Résumé

Dans ce travail, nous nous intéresserons à l'étude des réseaux de neurones RBF (*Radial Basis Function Neural Networks*, RBFNN) en problèmes de classification. Ce type de réseaux de neurones est parmi les plus connus et les plus développés et plusieurs algorithmes d'entraînement de RBFNNs sont utilisés. Ces algorithmes peuvent être appelés globaux, car ils essayent tous de faire apprendre au réseau la solution complète du problème.

Nous nous baserons sur une approche diférente qui consiste à appliquer le vieux dicton "diviser pour mieux régner". Nous diviserons le problème initial en une série de sous-problèmes. Pour celà, nous avons découpé la tâche d'apprentissage en trois phases:

La première consiste à séparer l'ensemble des données d'entraînements en un certain nombre de sous-ensembles. Pour constituer ces sous-ensembles, nous utiliserons des algorithmes de groupement (*clustering*), basés sur diférents critères.

Lors de la deuxième phase, nous résoudrons les sous-problèmes provenant de la phase précédente, entraînant un réseau pour chaque groupe. Ces réseaux ne sont pas globaux mais plutôt locaux: ce sont des réseaux spécialisés dans la résolution du sous-problème qui leur a été assigné. Des stratégies classiques d'entraînement sont utilisées pour ces réseaux locaux. La troisième phase est faite du rassemblement des solutions aux sous-problèmes en une solution globale. Nous utiliserons plusieurs techniques pour réunir les solutions locales.

Le but de ce travail est de comparer les diférentes méthodes d'entraînement resultants, avaluant l'influence de la complexité des diférents algorithmes de groupement utilisés et les techniques pour rassembler les solutions locales.

Ce travail a été financé par le projet espagnol du CICYT número TAP99-0747.

****

Abstract

The aim of this work is to study the effect of locality in classification tasks with radial basis function neural networks (RBFNN). The networks are trained in a three stage process. Firstly, the data are decomposed in their natural clusters, using clustering algorithms of different complexity. Secondly, a local RBFNN is fit to each cluster. These RBFNNs are local in the sense that they are modeling only a part of the problem, as given by the previous stage. Any RBFNN training algorithm can be used here. Thirdly, the local networks are fused together. We propose several simple techniques to do so. The results are analyzed in light of the following aspects: overall feasibility of the idea, influence of clustering algorithm complexity, influence of specific training algorithms, and selection of the fusing method.

This work is supported by the Spanish CICYT grant TAP99-0747.

Table des matières

1 Introduction **3**

2 Groupement supervisé et non supervisé **5**

2.1 Introduction 5

2.2 Algorithmes non supervisés 6

2.2.1 L'algorithme *K-Means* 6

2.2.2 L'algorithme *Expectation* - *Maximization* (EM) 7

2.2.3 *Maximum Certainty Data Partitioning* 11

2.3 Algorithmes supervisés 14

2.3.1 L'algorithme *Learning Vector Quantization* (LVQ) 14

3 Les réseaux de neurones de fonctions à bases radiales **17**

3.1 Les réseaux de neurones 17

3.2 Les réseaux de fonction à bases radiales (RBF) 18

3.2.1 Introduction 18

3.2.2 Base Mathématique 19

3.2.3 Réseaux de fonction à base radiale 20

3.3 Entraînement des RBFs 21

3.3.1 Entraînement en 2 étapes 21

3.3.2 Entraînement supervisé 24

3.3.3 Estimation de l'ordre du modèle. 26

4 Entraînement local de RBFNNs **27**

4.1 Phase 1 : Groupement des données 28

4.2 Phase 2 : entraînement de RBFNNs locaux 29

4.3 Phase 3 : Arrangement des RBFNNs locaux en un RBFNN global 30

4.3.1 Moyenne des RBFNNs locaux 30

4.3.2 Fusions des RBFNNs locaux 30

4.3.3 Conseil de RBFNNs locaux 31

5 Étude expérimentale **33**

5.1 Choix des données 33

5.2 Plan d'expérimentation 35

5.2.1 Déroulement général 36
5.2.2 Expérimentations sur Iris et Wine 37
5.2.3 Expérimentations sur Vowel 37
5.3 Résultats 38
5.3.1 Iris 38
5.3.2 Wine 40
5.3.3 Vowel 42
5.4 Discussion des résultats 45
5.4.1 Remarques générales 45
5.4.2 Groupement lors de la première phase 45

6 Conclusions **50**
6.1 Conclusions scientifiques 50
6.1.1 Avantages et inconvénients de l'entraînement en trois phases 50
6.1.2 Résultats 51
6.2 Travail futur 51
6.3 Conclusions personnelles 51
6.4 Coût 52

Chapitre 1

Introduction

Le modèle des réseaux de neurones artificielles est, à l'origine, inspiré de la façon dont le cerveau traite l'information. L'idée sous-jacente est d'essayer de résoudre des problèmes d'une manière semblable à celle utilisé par un cerveau, c'est-à-dire en se servant en parallèle d'un grand nombre d'unités de traitement relativement simples, les neurones. Parmi les avantages de cette façon de faire, nous avons, entre autres, sa capacité d'apprentissage. En plus de la comparaison avec le vivant, un modèle mathématique complet permet de les décrire. Cette formalisation présente les réseaux de neurones sous une forme complètement différente, celle des méthodes inductives d'approximations.

Dans ce travail, nous nous intéresserons à une catégorie particulière de réseaux de neurones connue sous le nom de réseaux de neurones RBF (Radial Basis Function Neural Networks, RBFNNs). Ce type de réseaux de neurones est parmi les plus connus et les plus développés. Pour qu'un RBFNN puisse résoudre le problème auquel il est confronté, il faut commencer par lui faire apprendre. L'apprentissage se fait de manière supervisée, c'est-à-dire en présentant au réseau une partie du problème et les solutions correspondantes. Actuellement, plusieurs algorithmes d'entraînement de RBFNNs sont utilisés. Ces algorithmes peuvent être appelés globaux, car ils essayent tous de faire apprendre au réseau la solution complète du problème.

Nous nous baserons sur une approche différente qui consiste à appliquer le vieux dicton “diviser pour mieux régner”.

Nous diviserons donc le problème initial en une série de sous-problèmes. Pour cela, nous avons découpé la tâche d'apprentissage en trois phases. La première consiste à séparer l'ensemble des données d'entraînements en un certain nombre de sous-ensembles ; c'est la phase de division du problème. Pour constituer ces sous-ensembles, nous utiliserons des algorithmes de groupement (clustering). Nous appliquerons aux données plusieurs types de groupement, basés sur différents critères. Citons l'Expectation - Maximization et le K-Means qui sont parmi les algorithmes les plus connus. Ces algorithmes sont décrits en détailles dans le chapitre 2.

Lors de la deuxième phase, nous résoudrons les sous-problèmes provenant de la phase précédente. Cette résolution sera menée à bien à l'aide des RBFNNs. Nous entraînerons un réseau pour chaque groupe. Ces réseaux ne sont donc pas globaux mais plutôt locaux. Ce sont des réseaux spécialisés dans la résolution du sous-problème qui leur a été assigné. Les stratégies classiques d'entraînement seront utilisées pour ces réseaux locaux. Nous essayerons plusieurs algorithmes d'entraînements de RBFNNs,

dans le but de pouvoir comparer les résultats. Notre choix s'est porté sur trois algorithmes parmi les plus connus, à savoir l'algorithme de descente du gradient (GD), le *orthogonal least squares* (OLS), et le *learning vector quantization* (LVQ). Les RBFNNs ainsi que les différentes manières de les entraîner sont expliquées dans le chapitre 3.

Enfin, la troisième phase est faite du rassemblement des solutions aux sous-problèmes en une solution globale. Une fois de plus, nous utiliserons plusieurs techniques pour réunir les solutions locales. Parmi ces techniques, certaines sont simples comme celle qui consiste à faire la moyenne des solutions locales, d'autres sont plus sophistiquées comme celles que nous avons appelées fusion de RBFNNs locaux ou conseil de RBFNNs locaux. La manière de rassembler ces réseaux ainsi que la descriptions détaillée des trois phases que nous venons de présenter font l'objet du chapitre 4.

Le but de ce travaille est de comparer les différentes méthodes d'entraînement entre elles. Nous ferons des comparaisons entre les méthodes d'entraînement en trois phases, suivant les algorithmes utilisés. Nous comparerons aussi les résultats obtenu par la division du problème à ceux provenant d'un entraînement global. L'étude expérimentale que nous avons faite est basée sur des jeux de données très connu dans le domaine des réseaux de neurones. Le chapitre 5 présente son déroulement ainsi qu'une discussion approfondie des résultats.

Finalement, nous terminerons ce document en présentant les conclusions qui peuvent être tirées de ce travail. Nous relèverons les points positifs et négatifs de notre entraînement local et présenterons quelques idées qui permettraient d'améliorer notre entraînement en trois phases.

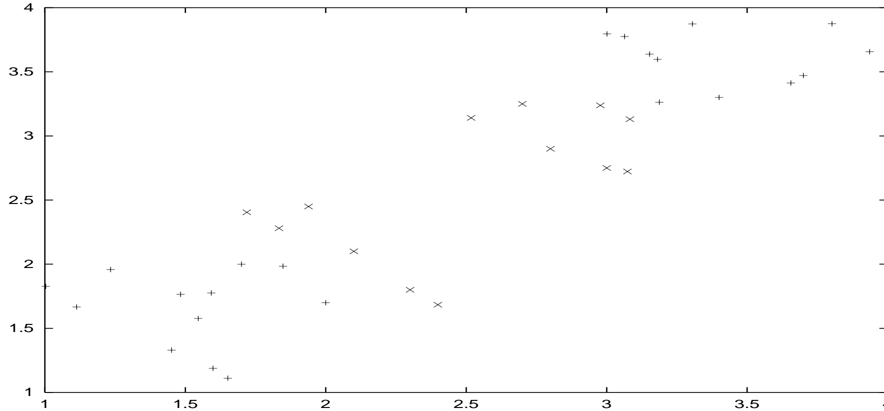
Chapitre 2

Groupement supervisé et non supervisé

2.1 Introduction

Dans ce chapitre, nous nous intéresserons de près au problème du regroupement des données. Lorsque nous parlons de groupement de données, nous nous référons au fait de rassembler les données similaires dans un même groupe. La notion de similarité est donc capitale au moment de s'intéresser aux problèmes de groupement. Pour commencer, il faut déterminer dans quel espace l'on va chercher la similarité. Il est possible de chercher à grouper les données semblables dans l'espace des inputs sans se préoccuper des variable de l'espace des outputs. C'est ce que l'on appelle un regroupement non supervisée ou, en anglais, clustering (par commodité, nous utiliserons cet anglicisme pour désigné le groupement). L'autre façon de faire, la classification supervisée, prend en compte les variables de l'espace des outputs pour regrouper les données selon leur similarité dans l'espace complet des inputs et des outputs.

Dans certains cas, les données de l'espace des outputs manquent et il est donc impossible de faire de la classification supervisée. Il est claire qu'une classification supervisée, qui a à disposition plus d'informations sur les données, présentera généralement de meilleurs résultats que son équivalant non supervisé. En effet, les groupes que peuvent former les données dans l'espace des inputs ne correspondent pas forcément à une manière intéressante de regrouper les données une fois connue la classification.



Par exemple, dans le cas de la figure, un algorithme de clustering non supervisé trouverai deux groupes alors qu'un algorithme supervisé en trouverai trois ou quatre ce qui, dans ce cas là, est nettement plus intéressant.

Ils existent un grand nombre d'algorithme de groupement. Nous avons fait un choix qui n'est evidemment pas exaustif.

2.2 Algorithmes non supervisés

Dans cette section, nous verrons un certain nombre d'algorithmes de groupement non-supervisé. Nous verrons pour commencer le K-Means qui se base sur une idée intuitivement facile à comprendre. Le k-Means est un cas particulier de l'algorithme connu sous le nom de *Expectation-Maximization* que nous présenterons aussi. Pour finir, nous nous intéresserons à un algorithme qui se base sur un critère tout à fait différent des deux précédents, à savoir le *Maximum Certainty Data Partitioning*

2.2.1 L'algorithme *K-Means*

L'algorithme connu sous le nom de K-Means est un algorithme de clustering non supervisé basé sur une procédure de réestimation itérative relativement simple. Supposons que nous voulons diviser un ensemble de N points en K groupes, chaque groupe étant représenté par un vecteur $\boldsymbol{\mu}_j$ avec $j = 1, ...K$. Pour cela, l'algorithme va ajuster les $\boldsymbol{\mu}_j$ de façon à minimiser la somme des carrés des distances entre les points et le centre du groupe auquel ils appartiennent. L'expression a minimiser est donc :

$$J = \sum_{j=1}^{K} \sum_{n \in S_j} \| \mathbf{x}_n - \boldsymbol{\mu}_j \|^2 \tag{2.1}$$

avec $\boldsymbol{\mu}_j$ la moyenne des points appartenant à l'ensemble S_j . Si le nombre N_j représente la cardinalité de l'ensemble S_j , alors

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{n \in S_j} \mathbf{x}_n . \tag{2.2}$$

Il existe principalement deux versions de cette algorithme et, pour chacune de ces versions, de nombreuses variations. Les deux versions sont la version *batch* et la version *online*.

L'initialisation de la version *batch* se fait en assignant, au hasard, chaque point à un des K ensembles. Puis, pour chaque ensemble, le vecteur $\boldsymbol{\mu}_j$ est calculé selon (2.2). Le pas suivant consiste à trouver, pour chaque point $\mathbf{x}_i$, le vecteur $\boldsymbol{\mu}_j$ dont il est le plus proche (selon la distance euclidienne) et a réassigner ce point à l'ensemble S_j correspondant. On commence l'itération suivante en recalculant les $\boldsymbol{\mu}_j$ selon (2.2). Sous forme algorithmique, la version *batch* du K-Means peut s'énoncer de la manière suivante :

1. **Initialisation.** Pour tout les $\mathbf{x}_n$, $n = 1, ..., N$, assigner $\mathbf{x}_n$ de manière aléatoire à un des ensemble S_j, $j = 1, ..., K$ de façon à ce qu'il y ait au minimum un $\mathbf{x}_n$ par ensemble S_j.
2. **Tant** qu'il y a des changement dans l'assignation des $\mathbf{x}_n$, **RÉPÉTER :**
   (a) **Calcul des $\boldsymbol{\mu}_j$.**
   Pour chaque S_j, $j = 1, ..., K$ calculer $\boldsymbol{\mu}_j$ selon (2.2).
   (b) **Assignation des $\mathbf{x}_n$.**
   Pour chaque $\mathbf{x}_n$, $n = 1, ..., N$, assigner $\mathbf{x}_n$ à l'ensemble S_j dont le centre $\boldsymbol{\mu}_j$ minimise la quantité $\| \mathbf{x}_n - \boldsymbol{\mu}_j \|$.

La version *online* est un peu différente bien que basée sur le même principe. On commence par une initialisation aléatoire des K vecteurs $\boldsymbol{\mu}_j$. Ensuite, pour chaque vecteur $\mathbf{x}_n$, $n = 1, ..., K$ on cherche le vecteur $\boldsymbol{\mu}_j$ le plus proche et on déplace ce dernier en direction de $\mathbf{x}_n$; on a donc

$$\Delta\mu_j = \eta(\mathbf{x}_n - \boldsymbol{\mu}_j) \tag{2.3}$$

ou η est le taux d'apprentissage. Sous forme algorithmique, la version *online* du K-Means s'énonce donc :

1. **Initialisation.** Assigner des valeurs aléatoires à chaque $\boldsymbol{\mu}_j$, $j = 1, ..., K$.
2. **Tant** qu'il y a des changement notable des $\boldsymbol{\mu}_j$ entre deux itérations **RÉPÉTER :**
   Pour chaque $\mathbf{x}_n$, $n = 1, ..., N$:
   (a) trouver le vecteur $\boldsymbol{\mu}_j$ qui minimise la distance euclidienne $\| \mathbf{x}_n - \boldsymbol{\mu}_j \|$.
   (b) mettre à jour le vecteur $\boldsymbol{\mu}_j$ selon (2.3).

Comme nous venons de le voir, la version *online* de l'algorithme implique le choix d'un taux d'apprentissage η. Le choix d'une valeur appropriée pour ce paramètre peut parfois s'avérer difficile. De plus, il faut aussi définir la signification de "changement notable". Une variante de l'algorithme avec un taux d'apprentissage ajusté dynamiquement a été développée pour éviter, entre autre, ce problème. Pour plus de détails sur la version *online* de l'algorithme et ses améliorations, voir [3].

Dans le cadre de ce travail, la version *batch* de l'algorithme, généralement considérée comme plus performante a été implémentée. La version *online* s'utilise principalement lorsque l'ensemble des données n'est pas disponible au moment du traitement. C'est le cas par exemple des applications comme la reconnaissance vocale.

2.2.2 L'algorithme *Expectation - Maximization* (EM)

L'algorithme *Expectation - Maximization*, connu dans la littérature sous le nom de EM, a été présenté pour la première fois sous ce nom par Dempster, Laird et Rubin dans un article paru en

1977 dans le journal de la *Royal Statistical Society* [7]. Cet algorithme est relativement intuitif et a été utilisé sous de multiples formes particulières, dont la première connue est de 1886, avant d'être présenté sous sa forme générale par Dempster, Laird et Rubin. Une des raisons de la grande popularité de cet algorithme, est sa capacité à traiter des problèmes dans lesquels il manque une partie des valeurs de certains attributs, ce qui est très courant en pratique. Le but de ce travail n'étant pas l'étude approfondie de cet algorithme (déjà très bien faite dans [7]), nous nous contenterons d'une courte présentation de l'algorithme en général et d'une étude plus détaillée de l'application que nous en ferons. L'application que nous verrons consiste à essayer de trouver les centres et matrices de covariances de K gaussiennes pour qu'elles représentent au mieux la fonction de densité des données.

L'algorithme EM : généralités

L'algorithme EM se base sur le concept de "vraisemblance maximum" (*maximum likelihood*). Comme son nom l'indique, l'algorithme *Expectation - Maximization* est composé de deux étapes. Après une initialisation des paramètres inconnus, la première phase, que nous appellerons E-Step, consiste à estimer les données manquantes à l'aide des données présentes et des valeurs actuelles des paramètres cherchés. La seconde étape, le M-Step, consiste à réévaluer les paramètres selon le principe de vraisemblance maximum. En répétant ces deux étapes, il est prouvé que l'algorithme converge vers un extremum local de la fonction de vraisemblance.

