Université de Montréal

Apprentissage machine efficace : théorie et pratique

par
Olivier Delalleau

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

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

Mars 2012

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée :

Apprentissage machine efficace : théorie et pratique

présentée par :

Olivier Delalleau

a été évaluée par un jury constitué des personnes suivantes :

Pascal Vincent	président-rapporteur
Yoshua Bengio	directeur de recherche
Pierre McKenzie	membre du jury
Yves Grandvalet	examinateur externe
Patrick Drouin	représentant du doyen

Résumé

Malgré des progrès constants en termes de capacité de calcul, mémoire et quantité de données disponibles, les algorithmes d'apprentissage machine doivent se montrer efficaces dans l'utilisation de ces ressources. La minimisation des coûts est évidemment un facteur important, mais une autre motivation est la recherche de mécanismes d'apprentissage capables de reproduire le comportement d'êtres intelligents. Cette thèse aborde le problème de l'efficacité à travers plusieurs articles traitant d'algorithmes d'apprentissage variés : ce problème est vu non seulement du point de vue de l'efficacité *computationnelle* (temps de calcul et mémoire utilisés), mais aussi de celui de l'efficacité *statistique* (nombre d'exemples requis pour accomplir une tâche donnée).

Une première contribution apportée par cette thèse est la mise en lumière d'inefficacités statistiques dans des algorithmes existants. Nous montrons ainsi que les arbres de décision généralisent mal pour certains types de tâches (chapitre 3), de même que les algorithmes classiques d'apprentissage semi-supervisé à base de graphe (chapitre 5), chacun étant affecté par une forme particulière de la malédiction de la dimensionalité. Pour une certaine classe de réseaux de neurones, appelés réseaux sommes-produits, nous montrons qu'il peut être exponentiellement moins efficace de représenter certaines fonctions par des réseaux à une seule couche cachée, comparé à des réseaux profonds (chapitre 4). Nos analyses permettent de mieux comprendre certains problèmes intrinsèques liés à ces algorithmes, et d'orienter la recherche dans des directions qui pourraient permettre de les résoudre.

Nous identifions également des inefficacités computationnelles dans les algorithmes d'apprentissage semi-supervisé à base de graphe (chapitre 5), et dans l'apprentissage de mélanges de Gaussiennes en présence de valeurs manquantes (chapitre 6). Dans les deux cas, nous proposons de nouveaux algorithmes capables de traiter des ensembles de données significativement plus grands. Les deux derniers chapitres traitent de l'efficacité computationnelle sous un angle différent. Dans le chapitre 7, nous analysons de manière théorique un algorithme existant pour l'apprentissage efficace dans les machines de Boltzmann restreintes (la divergence contrastive), afin de mieux comprendre les raisons qui expliquent le succès de cet algorithme. Finalement, dans le chapitre 8 nous présentons une application de l'apprentissage machine dans le domaine des jeux vidéo, pour laquelle le problème de l'efficacité computationnelle est relié à des considérations d'ingénierie logicielle et matérielle, souvent ignorées en recherche mais ô combien importantes en pratique.

Mots-clés : efficacité computationnelle, efficacité statistique, malédiction de la dimensionalité, arbres de décision, réseaux de neurones, apprentissage semi-supervisé à base de graphe, divergence contrastive, mélanges de Gaussiennes, appariement de joueurs

Abstract

Despite constant progress in terms of available computational power, memory and amount of data, machine learning algorithms need to be efficient in how they use them. Although minimizing cost is an obvious major concern, another motivation is to attempt to design algorithms that can learn as efficiently as intelligent species. This thesis tackles the problem of efficient learning through various papers dealing with a wide range of machine learning algorithms: this topic is seen both from the point of view of *computational* efficiency (processing power and memory required by the algorithms) and of *statistical* efficiency (number of samples necessary to solve a given learning task).

The first contribution of this thesis is in shedding light on various statistical inefficiencies in existing algorithms. Indeed, we show that decision trees do not generalize well on tasks with some particular properties (chapter 3), and that a similar flaw affects typical graph-based semi-supervised learning algorithms (chapter 5). This flaw is a form of curse of dimensionality that is specific to each of these algorithms. For a subclass of neural networks, called sum-product networks, we prove that using networks with a single hidden layer can be exponentially less efficient than when using deep networks (chapter 4). Our analyses help better understand some inherent flaws found in these algorithms, and steer research towards approaches that may potentially overcome them.

We also exhibit computational inefficiencies in popular graph-based semi-supervised learning algorithms (chapter 5) as well as in the learning of mixtures of Gaussians with missing data (chapter 6). In both cases we propose new algorithms that make it possible to scale to much larger datasets. The last two chapters also deal with computational efficiency, but in different ways. Chapter 7 presents a new view on the contrastive divergence algorithm (which has been used for efficient training of restricted Boltzmann machines). It provides additional insight on the reasons why this algorithm has been so successful. Finally, in chapter 8 we describe an application of machine learning to video games, where computational efficiency is tied to software and hardware engineering constraints which, although often ignored in research papers, are ubiquitous in practice.

Keywords: computational efficiency, statistical efficiency, curse of dimensionality, decision trees, neural networks, graph-based semi-supervised learning, contrastive divergence, mixtures of Gaussians, matchmaking

Table des matières

Liste des figures

Liste des tableaux

Abbréviations et notations

CD	Divergence Contrastive (*Contrastive Divergence*)
EM	Espérance-Maximisation (*Expectation-Maximization*)
NLL	Log-Vraisemblance Négative (*Negative Log-Likelihood*)
PCA	Analyse en Composantes Principales (*Principal Component Analysis*)
RBM	Machine de Boltzmann Restreinte (*Restricted Boltzmann Machine*)
SVM	Machine à Vecteurs de Support (*Support Vector Machine*)

$f(t)$	valeur de la fonction f appliquée à t
$f(\cdot)$	fonction f (ayant un argument)
$\ln(\cdot)$ ou $\log(\cdot)$	logarithme népérien
$\text{argmax}_u f(u)$	la valeur de u qui maximise $f(u)$
$\text{argmin}_u f(u)$	la valeur de u qui minimise $f(u)$
$\mathbb{R}$	ensemble des nombres réels
$v^T w$	produit scalaire entre les vecteurs v et w
M^T	transposée de la matrice M
M_{ij}	élément de la matrice M en i-ème rangée et j-ème colonne
$\mathbf{1}.$	fonction indicatrice, par exemple $\mathbf{1}_{i<j}$ vaut 1 si $i < j$, et 0 sinon
$\mathbf{I}$	matrice identité
L	matrice Laplacienne associée à un graphe
x_i	partie "entrée" du i-ème exemple d'apprentissage
x_{ij}	la j-ème composante de la partie "entrée" du i-ème exemple d'apprentissage
y_i	partie "étiquette" du i-ème exemple d'apprentissage
z_i	i-ème exemple d'apprentissage (si étiqueté : $z_i = (x_i, y_i)$, sinon : $z_i = x_i$)
$\mathcal{D} = \{z_1, \ldots, z_n\}$	ensemble d'entraînement
$\mathcal{T}$	ensemble de test
ℓ	nombre d'exemples étiquetés dans l'ensemble d'entraînement
$K(\cdot, \cdot)$	fonction noyau (ayant deux arguments)
$P(V)$	distribution de probabilité d'une variable aléatoire V
$P(v)$	raccourci pour $P(V = v)$ (probabilité discrète ou densité)
$P(v\|w)$	raccourci pour $P(V = v\|W = w)$ (probabilité discrète ou densité)
$\hat{P}$	distribution empirique
$v \sim P(V)$	indique que la quantité v est tirée de la distribution $P(V)$
$E_V[f(V)]$	espérance de $f(V)$, égale à $\int_v f(v)P(V = v)dv$
$E_V[f(V)\|w]$	espérance conditionnelle de $f(V)$, égale à $\int_v f(v)P(V = v\|W = w)dv$
$D_{KL}(P\|Q)$	divergence de Kullback-Leibler entre les distributions P et Q
$\mathcal{N}(\cdot; \mu, \Sigma)$	densité d'une Gaussienne de moyenne μ et covariance Σ
$\mathcal{E}$	fonction d'énergie dans une machine de Boltzmann restreinte
ℓ_1	type de régularisation qui pour un paramètre $\theta \in \mathbb{R}^k$ s'écrit $\sum_{i=1}^k \|\theta_i\|$
ℓ_2	type de régularisation qui pour un paramètre $\theta \in \mathbb{R}^k$ s'écrit $\sum_{i=1}^k \theta_i^2$

