et $E(t)$ l'ensemble de ses arêtes. À chaque sommet est associé un neurone caractérisé par un vecteur de poids synaptiques ${}_i\mathbf{w}$ ainsi qu'un signal d'erreur e_i. Ce dernier servira à accumuler l'erreur de modélisation attribuable au neurone i et guidera le choix de l'emplacement où nous ajouterons périodiquement de nouveaux sommets dans le graphe. Les arêtes du graphe, liant deux sommets i et j, correspondent quant à elles à des connexions entre les neurones sous-jacents. À chaque connexion $\{i, j\}$ est associé un âge $a_{i,j}$. Une connexion jeune implique une vraisemblance élevée de la relation topologique, alors qu'au contraire, une connexion âgée signifie une vraisemblance faible de cette relation. Lorsque l'âge d'une connexion dépassera un certain seuil, celle-ci pourra mourir et disparaître du graphe. Comme nous le verrons plus loin, à la fois les connexions et les neurones peuvent apparaître et disparaître du graphe tout au long du processus d'apprentissage. C'est pourquoi les ensembles V et E dépendent tous les deux du temps.

8.1 Algorithme

L'algorithme du GNG comprend un certain nombre de paramètres essentiels. Tout d'abord, il y a τ qui défini la période de temps entre les ajouts de neurone, c'est-à-dire qu'à toutes les τ itérations, nous ajouterons un neurone quelque part dans le graphe. L'apprentissage du GNG, tout comme celui du Kohonen (voir chapitre 7), est de type compétitif. Le neurone gagnant ainsi que ses voisins sont déplacés dans la direction d'un stimulus en proportion de sa distance et d'un certain taux d'apprentissage. Le GNG utilise deux taux distincts, un pour le neurone gagnant, η_g, et l'autre pour ses voisins immédiats, η_v. Contrairement au réseau de Kohonen, cependant,

[1]Bernd Fritzke, «A Growing Neural Gas Network Learns Topologies», publié dans *Advances in Neural Information Processing Systems 7*, G. Tesauro, D.S. Touretzky et T.K. Leen (éditeurs), MIT Press, Cambridge MA, 1995.

ces taux demeurent fixes tout au long de l'apprentissage. Également, le voisinage est fixé à 1, c'est-à-dire que seuls les voisins immédiats du neurone gagnant se déplacent. Les connexions entre les neurones ne peuvent dépasser un âge maximum $a_{\max}$, sinon elle meurent et disparaissent. Finalement, on utilise aussi un paramètre α pour contrôler l'oubli des signaux d'erreur associés aux neurones du réseau. Le rôle précis de chacun de ces paramètres est explicité dans les étapes suivantes :

1. Initialisation : $V = \{x, y\}$ et $E = \{\{x, y\}\}$ avec ${}_x\mathbf{w}(0)$ et ${}_y\mathbf{w}(0)$ initialisés aléatoirement avec de petits poids synaptiques dans l'espace des entrées ; $e_x(0) = e_y(0) = a_{x,y}(0) = 0$.
2. Fixer $t_{\max}$, τ, η_g, η_v, $a_{\max}$ et α.
3. $t = 1$.
4. **Répéter tant que** $t \leq t_{\max}$:
   (a) Choisir aléatoirement un stimulus $\mathbf{p}(t)$ parmi l'ensemble d'apprentissage.
   (b) Déterminer les deux neurones gagnants g_1 et g_2 les plus près de $\mathbf{p}(t)$:
$$g_1 = \arg\min_{i \in V(t)} ||\mathbf{p}(t) - {}_i\mathbf{w}(t)|| \tag{8.1}$$
$$g_2 = \arg\min_{i \in V(t) \setminus \{g_1\}} ||\mathbf{p}(t) - {}_i\mathbf{w}(t)|| \tag{8.2}$$
   (c) Incrémenter les âges de toutes les connexions adjacentes à g_1 :
$$\forall \{i, g_1\} \in E : \; a_{i,g_1} = a_{i,g_1} + 1 \tag{8.3}$$
   (d) Incrémenter l'erreur associée au premier gagnant :
$$e_{g_1} = e_{g_1} + ||\mathbf{p}(t) - {}_{g_1}\mathbf{w}(t)|| \tag{8.4}$$
   (e) Déplacer les vecteur de poids synaptiques de g_1 et de ses voisins immédiats dans la direction de $\mathbf{p}(t)$:
$$\Delta_{g_1}\mathbf{w}(t) = \eta_g[\mathbf{p}(t) - {}_{g_1}\mathbf{w}(t)] \tag{8.5}$$
$$\Delta_v\mathbf{w}(t) = \eta_v[\mathbf{p}(t) - {}_v\mathbf{w}(t)], \quad v \in \Lambda_{g_1} \tag{8.6}$$
   où $\Lambda_{g_1} = \{i \in V | i \neq g_1 et \{i, g_1\} \in E(t)\}$.
   (f) Si $\{g_1, g_2\} \in E(t)$, alors $a_{g_1,g_2} = 0$; autrement :
$$E(t) = E(t) \cup \{\{g_1, g_2\}\}, \quad a_{g_1,g_2}(t) = 0. \tag{8.7}$$
   (g) Retirer de E toutes les connexions pour lesquelles $a_{i,j} > a_{\max}$; retirer aussi tous les neurones qui se retrouveraient isolés (sans aucune connexion) suite à la mort de connexions.
   (h) Si $t \bmod \tau = 0$, alors :
      i. Déterminer le neurone q possédant l'erreur maximum :
$$q = \arg\max_{i \in V} e_i \tag{8.8}$$

ii. Déterminer le neurone r, voisin de q, possédant aussi l'erreur maximum :

$$r = \arg\max_{i \in \Lambda_q} e_i \tag{8.9}$$

iii. Insérer un nouveau neurone x à mi-chemin entre q et r :

$$_x\mathbf{w}(t) = \frac{_q\mathbf{w}(t) + {_r\mathbf{w}(t)}}{2}, \quad e_x = \frac{e_q}{2}, \quad e_q = \frac{e_q}{2} \tag{8.10}$$

iv. Remplacer la connexion $\{q, r\}$ par les deux connexions $\{q, x\}$ et $\{x, r\}$ avec $a_{q,x} = a_{x,r} = 0$.

(i) Réduire les signaux d'erreur de tous les neurones :

$$e_i = \alpha e_i, \quad \forall i \in V(t) \tag{8.11}$$

(j) $t = t + 1$.

8.2 Exemple

La figure 8.1 illustre la capacité qu'à le GNG à apprendre la topologie des données. Pour cet exemple, les données d'apprentissage ont été générées à partir de quatre distributions uniformes : la première dans un volume tridimensionnel en forme de prisme rectangulaire, la deuxième sur une surface rectangulaire apposée sur l'une des faces du prisme, la troisième le long d'un segment de droite au bout de la surface et la dernière le long d'un anneau au bout du segment de droite. La figure montre le graphe du réseau à différents instant durant l'apprentissage. Initialement (en haut à gauche), on commence avec un réseau de deux neurones liés par une seule connexion. Au fil des itérations, des neurones sont ajoutés dans le graphe aux endroits où l'erreur de modélisation est maximum. À la fin, nous seulement le réseau à appris à modéliser l'espace d'entrée, mais il a aussi réussi à apprendre la topologie des différentes formes échantillonnées !

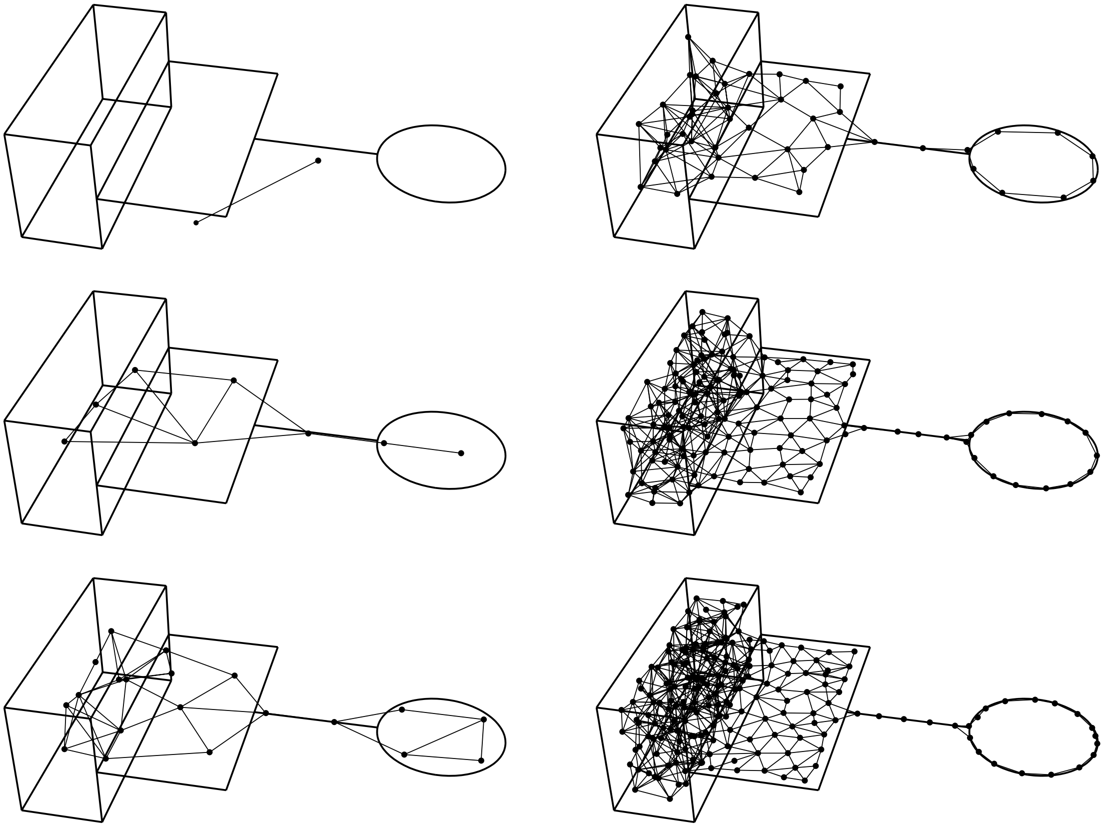


FIG. 8.1 – *Exemple d'un GNG entraîné sur des stimuli échantillonnés dans un volume en forme de prisme rectangulaire, sur une surface rectangulaire apposée perpendiculairement à l'une des faces du prisme et à une courbe en forme d'anneau au bout d'une tige apposée à l'une des extrémités de la surface.*

Chapitre 9

Architectures ART

Les architectures ART[1] sont issues des travaux de Stephen Grossberg et Gail Carpenter. Elles sont basées sur une théorie d'inspiration biologique[2] assez complexe sur laquelle nous n'insisterons pas. Elles se manifestent par différentes implantations spécifiques dont celles nommées «ART1», «ART2», «ART3», «fuzzy ART», «ARTmap», «fuzzy ARTmap», etc. L'architecture ART1[3] possède la particularité de n'accepter que des entrées binaires, alors que le ART2[4] accepte des entrées continues en incorporant aux mécanismes du ART1 différentes opérations complexes de normalisation. Quant au ART3[5], il développe le ART2 davantage en lui ajoutant un nouveau mécanisme de réinitialisation biologiquement inspiré. Ces trois architectures utilisent toutes un processus d'apprentissage non supervisé. Dans ce chapitre, nous allons nous concentrer sur une quatrième architecture, également non supervisée, nommée fuzzy ART[6], qui possède la relative simplicité du ART1 tout en offrant la capacité du ART2 à traiter des entrées continues.