Formellement, supposons que nous ayons une fonction de densité (que nous appellerons par la suite *pdf* de l'anglais *probability density function*) $p(\mathbf{y})$ dépendant d'un certain nombre de paramètres. Posons $\mathbf{\Psi} = (\psi_1, ..., \psi_d)^T$, $\mathbf{\Psi} \in \Omega$, le vecteur contenant les paramètres inconnus de la fonction de densité choisie (centres, matrices de covariances et coefficient d'un mélange de gaussiennes dans le cas particulier qui sera développé par la suite). Écrivons $p(\mathbf{y})$ sous la forme de $p(\mathbf{y}|\mathbf{\Psi})$ pour rendre la dépendance aux paramètres $\mathbf{\Psi}$ explicite. Le vecteur $\mathbf{\Psi}$ doit être estimé par la méthode de la vraisemblance maximum. Étant donné l'ensemble de N vecteurs de données $\mathbf{Y} = \{\mathbf{y}_1, ..., \mathbf{y}_N\}$ il est possible de calculer la fonction de densité conjointe des données qui est

$$p(\mathbf{Y}|\mathbf{\Psi}) = \prod_{n=1}^{N} p(\mathbf{y}_n|\mathbf{\Psi}). \tag{2.4}$$

La fonction de vraisemblance pour $\mathbf{\Psi}$ est donnée par $L(\mathbf{\Psi}) = p(\mathbf{Y}|\mathbf{\Psi})$, c'est-à-dire qu'elle peut être vue comme une fonction de $\mathbf{\Psi}$ pour un W donné. Un estimateur $\widehat{\mathbf{\Psi}}$ de $\mathbf{\Psi}$ peut être obtenu en résolvant l'équation de vraisemblance $\partial L(\mathbf{\Psi})/\partial\mathbf{\Psi} = 0$ ou, de manière équivalente qui permet de simplifier les calculs

$$\partial \log L(\mathbf{\Psi})/\partial\mathbf{\Psi} = 0. \tag{2.5}$$

L'algorithme EM est fait pour résoudre des problèmes avec des données incomplètes. La notion de donnée incomplète inclus évidemment le cas oú il manque des données (*missing data*) mais aussi le cas oú les données seraient générées par une expérience hypothétique ; par exemple, dans le cas d'un mélange de gaussiennes que nous verrons plus loin, chaque donnée "incomplète" $\mathbf{y}_n$ serait complétée par un vecteur $\mathbf{z}_n$ indiquant par quelle gaussienne a été généré le vecteur $\mathbf{y}_n$. Plus généralement, le vecteur $\mathbf{z}_n$ représente les données inobservables ou manquantes. Nous appellerons $\mathbf{x}_n$ le vecteur de donnée "complétée" composé de $\mathbf{y}_n$ et de $\mathbf{z}_n$ et $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ l'ensemble des données complétées.

Posons $p_c(\mathbf{X}|\Psi)$ la *pdf* de $\mathbf{X}$ connaissant Ψ. Alors, le logarithme de la vraisemblance maximum de Ψ si $\mathbf{x}$ était observable au complet est

$$\log L_c(\Psi) = \log p_c(\mathbf{X}; \Psi). \tag{2.6}$$

L'approche prise par l'algorithme EM est de résoudre l'équation (2.5) indirectement en agissant de manière itérative sur la fonction de vraisemblance maximum de Ψ sur les données complétées, cette-à-dire sur $\log L_c(\Psi)$. Comme la vraisemblance sur les données complétées n'est pas observable elle est remplacée par son estimation conditionnelle sur $\mathbf{z_n}$ utilisant le Ψ actuel.

Plus précisément, si $\Psi^{(0)}$ est la valeur initiale de Ψ, alors le E-Step (étape d'estimation) de la première itération de l'algorithme est fait du calcul de l'espérance de (2.6), connaissant les données observables

$$Q(\Psi; \Psi^{(0)}) = E_{\Psi^{(0)}}\{\log L_c(\Psi)|\mathbf{y}\}. \tag{2.7}$$

La seconde étape de l'algorithme (étape de maximisation), le M-Step consiste à trouver le vecteur $\Psi^{(1)}$ qui maximise (2.7) :

$$Q(\Psi^{(1)}; \Psi^{(0)}) \geq Q(\Psi; \Psi^{(0)}) \ \forall \Psi \in \Omega. \tag{2.8}$$

Les E-Step et M-Step sont ensuite répétés. A la (i+1)ème itération, les étapes E et M sont les suivantes :

E-Step. Calculer $Q(\Psi; \Psi^{(i)})$ tel que

$$Q(\Psi; \Psi^{(i)}) = E_{\Psi^{(i)}}\{\log L_c(\Psi)|\mathbf{y}\} \tag{2.9}$$

M-Step. Choisir le $\Psi^{(i+1)} \in \Omega$ qui maximise $Q(\Psi^{(i+1)}; \Psi^{(i)})$ c'est-à-dire,

$$\forall \Psi \in \Omega, \quad Q(\Psi^{(i+1)}; \Psi^{(i)}) \geq Q(\Psi; \Psi^{(i)}) \tag{2.10}$$

L'alternance E-Step / M-Step est répétée jusqu'à ce que l'amélioration de la vraisemblance sur les données incomplètes soit inférieur à un ϵ arbitrairement petit :

$$L(\Psi^{(i+1)}) - L(\Psi^{(i)}) < \epsilon. \tag{2.11}$$

La convergence est obtenue après un nombre fini d'itérations étant donné que Dempster, Laird et Rubin ont démontré que

$$L(\Psi^{(i+1)}) \geq L(\Psi^{(i)}) \ i = 0, 1, 2, \ldots. \tag{2.12}$$

Pour terminer ce survol théorique, signalons qu'il est un peu abusif d'appeler algorithme la démarche décrite par EM. En effet, EM décrit une procédure très générale pour chercher les paramètres inconnus d'un problème. Cette procédure se décompose en deux étapes dont le but est claire. Par contre, EM ne décrit pas les pas successifs à effectuer pour concrétiser ces deux étapes. La réalisation des deux étapes peut être très différente selon le type de problème et peut parfois soulever d'importantes difficultés analytiques. Pour cette raison, nous allons décrire en détail l'application au clustering à l'aide d'une modélisation gaussienne de la distribution de probabilité.

Application au clustering

Comme nous venons de le voir, l'algorithme EM est très général et peu s'appliquer à de nombreux problèmes. Dans le cadre de ce travail, il a été utilisé dans le but de grouper les données semblables de manière non supervisée (c'est-à-dire en ne prenant compte que la similarité dans l'espace des inputs). Pour cela, nous allons supposer que les données $\mathbf{y}$ ont été générées par un mélange de gaussiennes (*mixture modeling*). Chaque groupe sera donc représenté par une gaussienne déterminée par son centre $\boldsymbol{\mu}$ et sa matrice de covariances $\mathbf{\Sigma}$. Pour séparer les données d'entrée de dimension J en K groupes, il faut donc trouver les K vecteur $\boldsymbol{\mu}_k$ de dimension J représentant le centre des gaussiennes et les K matrices de covariances $\mathbf{\Sigma}_k$ de dimension $J \times J$. Il faut aussi connaître les coefficients $P(k)$ du mixture model :

$$p(\mathbf{y};\Psi) = \sum_k^K p(\mathbf{y}|k;\Psi)P(k) \tag{2.13}$$

Les K couples $(\boldsymbol{\mu}_k, \mathbf{\Sigma}_k)$ et les K coefficients du mélange sont donc les paramètres inconnus représenté par Ψ dans la présentation générale de l'algorithme ci-dessus. Les données incomplètes (observées) sont les points, c'est-à-dire des vecteurs $\mathbf{y}_i$ de dimensions J. Les données complétées sont représentées par des vecteurs $\mathbf{x}_i$ de dimension $J + K$. Les $\mathbf{x}_i$ sont composés du vecteur $\mathbf{y}_i$ et du vecteur $\mathbf{z}_i$ de dimension K. La k-ème composante du vecteur $\mathbf{z}_i$ est égale à 1 si le point observé $\mathbf{y}_i$ a été généré par la k-ème gaussienne ; il est égale à zéro si non. Le vecteur $\mathbf{z}_i$ est donc conceptuellement un vecteur binaire dont une seule composante est égale à 1 et toute les autres à 0. Nous verrons plus loin que, lors de l'application de l'algorithme, la k-ème composantes du vecteur $\mathbf{z}_i$ représente la probabilité que le point observé $\mathbf{y}_i$ ait été généré par la gaussienne de centre $\boldsymbol{\mu}_k$ et de matrices de covariances $\mathbf{\Sigma}_k$. Ce n'est donc plus un vecteur binaire mais la signification reste la même.

On admet donc que les données observée $\mathbf{y}_i$ ont été générées par une mixture de gaussiennes. La fonction de densité d'une loi normale gaussienne est donnée par

$$\phi(\mathbf{y}) = (2\pi)^{-\frac{d}{2}}\Sigma^{-\frac{1}{2}}\exp\{-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{y}-\boldsymbol{\mu})\}. \tag{2.14}$$

avec d la dimension de l'espace des inputs. Posons que k représente la gaussienne de centre $\boldsymbol{\mu}_k$ et de matrice de covariances $\mathbf{\Sigma}_k$. De (2.13), on peut déduire que le logarithme de la vraisemblance maximum est

$$\ln L(\Psi) = \sum_{i=1}^{I}\ln\sum_{k=1}^{K}P(\mathbf{y}|k)P(C_k). \tag{2.15}$$

Il faut donc trouver le Ψ qui maximise $L(\Psi)$. Pour cela, nous passons du logarithme de la vraisemblance sur le jeu de données considéré comme incomplet au logarithme de la vraisemblance sur le jeu de données complété par les z_{ik}. Ce qui donne

$$\ln L_c(\Psi) = \sum_{i=1}^{I}\sum_{k=1}^{K} z_{ik}\ln[P(\mathbf{y}|\mathbf{z}_i)P(\mathbf{z}_i)]. \tag{2.16}$$

En fait, si on admet que les z_{ik} ne sont plus des valeurs binaires mais représentent la probabilité

que le point $\mathbf{y}_i$ ait été généré par la gaussienne k. On a

$$z_{ik} = \frac{P(\mathbf{y} = \mathbf{y}_i | \boldsymbol{\mu} = \boldsymbol{\mu}_k, \boldsymbol{\Sigma} = \boldsymbol{\Sigma}_k) P(k)}{\sum_{l=1}^{K} P(l) P(\mathbf{y} = \mathbf{y}_i | \boldsymbol{\mu} = \boldsymbol{\mu}_l, \boldsymbol{\Sigma} = \boldsymbol{\Sigma}_l)} \tag{2.17}$$

$$= \frac{|\boldsymbol{\Sigma}_k|^{-\frac{1}{2}} \exp\{-\frac{1}{2}(\mathbf{y}_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{y}_i - \boldsymbol{\mu}_k)\} P(k)}{\sum_{l=1}^{K} P(l) |\boldsymbol{\Sigma}_l|^{-\frac{1}{2}} \exp\{-\frac{1}{2}(\mathbf{y}_i - \boldsymbol{\mu}_l)^T \boldsymbol{\Sigma}_l^{-1} (\mathbf{y}_i - \boldsymbol{\mu}_l)\}} \tag{2.18}$$

avec $\sum_{k=1}^{K} z_{ik} = 1$. On peut donc énoncer les deux étapes du (n+1)ème pas de l'algorithme comme suit :

E-Step : Calculer $E^{(n+1)}[L_c(\Psi^{(n)})]$ selon (2.16).

M-Step : Réestimé les $\boldsymbol{\mu}_k^{(n+1)}$ et $\boldsymbol{\Sigma}_k^{(n+1)}$: Pour $1 \leq k \leq K$,

$$\boldsymbol{\mu}_k^{(n+1)} = \frac{\sum_i^I z_{ik} \mathbf{y}_i}{\sum_i^I z_{ik}} \tag{2.19}$$

$$\boldsymbol{\Sigma}_k^{(n+1)} = \frac{\sum_i^I z_{ik} (\mathbf{y}_i - \boldsymbol{\mu}_k^{(n+1)})(\mathbf{y}_i - \boldsymbol{\mu}_k^{(n+1)})^T}{\sum_i^I z_{ik}} \tag{2.20}$$

On arrête l'algorithme lorsque (2.11) est vérifié. Une fois connu les différents paramètres composants $\boldsymbol{\Psi}$, on a un "soft-clustering", c'est-à-dire pour chaque $\mathbf{y}_i$ la probabilité qu'il appartienne au cluster C_k. Il est intéressant d'avoir un clustering qui soit "soft" car il permet d'avoir une mesure de fiabilité du clustering. Si une donnée $\mathbf{y}_i$ a une probabilité très légèrement supérieur à $\frac{1}{2}$ d'appartenir au cluster C_1 et qu'il n'y a que deux clusters, alors la classification de cette donnée ne peut pas être considérée comme fiable.

C'est une version encore un peu plus particulière de l'algorithme EM qui a été programmée dans le cadre de ce projet. En effet, dans l'implémentation qui a été faite, les variables ont été considérées indépendantes entre elles. Ceci implique que les matrices de covariances $\boldsymbol{\Sigma}_k$ sont diagonales, c'est à dire que, au lieu d'une matrice de covariance nous avons un vecteur de variances. Cette simplification est justifiée par le fait qu'il y a rarement assez de données pour pouvoir ajuster correctement le grand nombre de paramètres qu'implique une matrice de covariances complète par gaussienne. Pour le développement mathématique concernant les valeurs manquantes, pour attributs discrets ou continus, se référer à [4]. Pour un développement complet de la théorie de l'algorithme EM, voir [7].

Il est intéressant de constater que la version *batch* de l'algorithme du K-Means présenté dans la section précédente est, en fait, un cas particulier de l'algorithme EM. En effet, si l'on pose que toutes les gaussiennes ont la même variance unique σ^2 dans toutes les dimensions, alors l'algorithme EM présenté ci-dessus devient un K-Means *batch*.

2.2.3 *Maximum Certainty Data Partitioning*

Nous allons présenter l'algorithme appelé *Maximum Certainty Data Partitioning* (MCDP) qui a été développé en 1999 par une équipe de l'Imperial College of Science de Londres [9]. Cet algorithme est intéressant car il se base sur un critère complètement différent de celui utilisé par les deux algorithmes

vu précédemment. En effet, l'algorithme EM et son cas particulier qu'est le K-Means *batch* utilisent comme critère de similarité la distance euclidienne, ce qui n'est pas le cas du MCDP aussi appelé *Minimum Entropy Data Partitioning*. Comme le suggère ce deuxième nom, cet algorithme tend à minimiser l'entropie de Shannon des partitions sur le jeu de données. Comme cet algorithme est récent et peu connu, nous nous donnerons la peine de l'expliquer en détail.

Théorie

Considérons un ensemble de K partitions. La fonction de densité de $\mathbf{x}$ conditionnée par le jeu de partition est donnée par :

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}|k)p(k). \tag{2.21}$$

D'autre part, on sait que le chevauchement (*overlap*) entre deux distributions peut être mesuré par la fonction de Kullback-Liebler de ces deux distributions. Cette mesure est donnée par :

$$KL(p(\mathbf{x}) \parallel q(\mathbf{x})) = \int p(\mathbf{x}) ln(\frac{p(\mathbf{x})}{q(\mathbf{x})}) d\mathbf{x} \tag{2.22}$$

Remarquons que $KL(p(\mathbf{x}) \parallel q(\mathbf{x})) = 0$ si, est seulement si, $p(\mathbf{x}) = q(\mathbf{x})$. Si ce n'est pas le cas, cette fonction est strictement positive et augmente à mesure que le chevauchement entre les distributions diminue. Nous allons essayer de minimiser le chevauchement entre la fonction de densité $p(\mathbf{x})$ et la contribution de la k-ème partition à cette fonction. Notre mesure de chevauchement devient donc :

$$v_k = -KL(p(\mathbf{x}|k) \parallel p(\mathbf{x})). \tag{2.23}$$

Comme $\forall k \, v_k \leq 0$, nous pouvons définir un chevauchement total, qui est la moyenne pondérée des v_k :

$$\begin{aligned} V &= -\sum_{k=1}^{K} p(k) v_k & (2.24) \\ &= -\sum_{k} p(k) \int p(\mathbf{x}|k) \ln(\frac{p(\mathbf{x}|k)}{p(\mathbf{x})}) d\mathbf{x}. & (2.25) \end{aligned}$$

Un partitionnement "idéal" sépare les données de façon à que le chevauchement entre les partitions soit minimal. Comme V ne change pas de signe, la minimisation de V sur toute les données équivaut à la minimisation de V sur chaque donnée. A l'aide du théorème de Bays, nous pouvons donc écrire :

$$V = \int \left(-\sum_{k}^{K} p(k|\mathbf{x}) \ln p(k|\mathbf{x}) \right) p(\mathbf{x}) d\mathbf{x} + \sum_{k=1}^{K} p(k) \ln p(k) \int p(\mathbf{x}|k) d\mathbf{x} \tag{2.26}$$

On peut voir que dans cette expression la première somme sur k est l'entropie (de Shannon) sur le jeu de partition étant donné $\mathbf{x}$. C'est-à-dire

$$H(p(k|\mathbf{x})) = -\sum_{k} p(k|\mathbf{x}) \ln p(k|\mathbf{x}) \tag{2.27}$$

alors que la deuxième somme sur k est l'entropie négative des probabilités à priori des classes.

Dans le but de trouver un jeu de partitions "idéal", nous allons commencer par modéliser la fonction de densité des données par un mélange de J gaussiennes trouvées à l'aide de l'algorithme EM vu précédemment (2.2.2). Nous obtiendrons donc, pour chaque donnée $\mathbf{x}$ un ensemble de J probabilités $p^*(j|\mathbf{x})$ représentant la probabilité que le point $\mathbf{x}$ (connu) ait été généré par la j-ème gaussiennes. La notation avec étoile ("*") signifie que les probabilités sont évaluées dans la représentation des kernels trouvée à l'aide de EM. Il est claire qu'il n'est pas obligatoire d'utiliser un mélange de gaussiennes pour représenter la fonction de densité. Tout autre représentation à base de kernels peut être utilisée, néanmoins nous utiliserons le mélange de gaussiennes dans notre explication. Le but va être de trouver une transformation qui permette de passer de la représentations du mélange de gaussiennes fourni par EM à une représentation minimisant l'entropie. Nous allons donc chercher une matrice W tel que

$$\mathbf{p} = \mathbf{W}\mathbf{p}^* \tag{2.28}$$

ou $\mathbf{p}$ représente le vecteur de probabilité à posteriori des partitions, $\mathbf{W}$ la matrice de transformation (pas forcement carrée) et $\mathbf{p}^*$ le vecteur de probabilité à posteriori des kernels donné par EM. Nous avons donc, pour la composante i du vecteur $\mathbf{p}$

$$p_i = p(i|\mathbf{x}) = \sum_j W_{ij} p^*(j|\mathbf{x}). \tag{2.29}$$

Comme $\mathbf{p}$ est considéré comme un ensemble de probabilités à posteriori, les conditions suivantes doivent être satisfaites :

$$p_i \in [0,1]\, \forall i \text{ et } \sum_i p_i = 1. \tag{2.30}$$

Pour satisfaire la première de ces conditions nous imposerons que $W_{ij} \in [0,1]\ \forall i,j$. La seconde condition est vérifiée si

$$\begin{aligned} 1 = \quad & \textstyle\sum_i p(i|\mathbf{x}) = & \sum_i \sum_j W_{ij} p^*(j|\mathbf{x}) \\ = \quad & \textstyle\sum_j p^*(j|\mathbf{x}) \sum_i W_{ij} & = \sum_i W_{ij} \end{aligned}$$

c'est-à-dire que la somme des éléments de chaque colonne de $\mathbf{W}$ est égale à 1.