Remerciements

MERCI à tous ceux et celles qui ont contribué, de près ou de loin, à ce que cette thèse voie le jour. Je pourrais m'arrêter là, mais il serait ingrat de ma part de ne pas insister sur l'importance particulière de l'aide apportée par mon directeur de recherches, le Professeur Yoshua Bengio, sans qui je me demanderais encore quel pourrait bien être le sujet de ma thèse.

J'ai eu l'occasion de travailler dans des environnements variés, que cela soit parmi les étudiants et professeurs du Laboratoire d'Informatique des Systèmes Adaptatifs (LISA) de l'Université de Montréal, ou les équipes de ApSTAT Technologies et Ubisoft. Dans tous les cas, je n'ai jamais considéré que partir travailler le matin était une corvée, à de rares exceptions près essentiellement dues à quelques nuits trop courtes et aux aléas du transport en commun. Merci donc à vous tous qui participez à rendre l'ambiance de travail agréable, et aux employés de la STM qui tentent de minimiser les arrêts de service du métro et retards d'autobus.

Je m'étais déjà demandé jadis, étant jeune et naïf, pourquoi les auteurs d'autres thèses ou livres remerciaient systématiquement leur conjoint(e). Maintenant que je connais la réponse, je me dois de remercier Annie de m'avoir supporté – dans tous les sens du terme. Pour continuer dans la famille, un gros merci également à mes parents pour leur soutien indéfectible. Et je salue au passage mon frère Gaël, qui n'a pas grand-chose à voir avec cette thèse mais sera sûrement content d'en faire partie.

Finalement, m'étant inspiré de Nicolas Chapados en ce qui a trait à la durée de mon doctorat, il m'a semblé logique de réutiliser le modèle LaTeX qu'il a développé pour sa propre thèse. Il mérite toute ma gratitude pour cela*, car même si je laisserai à d'autres le soin de juger de l'aspect esthétique de ce document, une chose est certaine : je n'aurais pas fait mieux par moi-même.

*Pour m'avoir autorisé à ré-utiliser son modèle, pas pour m'avoir montré comment étirer un doctorat en longueur.

1 Introduction

Cette thèse aborde des sujets très variés, mais revenant systématiquement à un même thème central, celui de l'efficacité en apprentissage machine. Mon intérêt principal est de mieux comprendre les forces et limitations des algorithmes typiquement utilisés en apprentissage machine. Cette compréhension théorique n'a pas pour seul but de faire progresser la recherche d'améliorations algorithmiques : elle est également cruciale dans l'application judicieuse de ces algorithmes pour la résolution de "vrais" problèmes. Par "vrais" problèmes, je parle ici d'applications industrielles de l'apprentissage machine au-delà des problèmes de référence régulièrement utilisés dans les publications scientifiques. Ayant eu l'occasion de travailler sur de telles applications au travers de partenariats industriels avant* ainsi que pendant† mon doctorat, j'ai pu réaliser à quel point une telle compréhension est cruciale pour la réussite de ces projets.

Le terme "efficacité" peut être interprété de différentes manières. La première est sans doute la plus évidente : il s'agit de l'efficacité *computationnelle*, c.à.d. du temps de calcul et de la mémoire requis pour l'exécution d'un algorithme d'apprentissage. La seconde manière dont l'efficacité est envisagée dans cette thèse est du point de vue de l'efficacité *statistique*, c.à.d. du nombre d'exemples nécessaire pour apprendre à effectuer une certaine tâche. Cette mesure d'efficacité ne dépend pas seulement de la tâche en question, mais également de la manière dont elle est apprise (donc de l'algorithme d'apprentissage). Le but de ce premier chapitre d'introduction est de présenter les concepts de base de l'apprentissage machine permettant de comprendre ce phénomène.

Le second chapitre introduit les différents algorithmes qui seront utilisés dans les chapitres suivants (correspondant chacun à un article). L'ordre de présentation des articles suit – approximativement – une progression de l'efficacité statistique (plus théorique) vers l'efficacité computationnelle (plus pratique). Chaque article est précédé d'une mise en contexte, et suivi de commentaires résumant les enseignements principaux à en tirer par rapport au thème récurrent de l'efficacité.

*Avec Bell Canada.
†Avec ApSTAT Technologies et Ubisoft.

1.1 Objectif de l'apprentissage machine

L'apprentissage machine est une sous-discipline de l'intelligence artificielle dont l'objectif ultime est de reproduire chez l'ordinateur les capacités cognitives de l'être humain, *par le biais de l'apprentissage*. Ici, le terme apprentissage est à mettre en opposition avec des techniques d'intelligence artificielle basées sur des comportements (apparemment) intelligents "précodés", comme dans le fameux programme ELIZA (Weizenbaum, 1966) où l'ordinateur donnait l'illusion de pouvoir poursuivre une conversation intelligente avec un humain (à partir d'un système en fait très basique de détection de mots-clés et de réponses pré-définies). L'approche "apprentissage machine" consiste plutôt à programmer des mécanismes qui permettent de développer les connaissances (c.à.d. d'apprendre) à partir d'observations (les *exemples d'apprentissage*), de manière automatique.

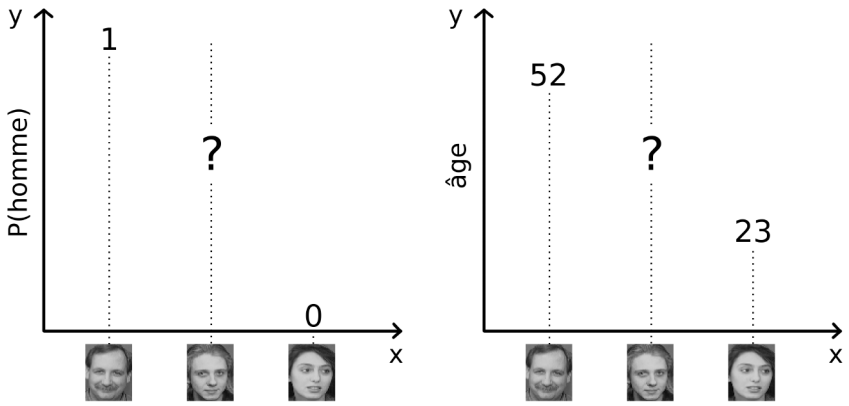
Les êtres humains et les animaux faisant preuve de capacités d'apprentissage impressionnantes, il est naturel que l'apprentissage machine s'inspire d'eux pour tenter de reproduire leurs comportements. En particulier, les travaux en neurosciences visant à comprendre les mécanismes d'apprentissage dans le cerveau sont suivis avec grand intérêt par la communauté de recherche en apprentissage machine. Une classe d'algorithmes d'apprentissage très populaire, les réseaux de neurones, a ainsi été inspirée de telles observations biologiques. Mais les détails du fonctionnement du cerveau restant pour l'instant en grande partie un mystère, nous sommes encore loin d'un "cerveau artificiel". Les algorithmes développés en apprentissage machine ont des objectifs moins ambitieux. Ils ont chacun leurs propres forces et faiblesses – souvent très différentes de celles des humains – et sont généralement prévus pour résoudre certains types de problèmes spécifiques (alors que le cerveau est capable d'accomplir une multitude de tâches différentes).