Nous aborderons ensuite une des versions supervisées des architectures ART : le fuzzy ARTmap[7] qui permet non seulement un apprentissage supervisé, mais également un apprentissage incrémental des connaissances, c'est-à-dire un apprentissage où tous les stimuli ne sont pas nécessairement disponibles en tout temps. Ainsi, on peut par exemple apprendre avec un premier sous-ensemble de stimuli, puis mettre ce dernier de côté et poursuivre l'apprentissage avec un autre sous-ensemble sans que les connaissances acquises précédemment soient oubliées par le réseau. Ceci n'est tout simplement pas possible avec d'autres architectures neuronales comme, par exemple, le

[1]En anglais : «Adaptive Resonance Theory».

[2]S. Grossberg, *Studies of Mind and Brain*, Boston : D. Reidel Publishing Co., 1982.

[3]G.A. Carpenter et S. Grossberg, «A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine», *Computer Vision, Graphics, and Image Processing*, vol. 37, p. 54-115, 1987.

[4]G.A. Carpenter et S. Grossberg, «ART2 : Self-Organization of Stable Category Recognition Codes for Analog Input Patterns», *Applied Optics*, vol. 26, no. 23, p. 4919-4930, 1987.

[5]G.A. Carpenter et S. Grossberg, «ART3 : Hierarchical Search using Chemical Transmitters in Self-Organizing Pattern Recognition Architectures», Neural Networks, vol. 3, no. 23, p. 129-152, 1990.

[6]G.A. Carpenter, S. Grossberg et D.B. Rosen, «Fuzzy ART : Fast Stable Learning and Categorization of Analog Patterns by Adaptive Resonance Theory», Neural Networks, vol. 4, p. 759-771, 1991.

[7]G.A. Carpenter, S. Grossberg et J. Reynolds, «Fuzzy ARTmap : A Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps», IEEE Transactions on Neural Networks, vol. 3, p. 698-713, 1992.

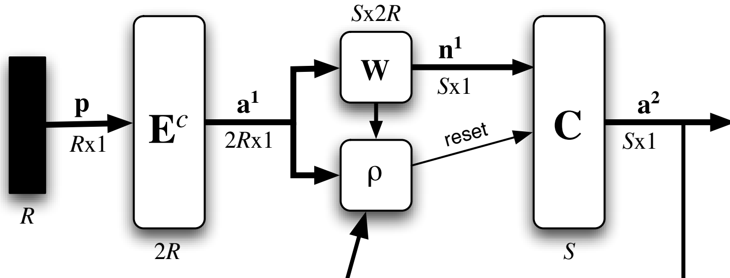


FIG. 9.1 – *Architecture du réseau fuzzy ART.*

perceptron multicouche qui, lors de la deuxième phase d'entraînement, oubliera très rapidement (en quelques périodes) tout ce qu'il a appris lors de la première phase. Pour permettre l'apprentissage incrémental, le fuzzy ARTmap combine deux réseaux fuzzy ART grâce à un carte associative («map» en anglais ; d'où le nom «ARTmap»).

9.1 Fuzzy ART

Le fuzzy ART est un réseau compétitif à deux couches de neurones tel qu'illustré à la figure 9.1. La première couche, notée $\mathbf{E}^c$, sert à coder les stimuli d'entrée avec un encodage dit «complémentaire». La deuxième couche est une couche compétitive semblable à celle du Kohonen. Cependant, tout comme les autres architectures ART, le fuzzy ART incorpore un mécanisme de rétroaction permettant de stabiliser les prototypes appris dans les vecteurs de poids qui relient les deux couches. Ce mécanisme dit de «résonance» est contrôlé par un paramètre ρ qui permet de ré-initialiser au besoin la couche compétitive. Pour chaque stimulus d'entrée, les sorties $\mathbf{a}^2$ du réseau spécifient une catégorie parmi S.

Un peu comme le GNG, le fuzzy ART est un réseau constructif où de nouveaux neurones sont allouées au fil de l'apprentissage. Généralement, on fixe au départ un nombre maximum de neurones S, ce qui fixe également un nombre maximum de catégories de stimuli. Initialement, aucun neurone n'est actif, le premier stimulus d'apprentissage activera le neurone associé à la première catégorie. L'allocation subséquente de nouvelles catégories dépendra à la fois des stimuli et des paramètres de l'algorithme.

Les entrées d'un fuzzy ART doivent être des ensembles flous définis sur un référentiel discret. Soit $\mathbf{p}$ la représentation vectorielle d'un ensemble flou E défini sur un référentiel discret $\mathcal{R} = \{r_1, r_2, \ldots, r_R\}$:

$$E = \{(r_j, p_j) | r_j \in \mathcal{R} \text{ et } 0 \leq p_j \leq 1\}, \quad j = 1, \ldots, R \tag{9.1}$$

où $0 \leq p_j \leq 1$ représente le degré d'appartenance à E de l'élément r_j du référentiel. Le vecteur $\mathbf{p} = [p_1\, p_2 \cdots p_R]^T$ peut ainsi être interprété comme un point dans un hypercube unitaire de R dimensions, comme à la figure 9.2 où $R = 2$. Pour pouvoir utiliser un fuzzy ART, il s'agit donc de transformer (normaliser) nos stimuli d'apprentissage de manière à les faire tous entrer à l'intérieur d'un hypercube unitaire, pour en faire des ensembles flous.

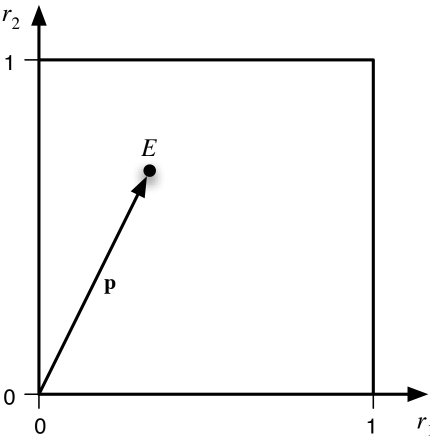


FIG. 9.2 – *Représentation vectorielle d'un ensemble flou E défini sur un référentiel de deux éléments.*

Le réseau fuzzy ART définit trois paramètres α, η et ρ :

$$\alpha > 0, \quad 0 < \eta \leq 1, \quad 0 < \rho < 1 \tag{9.2}$$

où α représente un paramètre de sélection, η le taux d'apprentissage et ρ le taux de vigilance. Nous allons expliciter la signification de ces paramètres ci-dessous.

L'algorithme du fuzzy ART se résume aux étapes suivantes :

1. Initialiser les poids $\mathbf{W} = [{}_1\mathbf{w}\; {}_2\mathbf{w} \cdots {}_S\mathbf{w}]^T$ avec des 1 ;
2. Fixer α, η et ρ ;
3. $t = 1$;
4. **Répéter tant que** $t \leq t_{\max}$:
   (a) Choisir aléatoirement un ensemble flou $\mathbf{p}(t)$ parmi la base d'apprentissage ;
   (b) Effectuer l'encodage complémentaire $\mathbf{E}^c$:

$$\mathbf{a}^1(t) = \begin{bmatrix} \mathbf{p}(t) \\ \mathbf{p}^c(t) \end{bmatrix} \tag{9.3}$$

   où $\mathbf{p}^c = [(1-p_1)(1-p_2)\cdots(1-p_R)]^T$ représente le complément flou de $\mathbf{p}$.

   (c) Calculer les niveaux d'activation n_i^1 des neurones de la première couche :

$$n_i^1(t) = \frac{|\mathbf{a}^1(t) \cap {}_i\mathbf{w}(t)|}{\alpha + |{}_i\mathbf{w}(t)|}, \quad i = 1, \ldots, S, \tag{9.4}$$

   où α est le taux de sélection, $|\cdot|$ désigne la norme l_1 du vecteur :

$$|\mathbf{x}| = \left|[x_1\, x_2 \cdots x_n]^T\right| = \sum_{i=1}^{n} x_i, \tag{9.5}$$

et $\mathbf{x} \cap \mathbf{y}$ représente l'intersection floue entre $\mathbf{x}$ et $\mathbf{y}$:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \cap \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \min(x_1, y_1) \\ \min(x_2, y_2) \\ \vdots \\ \min(x_n, y_n) \end{bmatrix} \tag{9.6}$$

(d) Calculer les sorties de la couche compétitive :

$$a_i^2(t) = \begin{cases} 1 & \text{si } n_i^1(t) = \max\limits_{j=1\cdots S} n_j^1(t) \\ 0 & \text{autrement} \end{cases} \quad , \quad i = 1, \ldots, S. \tag{9.7}$$

Soit $g = \arg\max\limits_j n_j^1(t)$, l'indice du neurone gagnant.

(e) Si le degré de résonance du neurone g avec les sorties de la couche d'entrée est inférieur au seuil ρ :

$$\frac{|\mathbf{a}^1(t) \cap {}_g\mathbf{w}(t)|}{|\mathbf{a}^1(t)|} < \rho, \tag{9.8}$$

alors bloquer temporairement le neurone g pour qu'il ne puisse plus gagner et retourner à l'étape 4d pour choisir le prochain gagnant.

(f) Libérer tous les neurones bloqués.

(g) Mettre à jour le prototype du neurone gagnant :

$$\Delta_g\mathbf{w}(t) = \eta \left[\left(\mathbf{a}^1(t) \cap {}_g\mathbf{w}(t) \right) - {}_g\mathbf{w}(t) \right], \tag{9.9}$$

où η est le taux d'apprentissage.