Minimisation de l'entropie

Nous allons voir maintenant comment minimiser l'expression (??). Pour commencer, nous allons prendre en compte les contraintes sur $\mathbf{W}$ vue précédemment. Pour cela, nous allons introduire un ensemble de variables θ_{ij} qui seront optimisées et qui représenteront les W_{ij}, au moyen de la fonction *softmax* :

$$W_{ij} = \frac{\exp(\theta_{ij})}{\sum_{i'} \exp(\theta_{i'j})}.$$

Le but que nous poursuivons est de trouver les valeurs des paramètres W_{ij} donc θ_{ij}, qui minimise l'entropie. C'est donc un problème d'optimisation de paramètres qui peut se résoudre de différentes manières. Entre autres, il est possible d'utiliser les méthodes de descente du gradient, les méthodes

dite de Newton et les méthodes "quasi-Newton". Ces méthodes nécessitent généralement le gradient de la fonction vectorielle à optimiser. Dans le cadre de ce projet, nous utiliserons une méthode quasi-Newton appelée procédure de *Broyden-Fletcher-Goldfard-Shanno* (BFGS). Nous ne décrirons pas cette méthode ici, pour plus de renseignement voir [1, chap. 7] ou [12]). Comme le gradient de V est nécessaires pour pouvoir optimiser les paramètres, nous allons nous intéresser de près à la dérivée de l'entropie V . La dérivée de V par rapport aux θ_{ij} est donnée par :

$$\frac{\partial V}{\partial \theta_{ij}} = \sum_{i'} \frac{\partial V}{\partial W_{i'j}} \cdot \frac{\partial W_{i'j}}{\partial \theta_{ij}}. \tag{2.31}$$

Posons que $\delta_{i'i}$ est le symbole de Kronecker qui vaut 1 quand $i' = i$ et 0 sinon. Avec cette notation, il est facile de voir que, en dérivant le dernier terme de (2.31), on obtient :

$$\frac{\partial W_{i'j}}{\partial \theta_{ij}} = W_{i'j}\delta_{i'i} - W_{i'j}W_{ij}. \tag{2.32}$$

A l'aide de (2.31) et de (2.32), l'on peut écrire après quelques transformation, la dérivée de V par rapport à θ_{ij} sous la forme suivante

$$\frac{\partial V}{\partial \theta_{ij}} = \frac{1}{N} \sum_{i'} (\delta_{i'i}W_{ij} - W_{i'j}W_{ij}) \sum_{n} \ln\left(\frac{p(i')}{p(i'|\mathbf{x}^n)}\right) p^*(j|\mathbf{x}^n) \tag{2.33}$$

Grâce à ces dérivées et à la méthode quasi-Newton, il nous est possible de trouver les W_{ij} qui minimise V ce qui implique un partitionnement optimal.

2.3 Algorithmes supervisés

2.3.1 L'algorithme *Learning Vector Quantization* (LVQ)

L'algorithme *Learning Vector Quantization* a été présenté au début des années nonante par le professeur Teuvo Kohonen et son équipe de l'université d'Helsinki. Il existe différentes versions de cet algorithme toutes basées sur le même principe. Nous présenterons ici la version de base, connue sous le nom de LVQ1, puis les versions qui ont été implémentées dans le cadre de ce projet qui sont le *optimized-learning-rate LVQ1* que nous désignerons pas son abréviation OLVQ1 et le LVQ3. Le fonctionnement de l'algorithme LVQ1 ressemble à celui de la version *online* de l'algorithme du K-Means que nous avons vu à la section 2.2.1. La principale différence est que LVQ1 prend en compte la classification au moment de déplacer les vecteurs représentant les différents groupes.

Généralités

Posons que I est le nombre de vecteur $\mathbf{m}_i$, appelé *codebook vectors* dans la nomenclature de LVQ, qui serviront à représenter les données. Généralement, il y a plusieurs *codebook vectors* par classe étant donné qu'il est tout à fait possible que des points appartenant à une même classe (c'est-à-dire ayant la même valeur dans l'espace des outputs) soient réparti en différents groupes dans l'espace des inputs. Après positionnement des codebook vectors à l'aide des algorithmes de la famille LVQ, lors de la classification, un vecteur $\mathbf{x}$ sera considéré comme appartenant à la même classe que le vecteur $\mathbf{m}_i$

dont il est le plus proche. Posons que c représente l'indice du *codebook vector* le plus proche (au sens de la distance euclidienne) de $\mathbf{x}$, c'est-à-dire :

$$c = \arg\min_i \{\| \mathbf{x} - \mathbf{m}_i \|\} \text{ avec } 0 < i \leq I \tag{2.34}$$

On peut donc dire que $\mathbf{m}_c$ est le *codebook vector* le plus proche de $\mathbf{x}$. Posons que $\mathbf{x}$ est un vecteur (pattern) d'entraînement et $\mathbf{m}_i(t)$ le *codebook vector* i au pas de temps discret t . A chaque pas de temps, c'est-à-dire à chaque nouveau $\mathbf{x}$, les *codebook vectors* $\mathbf{m}(t)$ sont actualisés selon la règle suivante :

$$\begin{aligned}
\mathbf{m}_c(t+1) &= \mathbf{m}_c(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)] && (2.35)\\
& \quad \text{si } \mathbf{x} \text{ et } \mathbf{m}_c \text{ appartiennent à la meme classe} && (2.36)\\
\mathbf{m}_c(t+1) &= \mathbf{m}_c(t) - \alpha(t)[\mathbf{x}(t) - \mathbf{m}_c(t)] && (2.37)\\
& \quad \text{si } \mathbf{x} \text{ et } \mathbf{m}_c \text{ appartiennent à des classes différentes} && (2.38)\\
\mathbf{m}_i(t+1) &= \mathbf{m}_i(t) \text{ pour } i \neq c. && (2.39)
\end{aligned}$$

avec $0 < \alpha(t) < 1$. Le pas d'apprentissage $\alpha(t)$ peut être soit constant soit décroître avec le temps de façon monotone. Dans la version de base, LVQ1, le paramètre $\alpha(t)$ est commun à tout les $\mathbf{m}_i$ et décroît linéairement avec le temps. La règle (2.39) représente le principe de base commun à tout les algorithmes de la famille LVQ, à savoir que, à chaque nouvelle donnée $\mathbf{x}$ le *codebook vector* le plus proche $\mathbf{m}_c$ est déplacé en direction du point $\mathbf{x}$ s'il appartient à la même classe que celui-ci, en direction opposée si ce n'est pas le cas.

OLVQ1

L'algorithme OLVQ1 qui a été implémenté utilise un $\alpha_i(t)$ différent pour chaque *codebook vector* $\mathbf{m}_i$ dans le but d'accélérer la convergence. Posons que $s(t) = +1$ si la classification est correcte et $s(t) = -1$ dans le cas contraire. A chaque itération de l'algorithme on actualisera le taux d'apprentissage du *codebook vector* le plus proche de la manière suivante :

$$\alpha_c(t) = \frac{\alpha_c(t-1)}{1 + s(t)\alpha_c(t-1)}. \tag{2.40}$$

Cette actualisation du taux d'apprentissage permet une convergence nettement plus rapide. Pour une justification mathématique de la vitesse de convergence de l'algorithme avec (2.40), se référer à [6]. L'algorithme OLVQ1 converge en moyenne 3 ou 4 fois plus vite que les autres versions ; c'est donc cette version qui est utilisée dans la plupart des applications pour lesquelles le temps de calcul est un paramètre critique.

LVQ3

La troisième version de LVQ se base sur le même principe que LVQ1, mais les deux *codebook vectors* les plus proches de $\mathbf{x}$ sont déplacés à chaque pas de l'algorithme. Si un des deux *codebook vector*, disons $\mathbf{m}_j$, appartient à la même classe que $\mathbf{x}$ alors que l'autre, $\mathbf{m}_i$, appartient à une classe différente, alors l'algorithme déplacera le vecteur $\mathbf{m}_j$ en direction de $\mathbf{x}$ et le vecteur $\mathbf{m}_i$ dans la direction opposée à

$\mathbf{x}$. Cette actualisation n'est faite que si $\mathbf{x}$ entre dans un fenêtre prédéterminée. Posons que d_i et d_j sont les distances euclidiennes entre le vecteur $\mathbf{x}$ et $\mathbf{m}_i$ et $\mathbf{m}_j$ respectivement. Alors, $\mathbf{x}$ entre dans la fenêtre de largeur w si :

$$\min(\frac{d_i}{d_j}, \frac{d_j}{d_i}) > s \text{ where } s = \frac{1-w}{1+w}. \tag{2.41}$$

Le largeur de la fenêtre w est à déterminer par expérience. Les auteurs de l'algorithme recommande l'utilisation d'un w compris entre 0.2 et 0.3.

Les règles de mise à jour des *codebook vectors* de LVQ3 peuvent s'énoncer de la manière suivante :

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) - \alpha(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] \tag{2.42}$$

$$\mathbf{m}_j(t+1) = \mathbf{m}_j(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] \tag{2.43}$$

avec $\mathbf{m}_i$ et $\mathbf{m}_j$ les deux *codebook vectors* les plus proche de $\mathbf{x}$; $\mathbf{m}_j$ et $\mathbf{x}$ appartiennent à la même classe alors que $\mathbf{m}_i$ appartient à une classe différente. De plus, il faut que $\mathbf{x}$ entre dans la fenêtre définie par (2.42). Si $\mathbf{m}_i$, $\mathbf{m}_j$ et $\mathbf{x}$ sont tout les trois de la même classe, alors la règle de mise à jour est :

$$\mathbf{m}_k(t+1) = \mathbf{m}_k(t) + \epsilon\alpha(t)[\mathbf{x}(t) - \mathbf{m}_k(t)] \tag{2.44}$$

pour $k = \{i, j\}$. Si $\mathbf{x}$ n'entre pas dans la fenêtre, alors aucune actualisation n'est faite. La valeur de ϵ doit être déterminée expérimentalement et semble dépendre de la largeur de la fenêtre (plus la fenêtre est large, plus ϵ doit être petit). Les auteurs de l'algorithme propose un ϵ compris entre 0.1 et 0.5 .

Cet algorithme a l'intéressante propriété d'être "auto-stabilisant" (*self-stabilizing*), c'est-à-dire que les codebook vectors ne change plus après avoir atteint leur position optimale. Dans cette version de l'algorithme, plusieurs fonctions sont possible pour $\alpha(t)$; dans le cadre de se projet, les deux versions proposées dans ([6]) ont été implémentée, à savoir :

$$\alpha(t) = \alpha(0)(1 - t/p) \tag{2.45}$$

et

$$\alpha(t) = \frac{c\alpha(0)}{c+t} \tag{2.46}$$

ou p est le nombres de pas total que l'algorithme doit faire (entre 50 et 200 fois le nombre de *codebook vectors* pour LVQ3) et $c = \frac{p}{100}$.

Chapitre 3

Les réseaux de neurones de fonctions à bases radiales

3.1 Les réseaux de neurones

Les premiers travaux sur les *réseaux de neurones* en intelligence artificielle ont été motivés par l'incroyable capacité de traitement de l'information d'un cerveau. En effet, un animal comme la chauve-souris peut, à l'aide de son "sonar", repérer un insecte en vol, connaître sa position et sa vitesse avec une précision que n'importe quel sonar humain équipé avec la dernière technologie lui envie. Une tel capacité de traitement de l'information dans un espace aussi réduit que le minuscule cerveau d'une chauve-souris invite à se poser des questions sur la façon dont est traitée cette information.

Au contraire des ordinateurs qui sont muni de quelques unités centrales de calculs qui opèrent à très grande vitesse d'une manière essentiellement séquentielle, le cerveau est composé de milliards de minuscules unités qui travaillent en parallèle. Cette architecture hautement complexe et parallèle a inspiré les chercheurs de différentes disciplines qui ont initialisé les recherches dans ce nouveau domaine.

Les avantages d'une architecture hautement distribuée comme celle des réseaux de neurones couplée à la capacité d'apprentissage de ceux-ci sont remarquables. Citons, parmi ces qualités, le fait que les réseaux de neurones ont une très bonne capacité d'*adaptation.* Ils peuvent changer en modifiant leurs poids synaptiques et s'adapter à un nouveau problème ou à une situation changeante sans altérer de la structure de base. De plus, le fait qu'il n'y ait pas d'unité centrale les rend plus tolérant aux fautes. Un réseau de neurone implémenté matériellement pourra continuer à assumer une partie de ces fonctions après avoir été endommagé[5].

Il est intéressant de constater que les biologistes en essayant de comprendre le fonctionnement du cerveau mirent en place des modèles qui peuvent être interprété sans peine par les mathématiciens. En effet, les réseaux de neurones peuvent être abordé de différentes manières. Les biologistes interprètent les analogies avec le fonctionnement du système nerveux alors que les mathématiciens y voient des techniques d'approximation de fonction dans des espaces du haute dimensionnalité. Après une quarantaine d'années d'existance, après avoir été pratiquement oublié, les réseaux de neurones artificielles commence à trouver des applications pratiques et à sortir des laboratoires universitaires

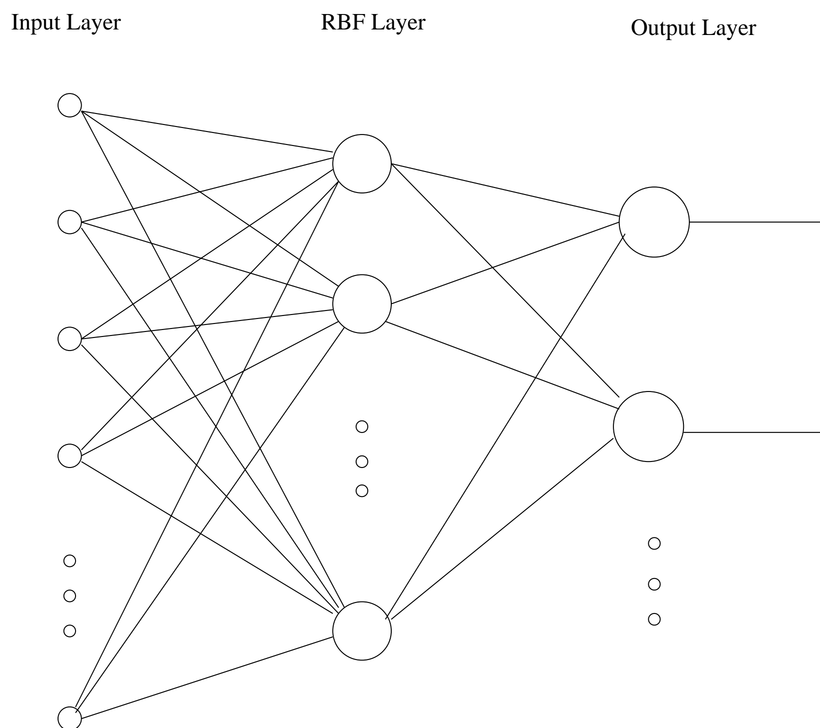
dans lesquels ils étaient cantoné jusqu'à présent. Les applications les plus prommeteuses sont celles liée à la reconnaissance statistique (*pattern recognition*), utilisée notamment dans les applications de reconnaissance de la paroles et de l'écriture.

Il existe plusieurs type de réseaux de neurones. Le plus connus de ces réseaux est probablement le MultiLayer Perceptron (MLP). Dans le cadre de ce travail, nous nous intéresserons à un autre type de réseau de neurone qui a le vent en poupe, les *Radial Basis Functions Neural Networks* (RBFNN).

3.2 Les réseaux de fonction à bases radiales (RBF)

3.2.1 Introduction

Les réseaux de neurones (Radial Basis Function Neural Networks, RBFNN) sont principalement utilisé pour résoudre des problèmes d'approximation de fonctions dans des espaces de grandes dimensions. Vu sous cet angle, apprendre revient à trouver une surface dans un espace multidimensionel qui s'adapte au mieux aux données. La capacité de généralisation du réseau et équivalente à la capacité d'interpolation de nouveau point de la surface. Un réseau de neurones de fonctions à bases radiales est donc un outil qui permet d'approximer une surface multidimensionelle à l'aide de fonctions à bases radiales.



Un RBF est composé de trois couches de neurones. La première couche est composée d'une série de senseur qui récoltent les informations et les transmettent aux neurones de la couche suivante. Cette couche représente les inputs et peut être ignorée si on admet que tout les neurones de la couches suivante, appelée couche RBF, reçoivent toutes les inputs. Chaque neurone de la couche RBF représente une fonction à bases radiale qui réagira aux activations transmisent par les neurones

d'entrée. Enfin, la couche de sortie est faites de neurones linéaires qui font une somme pondérées des excitations transmisent par la couche RBF[5]. Dans la suite de ce travail, nous considérerons que les RBFNN sont constitués de deux couches, à savoir la couche RBF, non linéaire, et la couche de sortie qui calcule la combinaison linéaire des sorties des neurones de la couche RBF.

3.2.2 Base Mathématique

Les méthodes utilisées dans les réseaux de fonctions à base radiale puisent leurs origines dans les techniques d'interpolation exacte de fonction dans un espace multidimensionnel à partir d'un jeu de donnée. Nous prendrons ce modèle comme point de départ de notre présentation du modèle mathématique sous-jacent aux RBFNNs.

Considérons une transformation d'un espace d'entrée en d dimensions à un espace de sortie à 1 dimension. L'interpolation exacte consiste à trouver une fonction tel que chaque point $\mathbf{x}$ de l'espace des entrées correspondent exactement à sa valeur cible t dans l'espace de sortie. Si le jeu de donnée est composé de N vecteurs d'entrée $\mathbf{x}^n$ avec leur valeur cible correspondante t^n alors le but de l'interpolation exacte et de trouve la fonction $h(\mathbf{x})$ tel que

$$h(\mathbf{x}^n) = t^n, \qquad n = 1, ..., N. \tag{3.1}$$

L'approche des fonctions à bases radiales s'appuie sur un jeu de N fonctions de bases, c'est-à-dire une par vecteur de données $\mathbf{x}^n$, de type $\phi(\| \mathbf{x} - \mathbf{x}^n \|)$ avec un certain nombre de restrictions sur $\phi(\cdot)$ que nous verrons par la suite. Nous pouvons d'ores et déjà constater que la fonction $\phi(\cdot)$ dépend de $\| \mathbf{x} - \mathbf{x}^n \|$, la distance (généralement euclidienne) entre la donnée $\mathbf{x}$ et le point $\mathbf{x}^n$ que nous appellerons le *centre* de la fonction. La fonction $h(\mathbf{x})$ sera définie comme étant une combinaison linéaire des fonction de base, c'est-à-dire

$$h(\mathbf{x}) = \sum_n w_n \phi(\| \mathbf{x} - \mathbf{x}^n \|). \tag{3.2}$$

Nous pouvons donc réécrire la condition de l'interpolation exacte (3.1) sous forme matricielle, ce qui nous donne

$$\Phi \mathbf{w} = \mathbf{t} \tag{3.3}$$

avec $\mathbf{t} \equiv (t^n)$, $\mathbf{w} \equiv (w_n)$ et Φ, la matrice carrée $N \times N$ composée de $\Phi_{nn'} = \phi(\| \mathbf{x}^n - \mathbf{x}^{n'} \|)$. Il a été montré (théorème de Micchelli) que pour un grand nombre de fonctions $\phi(\cdot)$ la matrice Φ est inversible si les points $\mathbf{x}^n$ sont distincts, ce qui nous permet d'écrire

$$\mathbf{w} = \Phi^{-1} \mathbf{t}. \tag{3.4}$$

Lorsque les poids w_n de (3.2) sont calculés à l'aide de (3.4), la fonction $h(\mathbf{x})$ représente une surface continue et dérivable passant exactement par chacun des points de donnée $\mathbf{x}^n$.

La généralisation pour plusieurs sorties est immédiate. Si à chaque vecteur d'entrée $\mathbf{x}^n$ doit correspondre un vecteur de sortie $\mathbf{t}^n$ de K composantes t_k^n, alors la condition (3.1) devient :

$$h_k(\mathbf{x}^n) = t_k^n, \qquad n = 1, ..., N \quad k = 1, ..., K \tag{3.5}$$

avec

$$h_k(\mathbf{x}) = \sum_n w_{kn}\phi(\| \mathbf{x} - \mathbf{x}^n \|) \tag{3.6}$$

ce qui, pour obtenir les poids nous donne

$$w_{kn} = \sum_{n'} (\Phi^{-1})_{nn'} t_k^{n'}. \tag{3.7}$$

Il est à remarquer que la matrice Φ^{-1} est la même pour toutes les fonctions $h_k(\mathbf{x})$.