1.2 Différents types d'apprentissage

1.2.1 Apprentissage supervisé

La forme d'apprentissage qui est considérée comme la plus intuitive est celle dite d'apprentissage *supervisé*. Cela signifie que la réponse attendue est observée dans les données collectées. Formellement, si l'on note z un exemple d'apprentissage, alors on peut écrire $z = (x, y)$ avec x la partie "entrée", c.à.d. les données que l'algorithme est autorisé à utiliser pour faire une prédiction, et y la partie "étiquette", c.à.d. la valeur correcte pour la prédiction. Par exemple, si la tâche consiste à déterminer le sexe d'une personne à partir d'une photo d'identité, x serait la photo et y soit "homme", soit "femme" (en supposant qu'il s'agisse des deux seules possibilités). Dans un tel cas où les valeurs possibles de l'étiquette y ne sont pas interprétables comme des nombres, on parle de tâche de *classification*. Si par contre la tâche était de

prédire l'âge de la personne plutôt que son sexe, y serait un nombre et on parlerait d'une tâche de *régression*. La figure 1.1 illustre ces deux situations.



◀ **Fig. 1.1.** *Exemples typiques d'apprentissage supervisé : classification (à gauche) pour prédire le sexe, et régression (à droite) pour prédire l'âge, Les photos aux deux extrémités de l'axe des x sont des exemples d'entraînement pour lesquels l'étiquette est connue, tandis que la photo du milieu est celle pour laquelle on veut obtenir une prédiction.*

L'algorithme est *entraîné* sur un ensemble de données pré-collectées (l'ensemble d'entraînement), de la forme $\mathcal{D} = \{z_1, \ldots, z_n\}$, avec $z_i = (x_i, y_i)$. De nombreux algorithmes supposent que ces exemples sont tirés de manière indépendante et identiquement distribuée (i.i.d.) d'une distribution P, c.à.d. $z_i \sim P(X, Y)$ (où la forme exacte de P n'est pas connue d'avance). Selon la tâche à résoudre, la prédiction d'un algorithme d'apprentissage supervisé va généralement tenter d'approximer l'une des trois quantités suivantes :

- $P(y|x)$: dans notre exemple de classification (trouver le sexe), il faudrait prédire la probabilité qu'une photo soit une photo d'un homme par un nombre p (la probabilité que ce soit la photo d'une femme étant alors $1-p$). Dans notre exemple de régression (trouver l'âge), la prédiction serait une densité de probabilité sur l'intervalle $]0, +\infty[$.
- $\text{argmax}_y P(y|x)$: dans notre exemple de classification, il s'agirait d'une prédiction binaire du sexe le plus probable ("homme" ou "femme"). Dans notre exemple de régression, il s'agirait de l'âge le plus probable.
- $E_Y[Y|x] = \int_y yP(y|x)dy$: cette quantité n'a de sens que pour la régression, et il s'agirait dans notre exemple de prédire l'âge moyen de la personne étant donnée sa photo.

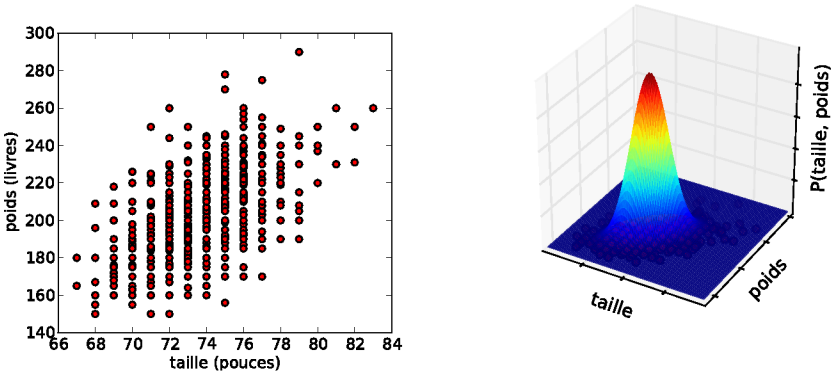
Il faut noter que la prédiction de $P(y|x)$ est la tâche la plus générale (dans la mesure où la résoudre permet également d'accomplir les deux autres), mais tous les algorithmes d'apprentissage ne sont pas capables d'estimer une distribution de probabilité.

1.2.2 Apprentissage non supervisé

Dans l'apprentissage *non supervisé*, un exemple $z_i = x_i \sim P(X)$ ne contient pas d'étiquette explicite. Il existe plusieurs types de tâches d'apprentissage non supervisé, parmi lesquelles les plus souvent rencontrées sont :

- L'estimation de densité : estimer $P(x)$, comme illustré en figure 1.2.

- La génération de données : tirer de nouveaux exemples d'une distribution la plus proche possible de $P(X)$.
- L'extraction de caractéristiques : trouver une fonction f telle que $f(x)$ soit "intéressant", par exemple pour :
  - compresser x, c.à.d. que $f(x)$ devrait être de dimension plus petite que x tout en permettant de reconstruire x par une fonction g telle que $x \simeq g(f(x))$,
  - simplifier l'apprentissage à partir de x, c.à.d. que $f(x)$ devrait être tel qu'un algorithme d'apprentissage (possiblement supervisé) utilisant $f(x)$ comme entrée au lieu de x donne de meilleurs résultats.
- Le regroupement ("clustering" en anglais) : partitionner les exemples en groupes $G_1, \ldots, G_k$ tels que tous les exemples dans un même groupe soient similaires (où plusieurs notions de similarité peuvent être utilisées, menant à des résultats potentiellement très différents).

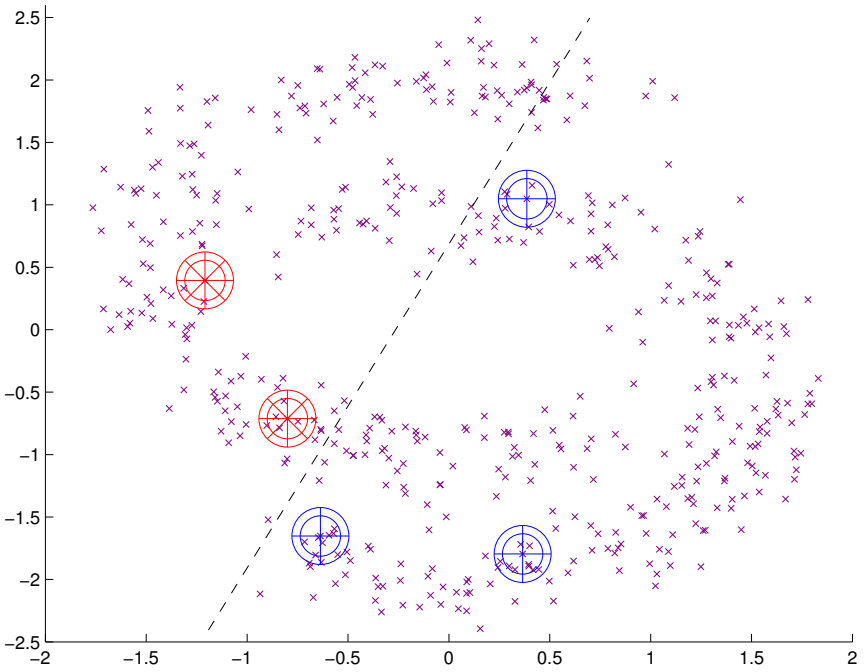


▶ **Fig. 1.2.** *Exemple d'apprentissage non supervisé : l'estimation de densité. À gauche, les données d'origine (taille et poids de 1033 joueurs de baseball aux États-Unis). À droite, l'estimation de la densité par une distribution Gaussienne.*