(h) $t = t + 1$;

Fin

La première chose que l'on peut remarquer à l'équation 9.3 est que la norme des vecteurs encodés demeure toujours constante :

$$|\mathbf{a}^1(t)| = \sum_{i=1}^{R} p_i + \sum_{i=1}^{R} (1 - p_i) = \sum_{i=1}^{R} p_i + R - \sum_{i=1}^{R} p_i = R \tag{9.10}$$

Le processus d'encodage permet donc de normaliser en amplitude les stimuli d'entrée.

Le niveau d'activation n_i^1 du neurone i de la couche compétitive, calculés à l'équation 9.4, permet de mesurer le degré de ressemblance entre le stimulus d'entrée $\mathbf{a}^1$, exprimé dans son encodage complémentaire, et le vecteur de poids ${}_i\mathbf{w}$. En posant ${}_i\mathbf{w} = [\mathbf{x}\ \mathbf{y}^c]^T$, où $\mathbf{x}$ et $\mathbf{y}$ sont deux vecteurs quelconques dans l'espace des stimuli, on obtient l'interprétation géométrique de la figure 9.3 pour le cas particulier d'un espace à deux dimensions. La région définie par le rectangle en trait plein correspond à la zone d'activité du neurone i. Pour tous les stimuli $\mathbf{p}$ situés à l'intérieur de cette zone, le numérateur de l'équation 9.4 sera égal à :

$$\mathbf{a}^1 \cap {}_i\mathbf{w} = \begin{bmatrix} \mathbf{p} \\ \mathbf{p}^c \end{bmatrix} \cap \begin{bmatrix} \mathbf{x} \\ \mathbf{y}^c \end{bmatrix} = \begin{bmatrix} p \cap x \\ (p \cup y)^c \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y}^c \end{bmatrix} = {}_i\mathbf{w} \tag{9.11}$$

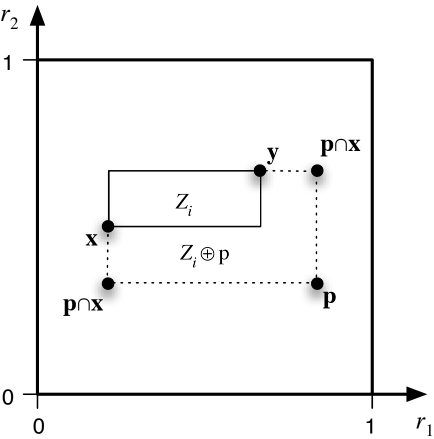


FIG. 9.3 – *Régions associées à* ${}_i\mathbf{w} = [\mathbf{x}\,\mathbf{y}^c]^T$ *(en trait plein) et à* $\mathbf{a}^1 \cap {}_i\mathbf{w}$ *(en trait pointillé).*

Par conséquent, on obtient dans ce cas $|\mathbf{a}^1 \cap {}_i\mathbf{w}| = |{}_i\mathbf{w}|$ et $n_i^1 = \frac{|{}_i\mathbf{w}|}{\alpha + |{}_i\mathbf{w}|}$, c'est-à-dire une valeur inférieure à 1. Pour un ${}_i\mathbf{w}$ fixé, plus α sera grand, plus le niveau d'activation sera petit et, inversement, plus ${}_i\mathbf{w}$ sera grand, plus le niveau d'activation tendra vers 1 pour un α donné.

Pour les stimuli $\mathbf{p}$ situés à l'extérieur de la zone, comme à la figure 9.3, l'opération $\mathbf{a}^1 \cap {}_i\mathbf{w}$ engendre la nouvelle région indiquée en trait pointillé. Pour cette région, nous aurons $|\mathbf{a}^1 \cap {}_i\mathbf{w}| < |{}_i\mathbf{w}|$ et le niveau d'activation calculé sera nettement inférieur à 1. Ainsi, les neurones dont la région associée englobe $\mathbf{p}$ gagneront la compétition avant les autres, et s'ils sont plusieurs à l'englober, c'est celui dont la région est la plus petite qui sera déclaré gagnant.

Soit Z_i la région associée à ${}_i\mathbf{w}$. Dans le cas général d'un espace des stimuli à R dimensions, il s'agit d'un hyper-rectangle défini respectivement par les coins $\mathbf{x}$ et $\mathbf{y}$ de ses composantes minimum et maximum. La dimension de cette région peut se calculer de la façon suivante :

$$|Z_i| \equiv |\mathbf{y} - \mathbf{x}| \tag{9.12}$$

Pour qu'un vecteur ${}_i\mathbf{w}$ puisse changer de valeur (équation 9.9), il faut que le neurone correspondant gagne la compétition de l'équation 9.7 et que celui-ci n'échoue pas le critère de résonance de l'équation 9.8. Dans ce cas seulement, la région Z_i associée à ce neurone pourra grandir et, éventuellement, englober le stimulus d'entrée. Si l'on pose un taux d'apprentissage $\eta = 1$ à l'équation 9.9, on parle dans ce cas d'apprentissage «instantané», alors la région Z_i sera agrandie juste assez pour englober le stimulus $\mathbf{p}$. Dans le cas particulier d'un neurone inactif (un neurone qui n'a jamais gagné), c'est-à-dire un neurone dont le ${}_i\mathbf{w}$ ne contient que des uns (étape 1 de l'algorithme), on obtiendra ${}_i\mathbf{w}(t+1) = \mathbf{a}^1(t)$ et la région Z_i sera limité au point $\mathbf{p}(t)$. Par la suite, l'apprentissage ne pourra qu'augmenter la taille de la région Z_i et, par conséquent, la norme de ${}_i\mathbf{w}$ ne pourra que diminuer. En fait, toujours sous l'hypothèse de $\eta = 1$, on peut montrer que la région Z_i correspondra toujours au plus petit rectangle (ou hyper-rectangle) englobant l'ensemble des stimuli qui vont faire gagner et résonner le neurone i au cours de l'apprentissage.

Le critère de résonance de l'équation 9.8 impose une contrainte sur la taille maximum que peut atteindre les régions associées aux neurones du fuzzy ART. En effet, le dénominateur $|\mathbf{a}^1(t)| = R$

est une constante et il faut que :

$$|\mathbf{a}^1(t) \cap {}_g\mathbf{w}(t)| \geq R\rho \tag{9.13}$$

pour que le neurone gagnant puisse subir une modification de son vecteur de poids. Or :

$$\begin{aligned}
|\mathbf{a}^1(t) \cap {}_g\mathbf{w}(t)| &= \left| \begin{bmatrix} \mathbf{p} \\ \mathbf{p}^c \end{bmatrix} \cap \begin{bmatrix} \mathbf{x} \\ \mathbf{y}^c \end{bmatrix} \right| \\
&= \left| \begin{bmatrix} \mathbf{p} \cap \mathbf{x} \\ \mathbf{p}^c \cap \mathbf{y}^c \end{bmatrix} \right| \\
&= \left| \begin{bmatrix} \mathbf{p} \cap \mathbf{x} \\ (\mathbf{p} \cup \mathbf{y})^c \end{bmatrix} \right| \quad (deMorgan) \\
&= |\mathbf{p} \cap \mathbf{x}| + R - |\mathbf{p} \cup \mathbf{y}| \\
&= R - (|\mathbf{p} \cup \mathbf{y}| - |\mathbf{p} \cap \mathbf{x}|) \\
&= R - |Z_g \oplus \mathbf{p}|
\end{aligned} \tag{9.14}$$

où $Z_g \oplus \mathbf{p}$ désigne la plus petite région (hyper-rectangle) qui englobe à la fois Z_g et $\mathbf{p}$ (voir figure 9.3 avec $i = g$). Par conséquent, on obtient l'expression suivante :

$$|Z_g \oplus \mathbf{p}| \leq R(1 - \rho) \tag{9.15}$$

qui spécifie la taille maximum qu'une région peut atteindre en fonction du taux de vigilance ρ. Plus il est grand, plus les régions sont contraintes à de petites tailles. À la limite, lorsque $\rho \longrightarrow 1$, le fuzzy ART apprendra les stimuli par cœur ! Plus il est petit, plus les régions pourront croître. À la limite, lorsque $\rho \longrightarrow 0$, un seul neurone pourrait couvrir tout l'espace d'entrée, et le réseau ne produirait plus qu'une seule catégorie. Lorsque le neurone gagnant ne respecte pas le critère de résonance, alors il est temporairement retiré de la compétition et le neurone suivant ayant le plus grand niveau d'activation est sélectionné à son tour. Ce processus est répété jusqu'à ce qu'on trouve un neurone qui «résonne». Ceci sera toujours possible dans la mesure où le réseau comporte encore des neurones inactifs (la preuve est laissée en exercice).

Généralement, dans le cas d'un stimulus qui ne ressemble à aucune des catégories actives, c'est-à-dire lorsque le neurone gagnant était précédemment inactif, on fixera automatiquement le taux d'apprentissage $\eta = 1$ pour passer en mode d'apprentissage «instantané». Sinon, si le stimulus engendre la résonance d'une catégorie existante, alors on fixera un taux $0 < \eta \leq 1$ tel que spécifiée par l'utilisateur.

Finalement, mentionnons que le paramètre de sélection α est habituellement fixé à une petite valeur telle que, par exemple, 0.01. Ainsi, le neurone gagnant sera celui dont la région associée requiert un minimum d'agrandissement relatif pour pouvoir englober le stimulus d'entrée. Lorsque $\alpha \longrightarrow \infty$, ce n'est plus le pourcentage d'agrandissement qui compte, mais seulement la dimension finalle.

