Université de Montréal

Boosting hiérarchique et construction de filtres

par
Marc-Olivier LaBarre

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2007

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Boosting hiérarchique et construction de filtres

présenté par:

Marc-Olivier LaBarre

a été évalué par un jury composé des personnes suivantes :

Philippe Langlais,	président-rapporteur
Balázs Kégl,	directeur de recherche
Yoshua Bengio,	codirecteur
Miklos Csüros,	membre du jury

Mémoire accepté le: 18 juin 2007

RÉSUMÉ

Le but du présent mémoire est d'explorer le potentiel qu'une structure hiérarchique apporterait aux algorithmes de *boosting*. Le boosting est souvent décrit comme un méta-algorithme, dans le sens où il s'agit d'une technique qui améliore la précision d'un algorithme d'apprentissage en en joignant plusieurs. Le but est simple : à partir d'une hypothèse, est-il possible de *booster* son efficacité en lui joignant une autre hypothèse, celle-ci entraînée spécifiquement pour pallier les faiblesses de la première ?

La technique de boosting est habituellement considérée comme une méthode d'ensemble, où plusieurs hypothèses votent (avec des pondérations différentes) sur un exemple. Le vote pondéré de ces hypothèses "faibles" constitue l'hypothèse "forte", le résultat du boosting. Un autre point de vue est celui décrit par l'algorithme de boosting lui-même, par l'évolution de la distribution de poids sur les données d'entraînement. Ainsi, chaque itération de boosting ajoute une hypothèse entraînée spécifiquement à améliorer la décision de l'hypothèse forte. Ces deux visions montrent la création d'une série d'hypothèses parallèles. Si nous représentons les hypothèses faibles comme de simples unités de traitement, ou encore comme des neurones, il est facile de voir le processus de boosting comme la génération d'une "couche" d'hypothèses. Avec les connaissances que nous avons au sujet des réseaux de neurones, nous poussons la question plus loin : "Pourquoi une seule couche ? Pourquoi pas plusieurs ?"

Nous répondons à ces questions par un nouvel algorithme, inspiré d'AdaBoost, qui construira un "réseau" d'hypothèses, plutôt qu'une seule couche. Nous décrivons dans ces pages ce nouvel algorithme, appelé *hboost* (pour *hierarchical boosting*), en prenant soin d'identifier les bases théoriques qui ont mené à sa création. Nous présentons également une méthode de groupement de caractéristiques inspirée par le traitement d'images, méthode qui sera utile au jeu hiérarchique de l'algorithme. Nous présentons également une comparaison de l'utilité d'architectures différentes pour le boosting à travers des expériences sur des jeux de données crédibles.

Mots clés: Réseaux de neurones, Apprentissage supervisé, Reconnaissance d'images, Classification multiclasse, AdaBoost, Prétraitement de données

ABSTRACT

This thesis' goal is to explore the potential of giving a hierarchical structure to boosting algorithms. Boosting techniques are generally described as a meta-algorithm, that is, an algorithm working to improve (boost) the accuracy of some learning algorithm, like neural networks or decision trees. Starting with one hypothesis, boosting tries to improve its power by training successive hypotheses, each one specializing on correcting the faults and limitations of the previous ones.

Boosting is usually considered as an ensemble method (like bagging), where many hypotheses vote (with different weights) on an example. The weighted votes of these "weak" hypotheses make the "strong" hypothesis that boosting produces. Another point of view on boosting comes from the algorithm itself, by the evolution of the weights on the training data. That way, each iteration of boosting adds a new hypothesis, which was specifically trained to improve the decision of the sum, considering the weak hypotheses already computed in the previous iterations. These two visions of boosting show the creation of a series of "parallel" hypotheses. If we represent these hypotheses as "black box" computing units, or neurons, it's easy to see the process of boosting as the generation of one "layer" of hypotheses. With our prior knowledge of neural networks, we raise the question: "Why one layer? Why not more?"

We answer these questions with a new algorithm inspired by AdaBoost, which will build a multi-layered "network" of hypotheses, rather than only one layer. We describe in these pages this new algorithm, named *hboost* (for *hierarchical boosting*), highlighting the theoretical bases that lead to such a creation. We present a comparison of what multiple layers can add to AdaBoost through experiments on benchmark datasets.

Keywords: Neural Network, Supervised learning, Image recognition, Multiclass classification, AdaBoost, Preprocessing

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

NOTATION

Général :

$\mathbb{R}$	L'espace des réels
d	Nombre de dimensions de l'espace d'entrée
$\mathbf{X} \subset \mathbb{R}^d$	L'espace des données
c	Nombre de dimensions de l'espace des étiquettes
$\mathbf{Y}$	L'espace des étiquettes
$\mathbf{x} \in \mathbf{X}$	Un point de donnée d'entrée
$\mathbf{y} \in \mathbf{Y}$	Une étiquette d'un point de donnée
n	Le nombre de données
$\hat{R}$	Erreur de classification

AdaBoost :

T	Le nombre d'itérations d'AdaBoost
t	L'itération courante
D_t	Distribution sur les données à l'itération t
h_t	L'hypothèse faible générée à l'itération t. $h_t : X \rightarrow Y$
α_t	La confiance sur la prédiction de h_t
ε_t	L'erreur de l'hypothèse h_t selon D_t
H	Hypothèse forte. $H : X \rightarrow Y$

hboost :

L	Nombre de couches d'hypothèses
l_s	Nombre d'unités sur la couche s
$D_{t,s}$	Distribution sur les données à l'itération t pour la couche s
$h_{t,s}$	L'hypothèse faible de la couche s générée à l'itération t
$\alpha_{t,s}$	La confiance sur la prédiction de $h_{t,s}$
$\varepsilon_{t,s}$	L'erreur de l'hypothèse $h_{t,s}$ selon $D_{t,s}$
H	Hypothèse forte. $H : X \rightarrow Y$
u_i	i^e unité (hypothèse) d'une couche.
ω_i	poids de l'unité u_i dans un filtre.
ω	le vecteur des poids des unités dans un filtre. $\omega = \{\omega_1, \ldots, \omega_l\}$
$f(u_i)$	fonction théorique du poids dans un filtre pour l'unité u_i.

REMERCIEMENTS

Cette thèse n'aurait pas été possible sans l'aide et les commentaires de mon directeur, Balázs Kégl. Je tiens également à remercier l'ensemble des membres des laboratoires LISA et GAMME de l'Université de Montréal, tant pour leurs aide, leurs commentaires, ou simplement leur discussion. Je reconnais gratieusement le support du Conseil de Recherche National en Sciences Naturelles et en Génie du Canada pour son aide financière.

Je souhaite également saluer mes parents, mes deux soeurs, et surtout Marie-Christine, qui ont toujours été là pour m'aider.

CHAPITRE 1

INTRODUCTION

La technique de boosting est maintenant bien connue dans le domaine de l'apprentissage machine. Le boosting est en fait un méta-algorithme – du fait qu'il ne s'agit pas d'un algorithme d'apprentissage complet, mais plutôt d'un algorithme qui utilise les résultats d'un autre algorithme d'apprentissage. Quoiqu'il existe plusieurs algorithmes de boosting différents et polyvalents, les applications techniques demandent toutes de *booster* "quelque chose", habituellement des arbres de décision ou des réseaux de neurones. Le boosting sert à améliorer les performances d'une hypothèse quelconque (la fonction apprise d'un algorithme d'apprentissage), en lui ajoutant de manière parallèle d'autres hypothèses apprises, chacune de celles-ci entrainée spécifiquement à combler les faiblesses et les limitations des hypothèses précédentes.

Cette façon de faire est simple et bien connue. On ajoute de nouvelles hypothèses de manière progressive et parallèle, toutes celles-ci étant sur un même niveau, dans le sens où les entrées sont les mêmes, et les sorties sont combinées ensemble. Cependant, lorsqu'on représente graphiquement le processus de boosting, on peut voir nos hypothèses comme faisant partie d'une "couche". Chacune de ces unités ayant des entrées et des sorties facilement identifiables, on peut représenter une hypothèse comme un neurone "boîte noire" (au comportement interne non spécifié). Par cette analogie, le boosting est dépeint comme une méthode de génération de réseaux de neurones[1]. Plus exactement, on génère un réseau de neurones d'une couche. D'où la question : comme la puissance d'une seule couche de neurones est souvent limitée et que l'on contourne habituellement ce problème avec une structure hiérarchique de couches superposées, serait-il possible de faire de même avec le boosting ? En d'autres termes, serait-il possible de doter le boosting d'une technique pour créer et gérer plusieurs couches d'unités apprenantes ? La réponse est oui, et nous présentons l'algorithme résultant dans ce mémoire.

[1]Ici, réseau de neurone est utilisé au sens large, et n'implique que la symbolique d'un réseau de neurones, et non son application pratique.

Le présent chapitre introduit l'environnement théorique de l'apprentissage machine dans lequel le boosting se situe, ainsi que certains concepts importants du domaine[2]. Nous décrivons également l'organisation du mémoire et sa répartition en chapitres.

1.1 Apprentissage machine

L'apprentissage machine (*machine learning* en anglais) est la formalisation informatique de théories psychocognitives de l'apprentissage. Son utilité est très simple : faire apprendre une machine, un ordinateur ou un robot, d'une manière semblable au processus d'apprentissage humain. Bien qu'on réfère souvent aux ordinateurs comme étant des entités "intelligentes", on pense surtout à leur caractère de logique mathématique ainsi qu'à leur mémoire "précise". Si ces deux éléments, la capacité de raisonnement logique et la connaissance, sont effectivement des composantes majeures de l'intelligence, le domaine de l'apprentissage et de l'adaptativité y sont également importants.

La notion d'apprentissage, tant pour un être humain que pour une machine, peut être essentielle à plusieurs activités. Il n'y a qu'un ensemble relativement limité, bien que très grand, de connaissances précises et définitives, dont l'application pratique est triviale et automatique. Il existe également un très grand nombre de situations non ou seulement partiellement documentées, et cette information n'est pas toujours disponible au moment opportun. D'un autre côté, les capacités mémorielles, que ce soit pour un humain ou une machine, sont limitées. Privé d'une source illimitée de données fiables et précises sur tout ce qui existe (et tout ce qui pourrait exister), il devient important pour notre machine de pouvoir apprendre, de pouvoir s'adapter à la tâche voulue[3]. L'informatique traditionnelle a souvent comme réponse à des problèmes d'encoder une solution fixe, une solution logique et claire, généralement le reflet de l'intelligence humaine. Cependant, plusieurs processus importants du cerveau animal sont difficilement transmissibles à l'ordinateur. Par exemple, tout ce qui a trait au langage naturel, ou aux données sensorielles ; bien

[2]Ceux qui sont le plus important pour le boosting.

[3]Dans le cas humain, l'apprentissage est tout simplement essentiel, car il n'est pas possible de transmettre les connaissances de manière "pure". Il faut donc apprendre la grande majorité des connaissances dont on a besoin, et du reste, on passe toute sa vie à apprendre, d'une manière ou d'une autre.

qu'il existe des moyens informatiques efficaces de saisir des données auditives ou visuelles, l'interprétation de ces signaux est loin d'être facile. Il y a aussi le problème de "l'intuition" d'un expert, qui se base sur des données précises, mais ne suit pas nécessairement un processus cognitif précis qu'il serait possible de traduire en algorithme. Enfin, il y a tout le reste : faire apprendre des éléments qui sont encore "invisibles" à la compréhension humaine, soit parce qu'ils demandent une plus grande capacité de traitement que ne le permet le cerveau biologique, soit simplement parce que, pour une raison ou une autre, personne n'y a songé avant.

L'apprentissage machine est généralement rattaché au domaine de l'intelligence artificielle, mais ses ramifications scientifiques sont plus nombreuses. Certaines techniques d'apprentissage viennent des statistiques et de l'analyse de données, alors que d'autres sont inspirées directement de l'intelligence "biologique", et tiennent de la neuroscience et de la psychologie cognitive. L'apprentissage machine, comme sa contrepartie "humaine", se divise en plusieurs philosophies. On peut parler d'apprentissage supervisé, d'apprentissage non supervisé, d'apprentissage semi-supervisé et d'apprentissage par renforcement.

L'apprentissage supervisé s'apparente à l'apprentissage scolaire classique. On possède des problèmes avec leurs solutions, et on entraîne la machine à donner la bonne réponse au problème, en supervisant l'entraînement : si le système donne une réponse correcte, on "récompense" son fonctionnement, sinon on le "punit". L'apprentissage dans ce cas est le moyen de modifier la prise de décision en fonction de ces récompenses. L'apprentissage semi-supervisé est similaire, mais plus distant : on possède les étiquettes (réponses) d'une petite partie des données, mais la plupart nous sont inconnues. On doit alors se baser sur la similarité de problèmes déjà rencontrés (avec étiquettes) pour estimer les nouvelles étiquettes. Ici, l'apprentissage demande déjà une qualité de généralisation, et il n'est pas possible de seulement apprendre par coeur les associations entre les problèmes précis et leur solutions exactes. Le cas de l'apprentissage non supervisé est un apprentissage plus abstrait : on ne peut pas apprendre de bonnes "réponses", car on n'a pas accès à cette information, si même elle existe. On veut plutôt apprendre directement le domaine des exemples. Cette vision d'apprentissage sert plutôt à modéliser

une distribution des cas possibles, d'une manière statistique. On peut aussi se servir de ce type d'apprentissage pour effectuer une classification "naturelle", c'est-à-dire séparer nos exemples dans des catégories distinctes, mais non symboliques a priori ; on peut toujours leur donner un sens par la suite. L'apprentissage par renforcement est plutôt différent : c'est le type d'apprentissage plus "robotique". Avec un agent (notre robot) dans un environnement, on n'obtient pas de réponse, de récompense, de nos actions directement, mais seulement lorsque le but ultime est atteint. Comme il faut une suite d'actions pour arriver au résultat, on doit donc trouver un moyen de diffuser l'apprentissage dans le "passé", de manière à ce que le comportement global soit récompensé (ou puni), et non seulement la dernière action.

Les problèmes et les algorithmes abordés dans ce mémoire ont trait à l'apprentissage supervisé[4], et la discussion qui suit se concentre sur cet aspect. Du point de vue de programmation informatique, l'apprentissage supervisé peut se définir grossièrement comme la construction d'une fonction qui donnera la sortie désirée pour les exemples présentés, et donnera de bonnes réponses sur de nouveaux exemples. Le but est bien sûr de trouver un lien entre les exemples (les données) et l'étiquette qui leur est attribuée. Les données fournies sont généralement de forme vectorielle, contenant à la fois les données et la réponse, l'étiquette, c'est-à-dire la sortie désirée. Une donnée $\mathbf{x}_i$ est généralement identifiée comme suit : $\mathbf{x}_i = (x_1, \ldots, x_d, y_1, \ldots, y_c)$, où d représente la dimension des données d'entrée et c la dimension des données de sortie[5].

Il existe deux types de tâches courantes : la classification et la régression. Le but de la classification, aussi appelée discrimination ou encore dans certains cas, reconnaissance de formes, est de déterminer l'appartenance d'un objet à une classe précise, classe qui représente généralement un sous-ensemble des données ainsi qu'une partie de l'espace vectoriel dans lequel les données sont placées. Le cas de classification binaire est le plus simple ; on doit départager deux classes, ou encore, en identifier une seule par rapport aux "autres". Le cas multiclasse est plus complexe. Dans le cas où chaque exemple ap-

[4]Leur application aux autres types d'apprentissage n'est cependant pas exclue, mais il dépasse le but de ce document.

[5]De manière courante, c sera 1. S'il est supérieur à 1, on parlera de classification multi-dimensionnelle, ou encore de multirégression.

partient à une seule classe, on doit identifier une classe non seulement par rapport à toutes les autres, mais en même temps départager ces autres classes entre elles. Dans les cas multiclasse à une seule étiquette (par exemple $y_1 \in 1,2,3,4$), on décompose souvent cette étiquette (multiclasse) en plusieurs étiquettes binaires. Ceci est non seulement plus simple du côté de la représentation, mais également du côté informatique tel quel. En effet, plusieurs valeurs peuvent être représentées comme choix discrets d'entiers, mais du côté de la machine, il est plus simple de ne pas s'encombrer de chiffres qui n'ont pas de sens numérique. Par exemple, dans le cas de classification de chiffres (le cas qui nous intéresse principalement), on aura habituellement une seule étiquette qui aura comme valeur possible de 0 à 9. Le problème ici est que pour l'ordinateur, ces valeurs sont numériques, et non simplement le reflet d'une énumération. Alors il est préférable de transformer cette simple étiquette d'énumération en vecteur d'étiquettes binaires, donc l'étiquette 4 deviendrait le vecteur multiclasse $(-1,-1,-1,-1,\ 1,-1,-1,-1,-1,-1)$. Ce faisant, on montre que notre donnée appartient à la classe "4", et n'appartient à aucune autre. Il existe finalement le cas de la classification multiclasse multiétiquette, dans lequel on veut pouvoir associer plusieurs étiquettes à un même exemple.

Le cas d'apprentissage de régression est l'apprentissage d'une fonction pour des étiquettes continues (non discrètes). Au lieu d'identifier l'appartenance à une classe, on désire calculer une mesure quelconque relative aux données, mais sans connaître a priori le lien qui unit les données à leurs mesures. Ce domaine est aussi intéressant que celui de la classification mais nous n'aborderons pas le sujet en profondeur ici. Le boosting, moyennant certaines modifications, permet l'apprentissage de régresseurs, mais nous nous limiterons dans ce document aux tâches de classification.

Le but de l'apprentissage est évidemment d'appliquer le "raisonnement" appris à de nouvelles situations. Le cas trivial d'apprentissage, par coeur, est possible et souvent extrêmement simple pour une machine, mais rarement souhaité. Nous souhaitons surtout que l'hypothèse apprise se généralise bien aux nouveaux exemples, aux situations nouvelles, inédites. C'est pourquoi l'apprentissage machine doit être vérifié et testé sur des données indépendantes. En l'absence de cette vérification, il est impossible d'avoir une assurance sur l'efficacité de la méthode sur des données nouvelles, absentes de l'en-

traînement. C'est pourquoi on sépare l'ensemble de données en deux parties distinctes : un ensemble d'entraînement et un ensemble de test. L'ensemble d'entraînement, comme son nom l'indique, sert à entraîner l'algorithme, et la qualité de l'algorithme sera vérifiée avec l'ensemble de test.

De manière générale, on peut formuler un processus d'apprentissage comme la création d'une fonction f qui prédira avec précision l'étiquette à donner à un exemple. La qualité, la précision, de cet apprentissage est donnée par le taux d'erreur empirique $\hat{R}$ de prédiction sur notre ensemble de test. Formellement, ça se présente sous la forme :

$$\hat{Y}_i = f(x_i) \tag{1.1}$$

$$\hat{R} = \frac{1}{n}\sum_{i=1}^{n} Id_{\{\hat{Y}_i \neq Y_i\}}. \tag{1.2}$$

La nature de la "fonction" apprise automatiquement dépend de l'algorithme et de ses paramètres et hyperparamètres. On peut aussi bien dire qu'un algorithme d'apprentissage cherche une solution dans une classe de fonctions, lesquelles peuvent être "produites" par l'algorithme en question. En apprentissage machine, plus on donne de degrés de liberté à un algorithme, plus il est facile d'apprendre les données d'apprentissage. Ces degrés de liberté représentent la richesse de la classe de fonctions, et en pratique, est proportionnel au nombre de paramètres ajustables dans l'algorithme. Du côté théorique, cette richesse a été étudiée comme la dimension VC [46], pour Vapnik-Chervonenkis, du nom des deux chercheurs qui ont établi cette mesure. Il est important de bien gérer la richesse de la classe de fonctions, car la précision du résultat dépend grandement des capacités de l'algorithme. D'un côté, il faut une assez grande richesse pour bien apprendre les données, c'est-à-dire créer une fonction qui possède la souplesse requise pour bien identifier les caractéristiques voulues des données. Si la classe de fonctions n'est pas assez riche, on obtiendra une fonction plus simple et moins précise. On parlera alors de **sous-apprentissage** *(underfitting)*, dans le sens où on n'a pas appris tout ce qu'on désirait des données. D'un autre côté, il faut éviter le cas opposé, où la richesse de notre classe de fonctions est trop grande. Avec suffisamment de latitude, il devient facile d'apprendre un jeu de données arbitraires. Cependant, cet apprentissage expert vient

souvent au prix d'une très grande complexité de représentation, complexité qui n'est pas souhaitable[6]. Suivant le principe du rasoir d'Occam, on cherche la fonction de qualité la plus simple possible. Dans le cas où deux hypothèses ont une précision similaire, on choisira l'hypothèse la plus simple. Comme le taux d'erreur sur un ensemble d'entraînement est une mesure biaisée (par définition), on s'attend à ce que la solution la plus simple généralise mieux sur de nouveaux exemples. En pratique, on voit souvent que le taux d'erreur sur l'ensemble d'entraînement diminue de manière monotone au fur et à mesure que la richesse de l'algorithme augmente. Cependant, le comportement de la courbe de l'erreur sur l'ensemble de test est plus intéressant. Si on constate qu'au début, celle-ci diminue de manière similaire à la courbe d'erreur d'entraînement, elle atteint normalement un minimum, et puis commence à augmenter, tel que montré dans la figure 1.1. C'est ce qu'on appelle le **sur-apprentissage** *(overfitting)*. Ceci s'explique du fait que l'algorithme se sur-spécialisera sur les exemples d'entraînement, et ce qui est gagné au niveau de l'entraînement est perdu au niveau du pouvoir de généralisation.

Le problème de sur-apprentissage est lié en grande partie à la taille de l'ensemble d'entraînement : plus on a d'exemples, plus on résistera longtemps au sur-apprentissage. Cependant, dans la plupart des cas réels, il est impossible d'avoir un nombre infini d'exemples valides, et il faut donc en conséquence gérer l'équilibre entre le sous-apprentissage et le sur-apprentissage. De manière pratique, sur-apprendre signifie que l'algorithme cesse d'apprendre des éléments significatifs du problème lui-même, et s'attarde sur des éléments insignifiants ou erronés des données. En cherchant à apprendre le mieux possible les données d'entraînement à tout prix, on risque d'apprendre des éléments singuliers de tel ou tel exemple, des coïncidences dues au nombre limité de données disponibles, ou encore du bruit présent dans ces mêmes données. Si en plus il existe des erreurs d'étiquetage dans l'ensemble d'entraînement, c'est-à-dire des exemples qui n'ont pas la bonne étiquette, le problème est plus flagrant. Il n'est maintenant plus souhaitable, théoriquement, d'obtenir un taux d'erreur d'entraînement nul, puisque ceci signifierait que les erreurs d'étiquetages ont été apprises comme si elles étaient correctes.

[6] l'apprentissage par coeur est dans cette catégorie. Bien qu'il soit pratiquement simple d'apprendre par coeur, cet apprentissage demande beaucoup de ressources.

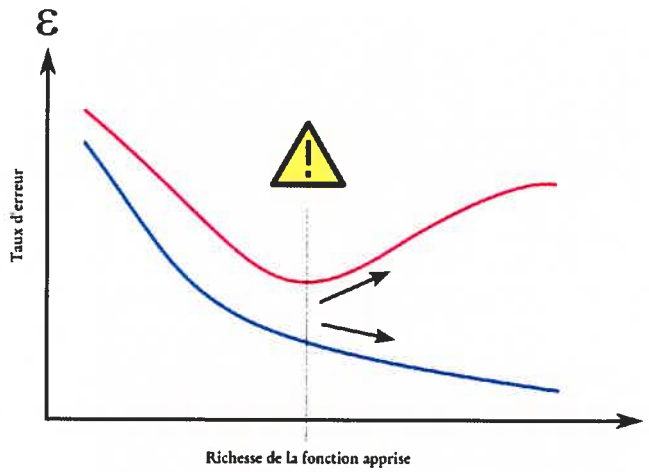

Figure 1.1 – Sur-apprentissage : courbe de l'erreur d'entraînement (bleu) et de l'erreur de test (rouge)

C'est dans ce contexte qu'existe le boosting. Comme nous l'avons déjà indiqué, c'est une technique qui entraîne successivement plusieurs hypothèses complémentaires pour améliorer le fonctionnement d'un algorithme d'apprentissage. Le boosting est déjà connu pour être particulièrement résistant au sur-apprentissage, mais la manière dont il structure les hypothèses qu'il génère est plutôt simple. Dans le cas d'hypothèses elles-mêmes très simples (avec peu de richesse), la structure de boosting, bien qu'efficace, ne crée pas un réseau d'hypothèse qui permettre vraiment d'enrichir la méthode globale. Nous proposons dans les chapitres suivants une variante de boosting qui permet justement d'enrichir la structure dans laquelle les hypothèses sont générées, de manière à pouvoir dépasser, via le boosting, les limitations de hypothèses faibles qui composent le résultat final.

1.2 Structure du mémoire

Le présent mémoire est structuré comme suit : nous allons d'abord énoncer les bases spécifiques de l'apprentissage machine qui sont nécessaires à la compréhension de l'algorithme de boosting hiérarchique. Cette exploration du domaine commencera par une description simple du boosting et de ses raffinements actuels. Nous passerons en re-

vue les différents points de vue sur la nature même du boosting. Nous décrirons ensuite sommairement les réseaux de neurones et les points communs qu'ils partagent avec le boosting. Nous partirons de cette comparaison pour ensuite essayer, sinon de combler ces différences, au moins de nous en inspirer pour enrichir le boosting d'une structure plus complexe, et potentiellement plus riche. Le chapitre 2 décrira ces étapes du cheminement intellectuel qui mène au boosting hiérarchique.

La section 2.4 présente le support représentatif que le traitement d'images peut ajouter à la compréhension d'une structure multicouche pour le boosting. Nous y présenterons certaines techniques communes du traitement d'images et décrivons comment des filtres sont utilisés pour améliorer la reconnaissance des formes. Nous étudions également un cas particulier qui a fait ses preuves lorsqu'il est jumelé au boosting soit les *filtres de Haar.*

Finalement, la section 2.6 présente de manière formelle l'algorithme général du boosting hiérarchique. Nous y définissons les paramètres et hyperparamètres du modèle, nous indiquons quels choix architecturaux doivent être pris en compte ainsi que leurs impacts sur l'algorithme comme tel. *AdaBoost* étant déjà un algorithme connu et apprécié qui s'accompagne de fortes garanties d'efficacité, nous tâcherons de profiter le plus possible de la filiation entre les deux algorithmes.