1.2.3 Apprentissage semi-supervisé

Comme son nom l'indique, l'apprentissage *semi-supervisé* est à mi-chemin entre le supervisé et le non supervisé : certains exemples $z_i = (x_i, y_i)$ $(1 \leq i \leq \ell)$ ont une étiquette, tandis que d'autres exemples $z_i = x_i$ $(\ell+1 \leq i \leq n)$ n'en ont pas (on dit qu'ils ne sont pas "étiquetés"). Les algorithmes d'apprentissage semi-supervisé tentent généralement de résoudre des problèmes d'apprentissage supervisé, mais en utilisant les exemples non étiquetés pour améliorer leur prédiction. L'idée sous-jacente, présentée plus en détails dans le prochain chapitre (section 2.5), est que la distribution des entrées $P(X)$ peut nous donner de l'information sur $P(Y|X)$ même en l'absence d'étiquettes. Un exemple typique où c'est le cas, pour une tâche de classification, est lorsque les exemples de chaque classe forment des groupes distincts dans l'espace des entrées, séparés par des zones de faible densité $P(x)$. La figure 1.3 montre ainsi deux classes dont la forme en croissant est révélée par les exemples non étiquetés. Un algorithme purement supervisé – donc

ignorant ces exemples – ne pourrait pas identifier correctement ces deux classes.



◀ **Fig. 1.3.** *Apprentissage semi-supervisé : ici seuls 5 exemples sont étiquetés (deux de la classe * en rouge, et trois de la classe + en bleu). Un algorithme n'utlisant pas les exemples non étiquetés (petits x mauves) séparerait les deux classes par exemple selon la ligne en pointillés, alors qu'un algorithme semi-supervisé (ou un humain) pourrait séparer les deux "croissants de lune" correspondant à chaque classe.*

Dans le cadre de l'apprentissage supervisé décrit précédemment, l'application d'un algorithme se fait typiquement en deux étapes :

1. L'algorithme va d'abord apprendre une tâche (phase d'*entraînement*) sur un ensemble $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$.

2. L'algorithme doit ensuite faire ses prédictions (phase de *test*) sur un ensemble $\mathcal{T}$ dont on ne fournit pas les étiquettes.

Un algorithme semi-supervisé peut également suivre ces deux étapes (en n'oubliant pas que $\mathcal{D}$ peut contenir également des exemples non étiquetés). On dit alors que la phase de prédiction sur l'ensemble de test se fait par *induction*. Mais puisque l'algorithme peut utiliser des exemples non étiquetés au cours de l'apprentissage, on peut également inclure $\mathcal{T}$ dans $\mathcal{D}$: on parle alors de *transduction*, et en général les performances seront meilleures qu'en induction puisque plus d'exemples sont disponibles pour l'apprentissage. Il existe malgré tout des applications où il n'est pas possible d'inclure $\mathcal{T}$ dans l'ensemble d'entraînement, par exemple lorsque les éléments de $\mathcal{T}$ sont générés en temps réel et que l'on a besoin de prédictions immédiates : si ré-entraîner l'algorithme avant chaque nouvelle prédiction s'avère trop lent, l'induction est alors la seule option possible.

1.2.4 Apprentissage par renforcement

L'apprentissage par renforcement est une forme d'apprentissage supervisé où l'algorithme n'observe pas une étiquette pour chacune de ses prédictions, mais plutôt une mesure de la qualité de ses prédictions (possiblement prenant en compte toute une séquence de prédictions). Cette thèse n'aborde pas l'apprentissage par renforcement, que je mentionne ici uniquement par souci d'exhaustivité. Le lecteur intéressé par ce sujet pourra se référer au livre référence de Sutton et al. (1998).

1.3 Généralisation

1.3.1 Sur-apprentissage

L'entraînement d'un algorithme d'apprentissage consiste à extraire, de manière explicite ou implicite, des caractéristiques de la distribution de probabilité P qui génère les données. Mais P étant inconnue, l'algorithme se base à la place sur un nombre fini d'exemples d'entraînement, c.à.d. sur la distribution discrète $\hat{P}$ des exemples disponibles dans $\mathcal{D}$ (appelée la distribution *empirique*). Lorsque certaines caractéristiques apprises sur $\hat{P}$ ne s'appliquent pas à P, on parle de *sur-apprentissage*, et on risque une mauvaise *généralisation*, c.à.d. que l'algorithme ne va pas obtenir une bonne performance sur de nouveaux exemples tirés de P.

Prenons l'exemple de la classification, lorsqu'un modèle estime $P(y|x)$ par une fonction $q_x(y)$. Afin de mesurer la similarité entre q_x et $P(\cdot|x)$, on peut par exemple utiliser la divergence de Kullback-Leibler D_{KL} (Kullback, 1959), définie par :

$$D_{KL}(P(\cdot|x)\|q_x) = \sum_y P(y|x) \ln \frac{P(y|x)}{q_x(y)}.$$

Cette quantité est toujours supérieure ou égale à zéro, et est égale à zéro si et seulement si q_x est égale à $P(\cdot|x)$. La minimisation de la divergence de Kullback-Leibler est donc un critère raisonnable pour obtenir une fonction q_x qui approxime $P(\cdot|x)$. Vu que le but est d'obtenir une bonne approximation pour toutes les valeurs de x qui pourraient être générées par P, il est logique de considérer la minimisation du critère

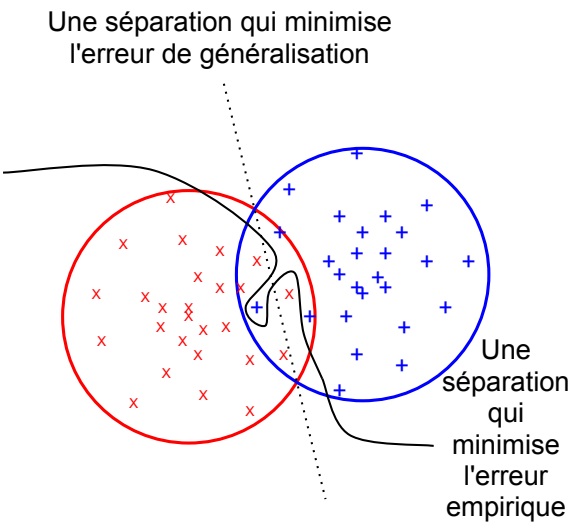
$$\begin{aligned} C(q) &= E_X[D_{KL}(P(\cdot|X)\|q_X] \\ &= \int_x \sum_y P(x)P(y|x) \ln \frac{P(y|x)}{q_x(y)} dx. \end{aligned} \tag{1.1}$$

$C(q)$ est ici ce que l'on appelle *l'erreur de généralisation*, c.à.d. l'erreur moyenne sur des exemples tirés de P. Puisque P est inconnue, on minimise en pratique un critère $\hat{C}$ défini de la même manière en remplaçant P

par $\hat{P}$. C'est le principe de minimisation du risque empirique (Vapnik, 1998), et $\hat{C}$ s'écrit ici :

$$\hat{C}(q) = \sum_{i=1}^{n} \frac{1}{n} \ln \frac{1}{q_{x_i}(y_i)} = -\frac{1}{n} \sum_{i=1}^{n} \ln q_{x_i}(y_i) \tag{1.2}$$

qui est appelée la *log-vraisemblance négative* (en anglais NLL, pour "Negative Log-Likelihood"). Ce critère est minimisé dès que $q_{x_i}(y_i) = 1$ pour tous les exemples d'entraînement $(x_i, y_i) \in \mathcal{D}$, *et ce quelle que soit la valeur de* $q_x(y)$ *pour des valeurs de* x *non observées dans* $\mathcal{D}$. Une fonction q peut donc minimiser $\hat{C}(q)$ sans nécessairement minimiser $C(q)$. Si c'est le cas, on est en situation de *sur-apprentissage*, illustrée en figure 1.4.