9.2 Fuzzy ARTmap

Le fuzzy ARTmap est un réseau supervisé capable d'effectuer un apprentissage incrémental. Il est constitué de deux réseaux fuzzy ART respectivement nommés ART$_p$ et ART$_d$, tel qu'illustré

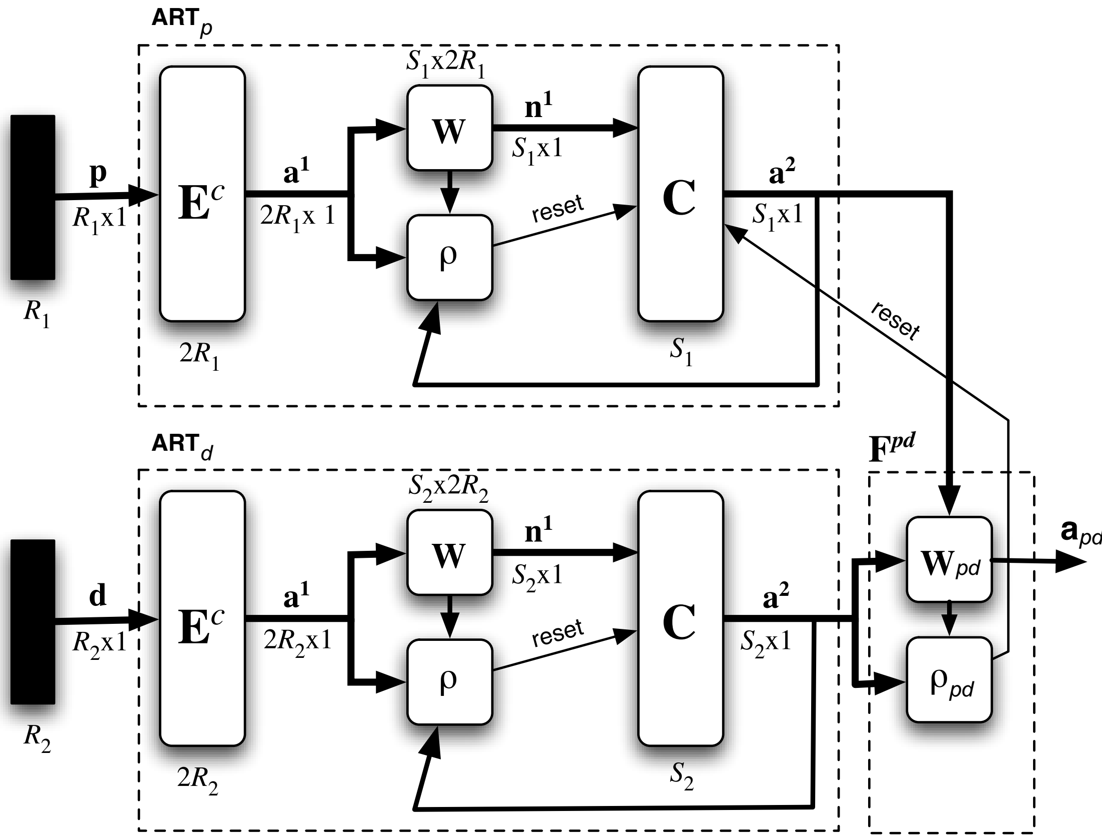


FIG. 9.4 – *Architecture du réseau fuzzy ARTmap.*

à la figure 9.4. Les réseaux ART$_p$ et ART$_d$ sont reliés entre eux par un champ associatif nommé $\mathbf{F}^{pd}$. Ce champ est utilisé pour effectuer des prédictions associatives entre les catégories du module ART$_p$, qui sont associées aux stimuli d'entrée, et celles du module ART$_d$, qui sont associées aux sorties désirées.

En mode d'apprentissage, on propage un stimulus d'entrée dans le module ART$_p$ et, simultanément, on propage le vecteur des sorties désirées dans le module ART$_d$. Les catégories produites par les deux modules sont alors comparées à l'aide du champ associatif $\mathbf{F}^{pd}$. La matrice $\mathbf{W}_{pd}$ de dimension $S_2 \times S_1$ contient des vecteurs de poids $\mathbf{w}_i$ associés à chacune des S_1 catégories du module ART$_p$. Lorsque la catégorie i produite par le ART$_p$ résonne avec la catégorie j produite par le ART$_d$, alors $\mathbf{w}_i$ est modifié pour ressembler davantage à $\mathbf{a}^2_d$, la sortie de ART$_d$. Sinon, le module de résonance de $\mathbf{F}^{pd}$ transmet un signal au ART$_p$ pour que celui-ci produise une nouvelle catégorie. Ce processus est recommencé jusqu'à ce que l'on trouve une association adéquate.

En mode de reconnaissance, on ne propage rien dans le module ART$_d$; celui-ci demeure inactif. Le stimulus d'entrée est simplement propagé dans le module ART$_p$ et la catégorie produite est utilisée pour sélectionner le vecteur correspondant dans $\mathbf{W}_{pd}$.

Plus formellement, voici toutes les étapes de l'algorithme d'apprentissage du fuzzy ARTmap :

1. Initialiser les poids $\mathbf{W}_{pd} = [\mathbf{w}_1 \, \mathbf{w}_2 \cdots \mathbf{w}_{S_1}]$ avec des 1 ;
2. Fixer les paramètres des modules ART$_p$ et ART$_d$;
3. Fixer η_{pd} et ρ_{pd} de $\mathbf{F}^{pd}$;
4. $t = 1$;
5. **Répéter tant que** $t \leq t_{\max}$:
   (a) Choisir un couple $(\mathbf{p}(t), \mathbf{d}(t))$ parmi les données de la base d'apprentissage ;
   (b) Propager $\mathbf{p}(t)$ dans le ART$_p$;
   (c) Propager $\mathbf{d}(t)$ dans le ART$_d$;
   (d) Calculer la sortie de $\mathbf{F}^{pd}$:
   $$\mathbf{a}_{pd} = \mathbf{a}_d^2 \cap \mathbf{w}_g^{pd} \tag{9.16}$$
   où $\mathbf{a}_d^2$ désigne le vecteur de sortie de ART$_d$ et g correspond à la catégorie produite par le ART$_p$;
   (e) Si le degré de résonance de $\mathbf{F}^{pd}$ est inférieur au seuil ρ_{pd}, c'est-à-dire si :
   $$\frac{|\mathbf{a}_{pd}|}{|\mathbf{a}_d^2|} < \rho_{pd}, \tag{9.17}$$
   alors augmenter ρ_p juste assez pour forcer le ART$_p$ à produire une nouvelle catégorie et retourner à l'étape 5d. Pour ce faire, il faudra fixer temporairement ρ_p légèrement plus grand que $|\mathbf{a}_p^1 \cap \mathbf{w}_g^p| / |\mathbf{a}_p^1|$.
   (f) Mettre à jour le vecteur gagnant de $\mathbf{F}^{pd}$:
   $$\mathbf{w}_g(t+1) = \rho_{pd} \, \mathbf{a}_d^2 + (1 - \rho_{pd}) \, \mathbf{w}_g(t) \tag{9.18}$$
   (g) S'il y a lieu, remettre ρ_p à sa valeur initiale ;

   Fin

Plus souvent qu'autrement, le fuzzy ARTmap est utilisé pour faire du classement. Dans ce cas, la dimension R_2 de l'espace d'entrée du module ART$_d$ correspondra au nombre de classes et les vecteurs d'entrée $\mathbf{d}$ seront restreints à des ensembles non flous, c'est-à-dire à des vecteurs dont une des composantes est 1, et toutes les autres sont 0 (coins de l'hypercube). Les taux de vigilance et d'apprentissage seront $\eta_d = \rho_d = 1$ de manière à ce que les vecteurs de sorties désirés soient appris par cœur et instantanément. Autrement dit, le module ART$_d$ ne sert presqu'à rien dans ce cas, sinon à produire un indice de classe. De son côté, le module ART$_p$ contiendra au moins autant de neurones qu'il y a de classes, mais typiquement de 4 à 10 fois plus, pour pouvoir créer des frontières de décision complexes formées de plusieurs hyper-rectangles entrelacés. Les paramètres du module ART$_p$ seront fixés de manière à ce que celui-ci puisse bien modéliser son espace des stimuli. On emploiera un taux d'apprentissage élevé ($\eta_p = 1$) si les données sont peu bruitées, et un taux plus faible (p.ex. $\eta_p = 0.5$) en présence de bruit. Le taux de vigilance sera d'autant plus élevé que l'on cherche à classer correctement toutes les données. Mais gare au sur-apprentissage ! À la limite, si $\rho_p = 1$ on apprendra littéralement par cœur toutes les données mais la généralisation sera

TAB. 9.1 – *Valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte de classement.*

Paramètres	Intervalles	valeurs suggérées
taux de sélection	$]0, \infty[$	$\alpha_p = 0.01$ $\alpha_d = 0.01$
taux d'apprentissage	$]0, 1]$	$0.5 \leq \eta_p \leq 1$ $\eta_d = 1$ $\eta_{pd} = 1$
taux de vigilance	$[0, 1]$	$0.5 \leq \rho_p \leq 0.9$ $\rho_d = 1$ $\rho_{pd} = 1$

TAB. 9.2 – *Valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte d'approximation de fonction.*

Paramètres	Intervalles	valeurs suggérées
taux de sélection	$]0, \infty[$	$\alpha_p = 0.01$ $\alpha_d = 0.01$
taux d'apprentissage	$]0, 1]$	$0.5 \leq \eta_p \leq 1$ $0.5 \leq \eta_d \leq 1$ $\eta_{pd} = 1$
taux de vigilance	$[0, 1]$	$0.5 \leq \rho_p \leq 0.9$ $0.5 \leq \rho_p \leq 0.9$ $\rho_{pd} = 1$

catastrophique en présence de bruit. Une valeur comprise dans l'intervalle $0.5 \geq \rho_p \geq 0.9$ produit généralement de bons résultats. Finalement, le taux d'apprentissage du champ associatif $\mathbf{F}^{pd}$ est souvent fixé à $\eta_{pd} = 1$ pour apprendre instantanément les bonnes associations de catégorie. Dans ce cas, la valeur du taux de vigilance ρ_{pd} n'a pas d'importance (on la fixe donc à 1). En conclusion, le tableau 9.1 donne des valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte de classement.

Dans un contexte d'approximation de fonction, le rôle du module ART$_d$ prend toute son importance. C'est lui qui produira des catégories de valeurs pour la fonction que l'on tente de modéliser. Pour une fonction scalaire, nous aurons $R_2 = 1$. Pour une fonction vectorielle, nous aurons $R_2 > 1$. En variant la vigilance ρ_d nous pourrons varier la précision de l'approximation. À la limite, en fixant $\rho_d = 1$, nous pourrons apprendre par cœur toutes les valeurs ayant servi à l'apprentissage. Il faut donc prendre garde au sur-apprentissage et limiter ce paramètre à la plage $0.5 \leq \eta_d \leq 0.9$, tout comme pour la vigilance du module ART$_p$. En ce qui concerne les paramètres du module $\mathbf{F}^{pd}$, il sont souvent fixés à $\eta_{pd} = \rho_{pd} = 1$, tout comme dans le contexte de classement. Le tableau 9.2 résume les valeurs suggérées pour les paramètres du fuzzy ARTmap dans un contexte d'approximation de fonction.