Nous nous penchons ensuite plus précisément sur les détails de l'algorithme en question. La section 3.1 décrit les formes architecturales qui peuvent être produites par notre algorithme. Les conséquences qui en découlent sont également abordées, ainsi que les libertés que nous permet une structure en couches bien définies. La section 3.2 aborde l'autre aspect important de l'algorithme, soit la géométrie à maintenir pour la génération automatique de filtres, et les différents types de filtres qui peuvent être envisagés. Nous avons deux méthodes pour gérer un espace : par coordonnées ou par distance. Cette géométrie définit ensuite le genre de filtres qu'on pourra utiliser. Les filtres déjà connus y sont définis d'une nouvelle manière, et d'autres filtres simples y sont présentés. Chaque type de filtre présente des contraintes différentes, du point de vue symbolique comme du point de vue pratique d'implémentation.

Ces paramètres offrent une grande quantité de combinaisons possibles. Nous décri-

vons dans le chapitre 4 les expériences menées avec différents ensembles de paramètres. Les résultats obtenus y sont également montrés et analysés.

CHAPITRE 2

DE L'APPRENTISSAGE MACHINE AU BOOSTING HIÉRARCHIQUE

2.1 Boosting

Le boosting vient, à l'origine, d'une question soulevée par Kearns et Valiant dans le modèle d'apprentissage théorique PAC (pour *Probably Approximatively Correct*[16] [45] [1]) quant aux principes d'apprentissage fort et d'apprentissage faible. PAC est un modèle théorique pour appliquer l'étude théorique d'efficacité computationnelle des algorithmes à l'apprentissage machine. Un "apprenant" (le nom que porte un algorithme d'apprentissage dans le modèle PAC) cherche à trouver une approximation bornée d'une fonction à estimer c, avec une grande confiance et ce, dans un temps polynomial. D'abord, la probabilité que l'hypothèse (l'approximation de c) se trompe sur un exemple (l'erreur) doit être inférieure à une valeur arbitrairement petite : $Pr(c(x) \neq h(x)) < \varepsilon$. Ensuite, il faut que l'apprenant puisse générer des hypothèses de cette précision arbitraire de manière consistante, c'est-à-dire, que la probabilité de générer une telle hypothèse est très forte. De ces deux tensions d'apprentissage vient justement le nom du modèle : Probably Approximately Correct.

La définition précédente réfère à un concept c (fortement) apprenable par un algorithme d'apprentissage. Un définition d'apprenabilité moins sévère existe également ; un concept est faiblement "apprenable" (*weak learnability*) si l'"apprenant" peut produire de manière consistante (avec une grande probabilité) une hypothèse qui a une performance meilleure que le hasard. Schapire décrit ces deux principes dans [36] et montre qu'ils sont équivalents, et introduit une technique qui permet de passer d'un apprentissage faible à un apprentissage fort : le boosting.

Venant d'un modèle théorique d'apprentissage machine, les apprenants faibles (*weak learner* en anglais) sont considérés de manière externe. Cette particularité fait du boosting un méta-algorithme d'apprentissage, dans le sens où on "*boost*" habituellement un certain type d'apprenant faible, par exemple des arbres de décisions. Ceci est particu-

lièrement pratique, puisqu'on peut utiliser un algorithme de boosting avec plusieurs apprenants faibles, de manière interchangeable, moyennant que certaines conditions soient respectées.

L'algorithme général de boosting évalue d'un regard extérieur l'optimisation d'une fonction d'apprentissage. Par ceci, on indique que la nature intérieure de l'apprenant n'est pas importante. La seule exigence est qu'il s'agisse d'un apprenant faible valide, donc comme spécifié plus haut, qu'il ait un taux de succès supérieur au hasard. On vise à améliorer la qualité de prédiction d'un apprenant en joignant plusieurs hypothèses entraînées successivement dans ce but.

AdaBoost, de *Adaptative Boosting*, créé par Schapire et Freund en 1995 [13] (aussi [9]), constitue la base du boosting algorithmique : il s'agit du premier algorithme fiable et relativement rapide (en temps polynomial) répondant au modèle théorique du boosting. Pour la suite de ce document, nous utiliserons souvent les termes boosting et AdaBoost de manière interchangeable. Notons que plusieurs algorithmes modernes de boosting sont en fait des variations d'AdaBoost plutôt que des créations entièrement différentes, ce qui est également le cas du nôtre.

Le résultat du boosting est une méta-fonction d'apprentissage, la somme pondérée du résultat de plusieurs fonctions d'apprentissage, ou d'hypothèses, selon le vocabulaire du domaine. Une itération t de boosting construit une nouvelle hypothèse, entraînée, entre autres, à pallier les faiblesses et les manquements des hypothèses précédentes – le résultat de l'algorithme jusqu'à maintenant. On identifie ces points sensibles, c'est-à-dire les points sur lesquels on doit concentrer l'apprentissage, par une distribution D_t sur les données d'entraînement. Cette distribution représente l'importance relative des données par rapport à la tâche présente. On entraînera la nouvelle hypothèse h_t avec la distribution D_t. Si l'entraînement d'une nouvelle hypothèse par l'apprenant faible ne peut utiliser directement cette distribution comme paramètre, on peut alors l'utiliser pour échantillonner les données, d'une manière similaire au *bootstrap*, afin d'obtenir un ensemble d'entraînement adéquat. Le *bootstrapping* est une technique pour générer un nouvel ensemble de données avec un ensemble initial. Il utilise cet ensemble comme une distribution de probabilité sur les exemples, de sorte qu'en sélectionnant uniformément des éléments

de l'ensemble initial, on peut se constuire un ensemble de données différent mais représentant des données semblables. De manière pratique, on pige au hasard des éléments de l'ensemble initial, avec remise. Pour l'appliquer au boosting, on utilise la distribution de poids sur les points de données (au lieu d'une distribution uniforme) ; on parlera alors de boosting par échantillonnage. Pour AdaBoost, la distribution sur les exemples est initialisée uniformément, chacun des n points de données i ayant comme poids initial $D_1(i) = 1/n$. Ces poids sont mis à jour à chaque itération, selon que chaque point a été bien classifié ou non par la nouvelle hypothèse : le poids d'un exemple augmente s'il a été mal classifié, et diminue s'il a été bien classifié, de sorte que les exemples qui sont mal classifiés d'une fois à l'autre verront leur poids augmenter, et que la prochaine hypothèse entraînée devra s'intéresser à eux.

AdaBoost repondère les données de manière à ce que les exemples mal classifiés représentent, après repondération, 50 % du total, et par conséquent, les autres 50 % iront aux exemples bien classifiés. Ce faisant, on s'assure de *décorréler* complètement la distribution des poids D_{t+1} de la dernière hypothèse h_t. Ceci a pour effet immédiat de s'assurer que la prochaine hypothèse se concentre d'abord et avant tout sur les points qui sont plus difficiles à bien étiqueter, mais aussi force l'algorithme d'apprentissage à produire une hypothèse différente de la précédente[1].

La qualité d'une hypothèse h_t est mesurée par son erreur, notée ε_t. Celle-ci correspond à la probabilité d'erreur de classification pour un élément i selon la distribution D_t. Il ne s'agit donc pas de son erreur indépendante (si cette hypothèse était considérée seule), mais de l'erreur relative au boosting, c'est-à-dire, calculée avec les poids des données. Formellement :

$$\varepsilon_t = Pr_{i\sim D_t}[h_t(x_i) \neq y_i] = \sum_{j:h_t(x_j)\neq y_j} D_t(j) \tag{2.1}$$

L'erreur d'une hypothèse valide de boosting doit évidemment toujours être stricte-

[1]Produire la même hypothèse que la précédente n'est pas permit. Une nouvelle hypothèse identique produirait les même décisions sur les exemples, mais la distribution des poids ayant été modifiée, donnerait 50 % d'erreur exactement (pour le cas binaire). Ainsi, cette hypothèse ne correspond pas au critère d'une hypothèse faible à cette itération, et ne peut être choisie.

ment inférieure à $1/2$. Les hypothèses produites par AdaBoost sont également pondérées. De manière assez logique, l'influence d'une hypothèse h_t, que nous noterons α_t, sera fonction de sa qualité. Plus une hypothèse est bonne (faible erreur ε_t), plus son influence sera grande.

Algorithme 1 AdaBoost

Avec comme points de données $(x_1, y_1), \ldots, (x_n, y_n)$ où $x_i \in X, y_i \in Y = \{-1, +1\}$
Initialiser $D_1(i) = 1/n$
pour $t = 1, \ldots, T$ **:**
  Entraîner un classifieur faible avec la distribution D_t
  Trouver l'hypothèse faible $h_t : X \to \{-1, +1\}$ avec l'erreur ε_t :

$$\varepsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$$

  Choisir la pondération α_t :

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

  Mettre la distribution à jour :

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{\sum_{j=1}^{n} (D_t(j) \exp(-\alpha_t y_j h_t(x_j)))} \end{aligned}$$

  # Le terme de normalisation Z_t est important pour que D_t demeure une distribution.
fin pour
L'hypothèse finale (globale) donne :

$$H(x) = \text{signe} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

L'idée, et un des grands avantages du boosting, est qu'on n'a pas besoin d'hypothèses très puissantes elles-mêmes ; on se contente même quelques fois de classifieurs très faibles qui donnent des réponses seulement légèrement meilleures que le hasard. Cela est dû au fait que le boosting gère la qualité de l'apprentissage à deux niveaux : au niveau, évident, des hypothèses, et au niveau des poids sur les exemples d'entraînement. Comme chaque itération est censée "raffiner" la solution, on devrait pouvoir améliorer

"indéfiniment" la solution en laissant le nombre d'itérations grandir à volonté. De fait, la théorie du modèle nous indique clairement qu'on n'a pas besoin de la puissance maximale de l'apprenant qu'on veut *booster*. Pour la même raison il y a un certain nombre, généralement grand, d'itérations dans l'application du boosting ; on désire souvent des algorithmes d'apprentissage "légers". Cela est dû tant à des considérations pratiques comme le temps d'entraînement et le temps d'exécution qu'à des considérations plus théoriques, par exemple éviter le sur-apprentissage.

La qualité première du boosting, venant de sa source dans le modèle théorique PAC, est qu'on peut mettre des bornes sur ses erreurs. En fait, un des fondements de l'algorithme même est sa capacité à réduire l'erreur d'entraînement à une vitesse exponentielle. Maintenant, il est également possible de borner l'erreur de généralisation de l'hypothèse finale de boosting en fonction de l'erreur d'entraînement, de la taille de l'échantillon, du nombre d'itérations ainsi que la richesse de la classe de fonction des hypothèses faibles. Cette borne, décrite en détail dans [12] prédit que l'algorithme va finalement sur-apprendre lorsque le nombre d'itérations T augmente. Cependant, en pratique, on remarque dans de nombreux cas que le sur-apprentissage n'arrive pas, ou encore qu'il se produit mais très lentement. En plus, on constate habituellement que l'erreur de test continue à diminuer bien après que l'erreur d'entraînement soit réduite à zéro. Cela semble contredire le principe du rasoir d'Occam ; cependant, il se trouve que le boosting a une action plus subtile. Des analyses plus poussées ont montré que le boosting travaille plus ou moins directement a augmenter les marges des exemples. La marge d'un exemple représente la distance entre celui-ci et la surface de décision. Une grande marge représente une grande confiance dans la décision, alors que les marges près de 0 représentent des cas douteux. Les marges négatives indiquent donc les décisions erronées. La marge d'un exemple est calculée empiriquement par la différence entre la somme des votes corrects et la somme des votes inccorects. Notons la marge γ_i d'un exemple (x_i, y_i) :

$$\gamma_i = \frac{y_i \sum_t \alpha_t h_t(x_i)}{\sum_t \alpha_t} = y_i \sum_t \tilde{\alpha}_t h_t(x_i) \tag{2.2}$$

Avec $\tilde{\alpha} = \alpha / \sum_t \alpha_t$, l'influence normalisée de l'hypothèse h_t. La marge sera positive

pour un exemple bien classifié et négative dans le cas contraire, et sa grandeur correspond à une mesure de confiance sur la classification. L'importance des grandes marges sur la qualité de la classification est déjà bien documentée [28] [29], et Schapire et al. [40] ont construit une borne de l'erreur de généralisation d'AdaBoost qui tient compte des marges. Ils ont également démontré que le boosting était très agressif à réduire les marges. Ainsi, bien après que l'erreur d'entraînement soit tombée à zéro, le boosting continue d'optimiser les marges des exemples. Certains algorithmes de boosting, dont AdaBoost* [34] et LPboosting [15] (LP pour programmation linéaire), tentent spécifiquement de maximiser la marge minimale des exemples.

Une des qualités effective du boosting est l'utilité de la distribution sur les données d'entraînement. Cette distribution se concentre sur les exemples difficiles à classifier et force les nouvelles hypothèses à se spécialiser sur les données qui se situent sur ou près de la frontière, en d'autres mots, les données avec de petites marges, ou des marges négatives. Cependant, cette caractéristique est à double tranchant si on se trouve en présence de bruit d'étiquetage : des données sont mal étiquetées. Dans ce cas, on se concentrera sur les exemples mal étiquetés et on risque de choisir de bien mauvaises hypothèses. Cette faiblesse du boosting face au bruit a été étudiée par Dietterich dans [7]. Pour pallier cette faiblesse et éviter de trop se concentrer sur certains exemples, certaines modifications ont été proposées [35] et le résultat a donné les algorithmes de "Gentle" AdaBoost et BrownBoost.

Le même article de Dietterich [7] note également une autre qualité du boosting, qu'on peut comprendre facilement à l'aide d'une comparaison avec l'algorithme de bagging de Breiman [2]. Le bagging est une méthode d'ensemble qui moyenne le vote de plusieurs hypothèses, chacune d'elle étant entraînée sur un ensemble légèrement différent des données, créé par la technique de *bootstrap*[2]. Le but de bagging est de réduire la variance de l'algorithme utilisé pour générer les hypothèses. Pour être utile, on doit choisir pour générer ces hypothèses un algorithme dit instable. Par ceci, on veut dire qu'un algorithme donnera des réponses différentes s'il est entraîné sur des jeux de données légèrement modifiés. Le boosting fonctionne de manière similaire, et permet dans certains cas de

[2]En fait, le nom même du bagging vient de *bootstrap aggregating*.

réduire la variance de l'algorithme qui crée ses hypothèses. En fait, le bagging se trouve être un cas dégénéré du boosting par échantillonnage où on ne modifie pas la distribution de poids sur les exemples d'entraînement. Dietterich résume ces qualités communes entre ces méthodes d'ensembles comme étant un mélange de précision et de diversité, par rapport aux hypothèses générées. Notons cependant que le bagging est efficace sur les algorithmes ayant une grande variance, alors que le boosting ne nécessite pas cette caractéristique et il arrive même quelques fois que le boosting augmente la variance. Breiman [3] lui-même a comparé le bagging et deux versions de boosting, au niveau des résultats, ainsi qu'au niveau d'une décomposition biais-variance. Une autre comparaison entre ces deux méta-algorithmes est examinée dans [31]. Nous reparlerons plus tard du lien qui existe entre ces deux techniques.

La qualité du boosting dépend clairement des données autant que de l'apprenant faible. On doit avoir une quantité suffisante de données d'apprentissage, ainsi que des hypothèses assez puissantes, mais pas trop complexes. Bref, le boosting dépend de plusieurs contraintes habituelles de l'apprentissage machine.

Le boosting travaille avec ce qu'on appelle un "apprenant faible", ou selon le vocabulaire plus général, un algorithme d'apprentissage, pour produire des hypothèses (faibles). Peu de restrictions sont posées sur cet apprenant faible, sinon qu'il doit produire des hypothèses faibles valides. Bien sûr, la nature même de cet apprenant a son importance, par exemple, la complexité de l'espace de fonctions décrit par l'apprenant est liée aux bornes sur l'erreur de généralisation. Habituellement, on "*boost*" un seul type d'apprenant dans une exécution de boosting. En d'autres mots, on "*boost*" un modèle en particulier. Certains apprenants ont été particulièrements populaires dans la littérature. En voici quelques-uns :

Réseaux de neurones Le modèle des réseaux de neurones est largement connu et très populaire. Son application au boosting se limite généralement aux perceptrons multicouches ou encore aux modèles similaires. Nous discutons d'un lien différent entre le boosting et les réseaux de neurones dans la section 2.2. Plusieurs résultats sur l'efficacité du boosting sur les réseaux de neurones sont présents dans la littérature [42] [25].

Arbres de décision Ce modèle, et plus particulièrement le C4.5 de Quinlan, est également très populaire comme apprenant faible de boosting. Plusieurs résultats d'expériences utilisant cet algorithme sont disponibles dans la littérature [20] [25]. Kearns, Mansour et Dietterich [19] [6] ont poussé plus loin la comparaison, en argumentant que les algorithmes d'arbres de décisions CART et C4.5 sont en fait eux-mêmes des algorithmes de boosting[3]. Cela donne en fait que *booster* des arbres de décision revient à *booster* du boosting.

Decision Stumps Un *decision stump*[4] est, comme son nom l'indique, un cas dégénéré d'un arbre de décision, ou encore plus précisément, est un arbre de décision à un seul noeud (et deux feuilles). On peut cependant décrire efficacement un *decision stump* sans avoir recours aux arbres de décisions. Il s'agit d'un seuillage ferme sur une seule caractéristique des données. On peut également, si on veut, représenter le *stump* comme un neurone-perceptron dégénéré, avec une fonction d'activation ferme, un biais nul et une seule entrée valide[5]. Les *decision stumps* sont très semblables aux "1-rules" de Holte [17] et représentent un exemple particulièrement "faible" d'apprenant faible. Il est en outre très facile de trouver le meilleur *stump* disponible, et dans le cas de classification binaire, tous les *stumps* sont automatiquement valides.

Bien que le boosting ne soit généralement appliqué qu'à un type d'apprenant en particulier, rien dans ses fondements théoriques n'empêche d'utiliser un apprenant plus "large" ou plus "puissant", qui par exemple combinerait plusieurs algorithmes, et effectuerait sa sélection parmi ceux produits par de sous-apprenants. Cependant, démontrer l'avantage pratique clair d'une telle pratique n'est pas trivial. De plus, cette pratique amène certains problèmes, comme une augmentation de la richesse de la classe de fonction dans laquelle les hypothèses sont tirées et en même temps une augmentation des paramètres internes à gérer, ainsi que du temps d'exécution.

[3]Certaines conditions s'appliquent. Lire les articles cités pour les bonnes explications.

[4]On pourrait traduire en français par "souche de décision", mais pour plus de clarté, nous garderons le terme anglais au lieu d'une traduction personnelle

[5]Le vecteur de poids associé à ce neurone n'a qu'une valeur non nulle. La valeur exacte de ce poids n'est pas importante, mais son signe indique le lien entre l'attribut et la décision.

Nous avons mentionné précédemment qu'il existe un lien algorithmique entre le boosting et les arbres de décisions. D'autres liens ont été établis avec d'autres domaines de l'apprentissage machine. Par exemple les SVM, pour *Support Vector Machines*, de Vapnik et Cortez [4] partagent avec le boosting la capacité d'optimiser les marges des exemples, quoique de manière différente [40]. En fait, le boosting ne tente pas directement, comme les SVM, de maximiser la marge minimale des exemples, mais son action sur les marges s'en approche. En fait, on peut montrer que les deux algorithmes travaillent sur des représentations similaires, quoique différentes, des marges sur les exemples[6]. La théorie des jeux peut également être appliquée à l'analyse du boosting [38] [39] [10]. On voit ainsi le boosting comme le jeu répétitif de plusieurs parties pour converger vers une stratégie gagnante. [14] explore de son côté le lien entre le boosting et des principes statistiques connus.

La version originale d'AdaBoost s'applique d'abord au cas de la classification binaire. Heureusement, la méthode est extensible à un usage beaucoup plus large. Moyennant certaines modifications, on peut également appliquer AdaBoost à des cas de régression [11], [22]. Souvent, on transforme le problème de régression en problème de classification, auquel cas un régresseur faible est un régresseur qui est précis à *epsilon*[7] dans plus de 50 % des cas. Bien que les premières tentatives utilisent la moyenne pondérée des régresseurs trouvés, certains résultats montrent qu'utiliser la médiane serait plus naturel, dans le sens où on a vraiment un algorithme qui retrouve AdaBoost comme cas spécial. En fait, les cas de classification peuvent aussi bien utiliser la médiane, puisque pour des réponses binaires, les deux mesures sont équivalentes.

On peut aussi étendre le type de classification au cas multiclasse. D'abord dans [13], puis dans [41], Schapire, Singer et Freund proposent des versions multiclasses d'AdaBoost. La version AdaBoost.M1 modifie légèrement les étapes d'AdaBoost, quant à la mise à jour des poids et au vote des hypothèses. AdaBoost.M1 s'accompagne d'un théo-

[6]Ce qui change est la définition des marges en question et l'algorithme interne qui permet l'optimisation. En fait, cest deux différences font des SVM et de boosting deux algorithmes d'apprentissage très différents, mais néanmoins liés au niveau conceptuel.

[7]Attention, il ne s'agit pas ici du même ε que pour l'erreur de boosting. La signification présente de précision du régresseur ne sera plus mentionnée dans le reste du document.

rème prouvant sa qualité de convergence. AdaBoost.M2 étend ce nouvel algorithme pour faciliter l'apprentissage multiclasse en introduisant, notamment, une nouvelle mesure d'erreur, appelée *pseudo-loss* (pseudo-perte), qui soulage l'apprenant faible d'avoir à fournir des hypothèses ayant plus de 50 % de précision. Cette nouvelle mesure est aussi accompagnée d'une preuve de convergence. Cet algorithme laisse les hypothèses donner des réponses plus complexes (dans les cas multiclasse), en donnant une liste de réponses possibles, ainsi qu'un degré de "plausibilité". Une autre version, AdaBoost.MH, traite le cas multiclasse comme plusieurs cas de classification binaire juxtaposés, et utilise une mesure de perte dite de Hamming. Cette méthode reste cependant très proche de l'algorithme original, mais utilise des poids sur les couples exemple-classe, avec la moitié du poids d'un exemple donné aux bonnes classes, et l'autre aux mauvaises, de manière uniforme. Une autre version, AdaBoost.MO, utilise une approche similaire aux "Error-correcting Output Codes" de Dietterich et Kong [5], [21] et aussi [37]. Enfin, une autre approche utilise des rangs *(ranking)* pour évaluer les prédictions, sous le nom d'AdaBoost.MR. Celle-ci est en fait une généralisation d'AdaBoost.M2.

Les mêmes Schapire et Singer [41] ont également étendu leur algorithme aux cas de classification où les hypothèses donnent une confiance sur leur prédiction, donc produisent des sorties réelles ($[-1,1]$) au lieu de sorties simplement binaires ($\{-1,1\}$). Le boosting a également été utilisé pour gérer des tâches plus lointaines, telles l'apprentissage non-supervisé [48] et l'estimation de densité [33].

2.2 Réseaux de neurones

Le modèle d'apprentissage machine le plus connu et le plus populaire, et encore un des plus importants dans le domaine de l'apprentissage machine, est le réseau de neurones. Il s'agit d'un modèle directement inspiré des sciences cognitives et de l'organisation biologique du cerveau humain, du moins de certaines théories sur le cerveau. Il est composé d'un certain nombre d'unités de traitement, appelées neurones (artificiels), reliées entre elles. Le modèle le plus commun présente un réseau de neurones comme un ensemble de couches de neurones, superposées de manière hiérarchique. Habituelle-

ment, cette structure, ainsi que le nombre de neurones, est fixée au départ, et l'apprentissage consiste à optimiser l'interaction, c'est-à-dire les liens, ou encore les poids, entre les neurones de différentes couches.

L'élément à la base d'un réseau de neurones, le neurone artificiel, fut introduit en 1943 par McCulloch et Pitts [30] comme représentation mathématique simplifiée d'un neurone biologique. Ce neurone artificiel reçoit un certain nombre d'influx en entrée, influx venant d'autres neurones, qui sont additionnés et passés dans une fonction d'activation. Cette fonction d'activation décide, à partir de l'influx global reçu, si le neurone s'active ou non, c'est-à-dire s'il émet un signal en sortie. La fonction d'activation du neurone artificiel de McCulloch et Pitts est une fonction de seuil (fonction de Heaviside). Conséquemment, la sortie du neurone est binaire. Appelons les entrées x_i et le seuil θ :

$$y = \begin{cases} 1 & \text{si } \sum_i w_i * x_i \geq \theta, \\ 0 & \text{sinon.} \end{cases} \tag{2.3}$$

Le premier modèle de réseau de neurones à proprement parler, le perceptron, fut développé à partir d'une variante du neurone de McCulloch-Pitts par Frank Rosenblatt en 1957 [32]. La principale différence entre le perceptron et le neurone de McCulloch-Pitts est que le perceptron a un algorithme d'apprentissage. Cet algorithme dépend du seuil du perception. Au lieu d'utiliser une fonction de Heaviside comme le neurone de McCulloch-Pitts, on utilise une fonction sigmoïdale. Cette fonction continue permet de dériver l'erreur de décision pour le neurone selon les poids des entrées et le seuil de

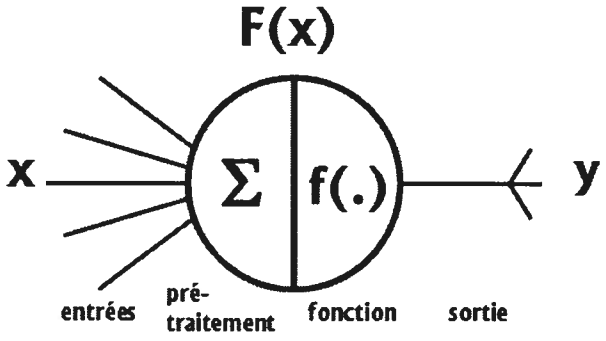

Figure 2.1 – Schéma d'un neurone

la fonction d'activation. En calculant ce gradient, on peut modifier graduellement les paramètres du neurone pour optimiser sa performance. Seul, cependant, ce modèle est limité dans ses usages possibles, car la surface de décision – la section de l'espace qui sépare celui-ci en deux parties, une de celles-ci activant le neurone – représentée par un neurone est linéaire. Cette surface est simplement $\sum_i w_i * x_i = \theta$ [8]. Il existe donc une classe de fonctions très importante qui est hors d'atteinte d'un tel algorithme d'apprentissage. Toutefois, une structure à plusieurs couches de neurones/perceptrons, appelée un perceptron multicouche (PMC), permet de briser la linéarité de la décision. Dans le cas d'un PMC, il existe un algorithme très connu pour entraîner tous les neurones en même temps, également par descente de gradient. Il s'agit de la rétro-propagation du gradient (*backpropagation* en anglais.). Il a même été prouvé plus tard [18] que cette structure permettait en fait, théoriquement, de représenter n'importe quelle fonction continue. Malgré ce bel élément théorique, il faut trouver un moyen de construire ou d'entraîner automatiquement un tel réseau.

$$\hat{Y} = s(\sum_i w_i f(n_i) - w_0) \tag{2.4}$$

L'équation 2.4 représente la fonction qui décide du comportement d'un neurone dans un réseau, dans laquelle $\hat{Y}$ représente la sortie du réseau, s, la fonction sigmoïdale voulue, w_i est le poids d'entrée venant d'un neurone n_i de la dernière couche, $f(n_i)$ la sortie de ce neurone, et w_0 est le biais, ou en d'autres termes, la valeur du seuil du neurone **n**. Pour plus de simplicité, on joint habituellement le biais au vecteur de poids, et on ajoute une valeur de -1 au vecteur d'entrées (les sorties de la couche précédente) de manière à ce que la sommation se transforme complètement en produit vectoriel entre un vecteur d'entrée et un vecteur de poids.