Il a été démontré que, théoriquement, un grand nombre de fonction non linéaire peuvent être utilisé pour l'approximation de la fonction $h(\mathbf{x})$. Les plus utilisées dans les RBFNN sont :

1. *Multiquadratiques :*
$$\phi(r) = (r^2 + c^2)^{\frac{1}{2}} \text{ pour } c > 0 \text{ et } r \in \Re \tag{3.8}$$
2. *Quadratiques inverses :*
$$\phi(r) = \frac{1}{(r^2 + c^2)^{\frac{1}{2}}} \text{ pour } c > 0 \text{ et } r \in \Re \tag{3.9}$$
3. *Gaussiennes :*
$$\phi(r) = \exp\left\{-\frac{r^2}{2\sigma^2}\right\} \text{ pour } \sigma > 0 \text{ et } r \in \Re \tag{3.10}$$

Il faut remarquer que les fonctions multiquadratiques (3.8) et gaussiennes (3.10) sont des fonctions *locales* c'est-à-dire que $\phi \to 0$ quand $|r| \to \infty$. Cette propriété facilite l'interprétation des fonctions de le cadre des RBFNN ce qui fait de ces deux fonctions les plus utilisés de ce domaine. Dans ce travail, nous nous concentrerons principalement sur les gaussiennes lors des expériences et des présentations théoriques. Néanmoins, tout les résultats présentés peuvent être obtenu à l'aide d'une autre de ces fonctions.

3.2.3 Réseaux de fonction à base radiale

Le développement mathématique vu ci-dessus permet l'interpolation exacte d'une fonction à l'aide de N fonctions à base radiale. En pratique, une interpolation exacte est rarement souhaitable. En effet, les données à traiter sont souvent perturbée par un du bruit et une interpolation exacte fait apparaître une fonction très oscillante et irrégulière avec une mauvaise capacité de généralisation, c'est-à-dire de prédiction de nouveau point n'appartenant pas à l'ensemble ayant servi à la construire. Un certain nombre de modifications ont été apportées au modèle vu précédemment pour arrivé aux RBFNNs. Ces modifications sont les suivantes :

1. Le nombre M de fonction de base n'est plus forcement égal au nombre N de donnée mais généralement nettement inférieur à celui-ci.
2. Le centre des fonctions n'est pas obligatoirement lié à la position d'un point de donnée. A la place, divers techniques sont utilisées pour trouver les meilleurs emplacements pour les centres des fonctions.
3. Au lieu d'avoir un paramètres communs σ, chaque fonction de base a ces propres paramètres σ_j dont les valeurs sont déterminées lors de la phase d'apprentissage que nous appellerons entraînement.

4. Un paramètre de biais w_{k0} est introduit dans la combinaison linéaire des fonctions.

Avec ces changement, nous pouvons écrire que la sortie d'un RBFNN est donnée par

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}. \tag{3.11}$$

Le biais w_{k0} peut facilement être intégré dans la somme en introduisant une fonction constante $\phi_0 = 1$.

Si nous considérons le cas des gaussiennes, nous avons, dans le cas général

$$\phi_j(\mathbf{x}) = \exp\left\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_j)^T(\Sigma_j^{-1})(\boldsymbol{\mu}_j)\right\} \tag{3.12}$$

avec Σ_j, la matrice de covariances et $\boldsymbol{\mu}_j$ le centre de la gaussiennes. Il est à remarqué que la fonction $\phi_j(\mathbf{x})$ n'a pas besoin d'être normalisée car le facteur de normalisation peut être absorbé dans les vecteurs de poids $\mathbf{w}$. Comme les matrices de covariances Σ_j sont symétriques, chaque fonction de la forme de (3.12) à $d(d+3)/2$ paramètres indépendants à comparer avec les $(d+1)$ paramètres de la forme moins générale avec une variance unique σ_j dans toutes les dimensions et des covariances nulles. Étant donnée que le nombre de paramètre doit rester dans tous les cas inférieur au nombre de données pour que le réseau garde une bonne capacité de généralisation, il y a un choix à faire entre un nombre élevé de fonctions avec peu de paramètres et un petit nombre de fonctions avec beaucoup de paramètres. Le choix du nombre de paramètres et/ou du nombre de fonction est très important et peut influencer significativement les performances du réseaux. En effet, un nombre élevé de paramètres permet une excellente approximation des données utilisées pour l'entraînement du réseau mais implique parfois une faible capacité de généralisation. Il n'y a pas de technique généralement applicable pour trouver le nombre optimal de paramètres, celui-ci étant fortement dépendant de la nature du problème. Il est donc généralement déterminé par expérimentation.

3.3 Entraînement des RBFs

La construction d'un RBFNN est constituée de plusieurs étapes. Premièrement, il faut choisir l'architecture exacte du réseau, c'est-à-dire déterminer le nombre de fonctions de base qui seront utilisées, leur type, et le nombre de paramètres de ces fonctions. Puis, il faut trouver, pour chacune de ces fonctions, les paramètres, $\boldsymbol{\mu}_j$ et Σ_j dans le cas des gaussiennes (3.12). Finalement, on cherchera la meilleure matrice de poids $\mathbf{W}$.

Un certain nombre de techniques d'entraînement de RBFNN ont été développées. Certaines sont basées sur un apprentissage entièrement supervisé (c'est-à-dire en prenant en compte les variables de l'espace de sortie) alors que d'autres se contentent d'un choix non supervisé des paramètres des fonctions .

3.3.1 Entraînement en 2 étapes

L'entraînement en 2 étapes (*two-stage training*) est une des manières les plus courantes d'entraîner un RBFNN. Comme son nom l'indique, l'entraînement est décomposé en deux étapes clairement séparées. Après avoir choisi l'architecture du réseau, la première étape consiste à chercher les paramètres

optimaux de façon généralement *non-supervisée.* La seconde étape consiste à trouver les poids de la combinaison linéaire (c'est-à-dire la matrice $\mathbf{W}$) les plus adéquats une fois fixé les paramètres des fonctions de la couche RBF.

Première étape : sélection des paramètres.

La première étape, celle de la sélection non-supervisée des paramètres est la plus difficile. Dans le cas des gaussiennes, il faut chercher les centres $\boldsymbol{\mu}_j$ et les matrices de covariances Σ_j. Il apparaît clairement que ce choix est intimement lié à la fonction de densité des données. Il existe plusieurs technique pour choisir ces paramètres. Nous en décrirons quelques unes par la suite.

Sous ensemble des points de données Cette procédure est la plus simple de toutes. Elle consiste à choisir comme centre $\boldsymbol{\mu}_j$ des gaussiennes un sous-ensemble sélectionné aléatoirement, des points de données. Évidemment, cette façon de faire très simple est loin d'être optimale ; pour arriver à une performance acceptable sur les données d'entraînement, il est souvent nécessaire de sélectionner un grand nombre d'entre elles. Les réseaux construits selon cette technique sont donc généralement trop grands et sous-optimaux. De plus, cette façon de faire ne permet pas de déterminer la matrice de covariances Σ_j. Lorsque cette procédure est appliquée, on admet généralement que toutes les fonctions partages un même écart type σ_j que l'on choisi comme étant un multiple de la distance moyenne entre les centres sélectionnés. De cette manière, on s'assure que les fonctions de bases se chevauchent en partie et on obtient ainsi une représentation relativement douce (*smooth*) de la distribution des données, condition requise pour que le réseau garde une bonne capacité de généralisation. Cet algorithme présent l'avantage d'être extrêmement rapide mais il est très nettement sous-optimal. Dans le cadre de se projet, il sera utilisé comme base de comparaison.

Orthogonal least squares Cette procédure se base elle aussi sur la sélection d'un sous-ensemble des données comme centre des gaussiennes. Elle a été décrite en 1991 dans [2] et propose une approche plus systématique pour sélectionner le sous-ensemble adéquat que celle vue au paragraphe précédent. La procédure commence en sélectionnant un point au hasard comme premier centre. Puis, en construisant une base orthogonale dans l'espace des vecteur choisi comme centres, il est possible de trouver le point de donnée qui doit être choisi pour minimiser la somme des erreurs au carré. Cette algorithme minimise une fonction d'erreur, il lui est donc nécessaire de connaître la valeur de la fonction de sortie ; c'est donc un algorithme *supervisé.* Dans certaines situations, il est possible d'obtenir un grand nombres de données dans l'espace des entrées mais que peu de données complètes, c'est-à-dire avec la fonction cible dans l'espace des outputs qui leurs corresponds. Ce cas de figure se présente par exemple quand l'obtention de la valeur cible nécessite un expert humain. Dans ce cas, le fait d'utiliser une procédure supervisé pour sélectionné les centres peut être vu comme un désavantage car il ne permet pas l'utilisation tout les données non traitée par l'expert humain [1].

Algorithmes de clustering Cette dernière technique est celle qui nous développerons le plus dans ce travail. Elle consiste à utiliser des algorithmes de clustering pour trouver un ensemble de centres qui représente au mieux la distribution des données. Différents algorithmes peuvent être utiliser. Le plus simple et le plus courant et l'algorithme du K-Means vu en 2.2.1. Celui-ci permet de trouver

une série de groupes de forme hypersphérique et de sélectionner les centres des gaussiennes comme étant les centres de ces hypersphères. Le K-Means est un cas particulier de l'algorithme *Expectation-Maximization* présenté en 2.2.2. Celui-ci permet la construction d'une approximation de la distribution de probabilité des données à l'aide de ce que l'on appelle un mélange de gaussienne o, en anglais un *gaussian mixture models.* Ces algorithmes ont pour but de trouver oú, dans l'espace des entrées uniquement, il y a la plus grande densité de données. Ils proposent, dans un certain sens, une approximation de la distribution des données. De cette manière, il est possible de mettre plus de centres dans les zones à haute densité de données et moins dans celle qui ne contiennent que peu de données.

Deuxième étape : optimisation des poids de la couche de sortie

Cette deuxième étape, qui se réalise de manière supervisée, est nettement plus simple que la première car il est possible de trouver une solution optimale au sens des moindres carrés. A partir de (??) il est possible d'écrire, en posant que $\phi_0 = 1$,

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x}) \tag{3.13}$$

qui, sous forme matricielle s'écrit de la manière suivante

$$\mathbf{y}(\mathbf{x}) = \mathbf{W}\phi$$

avec $\mathbf{W} = (w_{kj})$ et $\phi = (\phi_j)$. Comme les centres des fonctions ont été choisis lors de l'étape précédente, l'optimisation de $\mathbf{W}$ revient à résoudre un système de N équations linéaires à M inconnues pour chacune des K variables de sortie (fonctions) $y_k(\mathbf{x})$. Comme le nombre de centres M et toujours inférieur ou égal au nombre de données N, le système est généralement surdétérminé. La solution optimale d'un système surdéterminé est celle qui minimise la somme des erreurs au carré. Pour l'ensemble des variables de sortie, la fonction d'erreur est

$$E = \frac{1}{2}\sum_n \sum \{y_k(\mathbf{x}^n) - t_k^n\}^2 . \tag{3.14}$$

Résoudre les K systèmes de N équations à M inconnues revient donc à résoudre le système matriciel

$$\Phi^T\Phi\mathbf{W} = \Phi^T\mathbf{T} \tag{3.15}$$

ou $(\mathbf{T})_{nk} = t_k^n$. Formellement, la solution algébrique de se système est

$$\mathbf{W}^T = \Phi^\dagger\mathbf{T}. \tag{3.16}$$

En pratique, ce système se résous à l'aide de la technique de décomposition en valeurs singulières (*singular value decomposition*, SVD) pour éviter les pièges dû au fait que le problème peut être numériquement mal posé (*ill-conditioning* de la matrice Φ).

3.3.2 Entraînement supervisé

Nous allons utiliser une version de l'algorithme de descente du gradient pour résoudre les problèmes de classification. L'algorithme de descente du gradient se base sur la minimisation d'une fonction d'erreur. Il existe plusieurs types de fonction d'erreur, il est donc nécessaire d'en choisir une qui soit appropriée au problème à résoudre. La fonction d'erreur considérée comme la mieux adaptée à un problème d'interpolation n'est pas la même que celle qui sera utilisée pour résoudre un problème de classification. Dans le cas d'un problème de classification on utilise généralement la *cross-entropy*.

Considérons un réseau avec une sortie y_k par classe et une codification des classes de type 1 parmi c, de façon à ce que $t_k^n = \delta_{kl}$ ($\delta_{kl} = 1$ si $k = l$ et 0 sinon) pour un vecteur d'entrée $\mathbf{x}^n$ appartenant à la classe C_l. Avec ces notations, s'il y a N vecteurs d'entrée $\mathbf{x}^n$, la *cross-entropy* pour plusieurs classes ([1]) s'écrit :

$$E = -\sum_{n}^{N} \sum_{k=1}^{c} t_k^n \ln y_k^n \tag{3.17}$$

Le but est de trouver les paramètres optimaux permettant de minimiser cette erreur. Nous allons donc réécrire (3.17) en fonction des paramètres à optimiser. Pour résoudre des problèmes de classification, la fonction y_k que nous voulons approximmer est la probabilité à posteriori que $\mathbf{x}$ appartienne à la k-ème classe, c'est-à-dire $y_k = p(k|\mathbf{x})$. Il faut donc que la sortie y_k du RBFNN soit comprise entre 0 et 1 et que $\sum_k y_k = 1$. Pour cela, nous allons appliquer la fonction softmax aux sorties du réseau, à savoir :

$$y_k = \frac{\exp\{a_k\}}{\sum_{k'} \exp\{a_{k'}\}} \tag{3.18}$$

Dans cette équation, a_k représente la sortie "normale" du réseau de neurone, c'est-à-dire, dans le cas général en D dimensions avec J neurones dans la couche RBF :

$$a_k = \sum_{j}^{J} w_{jk} \Phi_j(\mathbf{x}) \tag{3.19}$$

Avec $\Phi_j(\mathbf{x}^n)$ la fonction à base radiale qui, dans notre cas sera la gaussienne, à savoir :

$$\Phi_j(\mathbf{x}) = \exp\left\{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_j)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_j)\right\} \tag{3.20}$$

Comme nous pouvons le voir, nous utilisons le cas général de la distance de Mahalanobis dont la distance Euclidienne est un cas particulier, qui correspond à une matrice Σ diagonale dont tout les éléments sont égaux (c'est-à-dire que toutes les variables sont indépendantes et ont la même variance). Nous voulons optimiser les paramètres de la fonction d'erreur 3.17 de façon à la minimiser. Pour cette optimisation, nous allons utiliser un méthode quasi-Newton appelée procédure de *Broyden-Fletcher-Goldfarb-Shanno*. Pour appliquer cette méthode, nous aurons besoin du gradient de la fonction, c'est-à-dire des dérivées partielles de la fonction par rapport à chacun des paramètres à optimiser. Ces paramètres sont d'une part les poids w_{jk} et d'autre part les paramètres des gaussiennes, c'est-à-dire les J vecteurs $\boldsymbol{\mu}_j$ et les J matrices de covariances Σ_j. Pour obtenir ces dérivées, nous utiliserons la

règle des dérivées en chaîne qui, dans ce cas, nous permet d'écrire, pour w_{kj} :

$$\frac{\partial E^n}{\partial w_{jk}} = \sum_{k'} \frac{\partial E^n}{\partial y_{k'}} \cdot \frac{\partial y_{k'}}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{jk}} \tag{3.21}$$

Occupons nous de chaque terme de cette équation. De (3.19) nous pouvons écrire

$$\frac{\partial a_k}{\partial w_{jk}} = \Phi_j(\mathbf{x}) \tag{3.22}$$

De (3.18) nous pouvons déduire que

$$\frac{\partial y_{k'}}{\partial a_k} = y_{k'}\delta_{k'k} - y_{k'}y_k \tag{3.23}$$

D'autre part, à partir de (3.17) nous obtenons

$$\frac{\partial E^n}{\partial y_{k'}} = -\frac{t_{k'}}{y_{k'}} \tag{3.24}$$

A l'aide de (3.23) et (3.24), nous obtenons

$$\frac{\partial E^n}{\partial a_k} = y_k - t_k. \tag{3.25}$$

A l'aide de cette dérivée, nous pouvons maintenant calculer la dérivée de E par rapport à w ce qui nous donne

$$\frac{\partial E}{\partial w_{jk}} = \sum_n \phi_j(\mathbf{x}^n)(y_k^n - t_k^n) \tag{3.26}$$

Nous allons maintenant nous intéresser aux dérivées de l'erreur par rapport aux paramètres des gaussiennes, à savoir les $\boldsymbol{\mu}_j$ et les Σ_j. Pour cela, nous allons dériver l'expression (3.17) par rapport à chacune des D dimensions du centre et de la matrice de covariances. Pour éviter que le nombre de paramètres ne soit trop grand et que, par conséquent, le réseau soit sur-entraîné et perde sa capacité de généralisation, nous nous contenterons d'une matrice de covariances diagonale. Ceci implique que nous admettons que les variables sont indépendantes entre elles et ont donc une covariance nulle. Dans certain cas pratique, nous irons jusqu'à supposer que les gaussiennes sont sphériques (c'est-à-dire avec une variance unique σ_j dans toutes les dimensions) ou que les neurones RBF ont tous la même matrice diagonale de covariance. En effet, un certain nombre de problème ont relativement peu de données et une matrice de covariance complète pour chaque neurone mène inévitablement à un surentraînement. Lorsque les matrices de covariance Σ_j sont symétriques il y a $D(D+3)/2$ paramètres par fonctions alors qu'il y en a seulement $D+1$ dans le cas d'une variance unique σ_j.

Pour l'optimisation, il est nécessaire de connaître la dérivée de E par rapport aux centres des gaussiennes $\boldsymbol{\mu}_j$

$$\frac{\partial E}{\partial \mu_{jd}} = \sum_n \phi_j(\mathbf{x}^n)\frac{(x_d^n - \mu_{jd})}{\sigma_{jd}^2} \sum_k w_{jk}(y_k^n - t_k^n) \tag{3.27}$$

Pour finir, pour la dérivée par rapport à chacun des écarts-types σ_{jd}

$$\frac{\partial E}{\partial \sigma_{jd}} = \sum_n \phi_j(\mathbf{x}^n) \frac{(x_d^n - \mu_j)^2}{\sigma_{jd}^3} \sum_k w_{jk}(y_k^n - t_k^n) \tag{3.28}$$

A l'aide de ces dérivées et d'une méthode d'optimisation, nous pourrons trouver les paramètres optimaux. Il faut remarquer que la recherche supervisée des paramètres des gaussiennes que nous venons de développé présente le désavantage de nécessité une grande quantité de calculs. En utilisant cette technique, on perd donc un des grands avantages des RBFNNs à savoir l'entraînement linéaire par (pseudo)inversion d'une matrice. Le temps d'entraînement en est considérablement allongé et peut devenir problématique. Entraîner un réseau RBF à l'aide d'une descente du gradient peut être plusieurs *centaines* de fois plus long qu'un entraînement en deux étapes.

3.3.3 Estimation de l'ordre du modèle.

Un des problèmes classiques des réseaux de neurones et le choix de l'ordre du modèle, c'est-à-dire, dans le cas des réseaux de neurones à bases radiales, le nombre de neurones dans la couche RBF. En effet, un nombre élevé de neurones RBFs permets une très bonne estimation de la fonction sur le jeu de données d'entraînements. L'erreur est même nulle dans le cas ou le nombre de neurones RBF est égale au nombre de données d'entraînements. Par contre, si le nombre de neurones RBF est trop élevé, la capacité de généralisation du réseau diminue, c'est-à-dire que l'erreur sur l'ensemble de teste augmente. Ceci est du au fait que l'approximation de la fonction n'est plus assez "douce" (*smooth*).