Mentionnons finalement que, dans un contexte d'approximation de fonction, contrairement à celui d'un problème de classement où la sortie a_{pd} du réseau fuzzy ARTmap indique directement

l'indice j de la classe associée au stimulus d'entrée, la valeur de la fonction recherchée doit être extraite du module ART$_d$ en allant chercher le vecteur de poids ${}_j\mathbf{w}^d = [\mathbf{x}\ \mathbf{y}^c]^T$ associé à cet indice de classe et en calculant le centre $(\mathbf{x} + \mathbf{y})/2$ de sa région.

Chapitre 10

ACP et apprentissage hebbien

L'analyse en composantes principales (ACP) est une méthode d'analyse des données qui permet de réduire la dimension d'un espace d'entrée en ne retenant que les axes où la variance est importante. Soit un ensemble de Q vecteurs $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_Q\}$ définis dans $\Re^n$. Ces vecteurs forment un nuage de points dans un espace à n dimensions. En choisissant de nouvelles bases, on désire représenter ces vecteurs dans $\Re^m$, avec $m < n$ tout en minimisant la perte d'information.

Shannon définit l'information contenu dans une variable aléatoire $X = \{x_1, x_2, \ldots, x_N\}$ à partir de son entropie $H(X)$:

$$H(X) = -\sum_{k=1}^{N} \text{Pr}(x_k) \log[\text{Pr}(x_k)] = -E[\log(\text{Pr}(x_k))] \tag{10.1}$$

où $\text{Pr}(x_k)$ désigne la probabilité de rencontrer la $k^{\text{ème}}$ réalisation de X et E représente l'espérance mathématique. L'entropie nous dit que plus un x_k possède une probabilité élevée, moins il contient d'information. À la limite, lorsque la variable devient déterministe, c'est-à-dire lorsque $\text{Pr}(x_k) \to 1$ pour un certain k et que, par conséquent, $\text{Pr}(x_j) \to 0$ pour $\forall j \neq k$, alors l'entropie tend vers 0. Cette définition suppose cependant que nous connaissions a priori la loi de densité de nos variables aléatoires ce qui, dans la pratique, n'est pas toujours le cas. Cependant, si l'on suppose qu'elles obéissent à des lois gaussiennes[1] :

$$\text{Pr}(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right], \tag{10.2}$$

où μ représente la moyenne et σ^2 la variance, alors l'entropie devient :

$$H(x) = E\left[\frac{1}{2}\log(2\pi\sigma^2)\right] + \frac{1}{2}E\left[\left(\frac{x-\mu}{\sigma}\right)^2\right] = \frac{1}{2}\log(2\pi\sigma^2), \tag{10.3}$$

et l'on observe qu'elle ne dépend plus que de la variance. Par conséquent, dans le cas de distributions gaussiennes, on peut conclure que la variance est synonyme d'information.

[1]Notez bien que la loi de Gauss s'applique à une variable aléatoire continue. Dans ce cas, il faut remplacer la sommation par une intégrale dans l'équation 10.1.

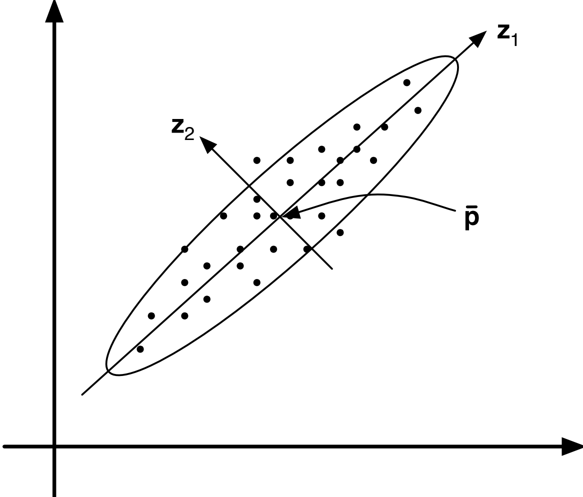


FIG. 10.1 – *Illustration des composantes principales pour un nuage de points en deux dimensions.*

Ceci nous amène à définir l'analyse en composantes principales en termes de la variance et de la covariance entre les différentes composantes de nos stimuli. Intuitivement, nous recherchons les directions dans nos nuages de points où la variance est maximale, tel qu'illustré à la figure 10.1 dans un espace à deux dimensions, où $\mathbf{z}_1$ et $\mathbf{z}_2$ donnent l'orientation des deux composantes principales et l'ellipse symbolise l'hypothèse de distribution gaussienne des vecteurs qui est sous-jacente à l'ACP.

Tout d'abord, calculons la moyenne $\bar{\mathbf{p}}$ de nos stimuli :

$$\bar{\mathbf{p}} = \frac{1}{Q}\sum_{k=1}^{Q}\mathbf{p}_k. \tag{10.4}$$

C'est le centre du nuage de points. La matrice de covariance $\mathbf{C}$ de nos stimuli est donnée par :

$$\mathbf{C} = \frac{1}{Q-1}\sum_{k=1}^{Q}(\mathbf{p}_k - \bar{\mathbf{p}})(\mathbf{p}_k - \bar{\mathbf{p}})^T = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \cdots & \sigma_{1n}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \cdots & \sigma_{2n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1}^2 & \sigma_{n2}^2 & \cdots & \sigma_{nn}^2 \end{bmatrix} \tag{10.5}$$

où σ_{ij}^2 représente la covariance entre les composantes i et j de nos stimuli, et n est leur dimension. Une matrice de covariance est toujours symétrique et positive définie (valeurs propres réelles et positives).

Pour trouver les composantes principales de nos données, il s'agit de déterminer les valeurs et les vecteurs propres de la matrice $\mathbf{C}$ (voir section 3.2.3). Les vecteurs propres de $\mathbf{C}$ définissent dans $\Re^n$ les orientations des composantes principales de nos stimuli lorsque l'origine de l'espace vectoriel est déplacé en $\bar{\mathbf{p}}$. Les valeurs propres, quant à elles, représentent l'importance de chacune de ces composantes. Elles correspondent aux variances des données lorsque projetées dans chacune

de ces orientations. Soit la matrice $\mathbf{Z}$ dont les colonnes contiennent les n vecteurs propres de $\mathbf{C}$:

$$\mathbf{Z} = [\mathbf{z}_1\, \mathbf{z}_2 \cdots \mathbf{z}_n]. \tag{10.6}$$

Alors $\mathbf{Z}$ est une matrice de rotation et $\mathbf{p}'_k = \mathbf{Z}^{-1}(\mathbf{p}_k - \bar{\mathbf{p}}) = \mathbf{Z}^T(\mathbf{p}_k - \bar{\mathbf{p}})$ représente le stimulus $\mathbf{p}_k$ après translation et rotation des axes dans la direction des composantes principales. Si l'on calcule la matrice de covariance $\mathbf{C}'$ des $\mathbf{p}'_k$, on obtient alors une matrice diagonale Λ dont les éléments correspondent aux valeurs propres de $\mathbf{C}$:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}. \tag{10.7}$$

Sans perte de généralité, supposons que les vecteurs propres $\mathbf{z}_i$ sont triés en ordre décroissant de leur valeur propre : $\lambda_i \geq \lambda_{i+1},\ i = 1, \ldots, n-1$, alors l'analyse en composante principale consiste à choisir les m premiers vecteurs propres associés aux plus grandes valeurs propres, c'est-à-dire ceux qui maximisent la variance. Pour réduire la dimension de l'espace de représentation de nos stimuli, il suffit donc de construire la matrice $\mathbf{W}$ suivante :

$$\mathbf{W} = [\mathbf{z}_1\, \mathbf{z}_2 \ldots \mathbf{z}_m], \quad m < n \tag{10.8}$$

et de s'en servir pour projeter les $\mathbf{p}_k$ de n dimension en $\mathbf{p}'_k$ à m dimension :

$$\mathbf{p}'_k = \mathbf{W}^T(\mathbf{p}_k - \bar{\mathbf{p}}), \quad k = 1, \ldots, Q. \tag{10.9}$$

La proportion de la variance des $\mathbf{p}_k$ contenue dans les $\mathbf{p}'_k$ se mesure habituellement par le ratio :

$$\frac{\sum_{i=1}^{m} \lambda_i}{\sum_{i=1}^{n} \lambda_i} > \tau, \tag{10.10}$$

que l'on peut contraindre, par exemple, aux m composantes principales qui expliquent au moins $\tau > 95\%$ de la variance des stimuli d'origine. La trace $\sum_{i=1}^{n} \lambda_i$ de la matrice Λ sert ici à mesurer la variance globale des stimuli d'apprentissage. Il s'agit d'une mesure de volume, ou d'hyper-volume, tout comme la norme est une mesure de longueur. Une autre mesure de volume pour un nuage de points consiste à calculer le déterminant de la matrice de covariance. Ainsi, on pourrait également choisir nos m composantes de la façon suivante :

$$\frac{\prod_{i=1}^{m} \lambda_i}{\prod_{i=1}^{n} \lambda_i} > \tau, \tag{10.11}$$

Mentionnons finalement qu'on peut aussi utiliser les valeurs propres de l'analyse en composantes principales pour effectuer un blanchiment de nos stimuli en effectuant l'opération suivante :

$$\mathbf{p}''_k = \Lambda_m^{-1/2}\, \mathbf{p}'_k, \quad k = 1, \ldots, Q, \tag{10.12}$$

où Λ_m représente la matrice diagonale des m premières composantes de Λ :

$$\Lambda_m = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_m \end{bmatrix}, \tag{10.13}$$

ce qui engendre une matrice de covariance unitaire pour les $\mathbf{p}''_k$.

10.1 Règle de Hebb

L'apprentissage par la règle de Hebb (voir section 4.2) exprime la variation de poids en fonction de la corrélation entre l'entrée $\mathbf{p}$ et la sortie a d'un neurone :

$$\Delta\mathbf{w} = \eta\,\mathbf{p}\,a. \tag{10.14}$$

Cette règle nous dit que plus la réponse du neurone sera forte vis-à-vis d'un stimulus, plus la variation de poids sera grande.

Dans le cas d'un neurone linéaire, nous avons la relation $a = \mathbf{w}^T\mathbf{p} = \mathbf{p}^T\mathbf{w}$. En interprétant $\mathbf{w}$ comme une direction dans l'espace des stimuli et en supposant que les stimuli d'entrée sont centrés sur leur moyenne[2], on peut se créer l'indice de performance F suivant :

$$F = a^2 = (\mathbf{w}^T\mathbf{p})(\mathbf{p}^T\mathbf{w}), \tag{10.15}$$

visant à maximiser la variance. Pour un module $||\mathbf{w}||$ fixé, on obtient une espérance $E(F)$:

$$E[F] = E[(\mathbf{w}^T\mathbf{p})(\mathbf{p}^T\mathbf{w})] = \mathbf{w}^T E[\mathbf{p}\mathbf{p}^T]\mathbf{w} = \mathbf{w}^T C\mathbf{w} \tag{10.16}$$

qui dépend de l'orientation de $\mathbf{w}$ et de la matrice de covariance $\mathbf{C}$ des stimuli. Clairement, on constate que F sera maximum lorsque $\mathbf{w}$ sera orienté dans la direction de la composante principale de $\mathbf{C}$.