Bien évidemment, le domaine des réseaux de neurones ne se limite pas aux perceptrons multicouches. Il existe un très grand nombre de réseaux de neurones qui diffèrent, parfois beaucoup, de ce modèle. Nous n'avons présenté que le minimum nécessaire pour comprendre la suite du développement du boosting hiérarchique.

[8] Dans un essai intitulé *Perceptrons* en 1969, Minsky et Papert montrèrent qu'un perceptron ne pouvait pas apprendre une fonction aussi simple qu'un *XOR* – un ou exclusif.

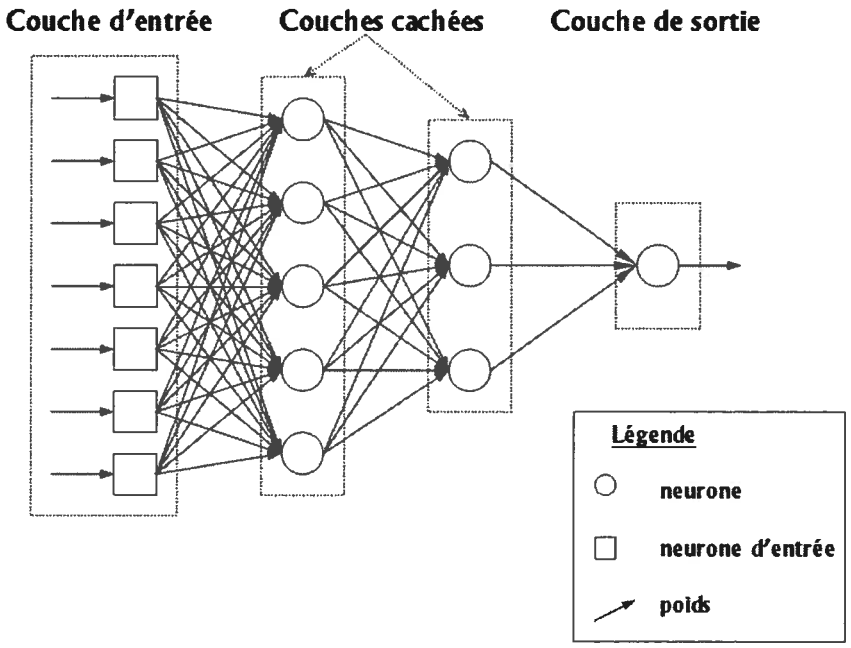

Figure 2.2 – Exemple d'un réseau de neurones

2.3 Boosting comme réseau de neurones

Le boosting, bien qu'ancré dans le modèle PAC, est souvent présenté sans ce cadre théorique et le sens qu'on veut donner à l'algorithme ne tient qu'au présentateur. Le boosting a ceci de sympathique qu'il est algorithmiquement simple et peut être présenté de différentes manières selon le contexte dans lequel on se trouve.

Le point de vue le plus commun sur le boosting est celui des méthodes d'ensembles. Tout comme AdaBoost, le bagging a été présenté dans les années 90, et leur procédure est très semblable. Assez que le bagging peut être considéré comme un cas dégénéré de boosting par échantillonage. De ce point de vue, le boosting, comme le bagging, construit un ensemble de classificateurs (hypothèses), lesquels votent, et la sortie globale correspond à la moyenne des votes. Les deux différences entre ces deux algorithmes sont l'ensemble sur lequel les algorithmes entraînent les hypothèses (bagging utilise le boostrap, alors que boosting utilise une distribution sur les exemples), et le fait que celles du boosting sont pondérées en sortie. La relation entre le boosting, le bagging et d'autres méthodes d'ensembles est analysée plus en détail dans [7] et [3].

Un autre point de vue commun est celui qui descend directement de l'algorithme, sinon de l'intuition originelle à partir du modèle PAC. Par la distribution des poids sur les exemples, le boosting entraîne chaque nouvelle hypothèse avec le but explicite de minimiser l'erreur de la somme des hypothèses combinées. Cette vision montre le côté itératif de l'algorithme et explique comment chaque hypothèse suit les précédentes, par rapport au bagging dont l'entraînement peut facilement être parallélisé et où les différentes hypothèses produites sont indépendantes.

Enfin, une troisième approche présente le processus de boosting plutôt comme une méthode de sélection des influences (α) des différentes hypothèses. De cette manière, on "part", en quelque sorte, avec toutes les hypothèses possibles d'une classe de fonctions, et le boosting, en fait, pondère leur action sur la décision finale. On fait également fi ici de l'action algorithmique itérative du boosting, et cette vision est proche de celle des méthodes d'ensembles, à la bagging, vues plus haut. Cette vision est très bien expliquée dans [11].

Ces trois interprétations montrent que le résultat du boosting est une série d'hypothèses parallèles, dans le sens où ces hypothèses ne sont pas reliées entre elles au niveau de l'exécution, et qu'elles ont les mêmes entrées et des sorties similaires.

Présentée de manière graphique, comme dans la figure 2.3, la représentation globale de l'algorithme AdaBoost est très similaire à un réseau de neurones, plus particulièrement à une couche de neurones. Si nous considérons les hypothèses faibles du boosting comme des "boîtes noires", nous pouvons aussi bien les voir comme des neurones. De cette manière, le boosting apparaît comme une méthode de génération de réseau de neurones. Pour compléter la comparaison, ajoutons un neurone identité en couche de sortie pour sommer les sorties de toutes les unités du système.

Pour se rapprocher encore plus d'un modèle de réseau de neurones précis, par exemple le perceptron multicouche, on peut aussi vouloir représenter les hypothèses faibles en deux parties, une préparant l'entrée, et l'autre la traitant. Au niveau d'un réseau de neurones, la première partie implique les poids d'entrée du neurone ; dans le cas usuel, il s'agit d'appliquer un produit scalaire entre le vecteur d'entrée $\mathbf{x}$ et le vecteur de poids $\mathbf{w}$. La seconde partie est la fonction d'activation (par exemple le sigmoïde). Si on réussit

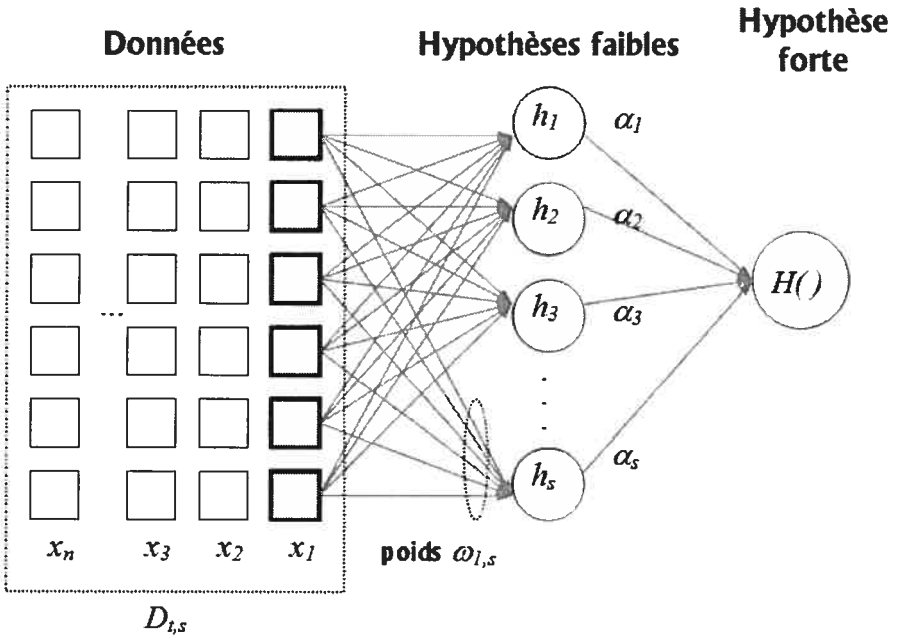

Figure 2.3 – Résultat du boosting présenté comme un réseau de neurones

à représenter nos hypothèses faibles de boosting comme une fonction agissant sur une sommation pondérée des entrées, on se donnera alors des "poids d'entrée" pour cette hypothèse. Pour bien identifier ces nouveaux poids, nous les noterons ω. Présenté de cette façon, le lien entre le boosting et un réseau de neurones est évident et ce, même au niveau des équations :

$$F(x) = s(\sum_i w_i F(x_i) - w_0) \tag{2.5}$$

$$H(x) = \text{signe}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right). \tag{2.6}$$

Nos unités, neurones ou hypothèses, sont ici regardées comme des "boîtes noires" et ne donnent pas de garantie sur les fonctions internes qu'elles peuvent représenter. En particulier, on ne garde pas nécessairement la propriété de continuité qui permet de dériver l'erreur d'apprentissage de l'unité elle-même par rapport à ses paramètres et ses entrées. Le résultat du boosting est donc un réseau de neurones du point de vue symbolique, mais

il ne permet pas nécessairement d'obtenir certaines caractéristiques tellement appréciées de ceux-ci, telle la possibilité d'utiliser un algorithme comme la rétropropagation du gradient. On peut cependant, dans le boosting, entraîner spécifiquement des hypothèses *neuronables*, c'est-à-dire des fonctions qui donneront des résultats cohérents avec une interprétation neuronale du résultat de l'algorithme. Évidemment, ce lien est trivial dans le cas où on *booste* des réseaux de neurones : un réseau de réseaux de neurones est un réseau de neurones. Le lien pratique entre les deux méthodes est alors relativement évident. On pourrait utiliser le boosting pour générer un réseau de neurones, et ensuite l'optimiser par *backpropagation* ou tout autre algorithme d'entraînement de réseaux de neurones. On pourrait également utiliser le boosting pour "modifier" un réseau de neurones existant en lui ajoutant des unités, et conserver notre réseau comme étant valide.

Ce lien entre les deux méthodes est passablement intéressant, mais vu les limitations qu'il impose d'un côté comme de l'autre, son application pratique n'est pas aussi naturelle qu'elle peut le paraître. Cependant, on peut analyser ce lien entre les deux méthodes de manière théorique. Prenons par exemple le cas simple d'AdaBoost avec des *decision stumps*. Chacun de ces *stumps* peut être considéré comme un neurone de McCulloch-Pitts dégénéré (les poids des liens d'entrée ne sont pas entraînés, un de ces liens étant 1, les autres étant nuls), ou encore comme un neurone-perceptron dégénéré c'est-à-dire avec un seuil ferme. Il s'ensuit qu'un seul *stump* induit les mêmes faiblesses qu'on impute au simple perceptron. Alors qu'en est-il du boosting, qui fait voter plusieurs *stumps* ? Ce boosting demeure la génération d'une seule couche d'unités. On constate ainsi que le boosting ne fabrique pas vraiment ce qui a été la solution au problème du perceptron – une hiérarchisation – tout en étant lié, d'une certaine manière, par ce problème. Nous y reviendrons plus tard.

2.4 Reconnaissance d'images

Les algorithmes d'apprentissage, réseaux de neurones et boosting compris, ont depuis longtemps été appliqués à la reconnaissance d'images. Les tâches sont multiples : reconnaissance de visages, détection d'objets, traitement vidéo, ou encore reconnais-

sance de texte et de caractères manuscrits. Cette dernière tâche en particulier fait partie des tâches de routine des tests d'algorithmes d'apprentissage. Il existe deux jeux de données très connus sur lesquels sont testés régulièrement les nouveaux algorithmes, soit MNIST [23] et USPS. Ces deux jeux de données ne sont pas particulièrement intéressants au niveau de la tâche elle-même, MNIST étant un problème presque totalement résolu, et dans le cas d'USPS, un problème plus bruité, on est maintenant très près du taux d'erreur humaine[9]. Ce qui nous intéresse est plutôt le moyen d'obtenir les résultats. On constate que plusieurs des "bons" résultats utilisent soit un prétraitement des données, soit des réseaux spécifiques à cette tâche de traitement d'image. Le résultat est similaire : nous utilisons notre connaissance a priori des données, qui sont des images, pour améliorer l'algorithme.

Le fait de connaître la nature des données nous donne un avantage certain. Les images peuvent être représentées comme des vecteurs, ce qui est le format traditionnel pour la plupart des algorithmes d'apprentissage. Cependant, une image est plus qu'un vecteur ; chacun des éléments représente en fait un pixel, qui est situé géométriquement dans l'image : il a une position, et une relation avec les autres pixels. De plus, les éléments de données d'une image sont spéciaux dans un sens – ils sont tous sur la même échelle de lumière, soit purement un pixel en tons de gris (de 0 à 255), ou encore en valeur replacée sur un intervalle ($[-1, 1]$)[10]. Ceci a de particulier qu'on peut jouer avec ces pixels de manière "intelligente" : modifier l'image, c'est-à-dire notre vecteur de données, afin qu'il soit plus facile à interpréter.

Ceci est un point important parce que, considérées uniquement comme des vecteurs (et non visuellement), deux images très semblables peuvent être des vecteurs très dissemblables. La figure 2.4 montre deux images de la lettre e très semblables ; cependant, les vecteurs correspondants à ces images seront très différents puisqu'à cause de la translation qu'à subi l'image, ce ne sont pas les mêmes pixels qui sont activés. Du point de vue algorithmique, on ne donne souvent aucun sens particulier à la position d'une dimension

[9]Le taux d'erreur humaine est l'erreur empirique d'humains à qui on a demandé la même tâche que le réseau, c'est-à-dire d'identifier les chiffres.

[10]Ceci est pour les images en tons de gris. La couleur pose un cas légèrement plus complexe, mais qui peut toujours être "dégénérée" en tons de gris.

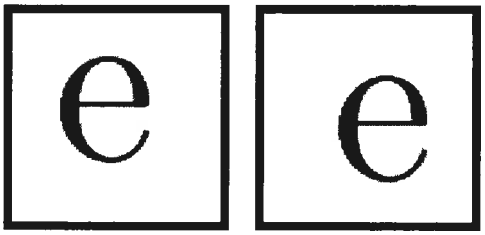

Figure 2.4 – Images semblables, vecteurs dissenblables

d'entrée. C'est tout le contraire avec une image : un pixel ne signifie pas grand chose par lui-même – c'est sa position qui lui donne un sens dans l'image. Malheureusement, c'est seulement le vecteur que "verra" l'algorithme. Certaines modifications légères des images ne devraient pas changer leur sens au niveau de l'apprentissage, mais ceci pose problème dans le cas où les vecteurs liés à ces images changent beaucoup.

Une autre propriété intéressante des images est que dans les images naturelles, les pixels ne sont pas indépendants. On remarque que la valeur d'un pixel est, en général, fortement corrélée à celle de ses voisins. Bien sûr, il existe souvent des contrastes clairs dans une image, mais ceux-ci sont relativement rares (au niveau des pixels), et l'image est généralement continue.

2.5 Filtres de Haar

AdaBoost avec des *decision stump* a été appliqué au traitement d'images, mais son efficacité sur des images n'est pas directe. Une grande partie de cette faiblesse vient du fait qu'un *stump* travaille sur une seule dimension du vecteur d'entrée, donc dans le cas des images, sur un seul pixel, et à moins d'avoir une image très précise, un pixel n'est pas d'une grande utilité. Non seulement, comme on vient de le voir, de faibles manipulations d'une image peuvent changer la valeur d'un pixel du tout au tout, mais lorsque les images sont bruitées, un pixel devient encore moins utile (précis), et ainsi beaucoup moins intéressant du point de vue de l'apprentissage.

Il est donc plus intéressant au niveau d'une image, de regarder dans une fenêtre plutôt

qu'à un endroit précis, c'est-à-dire un pixel. On ne peut pas s'attendre à ce qu'une information importante à propos de notre image se retrouve dans un seul pixel. Cependant, une fenêtre plus grande, une section de l'image, elle, peut s'avérer être d'une utilité bien meilleure. C'est là qu'interviennent les filtres de Haar *(Haar-like features)*, introduits pour la détection de visages par Viola et Jones dans [47], et étendus ensuite par Lienhart et Maydt [24]. Ces filtres géométriques, qui sont une incarnation de la représentation en vaguelettes *(wavelet representation [26])*, calculent rapidement le résultat de contrastes de filtres rectangulaires. Un filtre de Haar est en fait un rectangle divisé en zones positives et négatives (blanches et noires sur les illustrations), qui représentent un contraste. En délimitant des régions, les filtres permettent non seulement de dépasser le niveau des pixels pour prendre en compte une zone, mais aussi de relativiser cette importance par rapport à son entourage à l'aide du contraste. Les contrastes examinés par Viola et Jones sont les bordures verticales et horizontales, les lignes verticales et horizontales, ainsi qu'un schéma en damier. D'un point de vue algorithmique, le résultat d'un filtre de Haar sur une image est très rapide à calculer, d'ordre constant, en passant par une représentation dite "intégrale" de l'image.

Si nous avons une image P, avec $p_{i,j}$ représentant la valeur de l'intensité lumineuse du pixel à la position (i, j), alors la carte de la représentation intégrale de l'image, IP, sera de même taille, avec $ip_{i,j} = \sum_{k=1\ldots i}\sum_{j=1\ldots j}(p_{k,l})$. C'est-à-dire que la valeur aux coordonnées (i, j) de la représentation intégrale de l'image représente la somme des intensité de l'image dans le rectangle délimité par ce point et par l'origine. À travers cette représentation, il est facile de calculer l'intensité totale d'une section rectangulaire délimitée par les points (xa, ya) et (xb, yb), respectivement le coin supérieur gauche et le coin inférieur droit du rectangle :

$$\text{Intensité totale} = ip_{xb,yb} - ip_{xa-1,yb} - ip_{xb,ya-1} + ip_{xa-1,ya-1}. \tag{2.7}$$

Le résultat de l'application d'un filtre de Haar est la somme des différentes zones qui le constitue. Avec la formule 2.7, la somme de plusieurs rectangles juxtaposés permet de simplifier des termes dans l'équation globale.

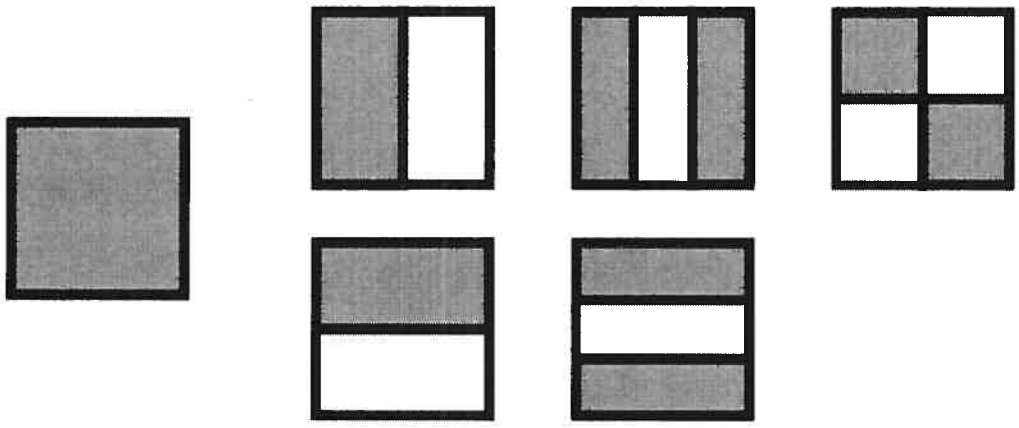

Figure 2.5 – Un filtre rectangulaire et les différents types de filtres de Haar

Au-delà de la représentation plus complète que présente un filtre par rapport à un pixel, il y a une autre justification simple à l'utilisation de ces filtres rectangulaires : ils sont moins sensibles au bruit. En effet, les filtres s'assimilent à des méthodes de réduction de bruit connues dans le domaine du traitement d'image. La plus simple de ces méthodes est celle de la moyenne : on choisit un voisinage de notre pixel, en général un cercle, et on calcule la moyenne des valeurs des pixels dans cette zone. La valeur du pixel au centre devrait, sans bruit, être proche de celle trouvée par ce calcul. La validité de ce calcul dépend de deux propriétés : il faut que les images soient continues localement (un pixel a une valeur près de celle des pixels voisins) et que le bruit soit aléatoire et de moyenne zéro. Une variation de cette méthode de réduction du bruit consiste à prendre la médiane des valeurs du voisinage. Une autre méthode, semblable mais plus coûteuse, est d'appliquer un filtre gaussien sur le voisinage, de sorte que les voisins proches ait une plus grande importance que les voisins lointains.

Les filtres de Haar ont été utilisés par Viola et Jones dans le cadre d'une application de détection de visage où la vitesse d'exécution était très importante. Ce système, entraîné par AdaBoost, se veut une cascade de classifieurs faibles qui agissent comme court-circuit : on arrête la détection dès qu'on obtient une réponse fausse. Ceci, associé à la rapidité d'exécution des filtres de Haar, permet de repérer les figures très rapidement.

2.6 Boosting hiérarchique

En reprenant les arguments présentés à la section 2.3, on constate que la création d'un algorithme de boosting multicouche peut très bien être avantageuse. Cela permettrait d'obtenir une plus grande profondeur, une plus grande richesse, un boosting gérant un nouveau niveau d'abstraction. Prenons AdaBoost comme algorithme de boosting de départ[11]. Pour s'approcher davantage d'un boosting de neurones, nous utiliserons des *decision stumps* comme hypothèses faibles, qui sont déjà, en quelque sorte, des neurones dégénérés. Comme un de nos buts est d'ajouter de la profondeur au résultat, utiliser à la base des hypothèses très puissantes ou profondes limiterait le gain potentiel d'une telle méthode. En quelque sorte, nous testons si l'architecture hiérarchique de boosting permet d'aumenter la richesse d'action de la combinason d'hypothèses (très) faibles. Pour créer une seconde couche de boosting, il suffit de *booster* à partir des sorties de la première couche.

Cela semble au départ assez naturel, et même plutôt facile, mais dans le cas simple où chaque hypothèse faible donne une sortie purement binaire (1 ou -1), les sorties seules font d'assez mauvaises entrées pour une nouvelle couche, en particulier dans le cas des *decision stumps*, où l'entraînement sur une valeur binaire ne peut générer que deux sorties différentes (ou dans le cas multiclasse, 2^c). C'est extrêmement faible, particulièrement si la donnée binaire qu'on utilise est une sortie d'AdaBoost, donc déjà utilisée dans l'hypothèse forte. Entraîner un *stump* sur le seul résultat d'un *stump* déjà existant ne donnera rien de neuf.

Pour contourner ce problème, on peut regrouper ces valeurs binaires pour en créer des variables plus riches. Ces regroupements, que nous appellerons filtres, doivent joindre plusieurs valeurs, d'une manière que l'on souhaite "intelligente". Reprenons les filtres de Haar. Un *stump* utilisant un de ces filtres se trouve non pas à prendre un seul pixel en entrée, mais une combinaison des valeurs de plusieurs pixels, certains ajoutés, d'autres soustraits. On somme plusieurs entrées dans leur zone, puis, selon le type, on les compare par soustraction. D'un point de vue vectoriel, nous avons un vecteur ω prenant comme

[11] Notons ici que bien que nous nous basons sur AdaBoost, notre algorithme s'adapte à plusieurs types de boosting sans changement majeur.

valeur soit 1 pour les pixels dans la zone positive du filtre, soit -1 pour ceux situés dans la zone négative, ou encore 0 pour ceux qui sont hors du filtre, et ce vecteur sert comme les poids d'un réseau de neurones, à pondérer les entrées pour en faire une seule valeur qui passera dans la fonction de seuillage. Il s'agit bien sûr dans le cas présent d'un filtre géométrique sur une image, mais nous pouvons nous en inspirer pour créer des filtres d'inspiration géométrique applicables à des données vectorielles quelconques.

Ces filtres de Haar sont une sorte de prétraitement sur les images. Numériquement, il existe un bien plus grand nombre de filtres possibles que de pixels dans l'image : de l'ordre[12] de $l * n_h^2$. Au niveau de nos hypothèses faibles, notons qu'on agit véritablement en deux étapes : on choisit un filtre, donc un vecteur ω, puis on entraîne notre *stump*. Les deux ne sont pas liés directement, contrairement à un réseau de neurones, où ces deux éléments sont appris en même temps. D'une certaine manière, l'utilisation des filtres nous permet d'exploser les dimensions d'entrées en les combinant de différentes manières. Au lieu de choisir parmi des combinaisons possibles d'entrées directement, on génère, à travers les filtres, des groupements sensés et on choisira parmi ceux-ci. Au lieu de choisir plusieurs dimensions d'entrées, on choisira un seul filtre parmi notre très grand ensemble de filtres.