◀ **Fig. 1.4.** *Situation de sur-apprentissage : classification binaire (les classes sont les cercles rouge et bleu, avec respectivement les* $\times$ *et les* $+$ *comme exemples d'entraînement). Une séparation idéale en terme d'erreur de généralisation serait la ligne en pointillés, mais un algorithme dont le but est uniquement de minimiser l'erreur empirique pourrait par exemple séparer les exemples selon la ligne pleine : la classification des exemples d'entraînement serait parfaite, mais l'erreur de généralisation serait plus élevée que celle de la ligne en pointillés.*

Pour une distribution fixe des données, deux facteurs principaux augmentent le risque de sur-apprentissage :

- Le manque d'exemples d'entraînement : moins il y a d'exemples, plus il existe de fonctions minimisant le critère $\hat{C}(q)$ (eq. 1.2), parmi lesquelles seulement un petit nombre seront vraiment proches de la "vraie" solution au problème.
- Pas assez de contraintes dans la forme de la fonction q : moins la classe de fonctions à laquelle q appartient est restreinte, plus l'algorithme d'apprentissage risque de tirer parti de la flexibilité de q pour apprendre des "détails" des exemples d'entraînement, qui ne se généralisent pas à la vraie distribution P. C'est le cas par exemple dans la figure 1.4 où la séparation tarabiscotée des exemples par la ligne pleine permet de minimiser l'erreur empirique, mais va mener à une plus grande erreur de généralisation qu'une simple ligne droite.

1.3.2 Régularisation

Un moyen de lutter contre le sur-apprentissage est d'utiliser une technique dite de *régularisation*. Il existe plusieurs méthodes de régularisation, mais elles partagent le même principe : rajouter au processus d'apprentissage des contraintes qui, si elles sont appropriées, vont améliorer les capacités de généralisation de la solution obtenue.

Reprenons par exemple le cas de la classification, où l'on cherche à minimiser le critère $C(q)$ (éq. 1.1), que l'on approxime par $\hat{C}(q)$ (éq. 1.2). Comme nous venons de le voir, ce problème est mal défini car il existe une infinité de fonctions qui minimisent $\hat{C}$, sans donner aucune garantie sur la valeur de C. Une première façon de régulariser le problème est donc de restreindre la forme de q : par exemple si $x \in \mathbb{R}^d$ et $y \in \{0, 1\}$ on peut se limiter aux fonctions de la forme

$$q_x(1) = \frac{1}{1 + e^{-w^T x}} \tag{1.3}$$

où $w \in \mathbb{R}^d$ est le vecteur de paramètres du modèle.

Notons que si les x_i de l'ensemble d'entraînement sont linéairement indépendants, alors cette contrainte sur la forme de q n'est pas suffisante, puisqu'il est toujours possible que $\hat{C}(q)$ soit arbitrairement proche de zéro sans pour autant avoir de garantie sur la valeur de $C(q)$. Une technique classique de régularisation consiste alors à rajouter au critère $\hat{C}$ une mesure qui pénalise la *complexité* de la solution, suivant le principe du rasoir d'Occam qui dit que les hypothèses les plus simples sont les plus vraisemblables (voir par exemple Blumer et al., 1987). Une possibilité est de minimiser

$$\tilde{C}(q) = \hat{C}(q) + \lambda \|w\|^2 \tag{1.4}$$

au lieu de $\hat{C}$, pour q défini comme dans l'éq. 1.3, afin d'empêcher le vecteur w de contenir des valeurs arbitrairement grandes (en valeur absolue). Le paramètre λ contrôle la force de cette contrainte (lorsque $\lambda \to +\infty$ la seule solution possible est la fonction constante $q_x(1) = q_x(0) = 0.5$, qui est la plus simple qu'on puisse imaginer). Le critère empirique $\hat{C}(q^*)$ pour la fonction q^* qui minimise le critère régularisé $\tilde{C}$ pourrait ne pas être proche de zéro, mais on peut souvent ainsi – pour *certaines* valeurs de λ – obtenir des valeurs plus basses du critère de généralisation C (celui qui nous intéresse vraiment). C'est le principe de la minimisation du risque structurel (Vapnik, 1998).

Dans cet exemple, nous avons utilisé $\|w\|^2$ pour mesurer la complexité de la fonction q définie à partir de w par l'éq. 1.3. En général, il n'existe pas une seule mesure de complexité universelle qui soit appropriée pour tous les algorithmes d'apprentissage, et le choix de la mesure de complexité à pénaliser joue un rôle très important. La *complexité de Kolmogorov* (Solomonoff, 1964; Kolmogorov, 1965), qui sera mentionnée dans cette thèse, est une mesure de complexité très générique qui est intéressante en théorie, même si en pratique elle est souvent impossible à utiliser directement. Elle consiste à dire que la complexité d'une fonction est la taille du plus petit

programme qui l'implémente. Un premier obstacle à l'utilisation de cette complexité est le fait qu'il faille choisir un langage de programmation approprié : par exemple si le langage choisi contient une fonction primitive qui calcule le produit scalaire, alors, dans notre exemple ci-dessus la plupart des fonctions q définies par l'éq. 1.3 ont la même complexité de Kolmogorov. Par contre, si le produit scalaire n'est pas une primitive du langage (et qu'il n'y a pas d'instruction de boucle), alors il faut l'écrire comme une somme de produits et q est d'autant plus complexe que w a d'éléments non nuls. Une autre difficulté est qu'il n'est en général pas possible d'optimiser la complexité de Kolmogorov de manière efficace, ce qui rend vaine son utilisation directe dans un processus d'optimisation. Elle a malgré tout de nombreuses applications, comme décrit dans le livre de Li et al. (2008).