Le problème majeur avec cette formulation de la règle de Hebb est que celle-ci est instable. Le module de $\mathbf{w}$ aura tendance à croître sans cesse et l'approche diverge presque toujours. Une solution consiste à normaliser $\mathbf{w}$:

$$\mathbf{w}(t+1) = \frac{\mathbf{w}(t) + \eta\,\mathbf{p}\,a}{||\mathbf{w}(t) + \eta\,\mathbf{p}\,a||} \tag{10.17}$$

10.2 Règle de Oja

Une autre solution consiste à adopter une approximation de l'équation 10.17, nommée «règle de Oja»[3] :

$$\Delta\mathbf{w} = \eta\,a(\mathbf{p} - a\,\mathbf{w}) = \eta(a\,\mathbf{p} - a^2\,\mathbf{w}) \tag{10.18}$$

Pour voir que cette règle possède bien le potentiel de trouver la composante principale des stimuli, il suffit de calculer l'espérance de la variation des poids :

$$\begin{aligned}
E(\Delta\mathbf{w}) &= E[\eta(a\mathbf{p} - a^2\mathbf{w})] && (10.19)\\
&= \eta\,E[\mathbf{p}(\mathbf{p}^T\mathbf{w}) - (\mathbf{w}^T\mathbf{p})(\mathbf{p}^T\mathbf{w})\mathbf{w}] && (10.20)\\
&= \eta\,E[(\mathbf{p}\mathbf{p}^T)\mathbf{w} - \mathbf{w}^T(\mathbf{p}\mathbf{p}^T)\mathbf{w}\mathbf{w}] && (10.21)\\
&= \eta\,(\mathbf{C}\mathbf{w} - \mathbf{w}^T\mathbf{C}\mathbf{w}\mathbf{w}) && (10.22)
\end{aligned}$$

[2]Si ce n'est pas le cas, il suffit de le faire à l'aide de l'équation 10.4.

[3]E. Oja, «A Simplified Neuron Model as a Principal Component Analyser», *Journal of Mathematical Biology*, vol. 15, p. 239-245, 1982.

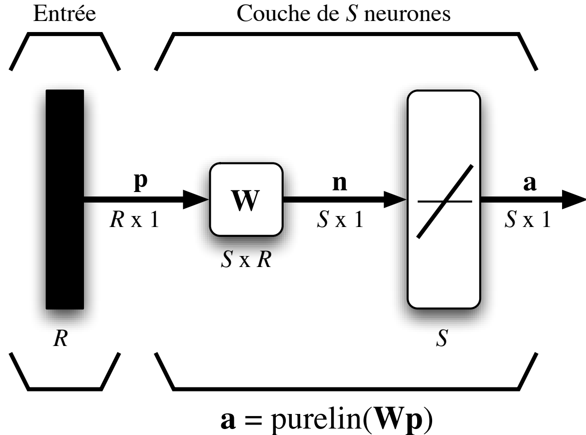


FIG. 10.2 – *Réseau permettant d'effectuer une analyse en S composantes principales.*

Après convergence, on obtient $E(\Delta\mathbf{w}) \longrightarrow 0$ et :

$$\mathbf{C}\mathbf{w} = (\mathbf{w}^T\mathbf{C}\mathbf{w})\mathbf{w} = \lambda\mathbf{w} \tag{10.23}$$

ce qui nous indique, par définition (voir section 3.2.3), que $\lambda = \mathbf{w}^T\mathbf{C}\mathbf{w}$ est une valeur propre de $\mathbf{C}$ et $\mathbf{w}$ le vecteur propre qui lui est associé. Finalement, en substituant l'équation 10.23 dans l'expression de λ, on obtient :

$$\lambda = \lambda\mathbf{w}^T\mathbf{w} = \lambda\,||\mathbf{w}||^2, \tag{10.24}$$

où $||\mathbf{w}||$ est la norme euclidienne de $\mathbf{w}$. Ceci implique que la norme de ce vecteur est unitaire, du moins une fois la convergence atteinte. On peut aussi montrer non seulement que cette convergence est assurée, mais que le vecteur propre obtenu sera associé à la plus grande valeur propre de $\mathbf{C}$, car seule celle-ci est stable.

10.3 Règle de Sanger

La règle de Oja nous permet de trouver la composante principale de nos stimuli. L'étape suivante consiste à trouver le moyen de déterminer les autres composantes en utilisant une couche de neurones linéaires comme à la figure 10.2. Un tel réseau permet de réduire la dimension de l'espace des entrées, de R dimensions à S dimensions ($S < R$). La matrice $\mathbf{W} = \{w_{i,j}\}$ représente les poids $w_{i,j}$ des connexions reliant les neurones i aux composantes j des stimuli d'entrée. La règle suivante, dite «règle de Sanger»[4], ou encore «Algorithme de Hebb généralisé», est une généralisation de la règle de Oja :

$$\Delta_i\mathbf{w} = \eta\left[a_i\,\mathbf{p} - a_i\sum_{k=1}^{i} a_k\,{}_k\mathbf{w}\right], \quad i = 1, \ldots, S. \tag{10.25}$$

[4]T. Sanger, «Optimal Unsupervised Learning in a Single Layer Linear Feedforward Neural Network», *Neural Networks*, vol. 12, p. 459-473, 1989.

En effet, dans le cas où $S = 1$, on retombe sur l'équation 10.18 :

$$\Delta_1\mathbf{w} = \eta \left[a_1\,\mathbf{p} - a_1 \sum_{k=1}^{1} a_1\,{}_1\mathbf{w}\right] = \eta\left[a_1\,\mathbf{p} - a_1^2\,{}_1\mathbf{w}\right] \tag{10.26}$$

Pour mieux visualiser l'équation 10.25, on peut la réécrire de la façon suivante :

$$\Delta_i\mathbf{w} = \eta\, a_i \left[\mathbf{p}' - a_i\,{}_i\mathbf{w}\right], \quad i = 1, \ldots, S, \tag{10.27}$$

où $\mathbf{p}'$ est une version modifiée du stimulus $\mathbf{p}$:

$$\mathbf{p}' = \mathbf{p} - \sum_{k=1}^{i-1} a_k\,{}_k\mathbf{w}. \tag{10.28}$$

qui dépend de l'indice i du neurone. Pour le premier neurone, $i = 1$, on obtient donc $\mathbf{p}' = \mathbf{p}$. Dans ce cas, la formule généralisée se réduit à la règle de Oja, et l'on sait que ce neurone recherchera la composante principale dans les stimuli.

Pour le deuxième neurone de la couche, $i = 2$, on obtient :

$$\mathbf{p}' = \mathbf{p} - a_1\,{}_1\mathbf{w}. \tag{10.29}$$

Sous l'hypothèse que le premier neurone a déjà convergé, on voit que l'on se trouve à retrancher de $\mathbf{p}$ une fraction a_1 de la composante principale des stimuli. Ceci ressemble étrangement à la procédure d'orthogonalisation de Gram-Schmidt (voir section 3.1.5). Le second neurone cherchera donc la composante principale des $\mathbf{p}'$, c'est-à-dire la seconde composante principale des $\mathbf{p}$. Et ainsi de suite pour les autres neurones qui chercheront dans un espace réduit des $i - 1$ composantes principales précédentes.

Dans la pratique, contrairement à ce qu'on laisse entendre ci-dessus, tous les neurones tendent à converger simultanément. Néanmoins, la convergence définitive d'un neurone i dépendant de celle du neurone $i - 1$, les poids ${}_i\mathbf{w}$ se stabiliseront dans l'ordre croissant de leur indice. Le temps total d'apprentissage sera cependant inférieur à ce qui serait nécessaire pour un apprentissage individuel des neurones.

En notation matricielle, la règle de Hebb généralisée permettant de faire une analyse en S composantes principales s'exprime de la façon suivante :

$$\Delta\mathbf{W}(t) = \eta\left(\mathbf{a}\,\mathbf{p}^T - \mathrm{LT}[\mathbf{a}\,\mathbf{a}^T]\,\mathbf{W}(t)\right), \tag{10.30}$$

où LT[.] désigne un opérateur matriciel qui met à zéro tous les éléments de son argument au dessus de la diagonale.

10.4 Apprentissage de Hebb supervisé

Nous terminons ce chapitre avec la version supervisée de l'apprentissage de Hebb où l'on remplace la sortie $\mathbf{a}$ du réseau par la sortie désirée $\mathbf{d}$:

$$\Delta\mathbf{W}(t) = \mathbf{d}(t)\,\mathbf{p}(t)^T \tag{10.31}$$

et où l'on fixe le taux d'apprentissage $\eta = 1$. En supposant que la matrice de poids $\mathbf{W}$ est initialisée à 0, on obtient :

$$\mathbf{W} = \mathbf{d}_1\mathbf{p}_1^T + \mathbf{d}_2\mathbf{p}_2^T + \cdots + \mathbf{d}_Q\mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{d}_q\mathbf{p}_q^T \quad (10.32)$$

après la présentation des Q paires $(\mathbf{p}_q, \mathbf{d}_q)$ d'apprentissage. En notation matricielle, on obtient :

$$\mathbf{W} = [\mathbf{d}_1\, \mathbf{d}_2 \cdots \mathbf{d}_Q] \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} = \mathbf{D}\,\mathbf{P}^T, \quad (10.33)$$

avec $\mathbf{D} = [\mathbf{d}_1\, \mathbf{d}_2 \cdots \mathbf{d}_Q]$ et $\mathbf{P} = [\mathbf{p}_1\, \mathbf{p}_2 \cdots \mathbf{p}_Q]$.