Comme il a été souligné au paragraphe précédent, les filtres de Haar sont possibles car ils s'appliquent sur des images. Qu'en est-il des autres types de données : par exemple, des données qui n'ont aucune structure géométrique naturelle, autre que celle qu'on veut bien leur donner. Nous pouvons généraliser, avec certaines conditions, l'applicabilité des filtres de Haar. Si on crée pour nos données une géométrie, on pourra alors les interpréter, d'une certaine manière, comme des images dégénérées. Et on pourra ainsi leur appliquer des filtres comme les filtres de Haar, géométriques.

Nous pouvons maintenant préciser la forme que notre algorithme de boosting multi-couche, *hboost*, prendra. En partant d'une couche de données C_0, nous combinerons ces données en les passant à travers un niveau de filtres pseudo-géométriques[13], et la sortie de ces filtres sera prise comme dimensions d'entrée pour une couche de boosting. On

[12] l représente ici le nombre de types de filtres de Haar et n_h le nombre d'unités sur la couche précédente.

[13] Ce niveau n'existe pas vraiment au sens pratique, puisqu'il s'agit seulement, en fait, de passer par les filtres pour choisir un vecteur ω sur les entrées

garde ensuite les sorties de chaque hypothèse faible comme une nouvelle dimension de données de la couche C_1, et de là on peut créer par AdaBoost une couche supérieure, C_2, basée sur ces nouvelles données. La figure 2.7 montre ces étapes successives.

La structure la plus simple de *hboost* va entraîner les couches d'hypothèses de manière successive. Cependant, au niveau d'une hypothèse, l'entraînement se fait selon les étapes suivantes : on place toutes les données d'entrées dans un espace, on crée des filtres sur cet espace, et finalement on prend le résultat de ces filtres comme véritables entrées de l'hypothèse. L'hypothèse est ensuite entraînée normalement (selon boosting, avec la distribution des poids sur les exemples d'entraînement), et sa sortie, lorsque calculée, est ajouté dans un ensemble de données, lui ajoutant une nouvelle dimension.

Pour les cas multiclasses, qui nous intéressent particulièrement, il faut modifier nos *decision stumps*. Il existe deux moyens simples d'étendre au cas multiclasse ces *stumps*. Le premier moyen est de traiter chaque classe séparément, toujours en n'utilisant qu'une unité d'entrée ; donc pour une classe i, on trouve le seuil θ_i qui sépare le mieux les éléments de la classe i et les éléments des autres classes. La seconde méthode est très similaire, mais n'utilise qu'un seul seuil θ pour toutes les classes. On choisit alors le seuil qui effectue la meilleure séparation.

C'est cette manière de faire que nous utiliserons. Comme il n'y a qu'un seuil, la sortie de l'hypothèse peut se décomposer en deux parties. Premièrement, on applique le seuil et on obtient une sortie binaire. Ensuite, on multiplie cette valeur par un vecteur de vote v lequel représente la décision de l'hypothèse pour chaque classe selon la position par rapport au seuil. De cette manière, on garde des hypothèses multiclasses qui donnent quand même une sortie intermédiaire (unique) binaire. Ceci n'est pas essentiel, mais simplifie la gestion de l'algorithme car chaque hypothèse n'a finalement qu'une sortie. Notons que ces hypothèses multiclasses (à un seuil unique) seront évidemment moins performantes individuellement que des hypothèses avec des seuils multiples.

Nous utiliserons, également dans le cas multiclasse, AdaBoost.MH, puisque celui-ci est une généralisation directe de l'algorithme original. L'algorithme *hboost* est décrit ici selon la version architecturale de hiérarchie forte, telle que présentée à la section 3.1.1 plus loin dans le texte. Notons que cette version de *hboost* conserve toutes les garanties

de convergence de AdaBoost, et que les formules de mise à jour des paramètres $\alpha_{t,s}$ et $D_{t,s}$ sont empruntées à AdaBoost.

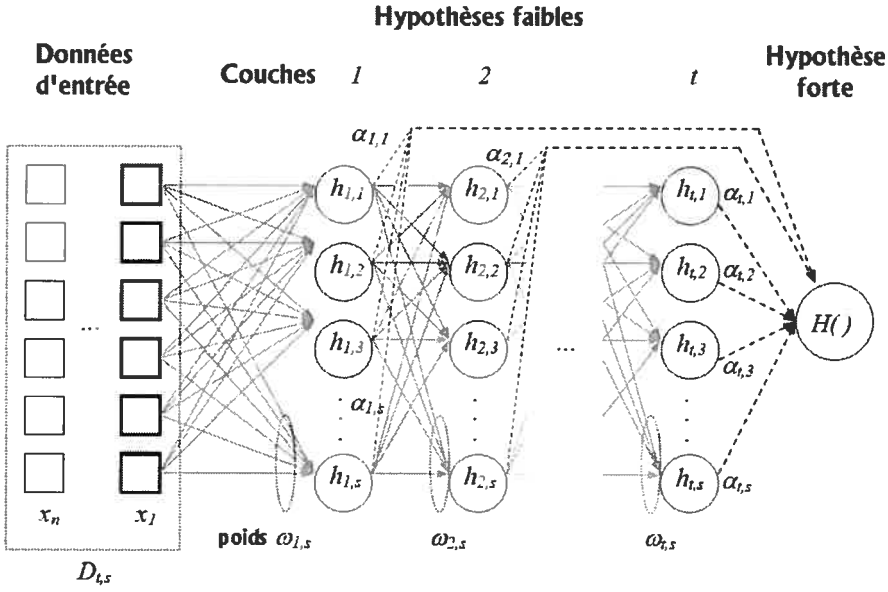

Figure 2.6 – Schéma du boosting hiérarchique

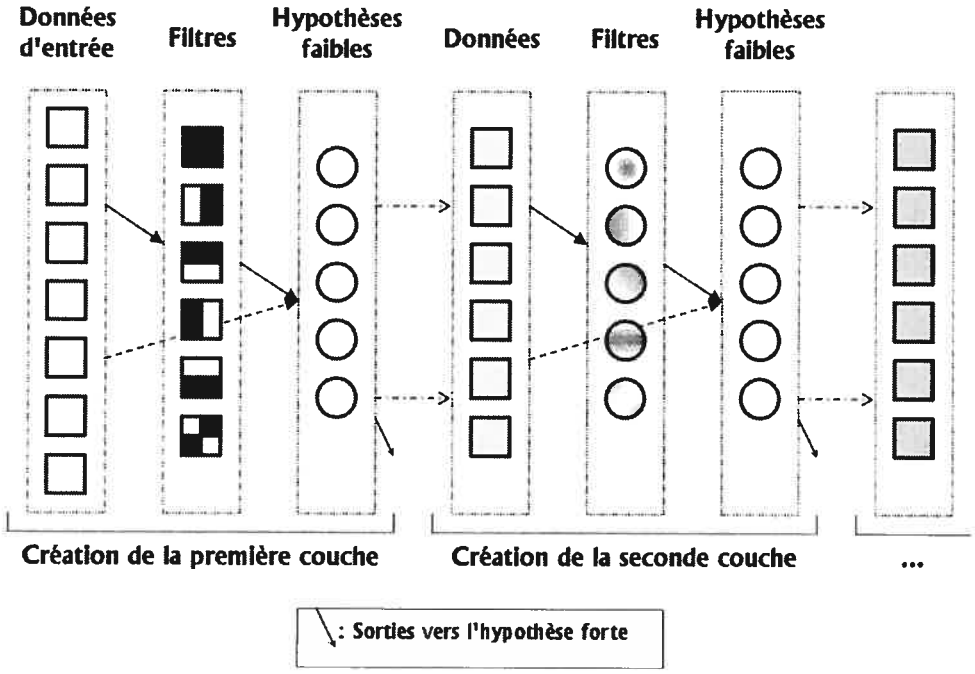

Figure 2.7 – Étapes de construction des couches hiérarchiques de boosting

Algorithme 2 hboost, hiérarchie forte

Avec comme points de données $(x_1,y_1),\ldots,(x_n,y_n)$ où $x_i \in X, y_i \in Y = \{-1,+1\}$
et avec $C_1,\ldots,C_L$ comme couches, chacune étant précisée pour l_j unités.
C_0 représente la couche de donnée d'entrée, avec $l_0 = d$ dimensions.
H_t représente l'ensemble des unités de la couche t.
$H_t(x)$ signifie passer le point x à travers cette couche.
$H_{\{1,t-1\}}(x_i)$ signifie ici la valeur de x_i lorsque passée à travers
les $t-1$ premières couches.
$H_{\{1,t-1\}}(x_i) = H_{t-1}(H_{t-2}(...H_2((H_1(x_i)))...))$.
Initialiser $D_{1,1}(i = 1..n) = 1/n$
pour couche$t = 1,\ldots,L$ **:**
  Si $t > 1$, $D_{t,1} = D_{t-1,l_{t-1}-1}$.
  pour unité $s = 1,\ldots,l_t$ **:**
    Créer un espace dans lequel les l_{t-1} unités sont placées (optionnel).
    Générer un ensemble de filtres F sur ces sorties.
    Entraîner un classifieur faible avec la distribution $D_{t,s}$ et les sorties des filtres générés à l'étape précédente $F(C_{t-1})$.
    Trouver l'hypothèse faible $h_{t,s} : X \to \{-1,+1\}$ qui minimise l'erreur $\varepsilon_{t,s}$:

$$\varepsilon_{t,s} = \Pr_{i\sim D_{t,s}}[h_{t,s}(H_{\{1,t-1\}}(x_i)) \neq y_i]$$

    Choisir la pondération $\alpha_{t,s}$:

$$\alpha_{t,s} = \frac{1}{2}\ln\left(\frac{1-\varepsilon_{t,s}}{\varepsilon_{t,s}}\right)$$

    Placer la nouvelles unité dans la couche C_t.
    Mettre la distribution à jour :

$$\begin{aligned} D_{t,s+1}(i) &= \frac{D_{t,s}(i)\exp(-\alpha_{t,s}y_i h_{t,s}(H_{\{1,t-1\}}(x_i)))}{Z_{t,s}} \\ &= \frac{D_{t,s}(i)\exp(-\alpha_{t,s}y_i h_{t,s}(H_{\{1,t-1\}}(x_i)))}{\sum_{j=1}^{n}(D_{t,s}(j)\exp(-\alpha_{t,s}y_j h_{t,s}(x_{j,t-1}))} \end{aligned}$$

  fin pour
fin pour
L'hypothèse finale (globale) donne :

$$H(x_i) = \text{signe}\left(\sum_{t=1}^{C}\left(\sum_{s=1}^{l_t}\alpha_{t,s}h_{t,s}(H_{\{1,t-1\}}(x_i)\right)\right)$$

CHAPITRE 3

ARCHITECTURE ET PARAMÉTRISATION

3.1 Couches, modèle et nombre

Maintenant que nous possédons un algorithme de boosting pour construire des couches hiérarchiques, nous devons décider quelle sera la structure hiérarchique que nous désirons modéliser. Le cas particulier d'AdaBoost est plutôt simple : puisqu'il n'y a pas de hiérarchie – une seule couche – la génération d'unités apprenantes se fait de manière linéaire. On ajoute des unités à la première couche, qui est en fait la seule qui existe, jusqu'à ce qu'un critère d'arrêt soit atteint.

Dans le cas du boosting hiérarchique, on doit planifier l'architecture que prendra le système : combien y aura-t-il de couches, quelle sera leur nature, leur taille, et comment ces couches seront-elles construites dans le temps ?

La première question qui se pose est au niveau du nombre de couches. Quel nombre de couches sera suffisant ? Comme on l'a vu avec les réseaux de neurones, une couche cachée permet en théorie d'atteindre le niveau de complexité voulu dans le cas des perceptrons multicouches. C'est bien sûr une théorie, et en pratique, on n'a pas encore abandonné le travail sur les réseaux de neurones à plus d'une couche cachée. Dans le cas du boosting hiérarchique, les couches ont une fonction symbolique particulière. Comme chaque unité d'une première couche est reliée à la sortie, elles sont déjà des décisions, et les unités de la seconde couche ont pour but de combiner des décisions "inférieures" en décisions "supérieures". De ce point de vue symbolique, on peut extrapoler que plus de couches donnent une plus grande puissance d'abstraction ; mais cette abstraction supplémentaire est-elle seulement utile, en pratique ? Serait-elle même possible avec la quantité limitée de données à notre disposition ? Un point de vue vorace nous suggérerait évidemment d'accepter autant de couches que l'entraînement trouve utile, en fait d'en ajouter tant que cet ajout se trouve être bénéfique. Cependant, des contraintes pratiques et de design militent plutôt dans le sens inverse, pour une structure fixe, ou du moins,

quelque peu limitée en taille.

Notre seconde préoccupation est la manière dont les couches seront construites. Doit-on y aller de manière progressive, une couche à la fois ? Ou encore, devrait-on laisser les couches pousser de manière concurrente ?

Notons la différence entre la première couche et les suivantes. La première couche se base directement sur les données brutes du problème. Les couches suivantes, dans notre modèle original, se nourrissent de la couche qui les précède, des décisions prises au niveau précédent. Cette structure ferme est cependant un choix de simplicité et il est possible de mettre des contraintes moins sévères quant à la gestion des couches.

3.1.1 Hiérarchie forte

La manière simple de gérer ce problème est de spécifier à l'avance le nombre de couches et leur taille respective. Ensuite, il suffit de procéder graduellement : la première couche est entraînée complètement, puis la suivante, et ainsi de suite. L'avantage majeur de cette approche est sa simplicité. Comme l'entraînement des différentes couches ne se superpose pas, la construction hiérarchique est triviale. Autre avantage, plus symbolique, la hiérarchie est complète : chaque unité d'une couche est entraînée sur une couche "complète", et aura accès aux mêmes données. Le désavantage est qu'on restreint la taille des premières couches. En fait, le boosting produit par cette technique est fermé dans le temps : contrairement à AdaBoost standard, qui peut produire une infinité d'unités, une hiérarchie forte remplira ses couches du nombre d'unités prédéfini, et ce, même si on voudrait ou on devrait continuer. Il faut donc en quelque sorte estimer la taille minimum dont notre structure de boosting aura besoin en fonction de la tâche à accomplir et de sa complexité. Il y a cependant deux moyens inélégants de continuer l'entraînement d'un boosting hiérarchique à hiérarchie forte. On peut d'abord laisser la dernière couche grandir autant que voulu, ceci ayant l'effet de déséquilibrer notre architecture. L'autre moyen est de laisser ouvert le nombre de couches lui-même, toute nouvelle couche ayant une taille égale à la dernière spécifiée.

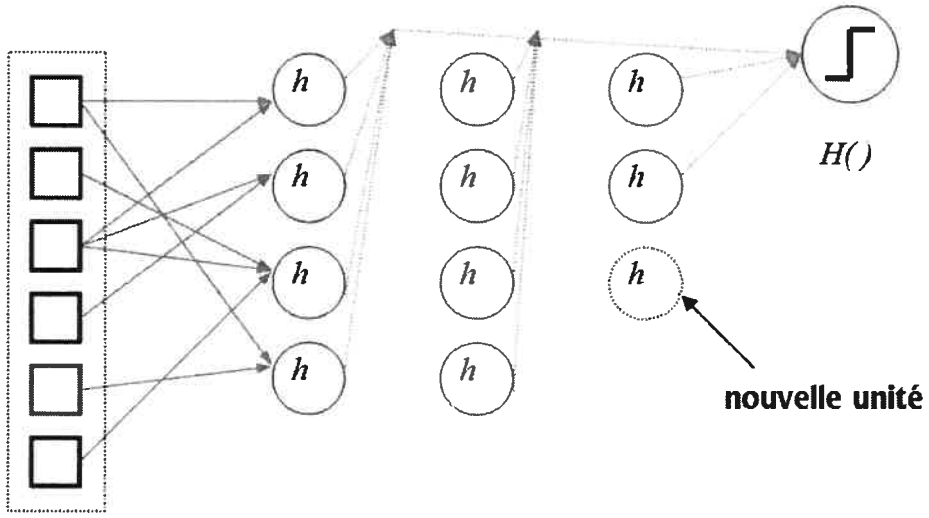

Figure 3.1 – Structure forte

3.1.2 Génération compétitive

Une variante de la hiérarchie forte permet de contourner le dernier problème soulevé ci-dessus. Il s'agit de laisser nos couches ouvertes, de taille indéfinie, en tout temps, et de choisir à chaque itération l'unité la plus utile, indépendamment de la couche sur laquelle elle se trouve. Ceci se rapproche davantage de la vision originale d'AdaBoost, comme sélection vorace, et c'est aussi plus naturel, non au sens de la hiérarchie, mais au sens de l'efficacité. Le léger défaut de cette approche est que les unités d'une même couche n'ont pas nécessairement eu accès aux mêmes données, puisque les couches inférieures ne sont pas complétées. Ceci n'est pas problématique du côté pratique, mais peut être moins élégant du côté théorique. Bien évidemment, la construction des couches supérieures devra attendre que leur source respective (la couche précédente) soit assez garnie. Cependant, avec le temps, on devrait voir les différentes couches se faire concurrence entre elles sur un pied relativement égal.

La construction "organique" des couches permet de voir une autre dimension du boosting hiérarchique : l'importance même des couches. AdaBoost ne crée qu'une seule couche, avec les faiblesses que l'on a notées plus haut. Avec plusieurs couches, créées de manière compétitive, on peut observer à quel point la hiérarchie semble utile. Dans

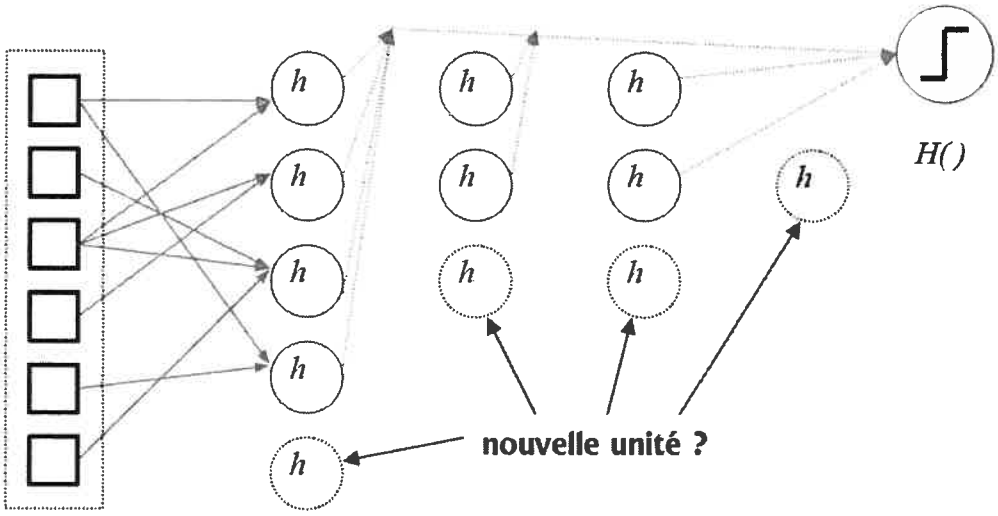

Figure 3.2 – Génération compétitive

un schéma de développement régulier, on s'attend à ce que la première couche se développe rapidement, puis que peu à peu la seconde se bâtisse et prenne plus de place, et ensuite que la troisième s'ajoute, etc. La croissance des couches inférieures devrait ralentir beaucoup, sans nécessairement cesser.

Il faut cependant se garder de considérer ces observations comme étant trop révélatrices. En effet, si les couches supérieures sont préférées rapidement, cela ne veut pas dire qu'elles sont nécessairement plus utiles. L'utilité finale ne pourra être observée qu'avec une vue d'ensemble du comportement de la fonction de sortie. N'oublions pas que le boosting est à la base une méthode vorace et qu'un avantage ponctuel, choisir une couche supérieure plutôt qu'une couche inférieure à une itération t par exemple, n'est pas significatif en général. On se considérera satisfait non pas si la courbe du taux d'erreur d'entraînement diminue plus vite, mais plutôt si celle du taux d'erreur de test atteint un plancher plus bas (qu'avec une seule couche).

3.1.3 Hiérarchie souple

Jusqu'ici, nous avons observé des architectures où les couches sont des niveaux hiérarchiques fixes et séparés. C'est une hiérarchie forte dans le sens où une unité d'une

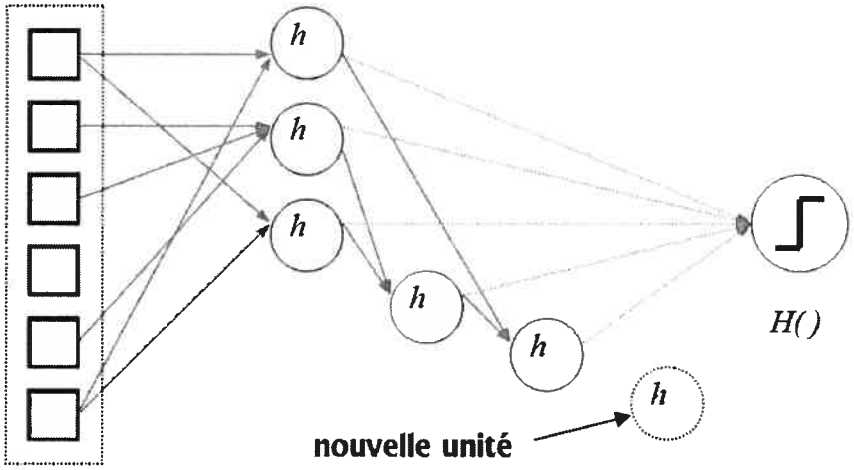

Figure 3.3 – Hiérarchie souple

couche aura comme entrée des valeurs provenant uniquement de la couche précédente, et sa sortie ira alimenter seulement les entrées de la couche suivante. On peut cependant faire autrement avec une hiérarchie souple.

Considérons des unités de différentes couches. Si leur nature est semblable et si leur comportement est également similaire, alors rien ne différencie vraiment ces unités en termes pratiques. La différence est en fait au point de vue conceptuel : ce sont les niveaux sur lesquels se situent les unités en question. Si, en plus, une unité a une sortie du même calibre que ses entrées, alors rien ne nous empêche d'ajouter cette unité dans le même "espace" que les unités qui l'ont nourrie, soit dans la même couche.

De cette manière, on ne crée qu'une seule couche au point de vue symbolique, mais elle s'alimente elle-même. Une nouvelle unité s'alimente à toutes celles existant déjà, peu importe sur quelle couche elles se trouvent. En fait, on se réduit à une ou deux couches seulement. On peut n'avoir qu'une couche dans le cas où les données d'entrée sont compatibles avec la sortie des unités apprenantes. Sinon, on devra avoir deux couches : une première qui apprend sur les valeurs d'entrées, une seconde qui apprend sur toutes les unités générées jusque-là. Évidemment, comme les unités sont apprises itérativement, et que ces unités ne sont pas modifiées par la suite, la création de boucles n'est pas possible.

Dans le cas particulier où nos unités sont des neurones, cette hiérarchie souple s'approche beaucoup de l'algorithme de *Cascade Correlation* [8].

3.1.4 Hiérarchie dégénérée

Il est également possible d'aller chercher un avantage de notre algorithme hiérarchique sans s'encombrer de la structure solide d'une hiérarchie forte. Nous pouvons utiliser la structure hiérarchique seulement pour créer des filtres des données brutes. Ainsi, toutes les unités prennent leurs données dans la couche de données de départ. La hiérarchie servira ici à créer des filtres de plus en plus complexes. La méthode est presque la même que pour une hiérarchie forte, mais lorsqu'on a choisi un filtre des unités de la couches précédente, on remplace ce filtre par un filtre qui se nourrira des données d'entrées. Ce nouveau filtre sera construit de manière récursive à travers les couches précédentes. Par exemple, avec des couches utilisant des filtres doubles (deux unités sont groupées ensembles[1].), un filtre pour la seconde couche d'unités comprend deux unités de la première couche. Ce filtre sera transformé pour prendre les entrées des unités qu'il regroupe. Ainsi, ce filtre, une fois transformé prendra les 4 entrées des 2 unités regroupées (lesquelles avaient chacune 2 entrées dans la couche de données directe). En fait, ce filtre transformé est l'union des filtres des unités qu'il regroupe. Avec cette méthode, nous n'avons plus de hiérarchie effective, mais chaque "couche" représente une sélection différente de filtres, les couches les plus hautes représentant des filtres plus élaborés.

Cette façon de faire entraîne malheureusement la perte de la qualité première du boosting hiérarchique, c'est-à-dire la possibilité de briser la linéarité des décisions. Cependant, l'idée de construction de filtres par couches dégénérées n'est pas nouvelle. Si relie un *stump* et un filtre de Haar, on voit qu'on peut créer un filtre de Haar en deux étapes avec les couches dégénérées : une première couche regroupe des "formes" rectangulaires unies (un filtre rectangulaire), et une seconde regroupe quelques-uns de ces rectangles avec un poids positif (zone noire), d'autres avec un poids négatif (zone blanche). En regroupant ces filtres rectangulaires sous un même filtre, on obtient un filtre

[1] Voir la section 3.2.2.3 pour plus de détails sur ces filtres.

qui est en quelque sorte une généralisation des filtres de Haar. Reprenons l'idée énoncée plus haut de généraliser les filtres de Haar à des données qui ne sont pas nécessairement des images. Nous avons maintenant un moyen de le faire.