Plusieurs techniques existent pour essayer de trouver le nombre optimum de neurones RBFs. Généralement, ces techniques ne permettent pas de prévoir à l'avance combien de neurones seront nécessaire. Elles impliques l'entraînement de plusieurs réseaux avec différent nombre de neurones et la comparaison des résultats. Dans ce projet, nous avons utilisé la technique appelée *bootstrap training* qui à l'avantage d'être applicable avec un faible nombre de donnée. Comme souvent, cette technique possède de multiples variantes. Nous nous contenterons de la plus simple.

Considérons que nous avons à disposition un ensemble X de N données d'entraînements. La procédure du bootstrap training peut se résumer sous forme algorithmique de la manière suivante :

- Initialisation : $E^0 = 100\%$. $m = 1$.
- Tant que $E^m < E^{m-1}$, faire :
  - Construire un réseau avec m neurones RBF.
  - Répéter K fois :
    - Constituer les ensembles X_E^k et X_V^k.
      - X_E^k : Sélectionner, aléatoirement et avec répétition, N données parmi l'ensemble X.
      - $X_V^k = X \setminus X_E^k$.
    - Entraîner le réseau avec X_E^k.
    - Tester le réseau avec X_V^k. E_k^m est le pourcentage d'erreur sur ce jeu de teste.
  - $E^m = \frac{1}{K} \sum_k^K E_k^m$.
  - $m = m + 1$.
- Choisir $m - 1$ comme nombre de neurones.

Signalons encore une fois que la version présentée est la plus simple et qu'il existe de multiples variantes.

Chapitre 4

Entraînement local de RBFNNs

L'objectif est d'étudier l'effet de la localité de la fonction à approximer et de la proximité des données qui la décrivent.

Le but de l'étude de la localité de la fonction à approximer est d'isoler les zones dans lesquelles la fonction se comporte de manière relativement uniforme de manière à pouvoir l'approximer localement à l'aide de modèles simples et faciles à trouver. En général, l'information complète, à savoir les valeurs connues de la fonction correspondant aux points diponiblent dans l'espace des entrées, est utilisée. Il s'agit donc d'un processus supervisé. Dans le cas des problèmes de classification auxquels nous nous intéressons dans ce travail, les parties intéressantes sont clairement les zones dans lesquelles l'étiquette de classe est la même.

Lorsque nous parlons d'étudier la proximité, nous nous référons à la recherche des groupes "naturels" de données dans l'espace des entrées uniquement. Il s'agit donc d'un processus non supervisé qui consiste à voir comment se groupe les réalisations disponibles de la fonction.

Lors de l'entraînement des RBFNNs, le problème principale est généralement le choix des centres des neurones de la couche RBF. Les algorithmes d'entraînement de RBFNNs présent dans la littérature se divisent en deux catégories (cf. 3.3) :

1. Les algorithmes qui se basent sur des méthodes non supervisées pour trouver les centres des neurones RBF. Une fois fixé les centres des neurones, les meilleurs poids de la combinaison linéaires sont calculés de manières supervisée. (entraînement en deux étapes, voir 3.3.1).
2. Les algorithmes entièrement supervisés. Les poids de la combinaison linéaire sont calculés soit conjointement à la recherche des centres, soit lors d'une seconde phase comme en 1. (cf. 24)

Dans le premier cas, les algorithmes sont principalement basé sur la proximité et la distribution générale des données dans l'espace des entrées, sans considérer l'espace des sorties. L'objectif général est l'aproximation ou l'estimation (si l'on veut une expression analytique) de la fonction de densité des données d'entrée. Les centres des neurones (et les autres paramètres s'ils existent) sont placés de façon à modéliser au mieux cette dernière.

Dans le deuxième cas, les algorithmes se basent sur la complexité de la fonction à approximer. Le but général est la construction d'une approximation de la fonction objectif qui minimise la différence entre cette dernière et son approximation faite par le réseau de neurones. La différence entre les deux

fonctions se mesurent à l'aide d'une fonction d'erreur, généralement la norme du vecteur d'erreur ou sa cross-entropy dans le cas des problèmes de classification.

Ces deux tactiques s'utilisent généralement de manière indépendante l'une de l'autre. Il parait donc intéressant de les combiner pour essayer d'améliorer autant que possible le processus d'entraînement. De plus, cette combinaison devrait pouvoir s'employer indépendemment des techniques concrètes utilisées par chacune des deux tactiques.

L'idée de base est simple : fragmenter la tâche totale (approximation d'une fonction dans l'espace des entrée dans son entier) en sous-tâche plus petites, et donc probablement plus simple, indépendantes les unes des autres puis combiner les résultats de ces sous-tâches d'approximation pour obtenir l'approximation complète. Finalement, il s'agit d'une variante du schéma classique "diviser pour mieux régner".

Pratiquement, ce processus est composé de trois phases que nous allons présenté en détail dans ce chapitre. La première phase est faite de l'analyse des données de manière non supervisée à l'aides des algorithmes de clustering vu en (2.2). La seconde phase consiste a entraîner un RBFNN pour chaque groupe définit lors de la phase 1. Finalement, la dernière étape consiste à regrouper les RBFNNs locaux de la phases 2 en un RBFNN globale capable d'effectuer la tâche de classification désirée.

4.1 Phase 1 : Groupement des données

La première phase, de notre entraînement en trois phases consiste à essayer de représenter la distribution naturel des données le mieux possible. Le but étant clairement de pouvoir s'intéresser de manière plus poussée au zone de l'espace d'entrée qui contiennent beaucoup de points et/ou de regrouper les points ayant quelques choses en commun.

La réalisation de cette phase est décomposée en plusieurs étape que sont l'application des différents algorithmes de clustering non supervisé vu en (2.2). Lors de la réalisation pratique de cette phase, nous irons de l'algorithme le plus simple au plus compliqué, allant à chaque fois un peu plus loin dans la sophistication, et donc dans l'efforts fourni lors du traitement des données.

Pour commencer, comme référence, nous ferons donc un regroupement aléatoire des données. Par regroupement aléatoire, nous entendons choisir K centres c_k au hasard parmi les données puis assigner chaque donnée x^n à l'ensemble k qui minimise la distance $\| c_k - x^n \|$. Cette façon de regrouper les données donne une approximation extrêmement grossière et peu fiable de la distribution de probabilité des données d'entrée. Néanmoins, elle n'est pas complètement inintéressante car comme les points de données sont des réalisations de leur propre fonction de densité ils représentent, dans une certaine mesure, la distribution générale des données.

Le pas suivant sera l'application de l'algorithme du K-Means vu en (2.2.1). Cet algorithme permet de trouver les zones de l'espace d'entrée dans lesquelles sont concentrées le plus de points. Les groupes sont formés, comme pour le regroupement aléatoire, en se basant uniquement sur le critère de la distance euclidienne entre le centre c_k et le points x^n. Par contre, les centres se placent de manière à être au centre de gravité de leur groupe ce qui représente une amélioration notable par rapport au regroupement aléatoire. Les groupes formé à l'aide de l'algorithme du K-Means étant basé sur la distance euclidienne uniquement, ils sont obligatoirement de forme hyper-sphérique ce qui dans certain cas peur représenter une limitation dérangeante.

En continuant à monter sur notre échelle de la sophistication, nous arriverons tout naturellement à l'algorithme EM (2.2.2) dont le K-Means est un cas particulier. A l'aide d'un mélange de gaussiennes, cet algorithme nous permet de modéliser la distribution de probabilité de manière beaucoup plus exacte que les deux précédents. Si la distribution de probabilité des variables d'entrée est de type gaussiennes, ce qui est souvent le cas pour les problèmes réels, et que le nombre de gaussiennes choisi pour modeler la distribution est adéquat, alors l'algorithme EM permettra de représenter la distribution de manière relativement satisfaisante. Le choix du nombre de gaussiennes ainsi que de leurs degrés de liberté sont bien évidemment des paramètres critiques. A l'aide des matrices de covariance diagonale que nous utilisons dans ce travail, il nous est possible de faire des groupes de données de forme hyperellispsoïdal ce qui représente un progrès non négligeable par rapport a l'algorithme du K-Means. L'initialisation de EM se fait à l'aide des centres fournit par le K-Means de l'étape précédente.

Pour finir, la dernière graduation sur notre échelle de l'effort fourni est représentée par l'algorithme $MCDP$ vu en (2.2.3). Cette algorithme se base sur un critère radicalement différent, à savoir la minimisation de l'entropie des probabilités à posterioris des partitions. La forme des groupes qui peuvent être formé à l'aide ce cet algorithme n'est plus basée sur une distance comme dans les cas précédents. Cet algorithme permet, par exemple, de séparer avec succès en *deux* groupes une couronne entourant un disque ce qui n'est évidemment pas possible à l'aide des algorithmes précédents.

4.2 Phase 2 : entraînement de RBFNNs locaux

Lors de la deuxième phase, nous entraînerons un RBFNN pour chaque groupe trouvé à la phase précédente. La motivation étant que les données appartenant à un même groupe aient des similarité qui facilite l'approximation de la fonction. Par exemple, dans le cas des problèmes de classification, on peut espérer que toutes les classes ne soient par présente dans un groupe donné. La dimension de l'espace des sorties est donc réduite et l'on peut espérer que la classification en sera facilitée. Nous utiliserons uniquement des méthodes d'entraînement supervisés lors de cette seconde phase.

La première de ces méthodes sera la descente du gradient vu en (3.3.2). Cette méthode présente l'avantage de proposer un moyen de trouver à la fois les centres des neurones, leurs matrices de covariances et les poids de la combinaison linéaire. Par contre, elle a le gros handicape d'impliquer des temps d'entraînement incomparablement plus long que les autres méthodes d'entraînement de RBFNNs. La puissance de calcule demandée peut s'avérée prohibitoire s'il y a beaucoup de données ou que la fonction d'erreur est très plate.

Le seconde technique que nous utiliserons est une variante de l'entraînement en deux étapes (3.3.1). Pour la première étape, la sélection des centres des neurones, au lieu d'utiliser un EM ou un K-Means, ce qui se fait généralement, nous utiliserons les algorithmes de la famille LVQ (2.3.1) qui sont des algorithmes basés sur la distance euclidienne. A la différence du K-Means, ces algorithmes sont supervisés. Les covariances des neurones RBF sont choisi proportionnelles à la distance moyenne entre le centre et les k centres voisins (*k-nearest neighbours*). Les poids de la combinaison linéaire sont fixés en résolvant le système surdéterminé à l'aide de SVD.

Pour finir, comme troisième technique d'entraînement, nous utiliserons l'algorithme OLS (3.3.1) qui permet de choisir comme centre les points de donnés qui réduisent le plus l'erreur. Les poids de la combinaison linéaire sont à nouveau trouvé en résolvant le système surdéterminé (??).

Nous utiliserons le bootstrap (3.3.3) pour sélectionner le nombre de neurones adéquats.

Le choix de ces trois techniques d'entraînement de nos réseaux locaux a été motivé par le fait que ces algorithmes sont parmi les plus connus. Cependant, signalons que la démarche générale de l'entraînement en trois phases que nous proposons est parfaitement indépendante des algorithmes d'entraînement choisi ainsi que de la façon de sélectionner le nombre de neurones. L'idée de cette phase est simplement d'entraîner un RBFNNs a résoudre un problème local.

4.3 Phase 3 : Arrangement des RBFNNs locaux en un RBFNN global

Cette phase est la plus expérimentale car c'est la moins bien documentée dans la littérature. Les techniques de clustering sont nombreuses et abondamment commentées dans divers types de publications scientifiques. Il en va de même pour les différentes façons d'entrainer un réseau RBF. Par contre, les possibilités de faire travailler ensemble plusieurs réseaux de neurones en les fusionnant ou d'une autre manière en sont à leurs débuts. Il existe les *commitees of networks* et les *mixture of experts* [1], que nous n'utiliserons pas dans ce projet. Nous nous intéresserons à trois techniques relativement simples : la moyenne des réseaux locaux, la fusion des réseaux locaux en un réseau global, et une troisième technique qui ressemble à l'approche prise par le *mixture of experts* que nous appellerons conseil de RBFNNs locaux.

4.3.1 Moyenne des RBFNNs locaux

Cette technique est la plus simple et la plus immédiate ; lorsque l'on veut obtenir la classification d'une nouvelle donnée, on la présente à l'entrée de chacun des réseaux locaux puis on fait la moyenne des sorties locales pour obtenir la moyenne globale. C'est réellement la manière la plus simple de combiner les réseaux locaux. Sa simplicité est aussi sont défaut. En effet, il semble évident que les performance obtenue à l'aide de cette technique seront relativement médiocre à cause du fait que les neurones des RBF situé "loin" de la donnée à classer ne réagiront que très peu. La fait de faire la moyenne affaiblira donc la réponse du réseau local le mieux placé. Néanmoins, nous garderons cette façon de faire comme base de comparaison.

4.3.2 Fusions des RBFNNs locaux

Cette façon de faire, plus sophistiquée que la première est nettement plus intéressante. Elle consiste à garder les centres des neurones des RBFNNs locaux et à crée un réseau global composé de tout les centres des réseaux locaux. Les poids de la combinaison linéaire sont recalculé à l'aide, une fois de plus, de SVD. Cette manière de faire offre plusieurs avantages : les neurones ont été placés de façon à estimer le mieux possible la fonction dans la zone qui leur a été assignée lors de la phase 1 sans être influencé par les valeurs de la fonction dans les autres zones. Ils permettes une estimation optimum de la fonction dans leur zone. D'autre part, le résultat de la fusion des RBFNNs locaux est lui même un RBFNN ce qui n'est pas le cas de la solution précédente. Ce dernier point est important car il permet de considérer cette méthode comme une façon d'entraîner les RBFNNs ce qui peut être intéressant.

4.3.3 Conseil de RBFNNs locaux

Cette façon de faire est motivée par l'envie de garder la totalité des "connaissances" de l'expert local sans les diluer dans une moyenne. Pour cela, nous allons garder les RBFNNs locaux sans les toucher et les combiner de façon à ce que l'expert qui connait le mieux la zone d'origine de la nouvelle donnée ait plus de poids au moment de prendre la décision sur la classification. Pour cela, nous allons ajouter une nouvelle couche de neurone linéaire qui relirons les différents experts. Cette couche contiendra autant de neurones qu'ils y a de classes dans le problème de classification. Chaque neurone de sortie d'un réseau local est relié à *un* neurone linéaire de la nouvelle couche. Nous aurons donc que la sortie du réseau global correspondant à la k-ème classe y_k^g sera donnée par

$$y_k^g = \sum_l^L w_{kl}^g y_k^l \tag{4.1}$$

ou y_k^l est la sortie du réseaux local correspondant à la zone l, L le nombre de zones, et donc de réseaux locaux (correspondant aux nombre de groupes formés lors de la première phase) et w_{kl}^g le poids correspondant au synapse reliant la k-ème sortie du l-ème réseau local au neurone linéaire "global" y_k^g. Constatons, pour commencer, que si $w_{kl}^g = \frac{1}{L}$ nous nous trouvons dans le cas de la moyenne des RBFs locaux que nous avons vu au début de cette section. La nouveauté est que, dans le cas du conseil de RBFNNs locaux, les poids sont des fonctions de l'entrée puisque l'on veut que le réseau qui connaît le mieux la zone d'où vient la donnée ait le plus de poids. Pour cela, il semble logique d'utiliser comme poids les probabilités à posterioris de la nouvelle donnée d'appartenir à la zone dont le réseau local est expert. Concrètement nous avons donc $w_{kl}^g(\mathbf{x}) = p(l|\mathbf{x})$ ce qui nous permet de réécrire (4.1) de la manière suivante

$$y_k^g(\mathbf{x}) = \sum_l^L p(l|\mathbf{x}) y_k^l(\mathbf{x}) \tag{4.2}$$

Du point du vue des réseaux de neurones, on ajoute une couche de neurones linéaires supplémentaire dont les poids sont fonctions des entrées. On peut remarquer que si les groupes sont déterminés par un algorithme de se basant sur les gaussiennes comme le cas particulier de EM présenté à la fin de (2.2.2) alors $p(l|\mathbf{x})$ correspond à la fonction gaussienne généralement utilisée par les neurones de la couche cachée des RBFNN. Dans ce cas, on aurait un neurone RBF dont la sortie serait le poids de la combinaison linéaire de notre réseau global.

Il est possible d'imaginer plusieurs réalistions de cette méthode qui peuvent s'illustrer à l'aide d'une métaphore. Imaginons que notre réseau global est une commission d'experts. Chacun de ces experts connaît bien une partie du problème, correspondant à une zone (déterminée lors de la phase 1) de l'espace des entrées. La commission peut alors prendre ces décisions de trois manières différentes :

1. Chaque expert donne un avis *nuancé* sur la classification. Nuancé dans ce cas signifie que l'avis de l'expert est un vecteur de probabilités donc chacune des composantes représente la probabilité, selon cet expert, que la donnée appartiennent à une des classes. La commission prend sa décision en pondérant l'avis des experts par la probabilité que la donnée soit dans leur domaine. Mathématiquement, on a donc (4.2) avec $y_k^l(\mathbf{x})$ qui est une variable continue représentant la probabilité que $\mathbf{x}$ appartienne à la classe k selon l'expert l . Nous qualifierons cette technique de *soft* car elle permet les avis nuancé, doux.

2. Chaque expert donne un avis *tranché* sur la classification. Tranché signifie que l'expert donne un avis te type oui ou non. Il dit que, selon lui, la donnée appartient à une certaine classe et pas au autre. Comme dans le cas précédent, la commission prend sa décision en pondérant l'avis des experts par la probabilité que la donnée soit dans leur domaine. Mathématiquement, on a l'équation (4.2) avec $y_k^l(\mathbf{x})$ qui est une variable binaire. Nous appellerons cette technique *mixte* car une des variables de l'équation est continue alors que nous avons appliqué un *winner-takes-all* à l'autre.
3. Seul l'expert qui est le plus probablement qualifié pour répondre décide de la classification. L'avis des autres experts n'est pas pris en compte. Cette façon de faire revient à appliquer une procédure *winner-takes-all* au vecteur de probabilité dont la l-ème composante est $p(l|\mathbf{x})$. C'est-à-dire que l'équation (4.2) devient $y_k^g(\mathbf{x}) = y_k^m(\mathbf{x})$ avec $m = \arg\max_l p(l|\mathbf{x})$. Nous désignerons cette technique par le terme *hard*.

Au cours de ce projet, nous testerons ces trois manières de combiner les réseaux locaux, afin de voir si les résultats présentent des différences significatives. Il est important de constater que les deux premières façons de faire, la soft et la mixte, ne sont utilisables que dans le cas ou l'algorithme de groupement de la première phase permet de calculer la probabilité à posteriori d'une instance d'appartenir à un des groupes. Dans le cas contraire (random clustering, K-Means), seul la façon dite hard est applicable.

Chapitre 5

Étude expérimentale

A l'heure de passer aux expérimentations, une multitude de choix importants se présentent. Des plus généraux (quels jeux de données va-t-on utiliser, combien d'expérimentations par jeu de donnée, ...) aux plus spécifiques (quelle valeur donner à σ dans le cas d'un entraînement en deux étapes, combien d'itérations pour le bootstrap, quelle tolérance pour la descente du gradient, ...), tous ont une grande influence qui peuvent modifié de façon significative les résultats. Nous avons évidemment fait les choix qui nous semblait les plus judicieux, en tenant compte des sévères contraintes de temps qui nous étaient imposées. Cette étude expérimentale n'est bien évidemment pas exhaustive et pourrait (devrait) être étendue à d'autres jeux de données et de paramètres. Néanmoins, elle donne déjà une idée de ce que l'on peut attendre, ou pas, de l'entraînement en trois phases décrits au chapitre 4.