L'équation 10.33 permet de construire ce qu'on appelle une mémoire associative linéaire. Elle permet d'apprendre à mémoriser Q associations entre un stimulus $\mathbf{p}_q$ et une réponse $\mathbf{d}_q$. Supposons d'abord que tous les stimuli sont orthogonaux et normalisés (longueur unitaire). Alors, la réponse de cette mémoire à l'un d'entre eux, par exemple $\mathbf{p}_k$, sera :

$$\mathbf{a} = \mathbf{W}\,\mathbf{p}_k = \left(\sum_{q=1}^{Q} \mathbf{d}_q\mathbf{p}_q^T\right)\mathbf{p}_k = \sum_{q=1}^{Q} \mathbf{d}_q \left(\mathbf{p}_q^T\mathbf{p}_k\right). \quad (10.34)$$

Or, puisque tous les stimuli sont orthogonaux et normalisés, on a :

$$\left(\mathbf{p}_q^T\mathbf{p}_k\right) = \begin{cases} 1 & \text{si } q = k \\ 0 & \text{autrement} \end{cases}, \quad (10.35)$$

et $\mathbf{a} = \mathbf{d}_k$. La réponse pour un stimulus d'apprentissage est donc la sortie désirée qui lui est associée, à condition que les stimuli soient orthonormaux. Dans le cas où ils ne seraient plus orthogonaux (mais toujours normalisés), on obtiendrais :

$$\mathbf{a} = \mathbf{W}\,\mathbf{p}_k = \mathbf{d}_k + \sum_{q \neq k} \mathbf{d}_q \left(\mathbf{p}_q^T\mathbf{p}_k\right), \quad (10.36)$$

où la somme ci-dessus représente un terme d'erreur par rapport à la réponse désirée, engendré par la non orthogonalité des stimuli.

10.4.1 Règle de la matrice pseudo-inverse

De nouveau, nous constatons que la règle de Hebb seule n'est pas suffisante pour produire le résultat souhaité dans le cas général, à savoir :

$$\mathbf{W}\mathbf{p}_q = \mathbf{d}_q, \quad q = 1, \ldots, Q. \quad (10.37)$$

Une méthode pour y arriver consiste à d'abord définir un indice de performance F quadratique à minimiser, semblable à ce que nous avons utilisé pour le perceptron (voir chapitre 5) :

$$F(\mathbf{W}) = ||\mathbf{D} - \mathbf{W}\,\mathbf{P}||^2, \quad (10.38)$$

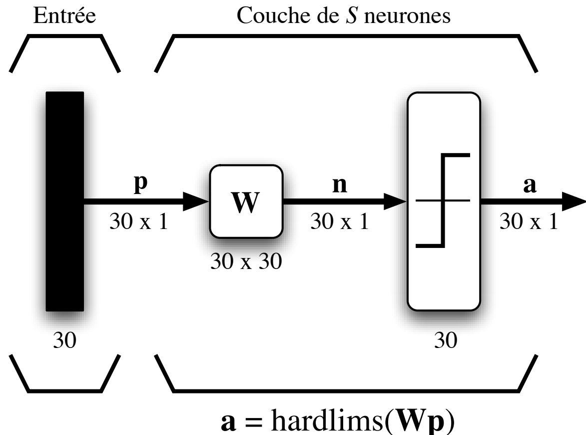


FIG. 10.3 – *Réseau auto-associatif pour la reconnaissance de chiffres.*

où $\mathbf{D} - \mathbf{W}\,\mathbf{P}$ est la forme matricielle de l'équation 10.37. Nous avons déjà démontré à la section précédente que $F(\mathbf{W}) = 0$ lorsque les stimuli d'apprentissage sont orthonormaux. Ensuite, pour minimiser l'équation 10.38 dans le cas général, il faudrait que $\mathbf{D} - \mathbf{W}\,\mathbf{P} \rightarrow 0$ et donc que $\mathbf{W} = \mathbf{D}\,\mathbf{P}^{-1}$. Or, la matrice $\mathbf{P}$ n'est généralement pas carrée (sauf si $P = Q$) et ne peut donc pas être inversée. Pour contourner cette difficulté, il faut faire appel à la matrice pseudo-inverse $\mathbf{P}^{+}$ de Moore-Penrose définie par :

$$\mathbf{P}^{+} = (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T \tag{10.39}$$

À condition que les stimuli de $\mathbf{P}$ soient indépendants, la matrice $(\mathbf{P}^T\mathbf{P})$ peut toujours être inversée et on obtient :

$$\begin{aligned} \mathbf{P}^{+}\mathbf{P} &= (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T\mathbf{P} \\ &= (\mathbf{P}^T\mathbf{P})^{-1}(\mathbf{P}^T\mathbf{P}) \\ &= \mathbf{I}. \end{aligned} \tag{10.40}$$

Ainsi, en fixant :

$$\mathbf{W} = \mathbf{D}\,\mathbf{P}^{+}, \tag{10.41}$$

on obtient la règle dite de la «matrice pseudo-inverse» pour construire notre mémoire associative linéaire. Contrairement à celle de l'équation 10.33, cette règle produira toujours la réponse désirée pour n'importe quel stimulus qui a servi à l'apprentissage. Pour un stimulus n'ayant pas servi à l'apprentissage, elle produira une réponse d'autant plus proche de celle d'un stimulus d'apprentissage que ce premier est proche de ce dernier.

10.4.2 Exemple d'auto-association

La figure 10.3 illustre un réseau auto-associatif permettant d'apprendre à reconnaître des chiffres représentés par une matrice binaire de 5×6 pixels (voir figure 10.4). Pour entraîner un tel réseau, il suffit de construire des stimuli et des réponses désirées en assemblant des vecteurs de bits à partir de la concaténation des lignes de pixels. Après entraînement avec l'équation 10.41,

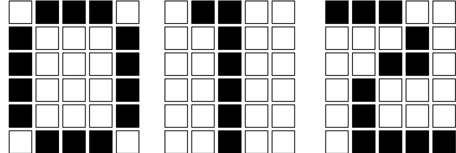


FIG. 10.4 – *Prototypes pour l'apprentissage auto-associatif des chiffres «0», «1» et «2».*

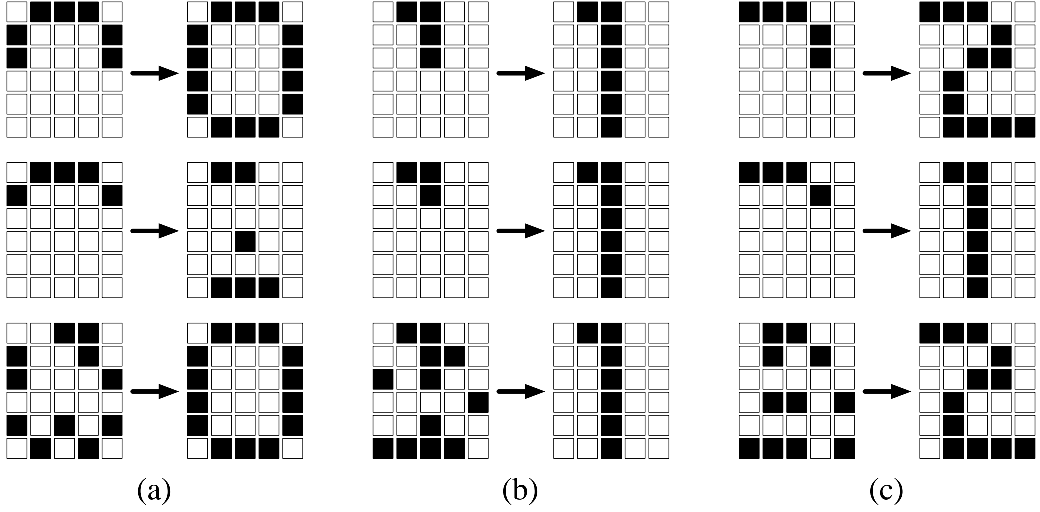


FIG. 10.5 – *Exemples de réponses du réseau auto-associatif de la figure 10.3 pour des stimuli dégradés ou bruités : (a) chiffres 0 ; (b) chiffres 1 ; et (c) chiffres 2.*

dans une certaine mesure, le réseau sera capable de produire en sortie des chiffres complets même si les stimuli d'entrée sont incomplets ou bruités. Par exemple, retirer la moitié inférieure des pixels des chiffres n'affecte aucunement la sortie du réseau, comme le montre la figure 10.5 (première ligne). Par contre, si l'on retire davantage de pixels (67% des pixels ; voir seconde ligne de la figure), alors seul le «1» est reconnu correctement. Dans le cas du «0», le réseau ne produit rien de cohérent alors que pour le «2», il produit un «1». Finalement, si l'on bruite les stimuli de nos trois chiffres (troisième ligne de la figure), on constate que le réseau n'a aucun problème à reconstruire les réponses désirées. Ceci illustre bien la robustesse d'une mémoire associative linéaire.

Chapitre 11

Réseau RBF

Dans ce chapitre, nous allons étudier les réseaux dits à «fonction de base radiale»[1]. Nous avons vu au chapitre 5 qu'il est possible d'approximer n'importe quelle fonction à l'aide d'un perceptron en intégrant une couche cachée de neurones sigmoïdes et une couche de sortie de neurones linéaires, comme à la figure 5.10 (page 50). Dans ce cas, on obtient les sorties suivantes pour le réseau :

$$\mathbf{a}^2 = \textbf{purelin}(\mathbf{W}^2\mathbf{a}^1 - \mathbf{b}^2) = \mathbf{W}^2\mathbf{a}^1 - \mathbf{b}^2 \tag{11.1}$$

Simplifions ce réseau au cas d'un seul neurone de sortie ($S^2 = 1$), posons $\mathbf{a}^2 = \hat{f}$ et annulons les biais de la couche de sortie pour simplifier ($\mathbf{b}^2 = \mathbf{0}$). On obtient alors :

$$\hat{f} = \mathbf{W}^2\mathbf{a}^1 = \sum_{j=1}^{S^1} w_{1,j}\, a_j^1, \tag{11.2}$$

où $\mathbf{a}^1 = [a_1^1\, a_2^1 \ldots a_{S^1}^1]^T$ correspond aux sorties des neurones de la couche cachée, et $w_{1,j}$ au poids qui relie le neurone caché j à la sortie unique de notre réseau. En interprétant les a_j^1 comme des bases (voir section 3.1.2), on remarque immédiatement que l'équation 11.2 permet d'approximer la fonction f à l'aide d'une combinaison linéaire de celles-ci. La problématique de l'apprentissage d'un tel perceptron consiste, premièrement, à trouver des bases adéquates pour effectuer l'approximation recherchée et, deuxièmement, à trouver les bon coefficients de la combinaison linéaire. Les bases engendrées par la couche cachée du réseau sont en fait des fonctions sigmoïdes que l'on positionne dans l'espace des entrées. Des travaux théoriques ont montré qu'un tel perceptron possède la propriété d'approximation universelle, c'est-à-dire qu'il peut approximer n'importe quelle fonction avec une précision arbitraire, à condition de disposer de suffisamment de neurones sur sa couche cachée.