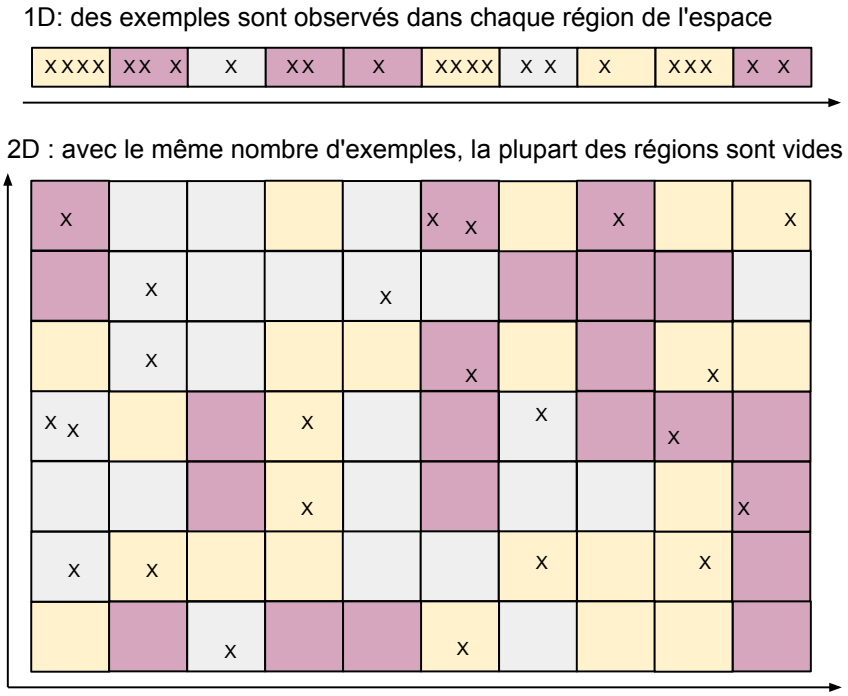
1.3.3 Malédiction de la dimensionalité

On peut observer empiriquement – et dans certains cas justifier mathématiquement – que plus la dimension d de l'entrée x est élevée, plus les tâches d'apprentissage machine ont tendance à être difficiles à résoudre. C'est ce qu'on appelle la *malédiction de la dimensionalité* (Bellman, 1961). Il existe plusieurs manifestations de cette malédiction. La plus importante dans le contexte de cette thèse est le fait que le nombre de combinaisons possibles des entrées augmente exponentiellement avec la dimensionalité d : en notant x_{ij} la valeur associée à la j-ème dimension de l'entrée x_i, si l'on suppose que ces entrées ne peuvent prendre qu'un nombre fini k de valeurs, alors le nombre de combinaisons possibles est égal à k^d. Un algorithme qui apprend "bêtement" à associer une valeur à chaque combinaison sans partager d'information entre les différentes combinaisons n'a aucune chance de fonctionner en haute dimension, car il ne pourra pas généraliser aux multiples combinaisons qui n'ont pas été vues dans l'ensemble d'entraînement. Dans le cas où x_{ij} n'est pas contraint dans un ensemble fini de valeurs, l'intuition reste la même pour certains algorithmes qui consistent à "partitionner" $\mathbb{R}^d$ en régions indépendantes (possiblement de manière implicite) : si le nombre de ces régions augmente exponentiellement avec d, alors un tel algorithme aura de la difficulté à généraliser pour de grandes valeurs de d. La figure 1.5 illustre ce phénomène en une et deux dimensions, et il faut garder à l'esprit que la situation peut s'avérer encore bien pire lorsque l'on manipule des entrées à plusieurs centaines de dimensions.

1.4 Différents types de modèles

1.4.1 Modèles paramétriques

En apprentissage machine, un modèle *paramétrique* est défini par un ensemble Θ de paramètres de dimension finie, et l'algorithme d'apprentissage associé consiste à trouver la meilleure valeur possible de Θ. Prenons



▶ **Fig. 1.5.** *Malédiction de la dimensionalité : si l'algorithme partitionne l'espace en régions indépendantes, le nombre d'exemples nécessaires pour remplir ces régions augmente de manière exponentielle avec la dimension. Ici, la couleur d'une région représente la classe majoritaire dans cette région, et un tel algorithme pourrait bien généraliser à partir de 23 exemples d'entraînement pour le problème du haut (1D), mais pas pour celui du bas (2D).*

l'exemple typique de la régression linéaire : le modèle tente d'estimer $E_Y[Y|x]$ où $y \in \mathbb{R}$ et $x \in \mathbb{R}^d$ par une fonction $f(x) = w^T x + b$, avec $w \in \mathbb{R}^d$ et $b \in \mathbb{R}$. On a alors $\Theta = \{w, b\}$ et un algorithme d'apprentissage possible serait de minimiser le critère suivant (qui contient un terme de régularisation comme justifié en section 1.3.2) :

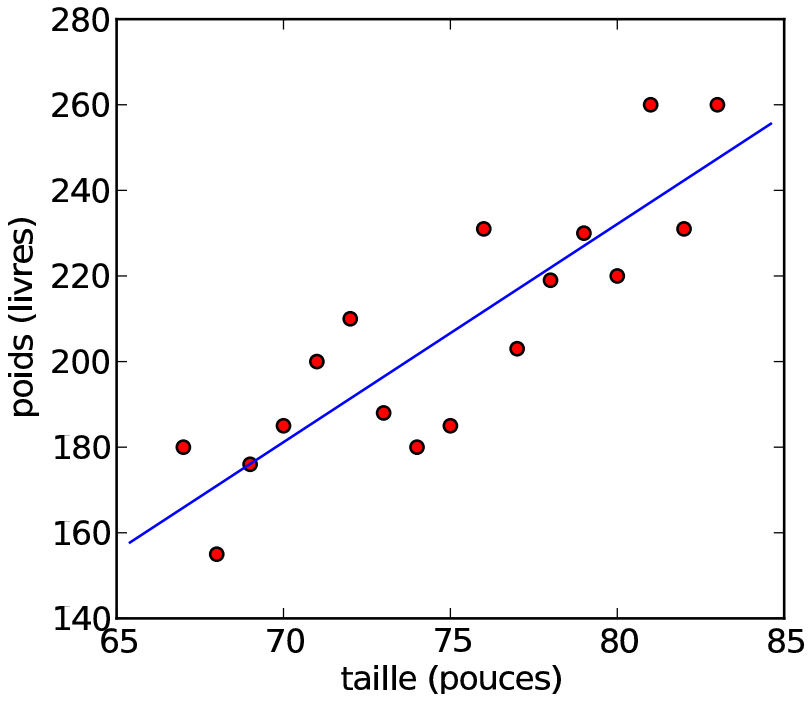
$$\sum_{i=1}^{n} (w^T x_i + b - y_i)^2 + \lambda \|w\|^2. \tag{1.5}$$

La figure 1.6 montre un exemple de régression linéaire en une dimension, sans régularisation.

1.4.2 Modèles non paramétriques

Un modèle *non paramétrique* n'a au contraire pas d'ensemble fixe de paramètres : le nombre de variables utilisées par le modèle augmente généralement avec le nombre d'exemples dans l'ensemble d'entraînement. Un exemple de modèle non paramétrique pour résoudre le même problème de régression que celui décrit en 1.4.1 est l'algorithme des fenêtres de Parzen, aussi appelé régression de Nadaraya-Watson (Nadaraya, 1964; Watson, 1964). Il consiste à écrire la prédiction du modèle comme

$$f(x) = \frac{1}{\sum_{i=1}^{n} K(x, x_i)} \sum_{i=1}^{n} K(x, x_i) y_i \tag{1.6}$$

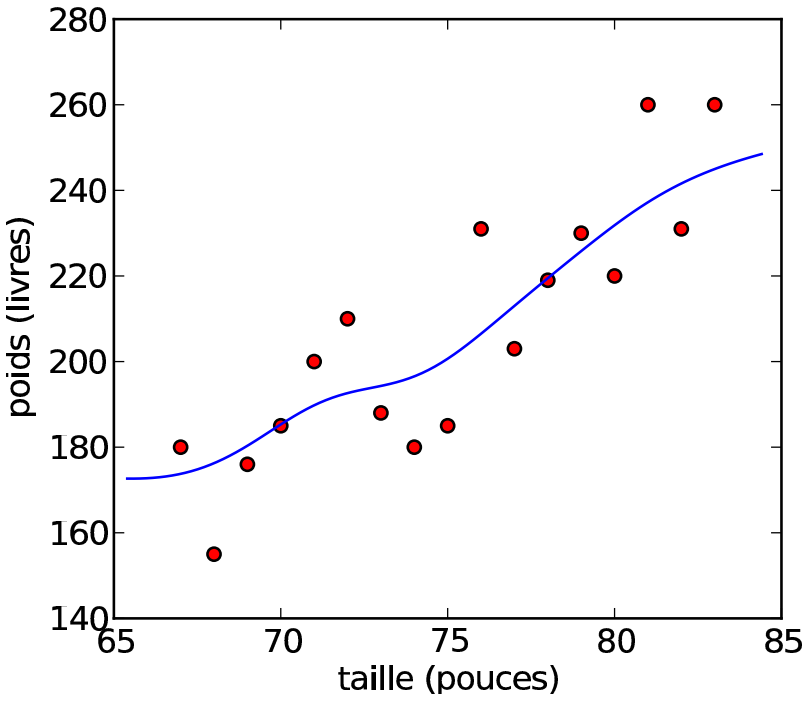


◀ **Fig. 1.6.** *Modèle paramétrique : la régression linéaire. Les données (points rouges) sont un sous-ensemble des mêmes données que dans la figure 1.2, et la tâche est ici de prédire le poids d'un joueur de baseball en fonction de sa taille. La prédiction du modèle minimisant le critère de l'éq. 1.5 (ici avec $\lambda = 0$, c.à.d. sans régularisation) est donnée par la ligne bleue.*