5.1 Choix des données

Il existe dans la littérature et sur internet, un grand nombre de jeux de données classiques utilisés pour mesurer et comparer les performances des algorithmes de classification supervisés et non supervisés. Nous avons été limité dans notre choix par plusieurs contraintes, certaines techniques, d'autres théoriques.

Pour commencer, nous nous sommes concentrés sur les problèmes de classifications. Bien qu'elle soit sans autre adaptable aux problèmes de régressions, l'application de notre entraînement en trois phases semble plus immédiate et intuitive dans le cas d'une tâche de classification. Nous avons donc laisser de coté les données traitant de problème de régression.

Comme nous voulions utiliser des algorithmes de groupement basé sur la distance euclidienne, nous avons du éliminer les problèmes contenant des attributs discret. La plupart de nos algorithme sont généralisable pour pouvoir traiter ce type de problème ; c'est donc une contrainte pratique plus que théorique. Cette limitation diminue énormément le nombre de jeux de données possibles. De plus, le manque de temps n'ayant pas permis l'implémentation du traitement des valeurs manquantes par les différents algorithmes utilisées, nous avons été contraint d'éliminer les jeux de données incomplets.

A ces contraintes théorique, il faut en ajouter d'autres d'ordre purement technique cette fois. Par exemple, nous avons laisser de coté les problèmes contenant un nombre de données très élevé. Le temps et la puissances de calcul à notre disposition ne nous permettaient pas de choisir des problèmes

contenant plusieurs milliers d'instances. En effet, il faut tenir compte du fait que si l'algorithme, lors d'un entraînement en trois phases, nous allons entraîner un réseau par groupe. Si les RBFNNs sont entraînés par un entraînement en deux étapes (3.3.1), grâce à la simple inversion d'une matrice, ceci n'est généralement pas un problème. Par contre, s'il on utilise un entraînement supervisé (3.3.2) de type descente du gradient, alors l'entraînement d'un grand nombre de réseaux devient problématique. D'autant plus si l'espace d'entrée est de grande dimension et/ou que le nombre de données est élevé. A cela, il faut ajouter que pour avoir une moyenne et un écart-type des performance un tant soit peu représentatifs, il faut, au stricte minimum 10 expériences par réseau global et par jeu de teste, plus si possible. Nous avons donc été contraint de nous limiter à des jeux de données de relativement petite taille.

Notre choix s'est porté sur trois jeux de données classiques respectant les nombreuses contraintes que nous venons de voir. Ces jeux de données sont :

1. **Iris.**
   Ce jeu de données décrit les caractéristiques morphologiques de trois sortes d'iris différentes. C'est un grand classique qui apparaît très souvent dans la littérature. Il contient 50 instances de chaque classe donc 150 instances au total. L'espace des entrées est composé de 4 variables continues et le codage 1 parmi m nous donnes 3 dimensions de sortie. Les classes sont relativement bien séparées et les taux de classification correctes espérés sont élevés. C'est un problème considéré comme facile. Nous avons séparé les données en 2 groupes, un de 100 pour l'apprentissage et au autre de 50 pour le test.
2. **Wine**.
   C'est un autre grand classique des problèmes de classification. Il contient 178 données en 13 dimensions d'entrée et 3 de sorties. Chaque instance décrit, à l'aide de variables continues, les proportions de différents composants chimiques d'un des trois vins possible. Ce problème est lui aussi considérer comme relativement facile et les résultats attendu sont dans tout les cas supérieur à 90%. Il nous permettra de voir si notre méthode d'entraînement en trois phases supporte avec succès une dimensionnalité plus élevée. Nous avons divisé les données en 100 données d'apprentissage et 78 de teste.
3. **Vowel.**
   Ce problème est aussi très connu et un grand nombre de résultats sont à disposition. Il consiste en 977 données de 10 dimensions d'entrée provenant de l'analyse des sons de 11 voyelles anglaises prononcées par différentes personnes. La dimension de l'espace de sortie est donc 11. Ce jeu de donnée est considéré comme très difficile et les résultats ne dépassent généralement pas les 50% dans le meilleur des cas.

Pour chacun de ces jeu de données, nous avons commencé par les centrés et les réduire de manière à ce que leur moyenne soit égale à 0 et leur écart-type à 1. Ceci s'obtient en applicant la formule suivante

$$x' = \frac{x - \mu_x}{\sigma_x} \tag{5.1}$$

à chaque donnée. Il faut noté que l'analyse et la transformation des données à l'aide d'outil statistique comme l'analyse en composante principale, par exemple, permet parfois de réduire la dimensionnalité de l'espace des entrées moyennant une perte d'information minime. Ce qui est connu sous le nom de

preprocessing peut avoir une grande influence sur les performances des réseaux de neurones. Malheureusement, ce sujet pourrait faire, à lui seul, l'objet d'un travail complet. Nous le laisserons donc de coté et nous limiterons à réduire les données au moyen de (5.1).

5.2 Plan d'expérimentation

La plan d'expérimentation suivit n'a pas été le même pour les deux premiers jeux de donnée que pour le dernier. La différence dans la méthode d'expérimentation est due à la taille et à la complexité nettement plus élevée du dernier jeu de donnée.

Nous avons considéré que pour présenter un résultat numérique dans ce document, il faut qu'il soit basé sur un stricte minimum de dix mesures. Pour avoir une moyenne correcte qui ne soit pas exagérément biaisée, il faudrait se baser sur un plus grand nombre de résultat. En effet, la plupart des algorithmes partent d'une initialisation aléatoire et un départ "malchanceux" peut les faire converger vers une solution qui n'est pas celle généralement espérée. Ceci est particulièrement vrai lorsque l'on entraîne un RBFNN en choisissant aléatoirement les centres parmi les données d'entraînement. Si une région de l'espace n'est couverte par aucun centre, alors les performances du réseaux seront catastrophiques.

Sauf indication contraire, les chiffres présentés dans la partie résultat sont des moyennes sur 15 expériences. La valeur mesurée n'est pas l'erreur comme c'est parfois le cas mais plutôt le pourcentage de classification correcte. Si rien n'est précisé, ce pourcentage se réfère à la réussite sur le jeu de teste.

La moyenne des pourcentages de bonne classification n'est pas la seul mesure sur laquelle nous nous arrêterons. En effet, la signification de l'écart-type nous parait, elle aussi, très importante. Dans les réseaux de neurones, il est important de savoir si une méthode d'entraînement donne un résultat fiable et peu variable à chaque utilisation ou si elle change énormément. Une méthode qui donne de manière très constante un bon résultat trouvera d'autres utilisation qu'une autre qui donne parfois un excellent résultat et d'autre fois non. Dans les cas des applications ou le temps n'a pas d'importance, on préférera la seconde quitte à devoir entraîner plusieurs fois le réseau alors que si l'on veut un résultat correcte en un seul entraînement, on utilisera la première. L'écart-type est donc, dans le cas des réseaux de neurones, une mesure tout à fait importante. Une diminution de l'écart-type peut être considérer comme un progrès au même titre qu'une augmentation de la moyenne. Il est néanmoins évident que le but premier reste l'amélioration du pourcentage de bonne classification.

Le pourcentage de succès ne sera pourtant pas l'unique mesure à laquelle nous nous intéresserons. Nous prendrons aussi en compte le temps nécessaire à entraîner le réseau ainsi que le nombre de neurones utilisés. En effet, s'il faut dix fois plus de temps pour entraîner un réseau dont les performances sont meilleur d'un demi pourcent, on peut remettre en question l'utilité réel de l'amélioration. Une fois de plus, le choix entre performance et temps d'entraînement dépend énormément de l'application. Certaine application peuvent se permettre d'attendre très longtemps (le temps qu'une descente du gradient donne une solution acceptable par exemple) alors que d'autre sont liées à ce qui est considéré comme un avantage des RBFs sur d'autre type de réseaux de neurones, à savoir, une fois les centres fixé, entraînement par simple inversion d'une matrice.

Avant de commencer les expérimentations "réels" sur les jeux de données présentés en (section 5.1), un certain nombre de testes on été fait avec des données artificielles (mélange de gaussiennes par

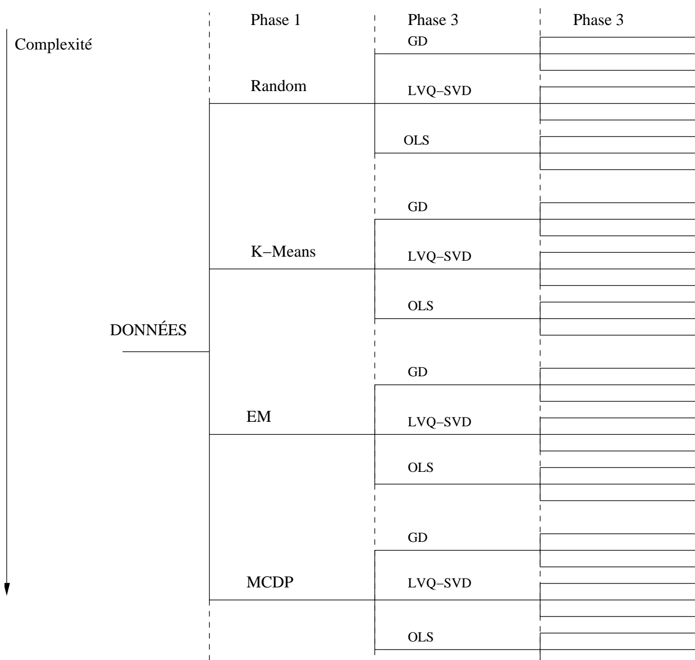


FIG. 5.1 – Plan d'expérimentation.
Cet arbre représente le plan d'expérimentation que nous allons appliqué. Les algorithmes de groupement de la phase 1 vont du plus basique, le Random, au plus évolué, le MCDP

exemple) dans le but de vérifier le fonctionnement correcte des algorithmes. Ces testes n'étant pas d'un grand intérêt pour le lecteur, ils ne seront pas présenté dans ce document.

5.2.1 Déroulement général

Nous avons à disposition quatre algorithme de groupement non supervisé qui ont été présentés au chapitre 2.2. Il s'agit du *random clustering*, du K-Means, de l'*Expectation - Maximization* et de l'algorithme appelé *Maximum Certainty Data Partitionning*. Chacun de ces algorithmes, qui constitue la première phase de notre entraînement en trois phases, et le départ d'une série d'expérience indépendante.

Pour une description détaillée des trois phases, voir le chapitre 4. Signalons que nous avons du faire une adaptation maison de l'algorithme OLS (qui consiste en fait autant de réseaux qu'il y a de classes) pour pouvoir l'utiliser pour des problèmes de classification. En effet, cet algorithme est normalement prévu pour résoudre des problèmes à une seule variable de sortie. Pour des raisons techniques qui découlent de cette adaptation, nous n'appliquerons pas la méthode appelée fusion de RBFNNs locaux aux réseaux entraîner à l'aide de cet algorithme.

L'algorithme MCDP a été implémenté comme présenté dans l'article ([9]). Cette implémentation pose énormément de problème. En effet, la fonction à minimiser possède un très grand nombre de minimums locaux auxquels il est très difficile d'échapper. En conséquence, il arrive extrêmement souvent que cette algorithme crée des partitions vides ce qui pose évidemment beaucoup de problème

pour la suite des expérimentation. Les résultats provenant de cette manière de faire le clustering initial sont tant disparates que nous ne les présenterons généralement pas. Après avoir écrit à l'auteur pour lui demander des précisions, nous avons eut connaissance d'un autre article décrivant une nouvelle manière de minimiser la fonction d'entropie à l'aide de *Markov Chain Monte Carlo* [10] qui semble-t-il permet d'éviter en grande partie lepiège des minimums locaux. Cette information arrivant trop tard, nous n'avons pu l'exploiter.

5.2.2 Expérimentations sur Iris et Wine

Les jeux de données Iris et Wine étant de petites tailles, il a été possible de faire 15 expérimentations pour chacune des configurations possibles. Nous avons ensuite fait des moyennes sur ces résultats.

Comme nous l'avons déjà signaler, nous n'avons pas de méthode précise pour connaître le nombre idéal de groupe à former lors de la première phase. Nous sommes donc obligé d'essayer avec plusieurs valeurs. Il faut tenir en compte le fait que pour chaque nouveau nombre de partition, ou ordre, il faut faire un cycle complet de partitionnement, d'entraînement des réseaux locaux, de construction des réseaux globaux et de teste. Ce cycle prend beaucoup de temps, matériel et humain. Nous nous limiterons donc, dans le cas des jeux de données Wine et Iris a faire ce cycle avec des nombres de partitions compris entre le nombre de classes et deux fois celui-ci.

5.2.3 Expérimentations sur Vowel

Ce jeu de donnée contenant beaucoup plus de données, nous ne pourrons pas utiliser le même plan d'expérience que nous avons appliqué aux deux autres jeux de teste. En effet, ces données sont composée de 10 variables d'entrée et 11 variables de sortie. Il nous est absolument impossible, matériellement, de faire le cycle complet d'expérimentations avec un nombre de groupes pour la première phase variant entre 11 et 22. L'entraînement de 20 RBFNNs à l'aide de GD en utilisant la technique du bootstrap pour déterminer le nombre de neurones est relativement longue. Nous ne disposons pas de la puissance de calcule nécessaire pour effectuer une quinzaine d'expériences par partitionnement comme pour les jeux de données précédents.

Nous agirons donc en deux étapes. Premièrement, nous ferons une première série de testes préliminaires pour essayer de déterminer quel est le nombre de partitions adéquats et quels sont les algorithmes qui donnent les meilleurs résultats sur ces données. Nous ne présenterons pas de résultats numériques de cette première étape car nous n'aurons pas assez de valeurs pour faire des moyennes crédibles.

La deuxième étapes consistera a faire les expériences proprement dites, dans le but d'en tirer des chiffres significatifs, avec les valeurs et les algorithmes sélectionnés lors de la première étape.

Il est évident que ce n'est pas la meilleur façon de faire et qu'il serait bien de pouvoir appliquer la même méthode de façon stricte à tout les jeux de données. Le problème est qu'il faut faire un choix entre utilisé de petit jeu de donnée sans grand intérêt mais qui ont le grand avantage de ne demander qu'un puissance de calcul limitée ou passer à des jeux de données plus sérieux qui posent les problème précédemment décrit. Nous avons choisi la seconde option.

5.3 Résultats

Nous allons donc cette section présenté les résultats des expériences réalisées. Comme nous pouvons le constater sur la figure (fig. 5.1) le nombre d'expérimentation réalisés est relativement important. Nous allons essayer de ne pas surcharger le lecteur de tableaux et de chiffre sans grand intérêt. Il y a plus d'une trentaine de réseaux globaux finaux pour chaque jeu de donnée. De plus, il y a un ensemble complet de réseaux globaux finaux pour chaque division initiale. Présenter plus d'une centaine de moyennes, écart-types, temps d'exécution, nombre de neurones n'aurait pas de sens. Nous essayerons donc de ne donner en détails que des résultats présentant un certain intérêt. Pour cela, nous résumerons certaines mesures de manière qualitative, sans entrer dans les détails. Nous développerons un peu plus la partie résultat concernant le dernier jeu de données car elle nous parait plus intéressant.

5.3.1 Iris

Lors de la présentation de ces résultats, il ne faut pas perdre de vue le fait que le jeu de teste n'est composé que de 50 instances et que donc une erreur de classification sur une instance présente une différence de 2% sur le pourcentage de réussite.

Moyenne de RBFNNs locaux

Pour commencer, nous allons nous intéresser aux résultats de la première des techniques, celle qui consiste à simplement faire la moyenne des RBFs locaux (4.3.1). Nous ne donnerons pas de résultats chiffré en ce qui concerne ce type de réseaux globaux car comme prévu, les résultats sont très médiocres. Néanmoins, signalons quelques points intéressants :

- La moyenne des réseaux locaux entraîner à l'aide de l'algorithme GD donne des résultats étonnement bon. Ils sont plusieurs fois supérieur à 90%. Par contre, il arrive parfois qu'il y ait des très mauvaises expérience, avec des résultats au alentour des 50%. Les écarts-types sont donc élevés.
- Les performances du réseau global entraîné à base de réseaux locaux OLS sont nulles. En effet, elles oscillent autour des 35% ce qui correspond à un tirage au hasard parmi les trois classes possibles.
- L'algorithme de clustering utilisé lors de la première phase semble n'avoir aucune influence (différences sur les moyennes inférieur à 1%). Plus étonnant, le nombre de groupes ne semble pas influencer les résultats.

Le point à souligner est l'étonnement bon résultat de la moyenne des réseaux locaux entraînés à l'aide de l'algorithme du GD.

Fusion des RBFNNs locaux

Intéressons nous maintenant à la technique qui consiste à supprimer les poids des combinaisons linéaires des réseaux locaux pour ne garder que les centres et, si l'algorithme de la phase 1 est EM les variances. Les poids du réseaux global sont recalculé à l'aide de SVD (4.3.2). Le tableau (tab. 5.1) montre les moyennes et écarts-types du pourcentage de classification correcte en fonction de l'algorithme de clustering utilisé durant la première phase et de l'algorithme d'entraînement des réseaux

phase 1 \ phase 2	Random	K-Means	EM
GD moyenne des performances	98.45	97.82	98.05
GD écart-type à la moyenne	1.42	2.96	2.44
LVQ-SVD moyenne des performances	92.95	91.11	87.35
LVQ-SVD écart-type à la moyenne	4.16	5.5	7.61

TAB. 5.1 – Iris : Fusion de RBFNN locaux.
Ce tableau montre les résultats de la fusions (4.3.2) des RBFNNs locaux sur le jeu de données Iris. Lors de la première phase, les données ont été divisée en 3, 4, 5 et 6 groupes. 15 expériences par divisons ont été réalisée. Dans ce cas, le nombre de groupes n'ayant pratiquement pas d'influence sur les résultats, on montre ici la moyenne sur les 60 expériences prisent ensemble. C'est le pourcentage moyen de classification correcte qui est représenté dans ce tableau.

locaux de la deuxième phase. Comme il semble que le choix du nombre de groupes n'affecte que très peu les performances (moins de 1%, non monotone), nous avons choisi de présenté les résultats indépendemment du nombre de groupes. Un total de 15 expériences ont été réalisées en faisant un clustering en 3 groupes lors de la première étape, la même chose en faisant 4, 5 et 6 groupes. Les résultats présentés sont donc des moyennes sur 60 expériences, ce qui représente un bon nombre qui permet de considérer les chiffres obtenus avec une certaine confiance.

Dans un premier temps, on constate que l'algorithme de descente du gradient (GD) permet d'obtenir des résultats significativement meilleurs que lors d'une recherche des centres avec LVQ et d'une résolution des poids à l'aide de SVD. Par contre, il faut tenir compte des temps d'entraînement qui n'apparaissent pas dans notre tableau. Le temps d'entraînement d'un réseau global à l'aide de GD est entre 10 et 100 fois supérieur à celui d'un LVQ-SVD. Dans le cas d'un jeu de donnée comme Iris, les temps restent tout à fait acceptable, mais c'est une information à prendre en compte lors du traitement de données nombreuses ou compliquées.