Mais les neurones sigmoïdes ne sont pas les seuls à posséder cette capacité d'approximation universelle. De nombreuses autres fonctions la possède aussi, dont les fonctions radiales qu'utilisent les réseaux RBF. Il est important de se rappeler qu'un neurone sigmoïde agit partout dans son espace d'entrée. Il passe une frontière de décision linéaire qui traverse l'espace de bord en bord. En ce sens, lorsqu'un un stimulus est présenté à la couche cachée d'un perceptron multicouche, tous les neurones de cette dernière peuvent contribuer à produire la réponse du réseau. Ils

[1]En anglais : «Radial Basis Function» (RBF).

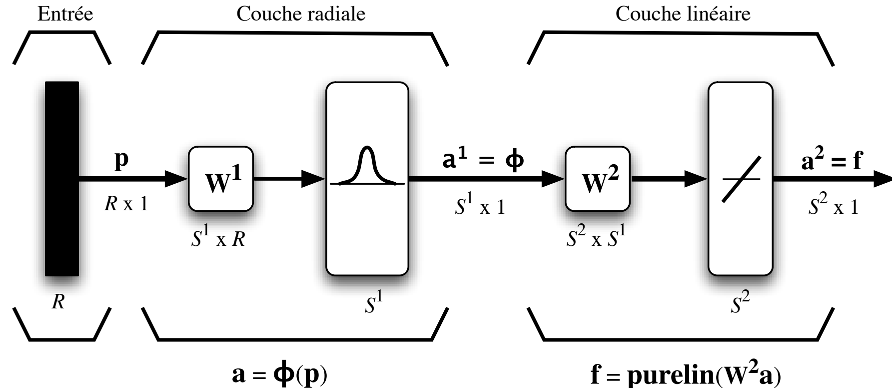


FIG. 11.1 – *Réseau RBF avec fonctions radiales gaussiennes.*

travaillent globalement. Ceci explique entre autres pourquoi on ne peut pas utiliser une perceptron multicouche pour faire de l'apprentissage incrémental, comme on peut le faire avec un réseau ART, par exemple.

Contrairement aux neurones sigmoïdes, les neurones «radiaux» travaillent localement dans l'espace des entrées. C'est la principale particularité des réseaux RBF. Plusieurs fonctions radiales peuvent être utilisées, mais la plus courante est une fonction ϕ de type gaussienne multivariée :

$$\phi(\mathbf{x}) = \exp\left(-\frac{(\mathbf{x} - {}_i\mathbf{w})^T \Sigma^{-1} (\mathbf{x} - {}_i\mathbf{w})}{2}\right), \tag{11.3}$$

où Σ désigne une matrice de covariance que l'on pose plus souvent qu'autrement égale à $\sigma^2\mathbf{I}$ et où ${}_i\mathbf{w}$ désigne la position (le centre) du neurone radial dans son espace d'entrée. Ainsi, la réponse à un stimulus $\mathbf{p}$ dépend d'un voisinage autour du centre, dont la variance σ^2 est égale dans toutes les directions, avec une décroissance exponentielle qui dépend du carré de la distance entre le stimulus et le centre :

$$\phi(\mathbf{x}) = \exp\left(-\frac{(\mathbf{x} - {}_i\mathbf{w})^T (\mathbf{x} - {}_i\mathbf{w})}{2\sigma^2}\right) = \exp\left(-\frac{||\mathbf{x} - {}_i\mathbf{w}||^2}{2\sigma^2}\right), \tag{11.4}$$

où $|| \cdot ||$ désigne la norme euclienne. Un réseau RBF peut alors approximer une fonction f avec l'expression suivante :

$$\hat{f}(\mathbf{p}) = \sum_{j=1}^{S^1} w_{1,j}^2 \, \phi_j(\mathbf{p}) \tag{11.5}$$

où $w_{1,j}^2$ est le poids de la deuxième couche qui relie le neurone j de la première couche au neurone de sortie 1, S^1 est le nombre de neurones sur la première couche et $\phi_j(\mathbf{p})$ est la fonction radiale associé au neurone j de cette dernière. Dans le cas vectoriel où l'on désire plusieurs sorties, on obtient l'expression suivante :

$$\hat{\mathbf{f}}(\mathbf{p}) = \mathbf{W}^2 \, \Phi(\mathbf{p}) \tag{11.6}$$

où $\Phi = [\phi_1 \, \phi_2, \ldots, \phi_{S^1}]^T$ dont le réseau équivalent est illustré à la figure 11.1.

11.1 Entraînement d'un réseau RBF

Pour entraîner le réseau RBF, on peut utiliser plusieurs stratégies. La première consiste à optimiser simultanément tous les paramètres du réseau, par exemple, en utilisant la rétropropagation des erreurs. Il s'agit de la position des centres des fonctions radiales, de leur variance et, finalement, des poids de la couche linéaire de sortie. Malheureusement, cette approche comportent certaines difficultés liées à la nature très différente de ces deux couches et de leur dynamique de convergence. La première couche, constituée de neurones non linéaires agissant localement dans l'espace des entrées, a plutôt tendance à converger lentement, alors que la seconde, avec ses neurones linéaires, converge généralement très rapidement. Ces dynamiques très différentes provoquent souvent une stagnation de l'apprentissage autour d'un minimum local parfois très éloigné de l'optimum global.

Ce qu'il importe de remarquer ici est que les deux couches du réseau RBF réalisent des fonctions distinctes. En ce sens, on peut très bien procéder à leur apprentissage en deux étapes également distinctes. La première consistant à estimer la position des centres des neurones radiaux puis à estimer leur variance, et la deuxième à estimer les poids de la couche linéaire.

Une première alternative pour le positionnement des centres consiste simplement à les distribuer uniformément dans l'espace des entrées. Cette solution comporte cependant des limitations évidentes, tout particulièrement lorsque cet espace d'entrée est de grande dimension.

Une seconde alternative est de fixer les centres sur certains stimuli $\mathbf{p}_k$ choisis aléatoirement parmi l'ensemble des données d'apprentissage. Dans ce cas, on peut imposer comme fonction radiale une gaussienne isotrope normalisée et centrées sur ${}_i\mathbf{w}^1 = \mathbf{p}_k$:

$$\phi_i(\mathbf{p}) = \exp\left(-\frac{S^1}{\delta^2_{\max}}||\mathbf{p} - {}_i\mathbf{w}^1||^2\right), \tag{11.7}$$

où S^1 correspond au nombre total de neurones radiaux, $\delta^2_{\max}$ au carré de la distance maximum entre leurs centres et ${}_i\mathbf{w}^1$ à la position de ces derniers. Ce choix de fonction radiale entraîne un écart type fixe de $\sigma = \delta_{\max}/\sqrt{2S^1}$ pour tous les neurones. Il permet de garantir des fonctions radiales ni trop pointues ni trop aplaties, ces deux extrêmes étant à éviter autant que possible. Il ne reste plus qu'à estimer les poids de la couche linéaire en utilisant, par exemple, la règle de la matrice pseudo-inverse (voir section 10.4.1) :

$$\mathbf{W}^2 = \mathbf{D}\,\mathbf{P}^+ \tag{11.8}$$

où $\mathbf{D} = [\mathbf{d}_1\,\mathbf{d}_2 \cdots \mathbf{d}_Q]$ est la matrice des réponses désirées pour le réseau, $\mathbf{P} = [\Phi_1\,\Phi_2 \cdots \Phi_Q]$ est la matrice des réponses de la couche radiale et $\mathbf{P}^+$ la matrice pseudo-inverse de $\mathbf{P}$.

Finalement, une troisième alternative consiste à positionner les centres des neurones radiaux à l'aide de l'une ou l'autre des méthodes d'apprentissage non supervisé étudiées aux chapitres 6 à 9 (nuées dynamiques, Kohonen, GNG ou Fuzzy ART). Une fois les centres positionnés, il ne reste plus qu'à estimer les σ_i en utilisant, par exemple, la partition (floue ou non floue ; voir chapitre 6) des stimuli engendrée par le processus compétitif des neurones, puis à estimer les poids de la couche linéaire à l'aide d'une méthode supervisée comme la règle LMS (voir section 5.2) ou, comme ci-dessus, celle de la matrice pseudo-inverse.

En conclusion, mentionnons que la principale difficulté des réseaux RBF concerne la question du nombre de neurones radiaux à utiliser pour une application donnée. A priori, il n'existe pas de méthode pour fixer leur nombre, et cette architecture souffre de façon particulièrement aiguë de ce qu'on appelle la «malédiction de la dimension»[2], à savoir l'augmentation exponentielle du nombre de neurones cachés requis en fonction de la dimension R de l'espace d'entrée. Lorsque R est grand, une façon d'atténuer ce problème consiste à remplacer les hyper-sphères qui résultent de l'imposition d'une variance fixe par des hyper-ellipses où la matrice de covariance n'est plus contrainte. On peut ainsi réduire le nombre de neurones à positionner au détriment du nombre de paramètres à estimer.

[2]En anglais : «curse of dimensionality».

Bibliographie

[1] M.T. Hagan, H.B. Demuth, M. Beale, «Neural Network Design», PWS Publishing Compagny, 1995.

[2] J.C. Principe, N.R. Euliano, W.C. Lefebvre, «Neural and Adaptive Systems : Fundamentals through Simulations», Wiley, 2000.

[3] Simon Haykin, «Neural Networks : A Comprehensive Foundation», IEEE Press, 1994.

[4] J.A. Freeman, D.M. Skapura, «Neural Networks : Algorithms, Applications, and Programming Techniques», Addison-Wesley, 1992.

[5] R.P. Lippmann, «An Introduction to Computing with Neural Nets», *IEEE ASSP Magazine*, pp. 4-22, avril 1987.

[6] R. Krishnapuram, J.M. Keller, «A Possibilistic Approach to Clustering», *IEEE Transactions on Fuzzy Systems*, vol. no. 2, p. 98-110, 1993.

[7] Bernd Fritzke, «A Growing Neural Gas Network Learns Topologies», *Advances in Neural Information Processing Systems 7*, G. Tesauro, D.S. Touretzky et T.K. Leen (éditeurs), MIT Press, Cambridge MA, 1995.

[8] G.A. Carpenter, S. Grossberg, N. Markuzon, J.H. Reynolds, D.B. Rosen, «Fuzzy ARTMAP : A neural network architecture for incremental supervised learning of analog multidimensional maps», *IEEE Transactions on Neural Networks*, vol. 3, no. 5, p. 698-713, 1992.