3.1.5 Prétraitement

Nous avons noté que plusieurs algorithmes d'apprentissage appliqués à des images utilisent un prétraitement des données. En fait, ce traitement des données est souvent une étape importante, voire essentielle, des algorithmes d'apprentissage. Ceci est particulièrement vrai dans le cas des réseaux de neurones. En fait, rien ne nous empêche de considérer une hypothèse apprise comme étant elle-même un prétraitement quelconque pour une autre étape. On peut l'entraîner avec un but intermédiaire, puis donner les résultats à un autre algorithme qui travaillera sur le problème principal. Dans des algorithmes fortement hiérarchiques, on peut aller plus loin, et identifier clairement des couches qui font du prétraitement, les premières. En fait, ceci est sous-entendu dans le cas de l'apprentissage multitâche d'un réseau de neurones. On ajoute une tâche secondaire justement pour que la couche intermédiaire extraie de meilleures caractéristiques *(features)* afin d'améliorer le résultat final. Dans le cas des réseaux auto-associatifs, une séparation des couches est encore plus facile : la première couche sert à l'encodage, et la seconde au décodage.

Ceci s'inscrit bien dans ce que le boosting hiérarchique nous permet de faire. Non seulement nous pouvons facilement utiliser l'algorithme pour produire des données prétraitées, mais nous pouvons aussi assigner à une ou plusieurs couches le statut de prétraitement, et commencer, ou recommencer, le véritable apprentissage après que ces couches soient entraînées.

3.1.6 Remplacement

Le remplacement est un complément à plusieurs des méthodes d'architectures présentées précédemment. Étant donné la nature du boosting lui-même, la hiérarchisation peut amener une certaine duplication des décisions. Comme chaque unité est déjà une

hypothèse faible valide, et donc naturellement liée à la décision de sortie, réutiliser (recycler) les sorties de ces unités, des décisions déjà compilées, peut sembler redondant. Il pourrait être souhaitable de "neutraliser" l'effet de ces unités sur la sortie une fois qu'elles ont été réutilisées par une couche supérieure. Par exemple, si nous avons cinq unités d'une couche A qui sont regroupées par une nouvelle unité de la couche suivante B, alors nous annulons l'effet de ces cinq unités sur la sortie (si ce n'avait pas déjà été fait), et ensuite nous calculons l'effet de la nouvelle unité sur la sortie. Cette technique nous rapprocherait encore plus d'un réseau de neurones. En utilisant des unités neuronales, comme des perceptrons, nous construirions en effet un réseau de neurones complet, toutes les sorties des couches précédentes étant alors naturellement neutralisées.

Cependant, même si cette approche peut sembler plus efficace, il faut bien considérer les avantages qu'elle nous force à abandonner. Premièrement, il ne s'agit plus vraiment d'une variation d'AdaBoost. On n'ajoute plus seulement une unité à la fonction d'agrégation de boosting, mais on en retire également. On peut cependant considérer que l'impact de la nouvelle unité sur la décision globale est en deux parties : une première, sa véritable sortie, et une seconde, le retrait des sorties de la couche précédente. De cette manière, on reste compatible avec AdaBoost. Le problème est qu'on rajoute un poids de temps de traitement sur l'algorithme, poids qui peut devenir prohibitif. Il ne faut plus seulement trouver la meilleure unité selon la pondération actuelle, mais retirer de la décision l'impact des unités alors neutralisées de la décision, recalculer la distribution des poids sur les exemples, et ensuite pondérer l'influence de cette nouvelle unité.

Dans le cas précis d'une construction avec des unités neuronales, ce poids en temps de calcul n'est pas si important, car en supposant qu'une unité soit liée à toutes les unités précédentes, alors le premier remplacement neutralisera toute la couche précédente, ce qui ne sera pas à refaire dans le futur, pour cette même couche. Dans ce cas, c'est l'équivalent de considérer chaque couche intermédiaire comme un prétraitement indépendant : on crée une couche, on annule sa contribution à la sortie, et on commence à neuf avec la prochaine couche, qui devrait, selon nos souhaits, être plus riche et apprendre mieux.

3.1.7 Réseau complet

Les deux sections précédentes indiquent comment l'algorithme, avec des hypothèses faibles neuronales, ou "neuronables", dans le sens ou on peut en faire des neurones pratiques, des fonctions dérivables, peut générer un réseau de neurones très similaire, au niveau du comportement extérieur, à un perceptron multicouche (PMC). Cependant, la construction d'un réseau par boosting multicouche demande une certaine vigilance. Dans un PMC, les unités cachées ont un rôle discret de représentation : elles extraient des caractéristiques importantes, mais non interprétables (pas facilement du moins), pour que la couche de sortie fasse une meilleure discrimination. Dans le cas du boosting, la stratégie de sélection vorace est un frein au développement précis d'une couche intermédiaire. La génération d'unités de boosting optimise directement la fonction de sortie, peu importe où ces unités se trouvent dans le réseau. Il faudrait donc, pour appliquer *hboost* à la génération de PMC, entraîner les couches intermédiaires avec un but clair de prétraitement, pour que l'entraînement de la dernière couche soit optimal. Au niveau des réseaux de neurones, cet entraînement "prétraitement" des couches médianes est automatique, mais ce n'est pas le cas pour le boosting. Le problème vient du fait qu'on entraîne une seule unité à la fois, indépendamment des autres, sans considérer son impact sur une hiérarchie.

On peut cependant modifier légèrement notre algorithme pour tenir compte de cette réalité différente. On veut maintenant que nos unités sur les couches médianes s'optimisent par rapport au réseau complet. Par exemple, avec un boosting hiérarchique partiel, on aimerait ajouter une unité à une couche médiane de manière à ce qu'elle améliore également les unités suivantes. On ne veut pas que notre hypothèse soit bonne seulement pour elle-même ; on veut qu'elle soit également utile pour les suivantes, qu'elle améliore les autres unités qui lui sont liées. Ainsi, il faudrait considérer les filtres que nous créons, non pas comme des manières de créer un vecteur ω fixe, mais extensible, qui peut s'augmenter de nouvelles valeurs non nulles. Le filtre n'est donc plus un prétexte pour former des groupements, mais une véritable fenêtre dans notre "espace", qui représente un ensemble de points qui peut évoluer.

Cette nouvelle méthode serait bien plus près d'une vraie génération d'un réseau de neurones comme un PMC, que le boosting hiérarchique plus simple décrit précédemment. Elle est cependant plus éloignée du boosting traditionnel, puisqu'on modifie le comportement d'unités déjà existantes. Ces autres unités affectées, sur les couches suivantes, ayant une nouvelle entrée, devront être modifiées et optimisées et leur état changeant, il faudra également changer les unités suivantes qui dépendent de celles-ci, et ainsi de suite. De plus, pour ajouter une unité significative dans le milieu d'un réseau, il faudra un moyen de calculer rapidement son impact sur tout le reste du réseau. Comme les unités sont souvent grandement interreliées, il faudrait alors utiliser une technique similaire à la rétropropagation du gradient, pour s'assurer de pouvoir identifier la nouvelle entrée qui serait la meilleure pour améliorer celles qui existent déjà. Ceci est une théorie qui pourrait s'appliquer, mais sort du cadre actuel de notre algorithme de boosting. Voilà pourquoi elle n'est ni traitée ni testée dans le présent mémoire.

3.2 Filtres

Nous avons jusqu'ici présenté les filtres de manière plus ou moins concrète, avec les filtres de Haar comme cas particulier. Nous allons maintenant préciser les caractéristiques que ces filtres prendront dans notre algorithme. Comme mentionné plus haut, nos filtres auront le plus possible une base géométrique, ou pseudo-géométrique. Pour ce faire, nous allons décrire les deux éléments qui rendront ceci possible : les mesures de l'espace dans lequel nos filtres agiront, et les formes que ces filtres peuvent prendre.

3.2.1 Mesures

Pour acquérir une signification dans notre espace, nos filtres doivent être définis à l'aide de paramètres. Pour ce faire, il faut donner à notre espace une certaine nature, des valeurs auxquelles rattacher ces paramètres. Il existe deux façons distinctes de procéder, quoiqu'interreliées : la première est de bâtir un espace à l'aide d'axes et d'y positionner nos unités (et filtres) selon des coordonnées, la seconde est de bâtir un espace avec une mesure de distance (ou une métrique), et les unités y trouvent place en fonction de la

distance qui les sépare des autres unités. Le type de mesure que l'on choisit pour définir l'espace est important car il détermine également les formes de filtres qui pourront être utilisées.

3.2.1.1 Coordonnées

Le moyen simple de construire un espace est d'extraire des coordonnées de nos unités. Ces coordonnées servent alors à définir les axes de notre espace. Il n'est pas nécessaire que les coordonnées soient sur des axes homogènes pour créer notre espace : par exemple, on peut combiner des axes réels avec des axes discrets, voire des axes booléens. Cependant, avoir des axes homogènes nous permettrait d'ajouter une métrique à notre espace. Ceci peut être nécessaire pour plusieurs formes de filtres. Notons que par la nature même de notre algorithme, chaque unité se trouve avoir des coordonnées par défaut, selon sa sortie produite sur l'ensemble d'entraînement. Ces coordonnées triviales ne sont pas nécessairement très utiles cependant, en tant que coordonnées même, puisqu'on se trouve alors dans un espace de très grande dimension, avec tout ce que cela engendre de problèmes.

3.2.1.2 Distances

Malheureusement, l'utilisation de coordonnées n'est pas nécessairement possible, ni même, parfois, souhaitable. Il est cependant possible de travailler dans un espace métrique[2], c'est-à-dire que les unités n'auront pas besoin de coordonnées pour se placer dans l'espace. Avec un espace défini uniquement à l'aide d'une métrique, on peut alors générer des filtres "coniques" (d'une section conique). Quoiqu'il ne soit pas nécessaire d'avoir à la fois des coordonnées et une mesure de distance, certains algorithmes, tels MDS[3] ou encore IsoMap [44] permettent de créer des coordonnées à partir de la métrique.

La mesure de distance que nous favorisons pour placer les unités est la distance de

[2]ou pseudo-métrique, car il n'est pas essentiel d'avoir une mesure de distance "métrique" à proprement parler, bien que ceci soit plus simple. Il est cependant important que cette mesure soit symétrique.

[3]Multi-Dimensional Scaling

Hamming entre les vecteurs de sorties binaires de nos unités. Une distance sur les sorties est importante puisque notre hypothèse de base est de considérer les unités commes des points de données, et ce sont ces sorties qui seront combinées de toutes façons. Ainsi, en prenant une distance de ces sorties, binaires, pour plaçons les unités comme des points de données dans l'espace.

3.2.2 Formes

L'utilité première d'un filtre dans notre algorithme est de sélectionner un sous-ensemble d'unités et d'assigner à chacune d'entre elles une importance, un poids. Mis à part les cas spéciaux de sous-ensembles triviaux, une seule unité ou toutes, nous devons définir quelles formes géométriquement valides vont prendre nos filtres. Chaque forme sera étudiée selon ses limitations et ses possibilités, également selon sa taille, calculée en nombre d'unités concernées, et le nombre de filtres qui peuvent être créés de cette manière.

Pour formaliser le filtrage, nous allons représenter un filtre comme une fonction qui choisit des poids ω_i (sans lien avec la distribution de boosting) pour les dimensions d'entrée i, ou encore un vecteur ω qui contient les valeurs pour chaque entrée possible. Ces poids auront une valeur nulle pour les unités d'entrée se trouvant hors du filtre et une valeur non nulle pour celles se trouvant à l'intérieur. Si les filtres admettent un contraste, ces valeurs pourront même être positives ou négatives. Par contraste ici, on parle d'un filtre qui ne fait pas que joindre deux unités (par addition ou union) mais peut également les comparer, par contraste. D'une certaine façon, nous pouvons considérer ces poids comme ceux d'un réseau de neurones, avec la valeur d'un filtre – en fait la valeur d'entrée de la fonction d'apprentissage présente – calculée à la manière de l'entrée d'un neurone. Plus formellement, on donnera le poids ω_i d'une entrée i (représentant le poids de l'unité u_i) dans un filtre par la formule générale suivante :

$$\omega_i = \begin{cases} f(u_i) & \text{si } u_i \text{ est à l'intérieur du filtre,} \\ 0 & \text{si } u_i \text{ est à l'extérieur du filtre.} \end{cases} \tag{3.1}$$

Ces poids sont rassemblés dans un vecteur $\omega = \omega_1, \ldots, \omega_l$ où l est la taille de la

couche dans laquelle le filtre travaille. Ce vecteur peut normalement servir à générer le résultat d'un filtre comme le vecteur de poids n'un neurone, traditionnellement, avec un produit scalaire. Les tableaux 3.1 et 3.2 à la fin de ce chapitre résument les caractéristiques des différentes formes filtres ainsi que leur taille.

3.2.2.1 Filtre simple

La première forme est une des formes triviales. C'est en fait le filtre qu'utilise un *decision stump* standard. Ce filtre prend en fait une seule unité I. Comme le poids de cette unité n'a aucune importance, il prendra la valeur 1 par défaut. Cette forme n'a pas besoin de la notion d'espace pour être significative. On peut cependant lui donner une interprétation géométrique comme "point", ou encore comme spécialisation (cas dégénéré) d'un filtre sphérique de rayon infiniment petit (quoiqu'en pratique, un rayon de zéro vaudrait tout autant). On peut créer autant de filtres simples qu'il y a d'unités dans l'espace (l), dans la couche précédente.

$$\omega_{i\{simple_I\}} = \begin{cases} 1 & \text{si } i = I, \\ 0 & \text{sinon.} \end{cases} \tag{3.2}$$

3.2.2.2 Filtre rectangulaire

Le filtre rectangulaire (en fait, hyper-rectangulaire) est une généralisation des filtres de Haar pour un espace de dimension arbitraire. Ce type de filtre exige un espace avec coordonnées. Le filtre représente une zone rectangulaire dans l'espace, un ensemble d'intervalles sur les axes (un intervalle par axe). Les unités localisées dans tous ces intervalles sont considérées comme faisant partie du rectangle. Pour créer un tel rectangle, on utilise deux unités dans l'espace avec coordonnées. On crée les intervalles du rectangle avec les différences entre les coordonnées des deux unités. De cette façon, chaque couple d'unités distinctes dans l'espace définit un rectangle. Ceci n'est pas tout à fait vrai dans le cas où deux unités ont une coordonnée égale. Dans ce cas, un des intervalles est de longueur nulle et le "volume" de notre rectangle est nul. Cependant, ceci n'indique pas que le rectangle est vide. L'intervalle n'est pas totalement vide : il représente un singleton. En fait,

chaque rectangle a l'assurance d'inclure au moins deux points I et J – les deux points qui l'ont généré. L'équation 3.3 montre formellement les poids des entrées du filtre, avec $z_{i,j}$ représentatant la position de l'unité u_i sur la dimension j.

$$\omega_{i\{rect_{IJ}\}} = \begin{cases} 1 & \text{si } \forall j \in \{1,\ldots,d\}, z_{i,j} \geq \min(z_{I,j}, z_{J,j}) \text{ et} \\ & z_{i,j} \leq \max(z_{I,j}, z_{J,j}), \\ 0 & \text{sinon.} \end{cases} \tag{3.3}$$

Il est possible, et même facile, d'inclure un élément de contraste dans un filtre rectangulaire. Il suffit de séparer un ou plusieurs axes pour diviser le rectangle en sous-rectangles de poids différents. On choisit en alternance des zones positives (noires) et des zones négatives (blanches), dont le poids des unités y étant présentes sera respectivement 1 et -1.

Le filtre rectangulaire se prête très bien au traitement des images. En effet, par la structure dense des images (une image est une carte de pixels où chaque couple de coordonnées est un pixel qui existe), on peut calculer facilement la somme des pixels dans un rectangle en passant par une image intégrale, comme le font les filtres de Haar. Les images se prêtent mieux également au jeu des contrastes, car on sait exactement le nombre d'unités, de pixels, présents dans un rectangle donné. On peut alors faire un contraste avec des zones positives et négatives de taille égale. Tel qu'indiqué ci-haut, on peut créer environ l^2 rectangles avec l unités, ou $l^2 f$ avec f types de subdivisions

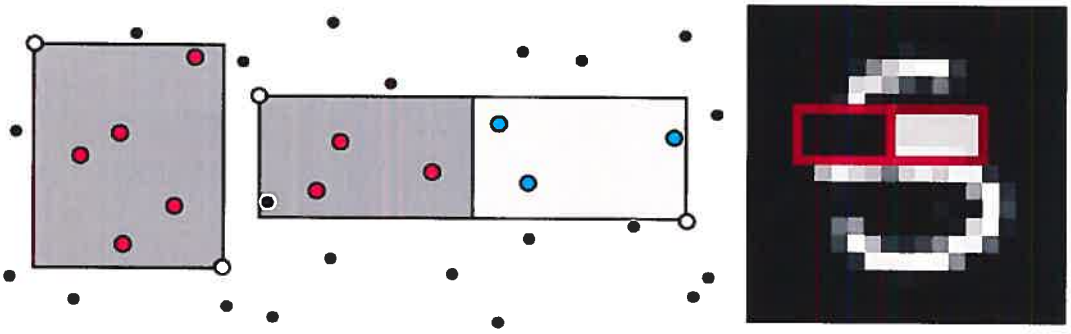

Figure 3.4 – Filtres rectangulaires, sans contraste, avec contraste, et filtre de Haar sur un 5 tiré de MNIST

contrastées. Chaque rectangle inclura de 2 à l unités.

On peut également créer des filtres rectangulaires sans utiliser les points, en choisissant les coordonnées de manière extérieure. Dans ce cas, cependant, on pourrait se retrouver avec des rectangles vides.

3.2.2.3 Filtre double

Le filtre double est une extension du filtre simple, et de cette manière, il n'a pas non plus besoin d'un espace. Au lieu de prendre une unité, on en prend deux. Leurs poids seront soit égaux (1), soit opposés (1 et -1), pour faire un contraste. Si ces entrées sont ensuite additionnées et qu'elles représentent des entrées binaires, alors la somme représentera en fait une opération binaire sur ces deux entrées : et, ou (avec contraste : "et non", "ou non"). Si nos unités sont placées dans un espace muni d'une distance, on peut aussi limiter la création selon la distance entre les deux unités, avec un maximum ou encore un minimum de distance requise. Il est possible d'obtenir un nombre de filtres de l'ordre de l^2 avec cette technique.

$$\omega_{i\{double_{IJ}\}} = \begin{cases} 1 & \text{si } i = I \text{ ou } i = J, \\ 0 & \text{sinon.} \end{cases} \tag{3.4}$$

$$\omega_{i\{doublecontraste_{IJ}\}} = \begin{cases} 1 & \text{si } i = I, \\ -1 & \text{si } i = J, \\ 0 & \text{sinon.} \end{cases} \tag{3.5}$$

En généralisant, on peut également créer des filtres "n-tuples", avec n éléments chacun, mais ceci engendre le problème bien simple d'une explosion des possibilités ($O(l^n)$).

3.2.2.4 Filtre sphérique

Le filtre (hyper-)sphérique est le filtre conique le plus simple et le plus naturel. Il nécessite un espace muni d'une distance d et deux paramètres : un centre et un rayon. Bien qu'on note généralement le centre d'une sphère par des coordonnées dans l'espace, on

peut aussi bien noter le centre de notre sphère de manière symbolique en la centrant sur une unité. Avec une matrice de distances des unités entre elles, il est simple d'identifier les unités incluses dans la sphère. De manière pratique, on créera un filtre sphérique par un couple d'unités, la première (x_I) représentant le centre, la seconde (x_J) identifiant de manière indirecte le rayon, par la distance entre les deux unités : $r = d(x_I, x_J)$. On obtient donc l^2 filtres possibles. On peut, bien sûr, choisir des rayons indépendamment des distances inter-unités, mais on n'obtiendra pas plus de filtres pratiquement différents de cette manière, parce que toutes les unités étant connues, tous les groupements circulaires possibles avec une unité comme centre seront générés par notre technique. Les autres rayons possibles ne créeront pas de groupements différents.

$$\omega_{i\{circ_{I,r}\}} = \begin{cases} 1 & \text{si } d(x_i, x_I) \leq r, \\ 0 & \text{sinon.} \end{cases} \tag{3.6}$$

Il est possible de créer un contraste avec nos filtres sphériques, mais ceci n'est pas aussi instinctif que dans les autres cas. Il s'agit de donner un deuxième rayon r_2, plus grand que le premier, qui induira une zone contour de poids négatif. Il est également possible de définir un contraste par plan si nous avons un espace avec coordonnées, mais ceci implique plus de limitations pour notre espace.

$$\omega_{i\{circcontraste_{I,r,r_2}\}} = \begin{cases} 1 & \text{si } d(x_i, x_I) \leq r, \\ -1 & \text{si } d(x_i, x_I) > r \text{ et } d(x_i, x_I) \leq r_2, \\ 0 & \text{sinon.} \end{cases} \tag{3.7}$$

On peut également utiliser le filtre sphérique, les distances et une gaussienne pour générer un filtre gaussien, ou "de Parzen"[4], en pondérant les unités selon leur distance. Cette manière de faire nous permet d'inclure tous les points dans notre sphère. Cependant, si cette technique nous libère de la contrainte d'un rayon, elle nécessite le choix d'un écart-type pour la gaussienne.

[4] Dans la vision où un cercle sphérique standard ressemble à un K-plus-proche-voisins, le filtre gaussien ressemble à une fenêtre de Parzen. Notons cependant que le filtre sert de regroupement, et non d'algorithme comme les K-plus-proches-voisins et les fenêtres de Parzen.

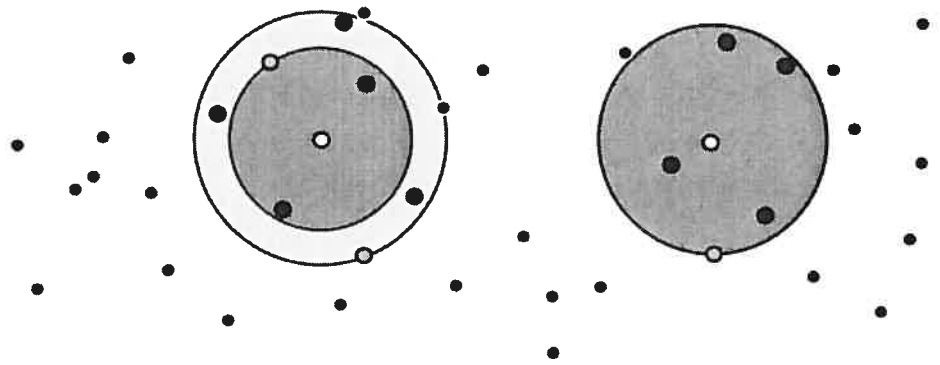

Figure 3.5 – Filtres sphériques, avec et sans contraste

$$\omega_{i\{circparzen_{I,\sigma}\}} = \frac{1}{\sigma\sqrt{2\pi}} e^{-d(x_i,x_I)^2/(2\sigma^2)} \tag{3.8}$$

3.2.2.5 Filtre ellipsoïdal

Le second type de filtre conique que nous allons utiliser est le filtre ellipsoïdal. Il se construit comme le filtre sphérique, mais nécessite trois points plutôt que deux : deux foyers, et un troisième point qui donnera le "rayon" r de l'ellipse. De cette façon, le nombre d'ellipses possible est de l'ordre de l^3. On peut également appliquer la technique "Parzen" à ce type de filtre, en utilisant la distance avec les deux foyers comme mesure.

$$\omega_{i\{ellipse_{I,J,r}\}} = \begin{cases} 1 & \text{si } d_{ellipse}(x_i,x_I,x_J) < r, \\ 0 & \text{sinon.} \end{cases} \tag{3.9}$$

Le grand avantage d'un filtre ellipsoïde est la construction de contrastes, qui est visuellement beaucoup plus significative. Une première forme inscrit l'ellipse dans une sphère[5], de sorte qu'on obtient quelque chose de similaire à un filtre de Haar en trois parties (pour 2 dimensions). Dans ce cas, si les deux foyers sont très rapprochés et si le rayon est grand, on aura une ellipse presque sphérique, réduisant ainsi grandement

[5]Le centre de cette sphère n'existe pas vraiment dans un espace sans coordonnées, mais on peut quand même se servir de la loi des sinus et de la loi des cosinus pour calculer la distance d'un point avec ce centre, avec l'aide des points focaux de l'ellipse, le centre du cercle étant positionné à mi-chemin entre les deux.

la zone de contraste. Dans le cas contraire, c'est l'opposé qui arrivera, un trop grand volume de contraste pour un volume d'ellipse trop petit. Une autre façon de créer du contraste est de séparer simplement l'ellipse en deux par un plan équidistant des deux foyers. De cette manière, on obtient deux zones de volume égal. Cependant, comme dans le cas des filtres rectangulaires, un volume égal ne signifie pas qu'on aura une densité de points égale dans chaque zone.

$$\omega_{i\{ellipse-sphere_{I,J,r}\}} = \begin{cases} 1 & \text{si } d_{ellipse}(x_i,x_I,x_J) \leq r, \\ -1 & \text{si } d_{ellipse}(x_i,x_I,x_J) > r \text{ et } d(x_i,centre), \\ 0 & \text{sinon.} \end{cases} \tag{3.10}$$

$$\omega_{i\{ellipse-plan_{I,J,r}\}} = \begin{cases} 1 & \text{si } d_{ellipse}(x_i,x_I,x_J) \leq r \text{ et } d(x_i,x_I) < d(x_i,x_J), \\ -1 & \text{si } d_{ellipse}(x_i,x_I,x_J) \leq r \text{ et } d(x_i,x_I) \geq d(x_i,x_J), \\ 0 & \text{sinon.} \end{cases} \tag{3.11}$$

Tout comme le filtre sphérique, on peut se servir de nos distances pour donner des poids à tous les éléments à l'aide d'une gaussienne.

$$\omega_{i\{ellipse-parzen_{I,J,r}\}} = \frac{1}{\sigma\sqrt{2\pi}} e^{-d_{ellipse}(x_i,x_I,x_J)^2/2\sigma^2} \tag{3.12}$$

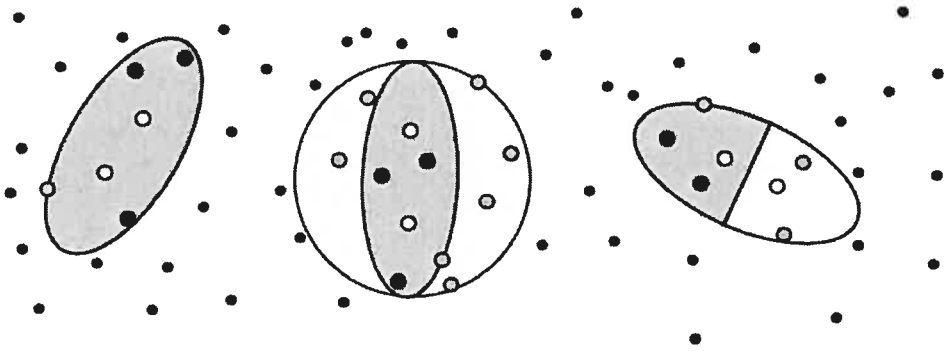

Figure 3.6 – Filtres ellipsoïdes, avec et sans contraste

3.2.2.6 Filtre libre

À l'opposé du filtre simple, l'autre cas trivial de sous-groupe est le groupe lui-même. Ceci donne une seule taille de filtre possible, un filtre représentant toutes les unités. Celui-ci est de toute façon représentable comme un filtre sphérique de rayon maximal (si la distance est bornée, sinon, il correspond à l'infini). Ceci ne nous donne pas un espace de filtres très riche (un seul filtre possible), ni très intéressant. Cependant, si on donne des poids arbitraires différents à chaque élément, on peut enrichir cet espace. Bien entendu, ceci nous donne alors une possibilité infinie de combinaisons de poids. Il existe cependant un algorithme bien connu pour trouver une bonne solution, un bon filtre, dans ces conditions : l'algorithme du perceptron.