où $K(\cdot, \cdot)$ est appelée la fonction noyau ("kernel" en anglais). Il s'agit donc d'une moyenne pondérée des étiquettes observées dans l'ensemble d'entraînement $\mathcal{D}$, où le poids est donné par le noyau K qu'on peut interpréter comme une fonction de similarité entre deux entrées. Les exemples $(x_i, y_i) \in \mathcal{D}$ font donc partie des variables utilisées par le modèle pour faire sa prédiction, même après que l'apprentissage est terminé (l'algorithme d'apprentissage le plus simple consiste à uniquement mémoriser les paires (x_i, y_i), mais il serait aussi possible d'optimiser certains paramètres du noyau K, si besoin est). La figure 1.7 montre un exemple de régression par fenêtres de Parzen en une dimension.

1.5 Retour sur l'efficacité statistique

Maintenant que les bases de l'apprentissage machine ont été posées, nous sommes en mesure de revenir à la question de l'efficacité *statistique* introduite au début de ce chapitre. De manière informelle, nous dirons qu'un algorithme d'apprentissage est statistiquement efficace s'il est capable d'obtenir une bonne capacité de généralisation avec un nombre limité d'exemples d'apprentissage. C'est une caractéristique extrêmement importante pour plusieurs raisons :



▶ **Fig. 1.7.** *Modèle non paramétrique : régression par fenêtres de Parzen. Les données sont les mêmes que dans l'exemple de la régression linéaire (figure 1.6). Le noyau K utilisé ici est un noyau de type Gaussien : $K(x, x_i) = e^{-\frac{(x-x_i)^2}{8}}$.*

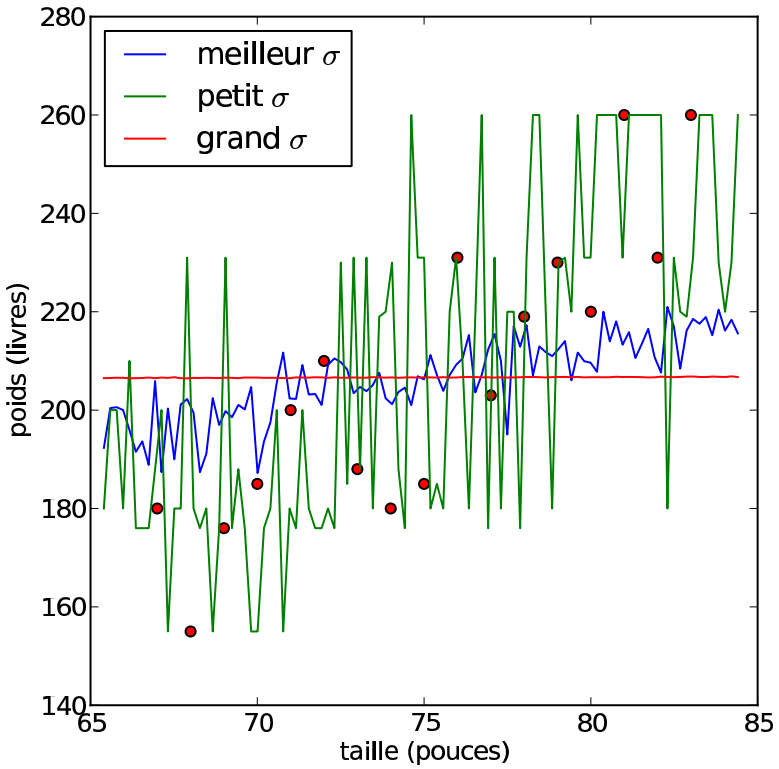
- La collecte de données est souvent coûteuse en temps et en argent.
- Même dans une situation où le nombre d'exemples d'entraînement disponibles serait infini, les limites computationnelles (temps de calcul et mémoire disponibles) pour exécuter un algorithme le contraignent à n'en utiliser qu'un sous-ensemble.
- Les humains sont reconnus pour leur capacité à généraliser à partir de très peu d'exemples. Si l'on souhaite reproduire l'intelligence humaine, il faut donc chercher dans la direction d'algorithmes ayant également cette capacité (principe du "one-shot learning", étudié par exemple par Fei-Fei et al., 2006).

Un exemple d'algorithme qui peut être statistiquement inefficace est la régression par fenêtres de Parzen (éq. 1.6), lorsque la fonction noyau K est le populaire noyau Gaussien, défini par

$$K(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}} \tag{1.7}$$

où $\sigma \in \mathbb{R}$ (paramètre du noyau). Intuitivement, on peut voir que la prédiction $f(x)$ définie par l'éq. 1.6 ne dépend alors que des exemples x_i proches de x (où la notion de proximité dépend de σ). Pour que la prédiction soit correcte, on s'attend donc à avoir besoin d'exemples d'apprentissage dans chaque région de l'espace où $P(x) > 0$ (où la taille d'une région est proportionnelle à σ). Ce nombre de régions pouvant augmenter de manière exponentielle avec

la dimension d de l'espace des entrées $x \in \mathbb{R}^d$ (c'est la malédiction de la dimensionalité introduite en 1.3.3), cela signifie que le nombre d'exemples d'apprentissage nécessaires pour avoir une bonne qualité de prédiction risque d'augmenter de manière exponentielle avec d. La figure 1.8 montre ainsi ce qu'il se passe lorsque des dimensions supplémentaires indépendantes de l'étiquette sont rajoutées : la similarité dans l'espace des entrées devenant moins indicative de la similarité des étiquettes, l'algorithme ne peut généraliser correctement avec peu d'exemples.