Il semble que l'enseignement principal à tirer de ces résultats se situe au niveau des écart-types. En effet, mis à par le résultat étonnement mauvais de la combinaison EM+LVQ-SVD, les performances sont relativement égales. Par contre, les écarts types sont, pour leurs parts, significativement différents. Quelques soit le nombre de groupes, quelques soit l'algorithme d'entraînement de la seconde phase, un groupement autour de centres choisis au hasard mène à une plus grande régularité que si les groupes sont formé à l'aide d'algorithmes plus sophistiqué. Nous essayerons d'expliquer ce surprenant résultat dans la section (5.4).

Conseils de RBFNNs locaux.

Les conseils de RBFNNs locaux permettent d'obtenir, pour ce jeu de données particulier, d'excellents résultats. Nous avons tester les trois types différents présenté en à la section (4.3.3). Pour ces données, il semble qu'il n'y ait pratiquement aucune différence entre les différentes formes de conseil, c'est-à-dire entre les conseil hard, mixte et soft. L'écart inférieur à un demi pourcent peut donc être ignorer. Le nombre initial de cluster ne semble pas non plus avoir une influence. Par contre, l'algorithme d'entraînement et, pour ça part un facteur discriminant évident. Ce qui n'est d'ailleurs pas vraiment étonnant aux vues des résultats précédents. La table (5.2) résume ces résultats.

Dans le cas des conseils de RBFNNs locaux, c'est toujours l'algorithme *EM* qui a été utilisé lors de la première étape. Dans le cas de LVQ-SVD, l'amélioration du résultat par rapport à (tab. 5.1) est

performances \ phase 2	GD	LVQ-SVD	OLS
Moyenne des performances	99.76	91.05	99.64
Écart-type à la moyenne	0.88	4.69	1.00

TAB. 5.2 – Iris : Conseil de RBFNNs locaux.
Les résultats des conseils de RBFNNs locaux sur le jeu de donnée Iris. La différence entre les conseils soft, mixte et hard étant pratiquement nulle, c'est la moyenne de tout les résultats qui est présentée ci-dessus. Ce sont donc des chiffres mesuré sur 60 expériences. Rappelons que dans le cas des conseils de RBFNNs, l'algorithme de la première étape est toujours *EM*.

notable (de 87.35% à 91.05% de réussite). En ce qui concerne GD, il n'est pas possible d'affirmer quoi que ce soit de façon claire, le résultat précédent étant déjà très élevé.

La comparaison avec des méthodes d'entraînement globale ne nous mène pas très loin dans ce cas. Il est en effet possible sans grand peine d'obtenir des résultats de 100% de classifications correctes. Un algorithme se basant sur une sélection aléatoire des centres aura parfois un raté qui fera baissé sa moyenne et augmenté son écart-type mais, en règle général, ce problème est trop simple pour pouvoir faire des comparaisons intéressantes.

Nous ne nous attarderons pas plus sur ces chiffres. La simplicité de ce jeu de données ne permet pas de savoir si une méthode est bonne. Elle permet simplement de se rendre compte qu'elle n'est pas sujette à de gros problème. En effet, une méthode qui ne résout pas ce problème avec plus de 80% de réussites peut être considérée comme très mauvaise.

5.3.2 Wine

Le jeu de teste Wine, bien que toujours considére comme très facile, est un petit peu plus compliqué que le précédent. Pour commencer, il possède 13 variables d'entrée comparées aux 4 du précédent. Comme pour le jeu de teste Iris, 15 expériences ont été faites pour chaque découpage en 3, 4, 5 et 6 groupes selon les algorithmes de la phase 1.

Moyenne des RBFNNs locaux

Comme pour le jeu de teste précédent, les résultats de la moyenne des RBFNNs locaux est très médiocre (au environs de 50%). Nous ne voyons aucune nouveauté par rapport à l'expérience précédente : le GD obtient parfois des résultats acceptable, d'autre fois non. Les deux autres restent constamment médiocre.

Fusion des RBFNNs locaux

Les résultats sont relativement difficile à interpréter. Une fois de plus, le nombre de groupe ne semble pas influencé de manière remarquable les performances. Nous avons résumé les résultats des expériences dans le tableau (tab. 5.3).On constate une fois de plus que le Random Clustering permet d'obtenir de meilleurs résultats. Non seulement les résultats sont bien meilleurs (plus de 3%) mais en plus ils sont beaucoup plus régulier, surtout en ce qui concerne le LVQ-SVD.

Il est aussi important de constater que la différence entre les performances du GD et du LVQ-SVD s'est nettement amoindrie. On peut même considérer qu'elle a disparue. Sachant que le temps

Phase 2 \ Phase 1	Random	K-Means	EM
GD Moyenne des performances	92.35	89.9	89.5
GD Écart-Type à la moyenne	5.73	6.05	6.22
LVQ-SVD Moyenne des performances	92.02	83.4	89.9
LVQ-SVD Écart-Type à la moyenne	6.75	9.08	10.2

TAB. 5.3 – Wine : Fusion de RBFNNs locaux.
Résultats de la fusion des RBFNNs locaux sur le jeu de donnée Wine. Les chiffres présentés sont la moyennes des pourcentages de classifications correctes sur tout les découpages possibles lors de la phase 1, c'est-à-dire sur 60 expériences. On constate que le random clustering permet d'obtenir de meilleurs résultats que les algorithmes de clustering plus sophistiqués.

d'entraînement d'un réseau, ou plutôt d'un groupe de réseaux locaux à l'aide de GD prend environ 100 fois plus de temps, on peut considérer que LVQ-SVD est généralement plus performant. Par contre, il faut aussi tenir compte du fait que l'écart-type sur les performances de LVQ-SVD est très élevé, nettement supérieur à celui de GD.

Dans le même ordre d'idée, signalons que le réseau global LVQ-SVD possède dans sa couche RBF un tiers moins de neurones que le réseau GD.

Conseils de RBFNNs locaux

Les résultats du conseil de RBFNNs locaux sont un peu plus intéressant pour ce jeu de donnée que pour le précédent. Ils permettent de mettre à jour quelques associations intéressantes.

Pour commencer, commentons quelques choses qui n'apparaît pas. Une fois de plus, il semble que le nombre de partition effectué lors de la première phase n'a que très peu d'influence, voire aucune. Ce fait est réellement étonnant car, intuitivement on a tendance à penser que ce paramètre est une valeur critique. Pourtant, à part le nombre de neurones qui augmente de façon régulière avec le nombre de partitions, les autres valeurs ne semblent pas influencées par ce paramètre. Bien évidemment, il faut garder à l'esprit que le jeu de données étant simple et de petite taille, on ne peut sans autre généraliser cette affirmation. Néanmoins, nous pouvons nous permettre, une fois de plus, de grouper les expériences et ne plus prendre en compte le nombre de partitions dans la suite de notre discussion. C'est ce que nous faisons dans le tableau (tab 5.4) ce qui nous permets de faire des moyennes sur 60 mesures.

Grâce à ces chiffres, il est possible d'émettre l'hypothèse suivante : les réseaux entraîner à l'aide de l'algorithme LVQ-SVD ne sont pas adapté à un conseil de RBFNNs locaux de type soft. L'écart de performance, sur 60 mesures, semble indiquer clairement qu'il y a une incompatibilité entre l'utilisation de LVQ-SVD pour l'entraînement des réseaux locaux et le principe du conseil soft pour le regroupement en un réseau global. Évidemment, il est impossible d'énoncer une règle générale en se basant sur un seul jeu de donnée mais c'est une hypothèse plausible qui ressort de ces expérimentations.

Nous n'avons pas, cette-fois ci, inclut les écarts-types dans notre tableau de résultats. Notons simplement à ce propos que OLS est très stable et performant. Sa moyenne est la plus haute des trois et son écart-type à la moyenne le plus faible. Ceci, malgré le fait que nous ayons fait une modification "maison" de cet algorithme pour l'adapter au problème de classification. Par contre, il utilise un grand nombre de neurones en comparaison avec ces deux concurrents. Le temps de calcule nécessaire à un entraînement complet à l'aide de cet algorithme est environs dix fois supérieur aux

Phase 2 \ Phase 3	Soft	Mixte	Hard
GD Moyenne des performances	96.37	95.34	93.58
LVQ-SVD Moyenne des performances	88.18	94.17	94.02
OLS Moyenne des performances	97.26	97.3	96.02

TAB. 5.4 – Wine : Conseils de RBFNNs locaux.
Résultats des différentes formes de conseils de RBFNNs locaux sur le jeu de teste Wine. On peut voir que le conseil dans sa forme "soft" ne convient pas bien aux réseaux entraîner à l'aide de LVQ-SVD alors que, au contraire, la forme "hard" semble de pas être idéale pour les réseaux de type GD. Le compromis, c'est-à-dire la forme "Mixte" semble convenir à tout le monde.

temps d'entraînement de LVQ-SVD et dix fois inférieur à celui de GD.

Les méthodes classiques d'entraînement global permettent d'atteindre une moyenne de 97.8% de classifications correctes. Ce résultat est meilleur que ceux que nous obtenons. Il n'est pas très surprenant que sur des jeux de données aussi simple que Iris et Wine, les méthodes plus complexe comme la notre soit un peu désavantagé. La comparaison sera plus intéressante sur un jeu de données plus complexe.

5.3.3 Vowel

Ce jeu de teste est réellement très difficile et les performances attendues sont très nettement inférieur à celle des jeux de données précédents. Nous reproduisons dans la table (tab. 5.5) de résultats fourni avec le jeu de données par A. J. Robinson ([11]). Parmi ces résultats, notons les deux RBFNNs. Le premier est munis de 528 neurones RBF ce qui correspond au nombre de donnée d'entraînement. Nous sommes donc pratiquement dans le cas d'une interpolation exacte (3.2.2). Aucune information n'est fournie sur la manière d'entraîner le réseau, particulièrement sur comment les variances des gaussiennes ont été fixée pour éviter un overfitting. Constatons aussi que le meilleur résultat est obtenu à l'aide d'un *nearest neighbour* qui est, d'une certaine façon, la plus simple manière de faire un classement que l'on puisse imaginer.

Première étape

Lors de la première étape d'expérimentation, nous avons effectuer trois cycles complets (correspondant à la figure (fig. 5.1)) d'expérimentation pour des partitionnement de l'espace d'entrée en 11, 15, 19 et 23 groupes. Comme expliqué en (5.2.3), nous ne pouvions faire des séries de 15 cycles complets pour chacun des ces partitionnements.

De cette première phase, nous pouvons tirer trois conclusions :

1. Les performances des réseaux entraînés à à l'aide de GD sont clairement inférieur à celles des réseaux entraînés à l'aide de LVQ-SVD.
2. Les réseaux entraîner sur 11 et 15 groupes de départ arrivent à des performances légèrement supérieur à ceux entraîner sur plus de groupes.

Ces deux conclusions nous arrangent car elles vont nous permettre d'éliminer l'algorithme GD pour la suite des expériences. C'est, du point de vue pratique, une nouvelle qui permet de réduire grandement le temps de calcul nécessaire aux expériences.

Classifier	# of hidden units	# correct	percent correct
Single-layer perceptron	-	154	33
Mult-layer perceptron	88	234	51
Mult-layer perceptron	22	206	45
Mult-layer perceptron	11	203	44
Modified Kanerva Model	528	231	50
Modified Kanerva Model	88	197	43
Radial Basis Function	528	252	55
Radial Basis Function	88	220	48
Gaussian node network	528	252	55
Gaussian node network	88	247	53
Gaussian node network	22	250	54
Gaussian node network	11	211	47
Square node network	88	253	55
Square node network	22	236	51
Square node network	11	217	50
Nearest neighbour	-	260	56

TAB. 5.5 – Vowel : Résultats fourni avec le jeu de données.
Ce tableau de résultats est fourni avec le jeu de donnée Vowel. Il a été obtenu par Tony Robinson dans sa thèse doctorale ([11]). Le premier RBFNN contient 528 neurones cachés, se qui correspond au nombre de données d'entraînements. On peut noter que la meilleur performance est obtenue par un simple “nearest neighbour”.

Deuxième étape

Au moment de passer à la phase suivante, nous restreindrons donc notre plan d'expérimentation de la manière suivante : Nous effectuerons 15 séries d'expérimentation sur des partitionnements initiaux en 11, 13 et 15 groupes. Nous n'appliquerons pas l'algorithme de descente du gradient pour l'entraînement des réseaux locaux. Ces limitations nous permettent de faire 15 entraînements complets pour chaque partitionnement initial. Les résultats de ces expériences sont présenté en continuation.

Moyenne des RBFNNs locaux

Les résultats de la moyenne des RBFNNs locaux restent très mauvaises. Le contraire eut d'ailleurs été étonnant. Lorsque l'algorithme d'entraînement de la phase 2 et LVQ-SVD, la performance du réseau global et au alentour des 20% ce qui est très mauvais. La performance est encore pire lorsque l'algorithme d'entraînement de la phase 2 est OLS car, dans ce cas le pourcentage de bonne classification stagne au alentour des 10%. Comme il y a 11 classes, on ne peut plus parler de classification mais plutôt de choix aléatoire de la classe.

Fusion des RBFNNs locaux

Pour ce jeu de donnée compliqué, c'est cette manière de construire le réseau global qui permet d'obtenir les meilleurs performances. Cette fois encore, il nous est impossible de trouver une relation claire entre le nombre de partitions, c'est-à-dire le nombre de réseau locaux, et les performance. Quelques soit le type de clustering utilisé lors de la première phase, les résultats sont très légèrement meilleurs pour 11 et 15 partitions que pour 13. Mais la différence étant très légère et le nombre

Phase 2 \ Phase 1	Random	K-Means	EM
LVQ-SVD Moyenne des performances	45.32	43.11	42.25
LVQ-SVD Écart-Type à la moyenne	3.54	4.1	2.81
Nombre moyen de neurones RBF	113.4	83	77.5

TAB. 5.6 – Vowel : Fusion de RBFNNs locaux.
Les résultats de la fusion des RBFNNs locaux sur le jeu de donnée Vowel. Ces chiffres sont les moyennes et écart-type à la moyenne des performances de 45 réseaux globaux. Le partitionnement de la phase 1 est réalisé soit au moyen d'un random clustering, d'un K-Means ou d'un EM. L'entraînement de la phase 2 se fait suivant l'algorithme LVQ-SVD.

de mesures limité on n'en tirera aucune conclusion. Signalons encore que l'écart-type a généralement tendance à légèrement diminuer quand le nombre de partition augmente. Fort de ce constat, nous nous permettrons une fois de plus, de présenter les résultats indépendemment du nombre de partitions.

Comme pour les deux autres jeux de données, les performances sont clairement meilleurs si l'algorithme de groupement de la première phase est un Random clustering (voir tab. 5.6). Ce constat s'impose pour la troisième fois, de plus avec un jeu de données dont les caractéristiques sont très différentes de celles des deux premiers. Nous pouvons donc considérer que l'effort consenti pour faire un "bon" clustering n'est pas seulement inutile mais qu'il est aussi contre-productif. Nous nous devons d'essayer de trouver une explication à ce résultat surprenant et peu intuitif.

Notons que le nombre moyen de neurones dans la douche RBF suit la même courbe que les performances. On pourrait dire que si l'on regarde les performances par neurones, le partitionnement à l'aide de EM permet la meilleur performance par neurone. Néanmoins, comme l'entraînement d'un RBF à l'aide de LVQ-SVD ne pause aucun problème de temps et que "l'économie" de neurones RBF n'a pas de sens, nous ne considérerons pas cela comme un élément positif en faveur de l'algorithme EM.

Le meilleur de nos résultats reste en dessous des résultats d'un entraînement global présenté dans la table (tab.5.5). Ne connaissant pas les conditions dans lesquelles ont été efféctués les testes de la table (tab. 5.5) nous avons aussi fait nos propres testes sur se jeux de données. Nous obtenons une moyenne de 46.1% avec 100 neurones placé à l'aide de K-Means. Ce résultat est donc meilleur que ce que nous obtenons à l'aide de notre méthode en trois phases.

Conseils de RBFNNs locaux

Les résultats obtenu à l'aide des différentes variantes des conseils de RBFNNs locaux sont relativement décevant. En effet, les résultats moyen ne dépasse pas les 40%. De plus, on ne peut pas dire, comme c'était le cas pour le jeu de teste Wine, que le conseil soft ne convient pas à LVQ-SVD. En effet, sur ces données, les meilleurs résultats sont obtenu par le conseil soft, que l'algorithme d'entraînement des réseaux locaux soit LVQ-SVD ou OLS. Les meilleurs résultats sont d'environ 39% de classifications correctes, aussi bien pour OLS que pour LVQ-SVD.

Nous ne présentons pas de tableau des résultats car il ne nous semble pas qu'il soit possible de faire des déductions intéressantes sur ces chiffres.

5.4 Discussion des résultats

Il est difficile de tirer des conclusions claires des résultats présenté jusqu'à maintenant. En effet, on ne peut dire qu'elle est la meilleur techniques pour rassembler les réseaux puisque les meilleurs réseaux globaux ne sont pas les mêmes selon les expériences.

5.4.1 Remarques générales

Une des rares affirmations que l'on peut faire de façon certaine, est que calculer la moyenne des sorties des réseaux locaux pour obtenir la réponse du réseau global n'est pas une bonne technique. Ce résultat n'est pas très surprenant car les réseaux experts dans des zones éloignées de celle d'où provient la donnée à classé auront des réponses faibles et souvent fausses qui constitueront un bruit. Rappelons que généralement, les neurones RBF ont la propriétés d'être locaux, c'est à dire de réagir uniquement aux entrées faisant partie d'une région de l'espace bien déterminée. Ce n'est pas forcement le cas des réseaux entraînés à l'aide de l'algorithme de descente du gradient dont une partie des neurones peuvent avoir un champ de réaction très large (c'est-à-dire des variances très élevées) ([1]). Ceci explique pourquoi, lors des expériences, la moyennes des réseaux de neurones locaux entraîné à l'aide de GD permet d'obtenir de moins mauvais résultats que si les réseaux locaux ont été entraîner avec d'autres algorithmes.

En observant de près le déroulement des expériences, nous avons constater que l'entraînement des réseaux locaux à l'aide de la version primitive du bootstrap qui a été implémentée s'arêtait pratiquement toujours à des pourcentages de classifications correctes très élevé, souvent supérieur à 90%. Cela semble normal dans le cas de problèmes simples comme Iris et Wine. Par contre, pour un problème compliqué comme le Vowel, un taux si élevé, et donc une très grande différence entre le résultat sur les données d'entraînement et sur celles de testes, implique de façon pratiquement certaine un overfiting. Pour remédier à ce problème, il est possible de spécifier que l'algorithme doit s'arrêter si le taux de succès dépasse un certain pourcentage. Le défaut de cette façon de faire et que l'on introduit un nouveau paramètre dont la valeur optimal dépendra de la difficulté du problème. Dans tout les cas, il vaudrait la peine de se préoccupé plus de cette partie est d'implémenter une des nombreuses versions plus sophistiquée du bootstrap.

5.4.2 Groupement lors de la première phase

Problème

La principale surprise, désagréable, a été de constater que, systématiquement, les performances sont meilleurs lorsque la première phase est un "Random Clustering" que lorsque on essaye de "préparer le terrain" à l'aide d'un K-Means ou d'un EM. Pour commencer, nous avons pensé qu'il s'agissait d'une erreur. Nous avons donc vérifier tout les paramètres, refait une partie des expériences mais il faut bien se rendre à l'évidence : l'entraînement en trois phase donne des meilleurs résultats lorsque le clustering est de type aléatoire c'est-à-dire que les centres des groupes sont choisis au hasard parmi les données d'entraînement et que les groupes sont formé autour de ces points selon le critère de la distance euclidienne minimum. Il nous faut donc essayer de trouver une explication.