Toutefois, la méthode du perceptron ne correspond pas à nos souhaits en matière de filtre. Tous les autres types de filtres sont construits à partir des données, et ce sont des filtres qui sont valides géométriquement parlant. Un filtre libre de type perceptron est quant à lui entraîné avec un but. Ceci devrait donner de meilleurs filtres, mais ne se justifie pas géométriquement parlant. C'est aussi contraire à la séparation que nous avons établie entre les filtres et l'algorithme d'apprentissage.

Forme de filtre	**Type de mesure nécessaire**	**Possibilité de contraste**	**Possibilité de nuances**
Simple	aucun	non	non
Rectangulaire	coordonnées	oui	non
Double	aucun	oui	non
Circulaire	distance	oui	oui
Ellipsoïde	distance	oui	oui
Libre	aucun	oui	oui

Tableau 3.1 – Caractéristiques des filtres

Type de filtre	**Nombre de filtres possibles**	**Nombre d'elements par filtre**
Simple	l	1
Rectangulaire	$O(l^2)$	$(2,l)$
Rectangulaire (contraste)	$O(l^2d)$	$(2,l)$
Double (avec ou sans contraste)	$O(l^2)$	2
sphérique	$O(l^2)$	$(1,l)$
Sphérique (contraste)	$O(l^3)$	$(2,l)$
Sphérique (parzen)	l ou ld	l
Ellipsoïde	$O(l^3)$	$(2,l)$
Ellipsoïde (contraste - cercle)	$O(l^3)$	$(2,l)$
Ellipsoïde (contraste - demie)	$O(l^3)$	$(2,l)$
Ellipsoïde (parzen)	l ou ld	l

Tableau 3.2 – Taille et nombre de filtres

CHAPITRE 4

TESTS ET EXPERIENCES

Nous exposons dans ce chapitre les divers résultats donnés par notre algorithme. Dans la partie 4.1, nous décrivons d'abord les jeux de données utilisés. Suivra la décription les différents hyperparamètres de notre modèle, ceux qui sont constants, dans notre cas, et ceux qu'il est possible d'optimiser. Nous montrons ensuite les résultats expérimentaux reliés aux hyperparamètres à choisir. Nous terminons avec quelques notes sur le comportement pratique de notre algorithme dans la section 4.3.4.

4.1 Jeux de données

Comme notre algorithme se base sur une technique de traitement d'images, il est naturel que notre comparaison de test se base sur le même genre de données : des images. En fait, pour tester au maximum la qualité de notre algorithme, il faudrait justement savoir si on atteint la même qualité avec un traitement explicite d'images, qu'avec notre méthode qui ne prend pas d'informations a priori sur la nature des données.

4.1.1 MNIST

Le jeu de données d'images par excellence est MNIST. Il s'agit d'un jeu de 70 000 images de chiffres manuscrits décrit par LeCun et al. [23] (60 000 données d'entraînement, et 10 000 de test, selon la partition officielle des données). Ces images ont été normalisées en tailles et centrées, pour faciliter le traitement. Ce jeu de données a été grandement utilisé pour comparer de nombreux algorithmes d'apprentissage et sert encore aujourd'hui pour valider de nouveaux algorithmes ou certaines propriétés "visuelles" d'apprentissage. Ce jeu de données ne pose plus lui-même de problème important, comme on arrive à bien classifier ses éléments avec des taux d'erreurs très faibles (Simard et al. ont obtenu 0.4 % d'erreur en 2003 [43]), mais il reste cependant assez intéressant (lire ici non trivial) pour que des comparaisons pratiques soient révélatrices.

MNIST est également un ensemble d'images (relativement) précises, celles-ci étant de 28×28 pixels (en tons de gris). Cependant, cette précision vient à un prix certain de gestion mémoire. On a donc 70000×784 données au total, ce qui n'est pas nécessairement prohibitif pour une machine moderne, mais qui dans notre cas, augure mal. Notre algorithme a comme particularité de générer de nouvelles données – pas de nouveaux exemples, mais de nouvelles dimensions – et la sortie de chaque unité pour chaque exemple est gardée en mémoire. Ceci signifie donc que dans le cas de MNIST, chaque itération, qui crée une nouvelle unité, ajoute 70 000 nouvelles données à mémoriser. Maintenant, dans un ordre de grandeur habituel de boosting, le nombre d'itérations est très grand (dans notre cas jusqu'à 10 000), et cela représente une quantité très impressionnante de mémoire à gérer. En fait, nous avons rencontré la limite pratique de mémoire dans plusieurs nos expériences multicouches avec MNIST[1]. Changer de jeu de données (utiliser USPS à la place) a été dans ce cas une solution pratique facile, qui représente environ 10 % moins de mémoire à gérer.

4.1.2 USPS

USPS handwritten digits, que nous noterons simplement USPS, est un jeu de données similaire à MNIST, quoique plus ancien. Il s'agit ici aussi d'un ensemble d'images de chiffres manuscrits, mais de dimension moindre (16×16), et également plus bruitées. Ceci donne une classification plus difficile. USPS contient 7291 valeurs d'entraînement et 2007 valeurs de test. Nous avons utilisé ce jeu de données, plutôt que MNIST, pour la totalité des expériences présentées ici.

4.2 Hyperparamètres

Une des qualités reconnues du boosting est le peu d'ajustement nécessaire pour en tirer le maximum. La version typique d'AdaBoost n'a qu'un seul hyperparamètre, soit le nombre d'itérations. De son côté, notre boosting hiérarchique complique la chose. Ce

[1] Ces limitations techniques sont également dépendantes de l'implémentation du code utilisée pour ce mémoire, lequel code n'est pas optimal dans son utilisation de mémoire. Notons cependant que l'algorithme *hboost* sera toujours très vorace en mémoire.

nouvel algorithme dépend de plusieurs heuristiques et de nombreux éléments peuvent être modifiés. Voici une liste résumant ces décisions :

- Architecture
  - Le nombre de couches
  - La taille de chacune des couches
  - Le type de structure (forte, souple ou dégénérée)
  - La stratégie d'apprentissage
- L'apprenant faible
- Filtrage
  - Nature de l'espace (distance ou coordonnées)
  - Mesures utilisées pour placer les éléments dans l'espace
  - Forme des filtres
  - Paramètres des filtres (restrictions, contraste, etc.)
- Le type de boosting
- Profondeur de la recherche (nombre de filtres testés parmi tous ceux qui sont générés)

4.2.1 Hyperparamètres constants

Bien évidemment, tester tous ces paramètres en même temps est un plan plutôt audacieux. Notons que nous omettons plusieurs de ces tests pour préciser nos expériences sur les nouveaux éléments. D'abord, comme nous avons mentionné au départ, nous utilisons AdaBoost comme base de boosting, et plus particulièrement AdaBoost.MH pour le multiclasse. D'autres versions du boosting seraient également compatibles avec la hiérarchisation que nous proposons. Le choix de notre apprenant faible est également fixé. Pour les raisons énoncées dans la section 2.6, nous utilisons le *decision stump* à la fois pour sa simplicité et pour son peu de profondeur. C'est à un tel apprenant faible que bénéficierait le plus la structure hiérarchique que nous proposons ici.

Dans la plupart des cas, nous utilisons un espace de distances pour placer nos unités. Ces distances sont générées à partir des sorties binaires de ces unités sur les données d'entraînement. Ce choix est principalement fait en fonction de la nature des groupe-

ments eux-mêmes. Comme ce sont les décisions des unités qui seront combinées, il est naturel que ce soit ces valeurs qui décident de l'organisation des unités dans l'espace. Dans le cas où nous bâtissons une hiérarchie non pas sur des *decision stumps* à filtre simple, mais avec filtres de Haar (sur la première couche), il nous est permis facilement de créer un espace de coordonnées. Les coordonnées que nous utilisons dans ce cas sont les paramètres de nos filtres de Haar. Ceci donne dix dimensions, deux pour la position du filtre, deux pour sa taille (hauteur et largeur), et six valeurs binaires pour le type de filtre.

Notons également que dans le cas des filtres de Haar, plutôt que de faire une recherche poussée dans l'espace des filtres, nous utilisons une optimisation locale discrète pour améliorer notre filtre. Il s'agit, à partir d'un filtre, de modifier légèrement ses coordonnées pour voir quel "voisin" est le meilleur, et on change alors pour le voisin, et ainsi de suite pour plusieurs itérations. Dans ces cas, nous limitons cette recherche à 30 itérations.

4.2.2 Hyperparamètres à optimiser

Il reste plusieurs hyperparamètres à choisir, et nous avons testé et comparé ces différents choix. Notons que le seul nombre de permutations de ces paramètres et heuristiques ainsi que le temps de calcul important nous empêchent de tester exhaustivement le problème. Cependant, la plupart des éléments discutés dans les chapitres précédents sont testés et exposés ici.

Tout d'abord, nous testons les différentes structures de réseaux de boosting hiérarchique. Ceci inclut le nombre de couches, leur type et la technique de génération des unités sur ces différentes couches (compétitive ou progressive). Nous testons également différents types de filtres, soit les filtre doubles, rectangulaires, sphériques et ellipsoïdaux. Pour chacune de ces formes, nous testons également ses paramètres, que ce soit les contraintes possibles sur les filtres ou les contrastes.

4.3 Tests

4.3.1 Résultats de référence

Avant de plonger dans les expériences du boosting hiérarchique, montrons ce que les techniques non hiérarchiques donnent comme résultats, et notons les espoirs qu'une méthode hiérarchique nous permet.

La figure 4.1 montre le comportement de trois exécutions de boosting traditionnel avec le jeu de données USPS. La première montre l'utilisation de *decision stumps* simples, c'est-à-dire, sur une seule dimension, donc ici un seul pixel. La seconde montre l'utilisation de *stumps* avec les filtres de Haar. Enfin, le troisième couple de courbes montre l'utilisation de filtres rectangulaires noirs. Il s'agit en fait de filtres de Haar dégénérés, utilisés sans contraste.

Expérience	**Taux d'erreur d'entraînement**	**Taux d'erreur de test**
1 valeur seulement	0,0 %	6,4 %
Filtres de Haar	0,0 %	4,0 %
Filtres noirs	0,0 %	5,9 %

Tableau 4.1 – Résultats de Référence

D'abord, notons que l'utilisation des filtres, de Haar ou noirs, apportent une amélioration significative quant au choix des valeurs à passer à travers le *stump*. Cependant, tous les filtres ne sont pas égaux, et les filtres de Haar performent bien mieux que les filtres noirs. Cela n'est pas une grande surprise, et incite plutôt à penser que la notion de contraste est très importante.

Quant aux niveaux hiérarchiques, notons que nous ne souhaitons pas nécessairement ici battre l'état de l'art de la classification sur ce jeu de données (USPS), mais plutôt voir s'il est possible d'atteindre le bon niveau de qualité des filtres de Haar par un filtrage qui ne dépend pas des images, un filtrage organisé, justement, par boosting hiérarchique.

Dans les tests subséquents, nous illustrons ainsi le progrès obtenu en comparant nos résultats à ceux de la version la plus simple de boosting avec des *decision stumps*, et à l'autre extrémité, celui des filtres de Haar.

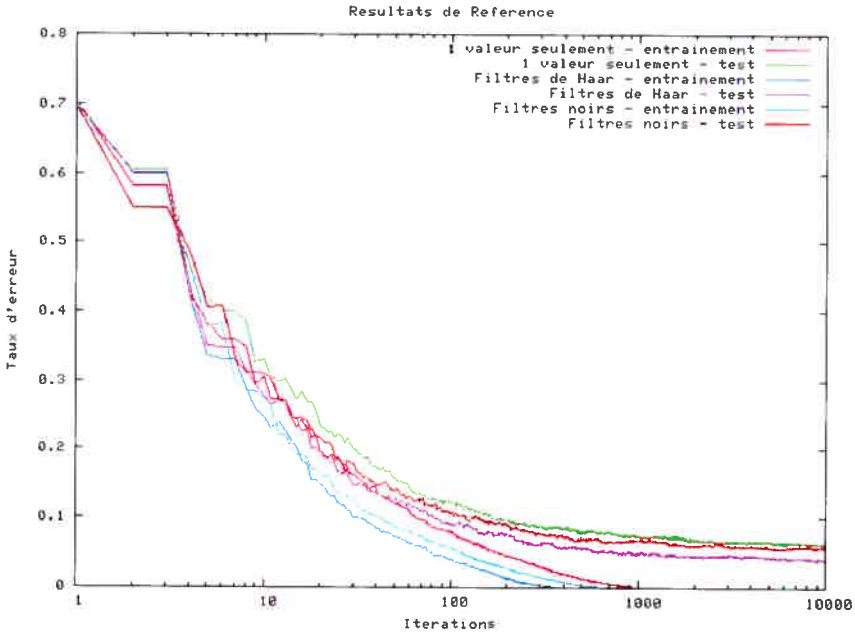

Figure 4.1 – Résultats de référence

4.3.2 Tests de structure

Le premier test à relever est l'utilité même de la hiérarchie. Au-delà de la question à savoir quel nombre de couches est optimal, il faut d'abord déterminer si ces couches sont d'abord bénéfiques à l'apprentissage. Les couches supérieures, ici, sont formées avec des filtres doubles alors que les premières couches ont des *stumps* simples.

Comme montré par les figures 4.2 et 4.3, la structure hiérarchique est clairement bénéfique (par rapport à aucune hiérarchie, sur les mêmes hypothèses et les même filtres). Nos expériences ici montrent la génération d'une seconde couche, celle-ci construite avec des filtres doubles, les filtres non triviaux les plus simples. Pour chacune des itérations de cette couche, nous avons sélectionné le meilleur filtre parmi mille. Ces expériences utilisent les mêmes hypothèses pour la première couche. Les couches supérieures sont lancées après mille itérations de la première couche de *stumps* standards. De fait, cette première couche semble avoir presque convergé et l'addition d'une nouvelle couche donne un nouveau souffle à l'apprentissage. Cependant, les autres couches subséquentes ne semblent pas avoir un effet majeur, et la différence entre cinq couches et deux n'est

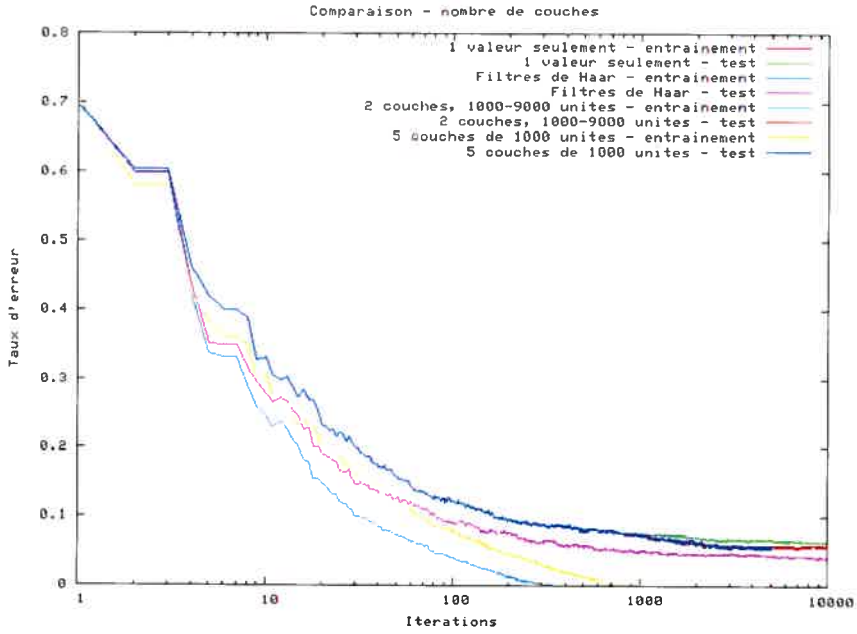

Figure 4.2 – Nombre de couches

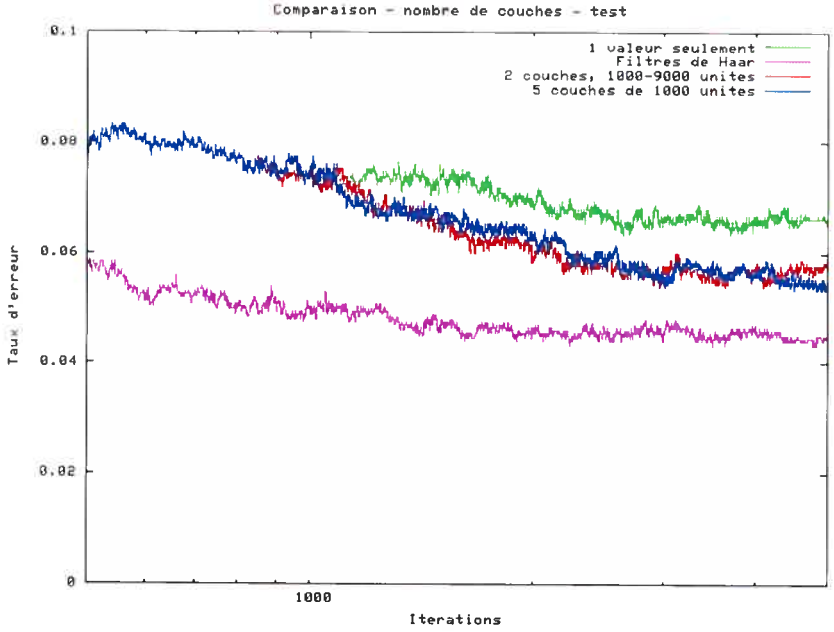

Figure 4.3 – Nombre de couches (zoom)

pas notable.

Nous avons testé les différentes structures selon plusieurs paramètres. Le premier test crée cinq couches fortes, générées séquentiellement. Le second crée également cinq couches fortes, mais générées de manière compétitive, à partir de 1000 unités sur la première couche. On répète ensuite le même processus, mais la génération est compétitive depuis le départ. Le dernier test montre la génération d'une hiérarchie souple, donc deux couches seulement, de manière compétitive. Toutes les couche supérieures ici sont composées de *stumps* avec des filtres doubles sans contraintes.

Les figures 4.4 et 4.5 illustrent le comportement de ces différents designs de structure. Ces graphiques montrent qu'il n'y a pas vraiment de vainqueur évident ici, et que si l'existence de couches supérieures est importante, la manière dont celles-ci sont créées l'est beaucoup moins.

4.3.3 Tests de filtres

Dans la section 3.2.2, nous avons décrit plusieurs types de filtres, plusieurs formes de filtres. Quels sont les avantages des uns par rapport aux autres ? Nous avons déjà noté les différentes caractéristiques de ceux-ci : taille, contraste, etc. Est-ce que ces caractéristiques peuvent rendre une forme meilleure que les autres ? C'est ce que nous allons voir ici.

4.3.3.1 Filtres doubles

Débutons ces tests par les filtres doubles (3.2.2.3). Il s'agit d'un type de filtre presque trivial, qui n'a pas besoin de géométrie pour assurer son existence. Nous commençons par ce type en particulier à cause de sa simplicité. La première comparaison a trait aux possibilités de contrastes. Les figures 4.6 et 4.7 montrent les résultats. L'espace des filtres contrastés ne semble a priori pas plus précis que celui des filtres non contrastés. Nous en donnerons la raison par la suite.

Nous avons déjà mentionné que les filtres doubles n'ont pas besoin de la notion d'espace pour exister, mais que nous pouvons leur donner un peu plus de sens, ou du

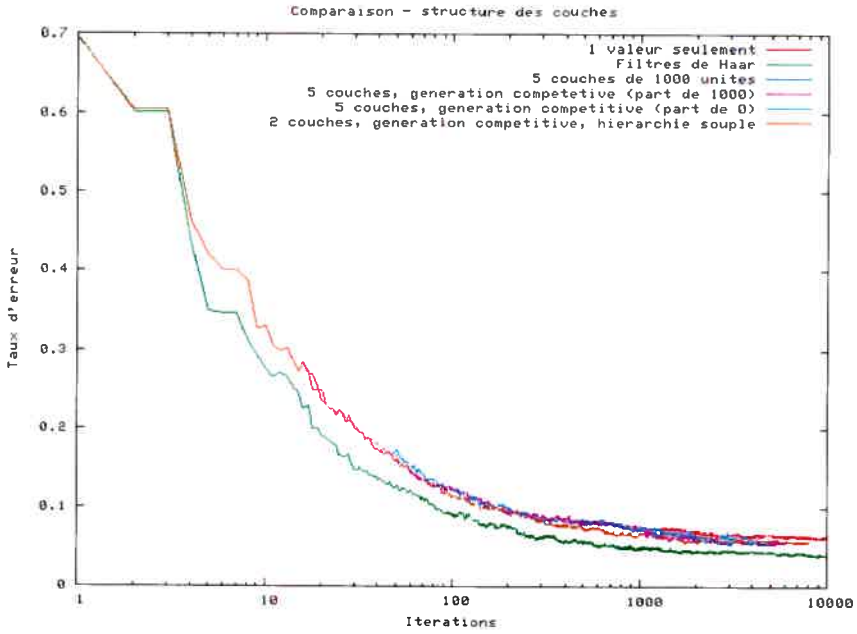

Figure 4.4 – Structure des couches

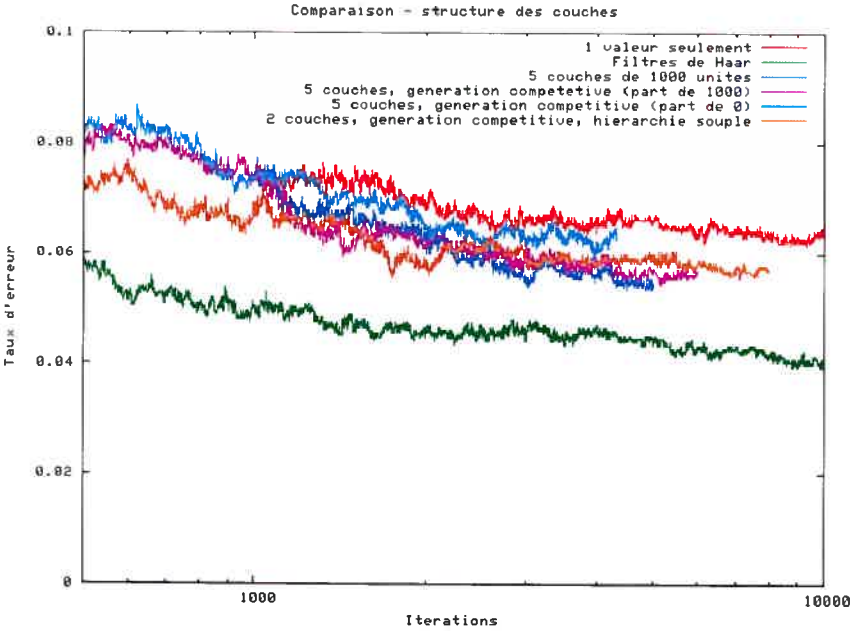

Figure 4.5 – Structure des couches (zoom)

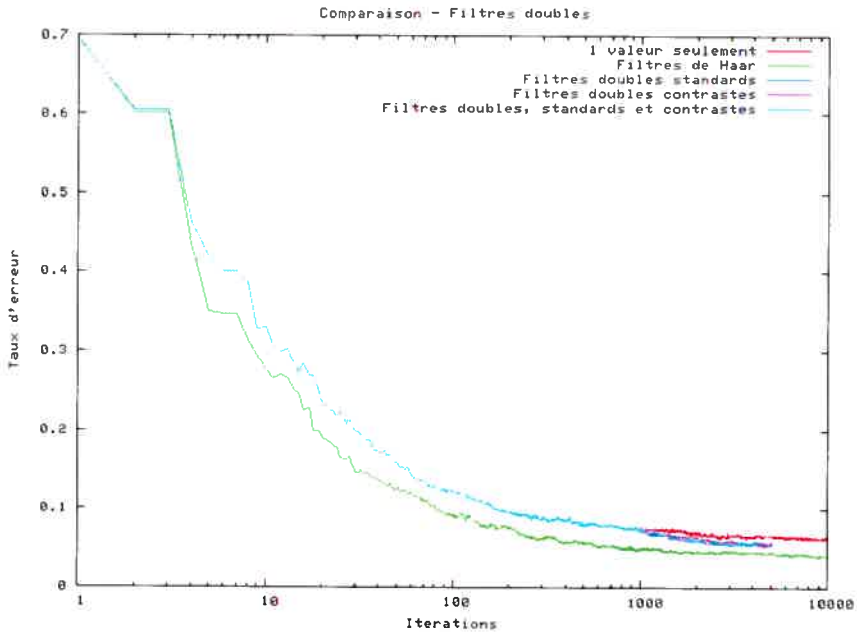

Figure 4.6 – Filtres doubles – sans contraintes

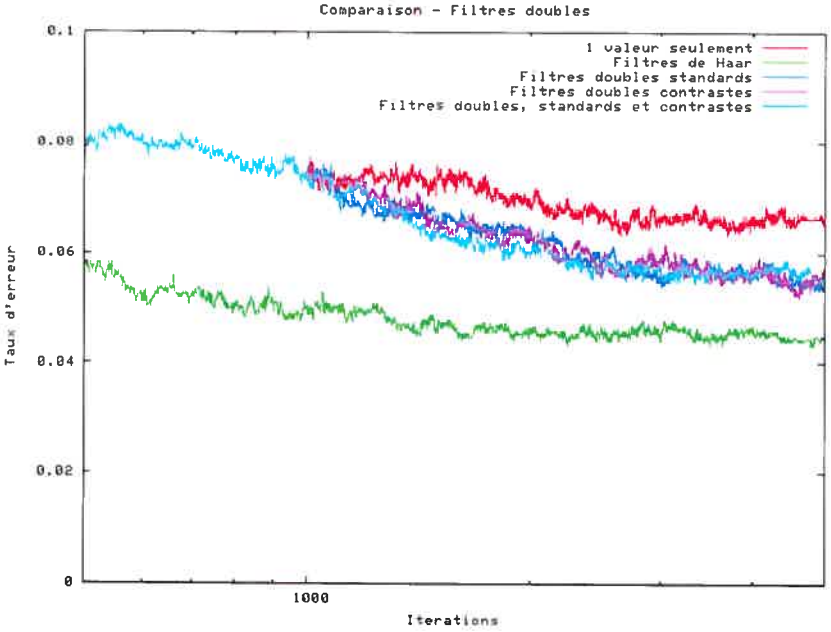

Figure 4.7 – Filtres doubles – sans contraintes (zoom)

moins, leur ajouter des contraintes. En effet, notre hypothèse de base par rapport aux filtres est qu'une géométrie vaut la peine d'être exploitée pour assister à la création des filtres. Pour les filtres doubles, une contrainte géométrique est possible : de limiter à un intervalle de distance les unités pouvant être jointe dans un filtre. Regardons ce que cela donne avec des filtres standards et avec des filtres contrastés.

Les figures 4.8, 4.9, 4.10 et 4.11 illustrent les résultats de ces tests de contraintes. On voit ici que parmi les trois contraintes (unités proches, moyennement éloignées, et très éloignés) testées dans les deux cas, un seul type de contrainte est vraiment profitable (sauf le cas "sans contraintes"). Et encore, c'est le même type de contrainte – les unités moyennement éloignées – qui donnent les meilleurs résultats dans les deux cas. Dans les autres cas, on s'approche des performances des *decision stumps* de la première couche.

On se serait attendus ici à ce que les valeurs proches donnent de bons résultats avec les filtres standards, et que les valeurs éloignées donnent de bons résultats avec les filtres avec contraste. Ceci semble infirmer notre hypothèse selon laquelle les valeurs proches font de bons groupements. Ici, comme ce sont les valeurs moyennement éloignées qui donnent de bons résultats, il paraît difficile d'en déduire une bonne règle géométrique de groupement.

4.3.3.2 Filtres circulaires

Passons maintenant aux filtres sphériques. Ceux-ci sont faciles à créer, tout en concrétisant les filtres géométriques que nous voulions avoir. Le premier test revoit encore les différentes structures, tel que montré avec les filtres doubles ci-haut. Nous avons utilisé pour ces tests des filtres circulaires sans contraintes et sans contraste, en prenant pour chaque hypothèse des couches supérieures, le meilleur filtre parmi cent. Ces tests montrent la génération séquentielle de même que la génération compétitive d'hypothèses dans une hiérarchie forte, sur 5 couches ou sur seulement 2.

Les figures 4.12 et 4.13 montrent les résultats de ces tests. On remarque d'abord que, comme dans le cas des filtres doubles, il y a peu de différence entre les différentes structures. Cependant, dans le cas des filtres circulaires, on remarque malheureusement que ceux-ci n'aident pas vraiment le boosting, et que ces niveaux supplémentaires ne

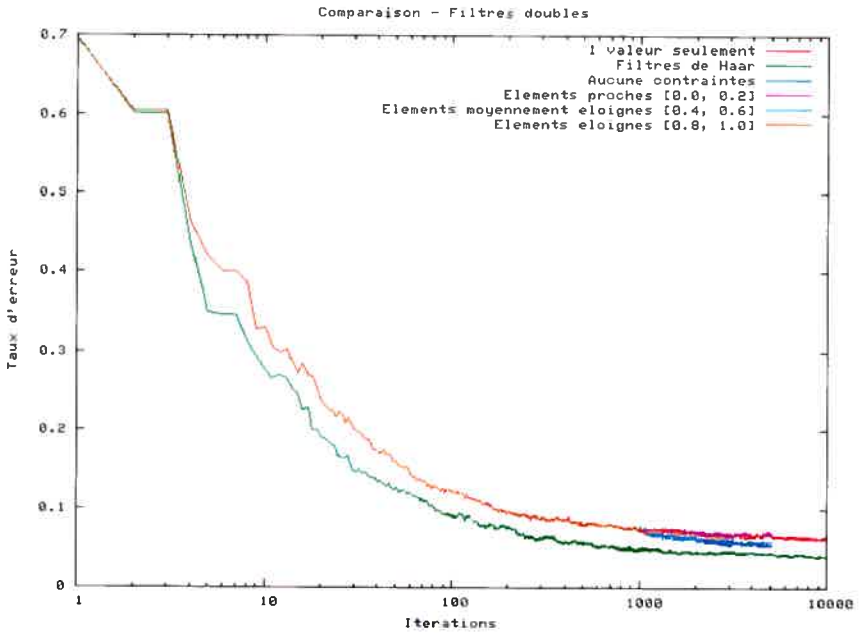

Figure 4.8 – Filtres doubles standards – contraintes

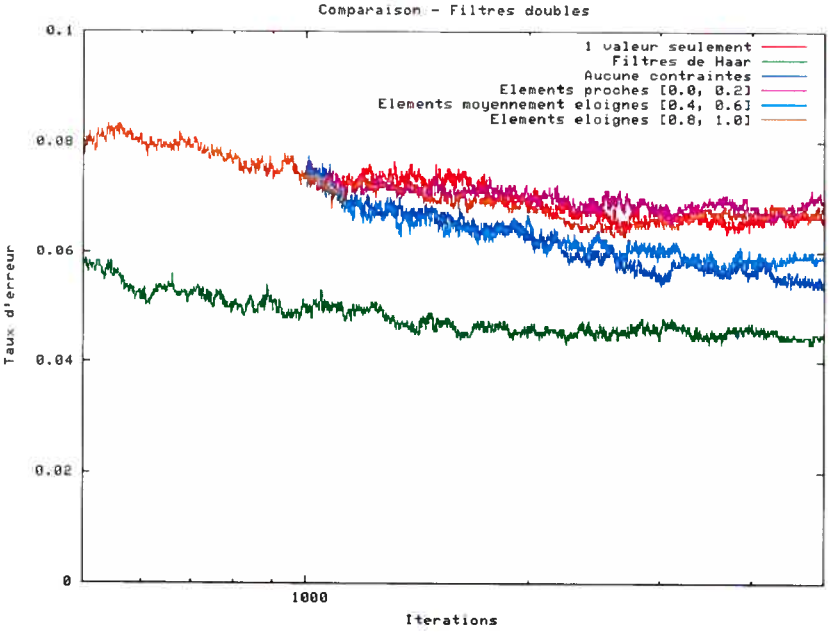

Figure 4.9 – Filtres doubles standards – contraintes (zoom)

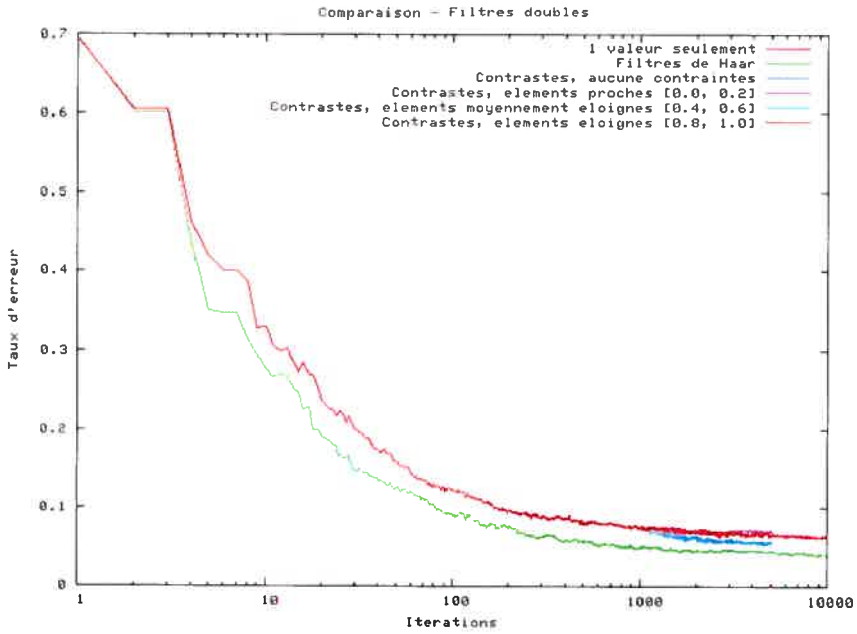

Figure 4.10 – Filtres doubles contrastés – contraintes

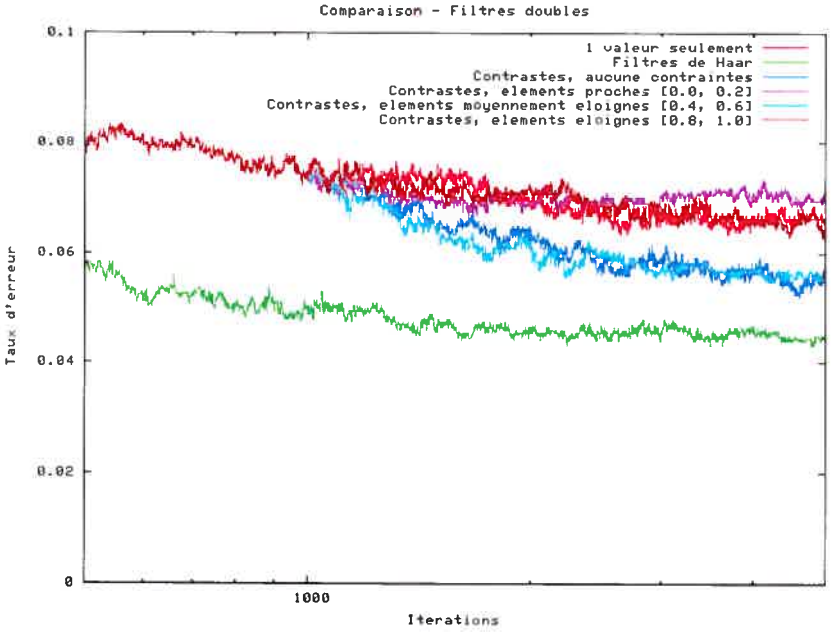

Figure 4.11 – Filtres doubles contrastés – contraintes (zoom)

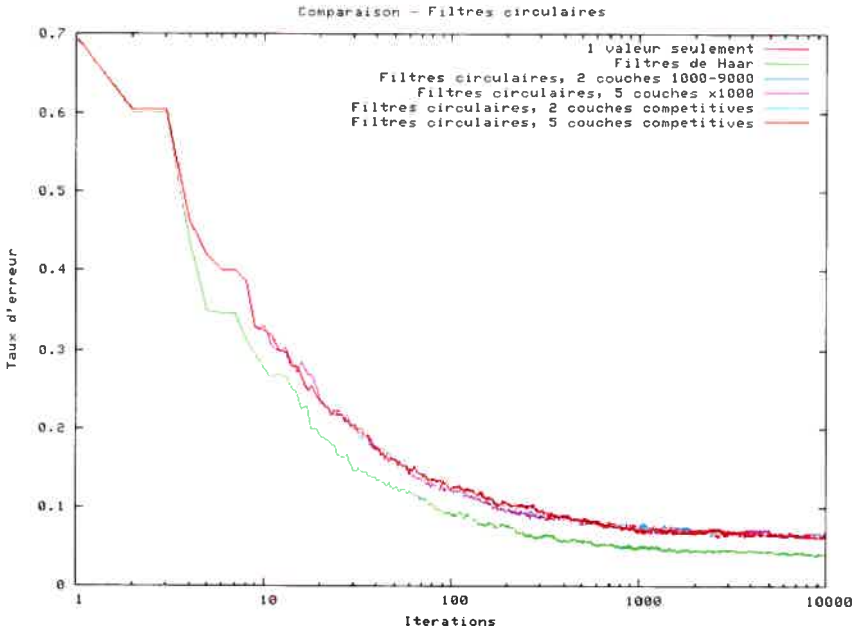

Figure 4.12 – Filtres circulaires, structures

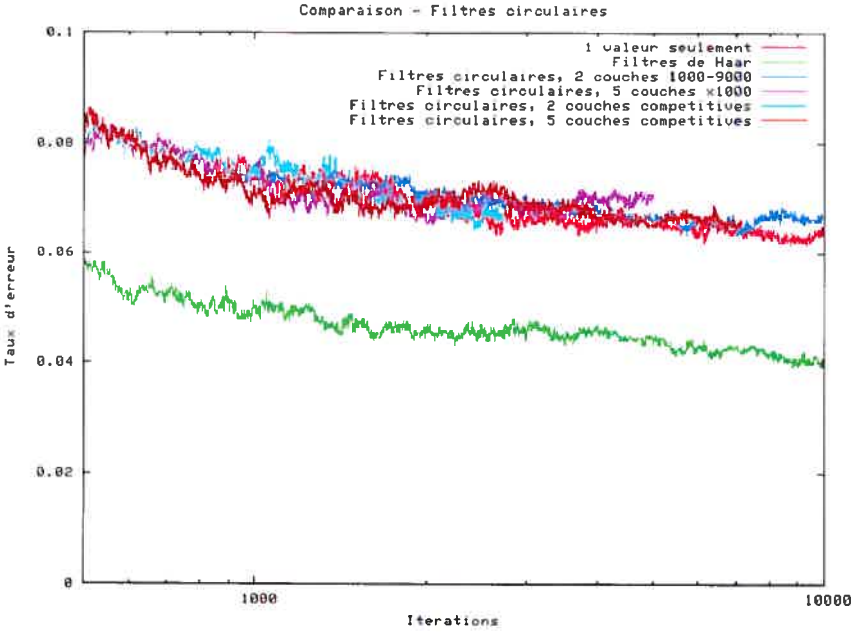

Figure 4.13 – Filtres circulaires, structures (zoom)

sont pas utiles.

Nous nous penchons ensuite sur les finesses que nous permettent ces filtres. Premièrement, nous avont limité le rayon maximal des filtres circulaire (à 0,15). Nous avons ensuite testé avec un contraste extérieur, avec un rayon maximal de 0,15 pour la zone positive et 0,30 pour la zone négative. Nous avons finalement testé la qualité de filtres circulaires gaussiens, en permettant un écart-type allant de 0,05 à 1,0.

Les figures 4.14 et 4.15 présentent ces nouveaux résultats. On y voit, malheureusement qu'aucune de ces techniques n'améliore ces filtres.

4.3.3.3 Filtres rectangulaires

Prenons ici quelque distance avec nos tests précédents. Le but jusqu'ici était d'atteindre, ou même dépasser, notre borne inférieure, c'est-à-dire un seul niveau de filtres de Haar. Tentons plutôt, puisqu'il s'agit d'une seule couche de filtres de Haar, d'améliorer ce résultat par une structure hiérarchique. Nous avons procédé à quatre séries de tests de filtres rectangulaires (avec et sans contraste). Les trois premières paires de tests ont la même structure forte de deux couches, mais avec une taille différente pour la première couche, respectivement, 1000, 500 et 250 (pour un total de 10 000 itérations). La dernière paire de tests s'attarde à la génération compétitive des deux couches.

Les figures 4.16 et 4.17 montrent les résultats des tests sans contraste. Aucun test ne bat clairement le résultat de référence, soit une seule couche de filtres de Haar. Quant au "point de départ" de la seconde couche, les résultats indiquent dans ce cas-ci qu'il est mauvais de commencer trop tôt. Cela tend à montrer que les unités de la première couche sont plus puissantes que celles de la seconde couche et que, dans le cas des filtres de Haar, la structure hiérarchique n'apporte pas de gain.

Même en introduisant des éléments de contraste, on n'arrive pas à faire mieux. Les figures 4.18 et 4.19 montrent deux des tests utilisés précédement, ainsi que leur pendant avec contraste. On voit à ce niveau que l'utilité du contraste pour les filtres rectangulaires n'est pas particulièrement importante, et semble même donner de moins bons résultats (que sans contraste), quoique ce ne soit pas de manière significative.

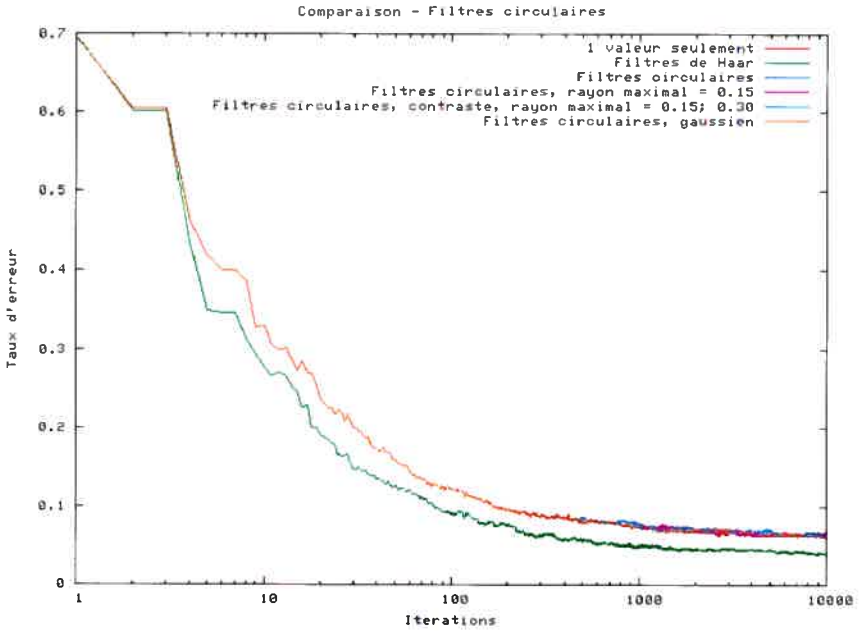

Figure 4.14 – Filtres circulaires, contraintes, contraste et gaussien

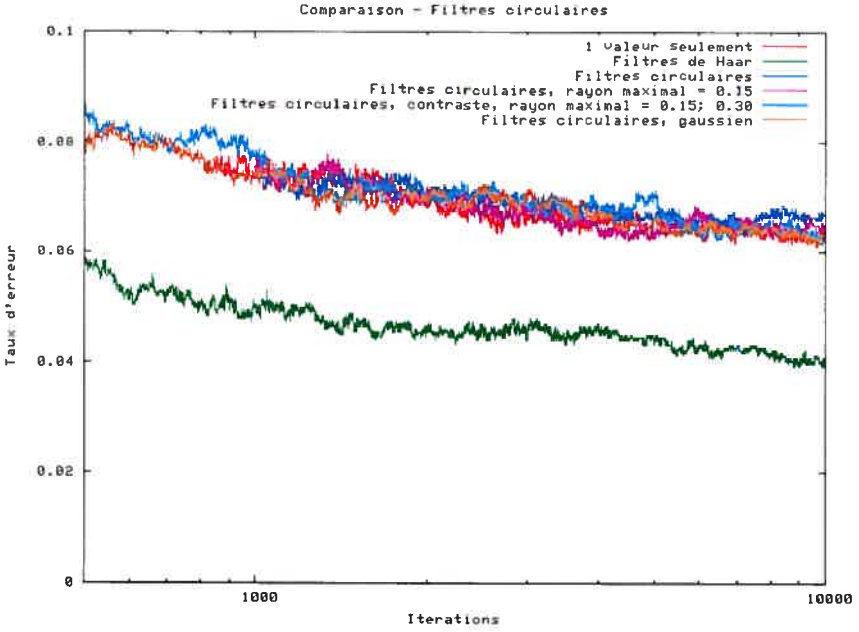

Figure 4.15 – Filtres circulaires, contraintees, contraste et gaussien (zoom)

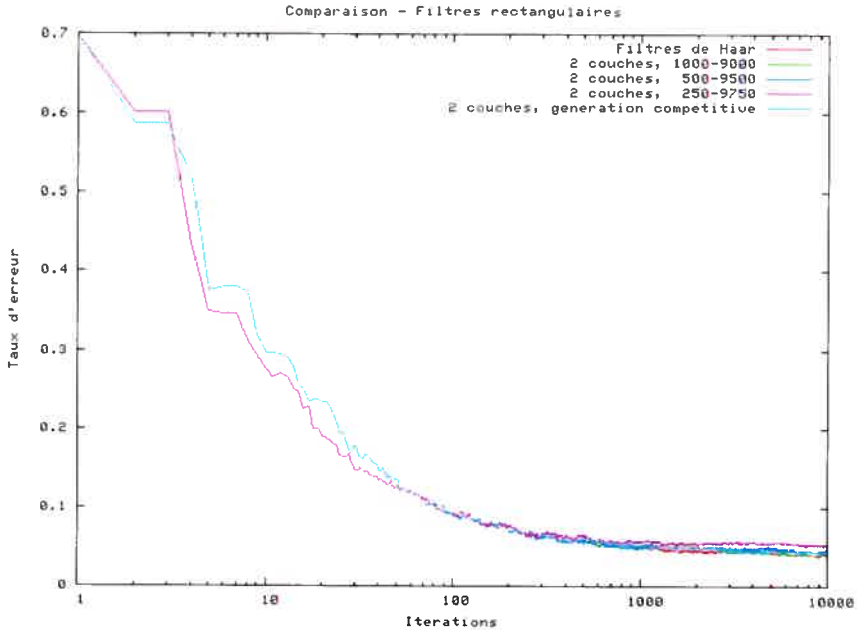

Figure 4.16 – Filtres Haar – filtres rectangulaires

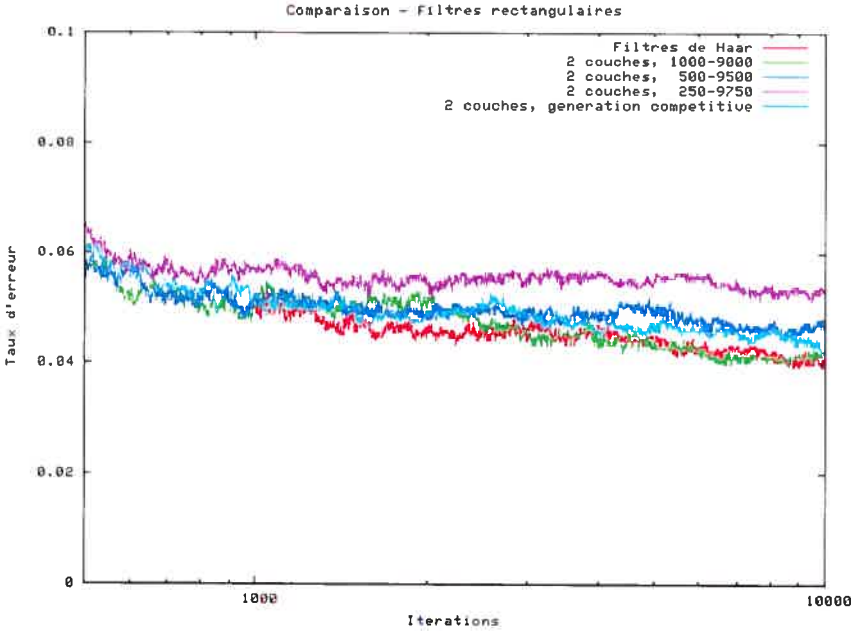

Figure 4.17 – Filtres Haar – filtres rectangulaires (zoom)

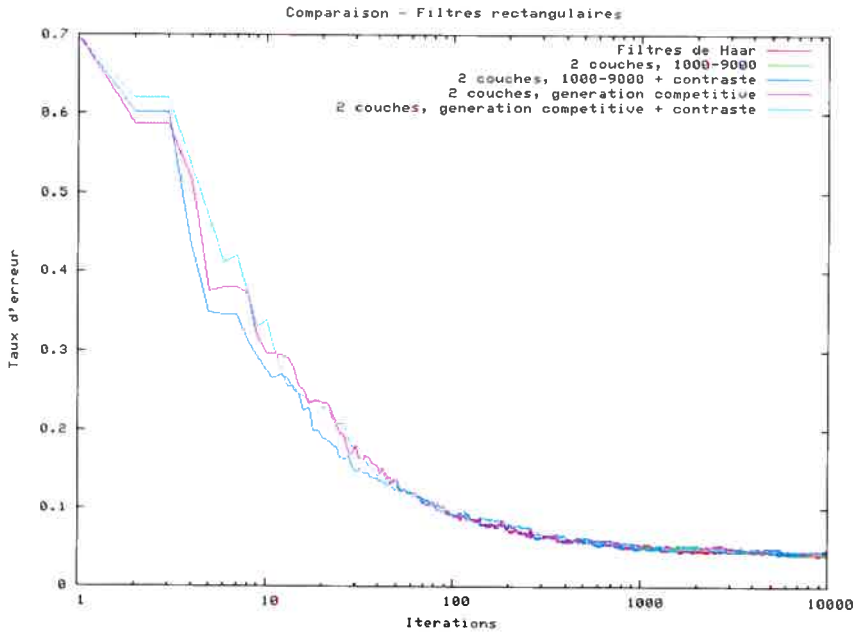

Figure 4.18 – Filtres Haar – filtres rectangulaires avec contraste

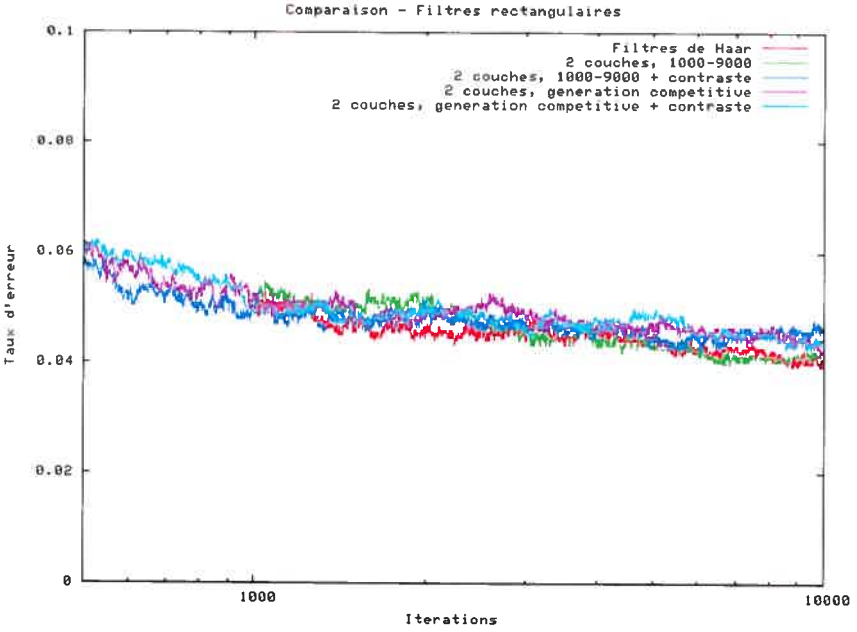

Figure 4.19 – Filtres Haar – filtres rectangulaires avec contraste (zoom)

4.3.4 Conclusion

Base	type de filtres	Nb couches	Type de structure	Erreur de test
1-stump	double	2	forte	5,7 %
1-stump	double	5	forte	5,4 %
1-stump	double	5	compétitive	5,7 %
1-stump	double	5	compétitive	6,3 %
1-stump	double	2	souple	5,7 %
1-stump	double – contrasté	2	forte	5,6 %
1-stump	double – standard et contrasté	2	forte	5,5 %
1-stump	double – proches	2	forte	6,9 %
1-stump	double – intermédiaire	2	forte	6,0 %
1-stump	double – éloignés	2	forte	6,7 %
1-stump	double – contrasté – proches	2	forte	7,0 %
1-stump	double – contrasté – intermédiaire	2	forte	5,5 %
1-stump	double – contrasté – éloingées	2	forte	6,5 %
1-stump	circulaire	2	forte	6,7 %
1-stump	circulaire	5	forte	7,0 %
1-stump	circulaire	2	compétitive	6,6 %
1-stump	circulaire	5	compétitive	6,5 %
1-stump	circulaire – rayon max	2	forte	6,5 %
1-stump	circulaire – contrasté – rayon max	2	forte	6,2 %
1-stump	circulaire – gaussien	2	forte	6,3 %
Filtre de Haar	rectangulaire – 1000-9000	2	forte	4,1 %
Filtre de Haar	rectangulaire – 500-9500	2	forte	4,6 %
Filtre de Haar	rectangulaire – 250-9750	2	forte	5,2 %
Filtre de Haar	rectangulaire	2	compétitive	4,2 %
Filtre de Haar	rectangulaire – contrasté – 1000-9000	2	forte	4,5 %
Filtre de Haar	rectangulaire – contrasté	2	compétitive	4,4 %

Tableau 4.2 – Résultats de hboost

Le tableau 4.2 cumule les résultats des expériences présentées dans ce chapitre. Il apparait que les filtres doubles sont clairement meilleurs que les filtres circulaires en général, et que la contrainte qui est la plus profitable pour ces filtres doubles est la distance moyenne entre les unités jointe dans un filtre. Les autres contraintes de distance ne sont pas utiles, et indiquement même que dans ces cas, la structure hiérarchique n'est pas utile.