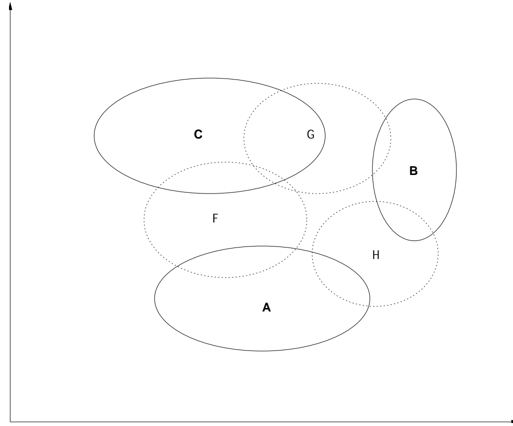


FIG. 5.2 – On a trois classes. Un algorithme de clustering non supervisé comme EM fera évolué les centres en direction des zones de l'espace dans lesquelles il y a le plus d'instances semblables, les point A, B et C par exemple. Les zones entourant les points F, G et H seront dépourvue de centres. Ce sont pourtant des zones difficiles dans lesquelles les instances se mélangent.

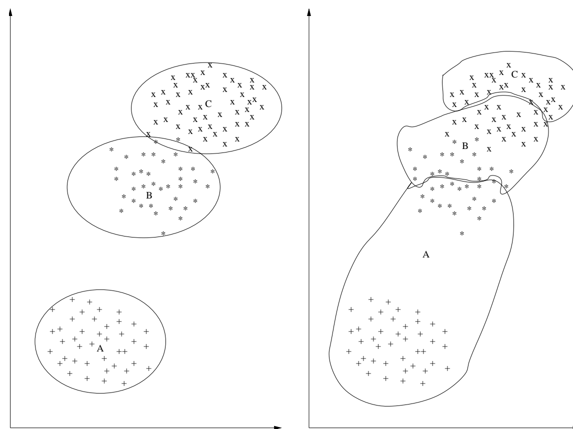


FIG. 5.3 – Sur cette figure qui représente un problème de classification simple, on voit deux manières différentes de grouper les données. A gauche, les donnée sont groupé de façon plus naturels. C'est ce clustering qui est obtenu à l'aide de EM. A droite, les données sont groupée selon un "anticlustering" (voir texte). Les centres des clusters sont situés sur les séparations des classes.

Hypothèse

Une hypothèse peut être formulée. Lorsque les groupes sont formés à l'aide d'un algorithme comme *EM*, on aura tendance, c'est le but recherché, à avoir principalement les points d'une seule classe, ou, au moins de peu de classes, dans un groupe donné. On essaye, dans un certain sens, de faire une première classification non supervisée pour faciliter la tâche à la classification supervisée qui suit. En agissant ainsi, on place les centres de nos groupes dans les zones de l'espace ou il y a une grande concentration d'instances. La proximité dans l'espace des entrées et une indication importante sur la classe de l'instance. N'oublions pas que les meilleurs résultats sur le jeux de teste Vowel ont été obtenu par un simple k-nearest neighbours. En plaçant les centres dans les zones a haute concentration d'instances semblables, on place les centres dans des zones relativement facile de l'espace des inputs et on fait que la plupart des instances d'un groupe donné appartienne à la même classe. Si au lieu de cela on essayait de placer les centres *entre* deux zones à grande concentration, on pourrait peut-être améliorer les résultats. En effet, les zones difficiles sont celles ou les données de deux groupes se mélangent.

Prenons l'exemple le la figure (fig. 5.2). Sur cette figure, on a représenté très schématiquement et de manière très simplifiée un problème de classification entre trois classes. Chaque classe a une zone de grande concentration représentées par les traits plein. Un algorithme non supervisé comme le K-Means ou l'*EM* fera évolué les centres en direction des points A, B et C. Les zones représentées par les traitillé, dans lesquelles il y a un moins haute concentrations, sont les zones de séparations entre les classes. Ce sont les parties de l'espace des entrées pour lesquelles la classification est la plus difficile car les instances de plusieurs classes s'y mélangent. C'est dans ces zones qu'il faudrait avoir plus de centre pour pouvoir dissocier les classes. Il serait donc intéressant d'avoir un algorithme qui fasse évoluer les centres en direction des points F, G et H.

Partant de cette idée, nous avons essayer de trouver rapidement un algorithme qui place les plus de centres dans les régions situées aux séparations entre les classes. Dans notre cas, il ne s'agit pas de placer les centres, mais plutôt les experts. En effet, les algorithme de clustering que nous utilisons lors de la première phase ont pour but de former des groupes d'instances semblables pour lesquels on entraînera un expert lors de la deuxième phase . Au lieu de chercher à faire un expert d'un groupe, nous allons essayer de faire un expert d'une séparation entre deux groupes d'instances semblables. Pour cela, nous avons utilisé l'algorithme LVQ pour trouver les centres des groupes puis nous mettons notre expert, notre nouveau centre, entre deux centres de LVQ. Pour fini, nous faisons le clustering selon la distance euclidienne des instances au nouveaux centres. Cette façon de faire aura tendance à placer plus d'experts centré au bord des concentrations et moins au centre. De cette manières, nous espérons obtenir des RBFNNs locaux spécialisés dans la séparation de deux classes. Nous appellerons cet algorithme "anticlusterer". La figure (fig. 5.3) montre un exemple de comment se comporterait *EM* et notre anticlusterer dans un cas simple. Évidemment, cet algorithme est très primitifs et pourrait probablement être amélioré sans peine. Mais c'est une première approximation permettant de tester la plausibilité de notre hypothèse.

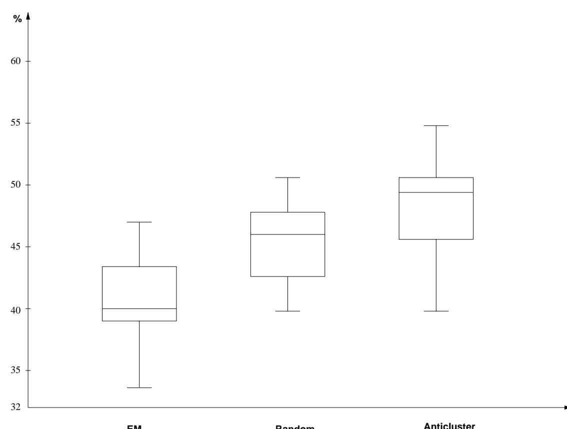


FIG. 5.4 – Cette figure représente un boxplote d'un entraînement en trois phases sur le jeu de données Vowel. Pour la première phase, on a appliqué soit EM, soit un Random clustering soit un "anti-clustering" (voir explication dans texte). Pour la seconde phase, on a utilisé LVQ-SVD et pour la troisième phase la fusion des RBFNNs locaux. (voir aussi 5.7).

Expériences

Malheureusement, le temps nous a manqué pour faire des testes complets avec cette nouvelle technique. Néanmoins, nous avons fait quelques expérimentations sur le jeu de donnée Vowel. Les résultats semblent indiquer que notre nouvelle hypothèse n'est pas fausse. Nous l'avons testée en entraînant 15 réseaux selon notre méthode en trois phases. Comme algorithmes de clustering pour la première phase, nous avons utilisé EM qui compose des groupement "naturel" d'instances, le Random Clusterer et notre anticlusterer qui fait quelques choses qui va dans une direction que l'on peut considérer d'une certaine manière comme contraire à ce que fait EM. On pourrait donc estimer que l'ordre EM - Random - anticlusterer représente une graduation du clustering le plus "naturel" au moins "naturel".

Les résultats que nous avons obtenu (fig. 5.4) semble indiquer que notre hypothèse est correcte. On voit en effet clairement que la moyenne de classifications correctes est nettement supérieur si l'algorithme utilisé lors de la première phase est notre anticlusterer. La moyenne est de 48.4% ; c'est un résultat tout à fait honorable sur un jeu de donnée aussi difficile que le Vowel. Il est aussi supérieur au résultat d'un entrainement global que nous avons obtenu. De plus, le meilleur résultat sur les 15 entraînements est de 54.79% c'est-à-dire mieux que le résultat présenté dans le (tab 5.5) pour les RBFNNs. Le nombre moyen de neurones utilisé est de 162. Ceci justifierait le constat que nous avons fait lors des expériences comme quoi le Random Clustering menait à de meilleurs résultats que les autres. Avec notre anticlusterer, nous obtenons des résultats significativement meilleurs que avec le Random Clustering. C'est un constat agréable car il était dérangeant de penser que laisser la chance décider procure de meilleurs résultats que notre travaille.

Évidemment, il est impossible de développé une théorie sur la base de 15 expériences, mais cette pistes semble intéressante et vaut la peine d'être suivie. De plus, il faut tenir compte du fait que l'anti-clustering proposé a été développé très rapidement et est relativement primitif. Il, peut sûrement être améliorer ce qui apporterait peut-être une augmentation du pourcentage de classifications correctes.

performance \ algorithme utilisé pour la première phase	EM	Random	Anticlusterer
Pourcentage moyen de classification correcte	41.04	45.51	48.4
Écart-type à la moyenne	3.96	3.48	3.88

Tab. 5.7 – Vowel : Différence de performance selon l'algorithme de clustering (EM-Radom-Anticlusterer).
Pourcentage de classification correcte sur le jeu de données Vowel en fonction de l'algorithme utilisé pour le clustering de la première phase. On peut constater que les résultats s'améliorent en fonction de la capacité de l'algorithme à placé des centres entre les clusters naturels, c'est-à-dire à faire le contraire du travail effectuer généralement par EM. Utiliser un EM lors de la première phase est donc clairement contre-productif. (voir aussi 5.4).

Chapitre 6

Conclusions

6.1 Conclusions scientifiques

6.1.1 Avantages et inconvénients de l'entraînement en trois phases

Avant même de parler de résultats, soulignons quelques points intéressants de l'entraînement en trois étapes proposé dans ce chapitre.

Le fait de décomposer le problème en plusieurs parties à résoudre indépendemment les unes des autres est intéressant pour plusieurs raisons. La première de ces raisons est la simplification du problème qui devrait en résulter. On peut espérer qu'une partie du problème sera plus simple à résoudre que le problème dans son entier. Le second avantage de cette décomposition est plus pratique que théorique mais son importance n'est pas moindre : en décomposant l'entraînement du réseau global en une série d'entraînements de réseaux locaux indépendant, on obtient un algorithme qui, de part sa nature même, se prête très bien à une résolution *parallèle*. Cette propriété est très importante du point de vue informatique car elle permet d'accélérer significativement l'entraînement. Il est très facile d'imaginer l'implémentation multi-processus ou multi-threads d'un entraînement en trois phases sur une machine à plusieurs processeurs ou sur un réseau de machine mono-processeur. De plus, si la technique utilisée pour rassembler les réseaux est une de celle présentée sous le nom de Conseil de RBFNNs locaux, rien n'empeche de généraliser l'entraînement en trois phases à d'autres type de réseaux de neurones, voire plus généralement d'autre types de "classifier". Le "diviser pour mieux régner" est une très vieille idée dont les domaines d'application sont infinis.

L'utilisation du bootstrap (3.3.3) permet, d'une certaine manière, d'éviter les difficultés de sélection de l'ordre du modèle pour les réseaux locaux. Par contre, l'entraînement en trois phases proposé implique un autre type de sélection tout aussi délicat : Combien faut-il faire de groupes lors de la première phase ? Certains algorithmes, comme le *Maximum Certainty Data Partitioning* proposent une façon de trouver le nombre de partitions le plus probable. Ce n'est pas le cas de la plupart des algorithmes utilisés dans ce projet. Le problème, classique des méthodes d'approximations, reste donc entier.

6.1.2 Résultats

L'idée est donc sans aucun doute intéressante, mais qu'en est-il des résultats dont l'amélioration reste tout de même le principale objectif d'une nouvelle méthode d'entraînement. Pour commencer, nous avons été un peu surpris, et déçu, des résultats obtenus. Ceux-ci se situent, au mieux, au niveau des performances des algorithmes d'entraînement globaux et parfois en dessous.

Cela n'implique pas que la méthode d'entraînement en trois phases n'est pas bonne.

Une des idées essentielles sur lesquelles nous sommes partis est la construction de réseaux qui soient des experts locaux. Cette idée, après avoir mené le projet à son terme, reste une idée de base très intéressante. Par contre, il semble que les critères de découpage des zones entre RBFNNs locaux (phase 1) sont à revoir. Nous pensons avoir démontrer dans ce travail que le découpage le plus intéressant n'est pas celui qui se base sur les groupes "naturel" comme le fait (dans le cas d'une distributions pouvant s'approximer à l'aide d'un mélange de gaussiennes) EM. C'est un résultat important. Nous avons émis l'hypotèse qu'un autre type de partitionnement basé sur la recherche des zones de séparation entre classes pourrait être utilisés. Cette hypothèse semble être confirmée par l'expérience alors même que l'algorithme proposé est imparfait et pourrait être amélioré.

6.2 Travail futur

Ce projet ouvre un chemin, une piste que l'on pourrait continuer à suivre en réalisant un autre projet de la même ampleur.

Du point du vue théorique, il serait intéressant de reconsidérer les critères du clustering de la première phase. En effet, il semble que leur influence sur les résultats finaux d'un entraînement en trois phases soit considérable. Cette piste parait intéressante et très plaisante à suivre.

Du point de vue pratique, beaucoup de chose pourrait être réalisée en continuation de ce qui a été fait. Il serait intéressant par exemple, de généraliser les algorithmes au traitement des valeurs perdues et des types énumérés. Il existe dans la littérature plusieurs techniques qui traitent ces problèmes.

En ce qui concerne l'implémentation informatique, l'implémentation parallèle de l'entraînement en trois phases seraient un plus évident qui permettrait d'exploiter une des caractéristiques fondamentale de cette nouvelle méthode, à savoir la division du problème.

6.3 Conclusions personnelles

Sur le plan personnel, je peux dire que ce projet m'a beaucoup intéressé. Et cela pour plusieurs raisons.

Premièrement, les réseaux de neurones représentent un domaine de l'informatique qui m'attirent et qui me semble très prometteur. Malheureusement, les applications sont encore relativement rares et il n'est pas facile de trouver un emplois qui soit en rapport avec se domaine en dehors du domaine universitaire. J'étais donc très content de pouvoir profiter de l'occasion de m'imméger durant trois mois dans ce monde.

De plus, c'est un domaine qui se trouve à cheval entre l'informatique, les mathématiques, et biens d'autres domaines. Ayant une mentalité et un esprit plutôt de type polyvalent que spécialiste, je me suis senti à l'aise dans cette branche. D'autant plus que, après avoir passé quatre mois immergé

dans les RBFNNs, je me sent tout de même un peu spécialiste... d'une branche qui demande de la polyvalence.

Au vue des résultats, l'envie de continuer la recherche sur la base de ce travail est très présente et il j'aimerais beaucoup pouvoir y consacrer une partie de mon temps.

6.4 Coût

Il serait un peu absurde d'essayer de mettre un coût économique à un projet comme celui-ci qui tient plus de la recherche que d'une réalisation pratique. Nous donnerons donc plutôt une estimation de la répartition du temps entre les différentes tâches.

Pour commencer, un important travail de documentation a du être réalisé. Mes connaissances sur les réseaux de neurones en général et les RBFNNs en particulier étaient faibles alors que le clustering était un domaine qui m'était pratiquement inconnu. Il m'a donc fallu lire beaucoup d'articles et une grande partie de l'incontournable livre de C. Bishop [1].

L'implémentation, à l'aide du langage de programmation Java, des différents algorithmes utilisés tout au long de ce projet a été une tâche très consommatrice en temps. En effet, programmer des algorithme tel que l'*Expectation - Maximization* ou le *Maximum Certainty data Partitioning* implique une compréhension en profondeur du fonctionnement de ces derniers et pose des problèmes relativement complexes.

La partie d'expérimentation, comprenant le choix des données, leur pré-traitement, et la récolte des résultats occupe elle aussi une bonne place dans mon projet.

L'analyse des résultats, qui comprend le calcule de moyennes, l'écart-types, et le choix des valeurs intéressantes n'est pas négligeable. D'autant plus que dans certain cas, l'analyse des résultats nous a donné l'envie de réaliser d'autres expériences pour confirmer ou infirmer une hypothèse suggérée par les premières expériences. Ce processus cyclique, typique de la recherche scientifique, peut prendre plus de temps que prévu au départ.

Pour finir, l'écriture du mémoire peut être considéré comme un travail relativement important tenant compte du fait que ce projet est plutôt théorique et qu'il y avait un grand nombres d'algorithmes a décrire.

Au temps passé devant des livres ou devant l'ordinateur, on pourrait ajouter un certain nombre d'heures de réflexion, impossible à quantifier, dans le métro, sous la douche et durant une grandes parties des activités qui laisse libre notre cerveau.

Au total, le temps passé sur ce projet est estimé entre 600 et 700 heures réparties sur les quatre mois d'une semestre universitaire. Une répartition approximative du temps passé pour chaque tâche est donnée dans la table (tab. 6.1)

tâche \ temps	%	temps en heures
Documentation	22.5	135
Implémentation	30	180
Expérimentation	15	90
Analyse des résultats	12.5	75
Écriture du mémoire	20	120
Total	100	600

TAB. 6.1 – Répartition approximative du temps passé sur les différentes parties du projet.

Liste des tableaux

5.1 Iris : Fusion de RBFNN locaux. 39
5.2 Iris : Conseil de RBFNNs locaux. 40
5.3 Wine : Fusion de RBFNNs locaux. 41
5.4 Wine : Conseils de RBFNNs locaux. 42
5.5 Vowel : Résultats fourni avec le jeu de données. 43
5.6 Vowel : Fusion de RBFNNs locaux. 44
5.7 Vowel : Différence de performance selon l'algorithme de clustering (EM-Radom-Anticlusterer). 49

6.1 Répartition approximative du temps passé sur les différentes parties du projet. 53

Bibliographie

[1] Bishop C., 1995, *Neural Networks for Pattern Recognition,* Oxford University Press, Oxford.

[2] Chen S., Cowan C. F. N., Grant P. M., 1991, *Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks,* IEEE Transaction on Neural Networks, vol. 2, no. 2., p. 302-309.

[3] Chinrungrueng C., Séquin C. H., 1995, *Optimal adaptive K-Means Algorithm with Dynamic Adjustment of Learning Rate,* IEEE Transaction on Neural Networks, vol. 6, no. 1., p. 157-168.

[4] Ghahramani Z., Jordan M., 1994, *Learning from incomplete data,* MIT, Massachusetts.

[5] Haykin S., 1999, *Neural Network : a Comprehensive Foundation*, Prentice Hall, New Jersey.

[6] Kohonen T., Hynninen J., Kangas J., Laaksonen J., Torkkola K., 1995, *The Learning Vector Quantization Program Package,* Helsinki University of Technology, Finland

[7] McLachlan G., Krishnan T., 1997, *The EM Algorithm and Extensions*, Wiley & Sons, New York.

[8] Morgenthaler Stephan, 1997, *Introduction à la statistique,* Presses polytechniques et universitaires romandes, Lausanne

[9] Roberts S. J., Everson R., IHead R., 1999, *Maximum Certainty Data Partitioning*, Pattern Recognition

[10] Roberts S. J., Holmes C., Denison D., 2000, *Minimum-Entropy Data Clustering using Reversible Jump Markov Chain Monte Carlo.*

[11] Robinson A. J., 1989, *Dynamic Error Propagation Networks*, Cambridge University Engineering Department, Thèse doctorale.

[12] Press W.H., Flannery B.P., Teukolsky S.A, Vetterling W.T., 1992, *Numerical Recipes in C : The art of scientific computing*, Cambridge University Press.