Les filtres circulaires donnent tous des résultats décevants, qui ne battent pas une seule couche de départ. Les filtres rectangulaires n'aident pas non plus à améliorer la première couche de laquelle ils partent, cette fois-ci une couche de filtres de Haar. Dans tous les cas, il semble que l'utilisation de contraste dans les filtres ne se trouve pas être un élément significatif.

CHAPITRE 5

CONCLUSION

Nous avons proposé un ajout aux algorithmes de boosting qui permet d'utiliser ce mécanisme pour créer un réseau hiérarchique d'hypothèses. Dans notre cas où il s'agit d'hypothèses très faibles, des *decision stumps*, nous avons montré qu'on peut recombiner des décisions de niveau inférieur pour améliorer la classification par rapport à une seule couche de simple *stumps*. Cependant, cette amélioration reste modeste, et il nous a été impossible d'atteindre le niveau de qualité des filtres de Haar utilisés directement sur les images.

De l'autre côté, deux hypothèses importantes de nos techniques de combinaisons d'unités apparaissent comme non fondées. Ces filtrages, que nous voulions comme généralisation des filtres de Haar, ne semblent pas donner les résultats escomptés. D'un côté, l'utilisation de contrastes ne semble pas améliorer significativement les filtres, et de l'autre, les groupements d'unités proches ne sont pas particulièrement puissants.

Examinons la première limitation, celle des contrastes. L'importance de ces contrastes au niveau des images est assez simple à exprimer. Une même image peut contenir des informations numériques, la valeur des pixels, très différentes selon leur contexte, selon l'intensité lumineuse de l'image. Dans ces cas, un certain ton de gris, pour des images en noir et blanc, peut être foncé dans une image, et pâle dans une autre. Dans le cas qui nous intéresse, cette particularité ne se transfère pas vraiment. Nous voulons combiner des décisions, décisions qui sont binaires. Il n'y a pas, dans ces cas, de "pâle" ou de "foncé", pas de tons. Cependant, cela n'indique pas que les contrastes sont nécessairement inutiles. Prenons par exemple un filtre double incluant les unités a et b. Le filtre double standard donnera comme valeur filtrée $a+b$, alors que le filtre contrasté donnera $a-b$. Rappelons qu'étant donné que nous traitons des valeurs binaires, ces opérations arithmétiques ont une interprétation binaire également. Un stump sur $a+b$ correspond soit à "a et b" ou encore à "a ou b", alors que $a-b$ donne le penchant avec le non binaire, $a+(-b)$. D'un point de vue pratique, la présence de contraste nous permet un

plus grand jeu avec les valeurs binaires, et dans le cas des filtres doubles, où les unités sont prises au hasard plutôt que par un véritable filtrage géométrique, le contraste met à notre disposition le complément binaire de toutes nos décisions.

Est-ce que cela est avantageux ? Par défaut, nous répondons que plus de possibilités n'est pas a priori négatif. Mais comme les tests nous l'ont indiqué, la présence de contraste pour les filtres doubles n'apporte rien de plus. Cela est dû en grande partie aux unités que nous avons combinées. Par les distances, on voit que ce sont les unités moyennement éloignées (d'une distance de 0,4 à 0,6) qui font les meilleurs groupements. Dans ces cas, notons que notre distance, qui est une distance de Hamming entre des vecteurs binaires, donnera par contraste, des unités "positives" presqu'aussi éloignées. Ainsi, l'addition ou la soustraction des deux unités ne pose qu'un choix peu important.

La seconde limitation est d'une nature différente. Penchons-nous sur l'utilité de choisir un voisinage comme filtrage. Dans les données imagées, ces voisinages permettent de réduire le bruit en faisant la moyenne de pixels sur des "dimensions" proches une information redondante. Cependant, comme nous l'indiquent les résultats sur les contraintes des filtres doubles, ce ne sont ni les unités éloignées, ni les unités rapprochées qui forment de bonnes combinaisons. Ce sont les unités qui sont moyennement éloignées qui forment les meilleurs groupements. Pour quelle raison ? Les unités proches devraient en effet représenter la caractéristique recherchée de redondance. Cependant, ces unités sont des hypothèses, donc leurs sorties représentent des décisions, des décisions déjà comprises dans le processus de boosting, donc prises en compte dans les poids qui servent à calculer la qualité et l'impact de la nouvelle hypothèse. Comme ces unités existent déjà, et sont déjà comprises dans la somme des hypothèses du boosting, préciser ses décisions en ajoutant une nouvelle hypothèse les joignant est justement redondant. Comme dans tous les cas que nous avons testés, on recycle des décisions déjà prises, mais sans les retirer auparavant du total.

Les groupements les plus utiles, selon les filtres doubles, sont les unités moyennement éloignées, celles qui partagent une partie des informations, mais sont autant en désaccord qu'elles sont en accord. Dans ce sens, joindre deux décisions ainsi différentes donne un plus grand terrain pour trouver de nouvelles informations. Si on se réfère à

la comparaison de boosting et de bagging effectuée par Dietterich [7], l'avantage du boosting est de générer, sinon des hypothèses précises, du moins des hypothèses variées. Dans ce sens, le recyclage décrit plus haut et la précision d'unités proches, ne sont pas, dans le cadre du boosting, des avantages très importants. Dans cette optique cependant, le fait que ce soient des unités grandement décorrélées qui donnent les meilleurs résultats est significatif ; c'est là que se crée la variété.

Bien sur, l'utilité géométrique des nos filtres est dépendente de cette géométrie même. Ainsi, nous avons calculé nos distances entre deux unités par la distance de Hamming entre leurs sorties binaires. Ceci était conséquent avec notre hypothèse de départ, qui semble malheureusement mal fondée. Cependant, notons qu'une mesure de distance différente pourrait donner des résultats différents. En fait, selon nos tests sur les filtres doubles, nous trouvons une utilité géométrique, seulement, pas celle que nous souhaitions. La validité de notre hypothèse dépendant de la mesure de distance elle-même, une autre mesure, plaçant proche les éléments les plus susceptible de fournir des groupement précieux, pourrait valider l'hypothèse. Cependant, cette nouvelle distance devient alors dépendante de sa qualité pour former une hypothèse, et on perdrait avec celle-ci toute assomption honnête quant à une création de filtres indépendante du processus qui nous intéresse ici, c'est-à-dire le boosting.

Notons également que l'échec de nos géométries permet également de justifier en partie l'échec des contrastes, eux-mêmes basés sur ces géométries.

Le boosting hiérarchique que nous avons présenté fonctionne par recyclage, et nous avons observé effectivement une amélioration de la qualité par rapport à un seul niveau d'hypothèses. En effet, il est possible de créer de la nouveauté à un niveau hiérarchique supérieur, et cette nouveauté est de qualité intéressante, assez pour qu'on trouve un avantage à ce recyclage, plutôt qu'à chercher encore dans les données directement.

Nous avons présenté notre algorithme comme une variation du boosting standard, dans le sens où on ajoute des niveaux hiérarchiques à la structure linéaire typique du boosting. Cependant, on ne modifie pas directement le coeur de l'algorithme de boosting, de sorte que si nous avons utilisé AdaBoost.MH dans ce cas, plusieurs autres algorithmes de boosting peuvent également s'adjoindre la même structure hiérarchique que nous

avons proposée. Alors est-ce que notre boosting hiérarchique est une généralisation ou une spécialisation de boosting ? Les deux, en fait, comme nous le permet la définition assez ouverte du boosting.

C'est une généralisation dans le sens où le boosting possède une structure qui est véritablement une seule couche d'hypothèses parallèles. Cependant, cette caractéristique n'est pas clairement définie dans le boosting et rien ne proscrit la réutilisation des hypothèses. D'une certaine manière, nos hypothèses situées sur plusieurs couches sont encore parallèles dans le sens où elles sont toutes ajoutées au même niveau pour construire l'hypothèse forte de boosting, et qu'elles sont toutes, directement ou indirectement, liées aux mêmes données d'entrée. Le boosting, comme un méta-algorithme, laisse une grande liberté aux "hypothèses faibles", et si une particularité multicouche des hypothèses entre elles n'est pas indiquée, elle n'est pas non plus proscrite. Nous avons présenté jusqu'ici la qualité hiérarchique du boosting comme étant extérieure aux hypothèses, et donc faisant partie du boosting, mais cette caractéristique peut faire partie des hypothèses elles-mêmes. On utilise habituellement le boosting pour *booster* un type d'algorithme en particulier. Il y a peu de contact entre le boosting et l'apprenant faible lui-même, peut-être justement parce que ces contacts ne sont pas nécessaires. Cependant, par le boosting hiérarchique, nous avons présenté une structure hiérarchique, permise par le boosting (car ce sont les hypothèses que nous recombinons), mais qui ne modifie pas directement l'algorithme de boosting.

La structure hiérarchique pouvant être confinée à l'apprenant faible, regardons ce que cette même structure hiérarchique donne de ce point de vue lorsqu'on ajoute une couche. En construisant la première couche, on ajoute toujours des unités provenant du même bassin d'hypothèses, de la même classe de fonctions. Cependant, lorsqu'on passe à la seconde couche, on change de classe de fontions. Si les nouvelles unités de la seconde couche sont considérées sur un même point, du point de vue du boosting pur, donc comme parallèles à celles de la première couche (les mêmes entrées, directement ou indirectement, sorties au même "endroit"), leur action est différente. Au niveau effectif, ces hypothèses ne prennent pas le même genre de décisions, et bien qu'elles soient dans notre cas le fruit du même genre de *decision stumps*, on se trouve dans une autre

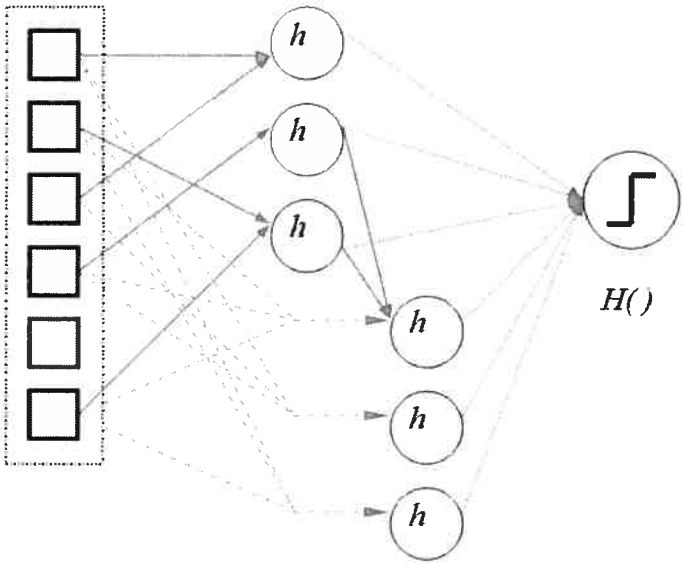

Figure 5.1 – Hypothèses de différents niveaux

classe de fonctions : nous avons ici, indirectement, des hypothèses différentes, du fait que leur entraînement est différent. Ceci fait en sorte qu'une hypothèse de la seconde couche tient d'une classe de fonctions à deux niveaux, bien que celles-ci soient limitées à utiliser comme "couche intermédiaire" des unités de la première couche. Un changement de couche offre donc, dans cette vision, un changement de la classe de fonction dans laquelle on cherche.

Dans le cas simple où nous nous trouvons, le premier niveau de *decision stumps* constitue un apprenant particulièrement faible, et on pourrait lier la convergence de cette première couche au fait qu'on épuise peu à peu la classe de fonctions de ces *decision stumps*[1]. De fait, lorsqu'on change de classe de fonctions, on augmente la diversité, relativement aux hypothèses déjà apprises. Non seulement on change d'espace de fonctions, mais on passe d'un espace de richesse faible à un espace plus riche. Dans les cas de génération compétitive des couches, on ne "change" pas directement de classe de fonctions, mais on enrichit celle dans laquelle on cherche. Selon cette analyse, le boosting hiérarchique sert en quelque sorte à dynamiser l'espace de fonctions dans lequel on cherche,

[1] En fait, on épuise la classe de fonctions dans le sens ou le boosting n'est plus capable d'aller chercher assez de puissance pour améliorer son pouvoir de généralisation.

de l'enrichir lorsque ceci est nécessaire.

Cependant, nous avons noté également que les niveaux hiérarchiques n'ajoutaient pas de puissance dans le cas où on combine des filtres de Haar. Dans ce cas, il nous semble que le problème soit dû au fait que les hypothèses de la seconde couche, bien que d'un niveau "hiérarchique" supérieur, ne constituent pas un espace de fonctions plus riche que celui de la première couche, et par conséquent cette couche supplémentaire n'est pas avantageuse pour l'approche vorace du boosting.

Notons que cette croissance de la richesse de la classe de fonctions, ou encore simplement le changement de cette richesse, nous oblige à perdre certaines garanties théoriques du boosting, y compris certaines bornes de l'erreur de généralisation qui prend pour acquis la stabilité de la richesse de la classe de fonction dans le temps. Plus particulièrement, ceci ouvre toute grande la porte au sur-apprentissage. On l'a déjà vu, la richesse de notre classe de fonctions est l'ennemie du sur-apprentissage, et augmenter la richesse de la classe de fonctions dans laquelle on pêche nos hypothèses constitue à ce titre un pas dans la mauvaise direction.

Cependant, nous l'avons vu empiriquement, le boosting original, à une couche, de nos tests est clairement limité, et converge vers une valeur clairement sous-optimale. Dans ce cadre-là, augmenter la richesse de l'espace de fonctions nous a permis d'aller chercher un plus grand pouvoir de généralisation, et d'atteindre une erreur de test moindre. Ainsi, nous nous trouvons entre le sur-apprentissage et le sous-apprentissage, avec une possibilité de doser notre présence entre ces deux démons. L'augmentation de la richesse de la classe de fonctions n'est mauvaise en fait que lorsqu'on a passé le seuil optimal entre sous-apprentissage et sur-apprentissage. C'est pourquoi nous recommandons de bien valider tout augmentation de la richesse de la classe de fonctions des hypothèses.

5.1 Améliorations possibles

Nous avons montré que notre méthode de boosting hiérarchique permet certains gains, quoique limités. Cependant notre technique de filtrage géométrique ne semble

pas appropriée dans ces cas du boosting, du moins pour les couches supérieures. Il reste possible de trouver une utilité à ces filtres pour des données homogènes qui ne sont pas des images. Nous croyons possible d'atteindre une plus grande puissance à travers le boosting hiérarchique, mais certaines modifications sont nécessaires.

Premièrement, il est important de construire une manière plus efficace et rapide, de refondre le code et de réduire la taille en mémoire autant que le temps d'exécution de l'algorithme. Ce sont des considérations pratiques, mais il serait bénéfique de pouvoir tester et valider des hypothèses beaucoup plus rapidement. Dans l'état actuel, un test peut monopoliser une machine moderne durant plusieurs semaines.

Une fois cette amélioration technique accomplie, il serait intéressant de tester les différents éléments de structures présentés dans la section 3.1, spécialement pour ce qui a trait au remplacement. Cette technique demande particulièrement plus de puissance machine.

Il serait également intéressant de joindre ce remplacement à une technique d'élagage, comme le montre [27] par exemple. Ceci offrirait une méthode différente de réduire à la fois la taille du réseau et l'influence des unités inférieures.

5.2 Travail futur

Nous pensons souhaitable d'explorer plus directement le lien entre la richesse des classes de fonctions (générées par les "couches") et le boosting plus en profondeur. Est-ce que l'augmentation contrôlée (et validée) de la richesse de la classe de fonctions parmi laquelle les hypothèses sont tirées permet une bonne amélioration de la puissance de généralisation ?

Une autre possibilité intéressante est d'utiliser un entraînement différent pour chaque couche. En d'autre mots, soit entraîner des couches intermédiaires de manières non-supervisées, soit leur ajouter une autre tâche (faire du multi-tâche), pondérée de manière à ce que les unités intermédiaires soit créent des caractéristiques utiles à la hiérarchie, alors que les unités supérieures soient axées quant à elles complètement sur la tâche principale.

De plus, il serait utile compléter le rapprochement pratique entre le boosting et les réseaux de neurones. Ceci nous permettrait peut-être de mettre en pratique des hypothèses un peu éloignées que nous avons mentionnées dans la section 2.3. On pourrait aussi fusionner les deux méthodes, soit recalibrer directement des poids et des liens de boosting par l'algorithme de rétropropagation. Plus simplement cependant, ce rapprochement offrirait des possibilités intéressantes dans l'ajout de neurones à un réseau existant. La création d'un réseau lié par deux concepts aussi puissants que les preuves théoriques du boosting et la grande expérience des réseaux de neurones ne peut être que très intéressante.

BIBLIOGRAPHIE

[1] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[2] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):1123–1140, September 1994.

[3] Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

[4] Corinna Cortes et Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[5] Thomas G. Dietterich et Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2: 263–286, 1995.

[6] Thomas G. Dietterich, Michael Kearns et Yishay Mansour. Applying the weak learning framework to understand and improve c4.5. Dans *International Conference on Machine Learning*, volume 13, pages 96–104, 1996.

[7] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees : Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000.

[8] S. E. Fahlman et C. Lebiere. The cascade-correlation learning architecture. Dans *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, 1990.

[9] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and computation*, 121(2):256–285, 1995.

[10] Yoav Freund et Robert Schapire. Adaptive game playing using multiplicative weights. *Games and Exonomic Behavior*, 29:79–103, 1999.

[11] Yoav Freund et Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. Dans *European Conference on Computational Learning Theory*, pages 23–37, 1995.

[12] Yoav Freund et Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[13] Yoav Freund et Robert E. Schapire. Experiments with a new boosting algorithm. Dans *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156, 1996.

[14] J. Friedman, T. Hastie et R. Tibshirani. Additive logistic regression : a statistical view of boosting, 1998.

[15] Adam J. Grove et Dale Schuurmans. Boosting in the limit : Maximizing the margin of learned ensembles. Dans *AAAI/IAAI*, pages 692–699, 1998.

[16] David Haussler. Probably approximately correct learning. Dans *National Conference on Artificial Intelligence*, pages 1101–1108, 1990.

[17] Robert C. Holte. Very simple classification rules perform will on most commonly used datasets. *Machine Learning*, 11(1):63–91, 1993.

[18] K. Hornik, M. Stinchcombe et H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[19] Michael Kearns et Yishay Mansour. On the boosting ability of top-down decision tree learning algorithm. Dans ACM Press, éditeur, *Annual ACM Symposium on the Theory of Computing*, volume 28, pages 459–468, 1996.

[20] Ron Kohavi et Clayton Kunz. Option decision trees with majority votes. Dans Doug Fisher, éditeur, *Machine Learning : Proceedings of the Fourteenth International Conference*. Morgan Kaufmann Publishers, Inc., 1997.

[21] Eun Bae Kong et Thomas G. Dietterich. Error-correcting output coding corrects bias and variance. Dans *International Conference on Machine Learning*, pages 313–321, 1995.

[22] Balázs Kégl. Robust regression by boosting of the median. Dans *Computational Learning Theory*, volume 16, pages 258–272, 2003.

[23] Yann LeCun. The MNIST database of handwritten digits. URL `http://yann.lecun.com/exdb/mnist`.

[24] Rainer Lienhart et Jochen Maydt. An extended set of haar-like features for rapid object detection. Dans *IEEE International Conference on Image Processing 2002*, volume 1, pages 900–903, September 2002.

[25] Richard Maclin et David Opitz. An empirical evaluation of bagging and boosting. Dans *Proceedings of the 14th National Conference on AI*, pages 546–551, 1997.

[26] Stephane G. Mallat. A theory for multiresolution signal decomposition : The wavelet representation. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 11(7), 1989.

[27] Dragos D. Margineantu et Thomas G. Dietterich. Pruning adaptive boosting. Dans *Proc. 14th International Conference on Machine Learning*, pages 211–218. Morgan Kaufmann, 1997.

[28] Llew Mason, Peter L. Bartlett et Jonathan Baxter. Direct optimization of margins improves generalization in combined classifiers, 1998.

[29] Llew Mason, Peter L. Bartlett et Jonathan Baxter. Improved generalization through explicit optimization of margins. *Machine Learning*, 38(3):243–255, 2000.

[30] Warren S. McCulloch et Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115–133, 1943.

[31] J. R. Quinlan. Baggin, boosting and c4.5. Dans *Proceedings of the 13th National Conference on Artificial Intelligence*, volume 13, pages 725–730, 1996.

[32] Frank Rosenblatt. The perception : a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1957.

[33] Saharon Rosset et Eran Segal. Boosting density estimation. Dans *Advances in Neural Information Processing Systems*, volume 15, 2002.

[34] Gunnar Rätch et Manfred K. Warmuth. Efficient margin maximizing with boosting. *Journal of Machine Learning Research*, 6:2131–2152, 2005.

[35] Gunnar Rätsch, Takashi Onoda et Klaus-R. Müller. Soft margins for adaboost, 1998.

[36] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2): 197–227, 1990.

[37] Robert E. Schapire. Using output codes to boost multiclass learning problems. Dans *Proceedings of the 14th International Conference on Machine Learning*, volume 14, pages 313–321. Morgan Kaufmann, 1997.

[38] Robert E. Schapire. Drifting games. Dans *Computational Learing Theory*, pages 114–124, 1999.

[39] Robert E. Schapire. Theoretical views of boosting. *Lecture Notes in Computer Science*, 1572:1–10, 1999.

[40] Robert E. Schapire, Yoav Freund, Peter Bartlett et Wee Sun Lee. Boosting the margin : A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

[41] Robert E. Schapire et Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[42] Holger Schwenk et Yoshua Bengio. Boosting neural networks. *Neural Computations*, 12(8):1869–1887, August 2000.

[43] Patrice Y. Simard, Dave Steinkraus et John Platt. Best practice for convolutional neural networks applied to visual document analysis. Dans *International Conference on Document Analysis and Recogntion (ICDAR)*, pages 958–962, 2003.

[44] Joshua B. Tenenbaum, Vin de Silva et John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[45] L. G. Valiant. A theory of the learnable. Dans *STOC '84 : Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445, 1984.

[46] Vladimir N. Vapnik et Alexei Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[47] Paul Viola et Michael Jones. Rapid object detection using a boosted cascade of simple features. Dans *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, 2001.

[48] Max Welling, Richard S. Zemel et Geoffrey E. Hinton. Self supervised boosting. Dans MIT Press, éditeur, *Advances in Neural Information Processing Systems*, volume 15, 2002.