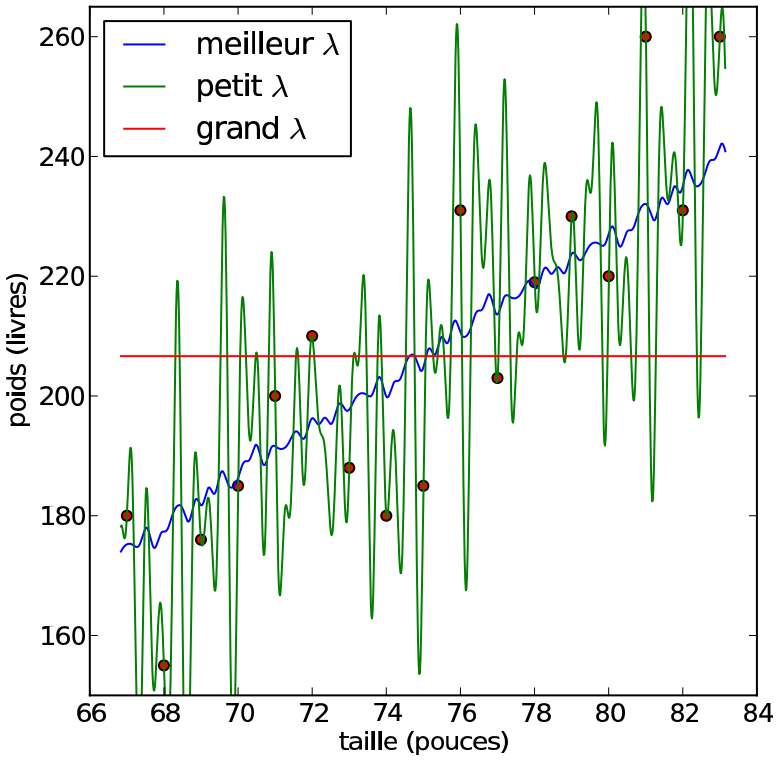


◀ **Fig. 1.8.** *Inefficacité statistique des fenêtres de Parzen. Les données originales ont été perturbées par l'ajout de 10 dimensions aléatoires en entrée. Une petite valeur de σ rend la prédiction bien trop bruitée, tandis qu'une grande valeur correspond à une prédiction constante. La meilleure valeur de σ (qui minimise l'erreur de généralisation estimée à partir d'autres exemples tirés de la même distribution) mène également à une prédiction médiocre à cause du faible nombre d'exemples d'entraînement.*

Les modèles paramétriques peuvent également être statistiquement inefficaces. En général, le nombre d'exemples d'apprentissage nécessaires pour optimiser k paramètres est de l'ordre de k^*. S'il y a moins d'exemples, alors le problème est sur-paramétré et on risque le sur-apprentissage : le modèle pourrait apprendre des paramètres taillés "sur mesure" pour les données d'entraînement, mais qui mèneront à une mauvaise généralisation. La conséquence de cette observation est qu'en général, un modèle avec un grand nombre de paramètres est statistiquement inefficace (puisqu'il a besoin d'un

*La formalisation de cet énoncé dépend de l'algorithme exact ainsi que de la distribution des données : nous nous contenterons donc ici de la version intuitive.

grand nombre d'exemples d'apprentissage pour que les paramètres appris généralisent bien). Par exemple, dans la régression linéaire (éq. 1.5), si le nombre d'exemples est plus petit que la taille de w et que les exemples sont linéairement indépendants, alors il existe une infinité de vecteurs w tels que $w^T x_i + b = y_i$ pour $i \in \{1, \ldots, n\}$. La régularisation (ici le terme $\lambda \|w\|^2$) s'avère alors essentielle : on peut la voir comme un moyen de limiter implicitement le nombre de paramètres effectifs du modèle, ce qui permet d'avoir besoin de moins d'exemples pour éviter le sur-apprentissage (voir figure 1.9).



▶ **Fig. 1.9.** *Inefficacité statistique d'un modèle de régression linéaire sur-paramétrisé. Les données originales ont été augmentées par l'ajout de 20 dimensions de la forme* $\sin(kx)$ *où* x *est l'entrée d'origine (la taille d'un joueur), et* k *un entier de 1 à 20. Sans régularisation (petit* λ*) il n'y a pas assez d'exemples par rapport au nombre de paramètres, et la prédiction – bien que parfaite sur les exemples d'entraînement – généralise mal. La régularisation permet de limiter le nombre de paramètres effectifs et d'atténuer le bruit, mais* λ *doit être choisi avec soin (s'il est trop grand, la prédiction sera constante).*

Notons que dans ces deux exemples – fenêtres de Parzen en figure 1.8 et régression linéaire en figure 1.9 – l'inefficacité statistique des algorithmes considérés est une manifestation de la malédiction de la dimensionalité vue en section 1.3.3. Nous étudierons dans cette thèse d'autres résultats similaires, également reliés à ce phénomène.

Bibliographie

Bellman, R. 1961, *Adaptive Control Processes : A Guided Tour*, Princeton University Press, New Jersey.

Blumer, A., A. Ehrenfeucht, D. Haussler et M. Warmuth. 1987, "Occam's razor", *Inf. Proc. Let.*, vol. 24, p. 377–380.

Fei-Fei, L., R. Fergus et P. Perona. 2006, "One-shot learning of object categories", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, n^{o} 4, p. 594–611.

Kolmogorov, A. N. 1965, "Three approaches to the quantitative definition of information", *Problems of Information and Transmission*, vol. 1, n^{o} 1, p. 1–7.

Kullback, S. 1959, *Information Theory and Statistics*, Wiley, New York.

Li, M. et P. Vitányi. 2008, *An Introduction to Kolmogorov Complexity and Its Applications*, 3^{e} éd., Springer, New York, NY.

Nadaraya, E. A. 1964, "On estimating regression", *Theory of Probability and its Applications*, vol. 9, p. 141–142.

Solomonoff, R. J. 1964, "A formal theory of inductive inference", *Information and Control*, vol. 7, p. 1–22, 224–254.

Sutton, R. et A. Barto. 1998, *Reinforcement Learning : An Introduction*, MIT Press.

Vapnik, V. 1998, *Statistical Learning Theory*, Wiley, Lecture Notes in Economics and Mathematical Systems, volume 454.

Watson, G. S. 1964, "Smooth regression analysis", *Sankhya - The Indian Journal of Statistics*, vol. 26, p. 359–372.

Weizenbaum, J. 1966, "ELIZA-a computer program for the study of natural language communication between man and machine", *Commun. ACM*, vol. 9, n^{o} 1, p. 36–45.

2 Algorithmes d'apprentissage

Il existe des milliers d'algorithmes d'apprentissage, et le but de ce chapitre n'est évidemment pas de tous les recenser. Les algorithmes présentés ici sont ceux qui apparaissent dans les articles constituant le corps de cette thèse. Par souci de concision, la description de chaque algorithme est volontairement succinte, se concentrant sur les éléments importants pour la compréhension des articles qui y sont reliés. Des références offrant une présentation plus complète de chaque algorithme seront fournies pour satisfaire la curiosité du lecteur avide de détails. L'ordre dans lequel les algorithmes sont passés en revue dans ce chapitre est basé sur l'idée d'introduire d'abord les concepts les plus simples, sur lesquels se basent les algorithmes qui suivent.

2.1 k plus proches voisins

L'algorithme des k plus proches voisins est un algorithme non paramétrique utilisé pour la régression et la classification. Étant donnée une mesure de distance dans l'espace d'entrée $\mathbb{R}^d$ – souvent prise comme la distance Euclidienne $\|x_i - x_j\|$ – la prédiction du modèle sur un exemple de test $x \in \mathcal{T}$ dépend uniquement des k plus proches voisins de x dans l'ensemble d'entraînement $\mathcal{D}$. En notant $i_1(x), \ldots, i_k(x)$ les indices des k exemples de $\mathcal{D}$ les plus proches de x selon la distance choisie (ses "voisins"), la prédiction du modèle en régression est la moyenne des étiquettes observées chez ces k voisins :

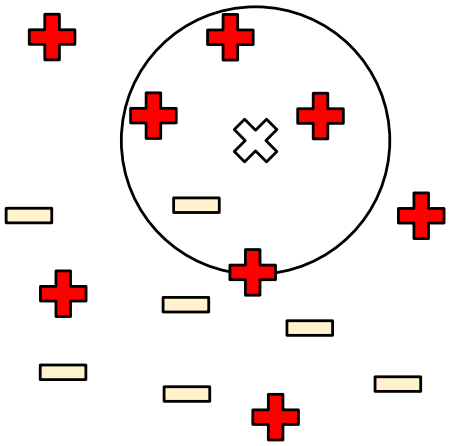
$$f(x) = \frac{1}{k} \sum_{j=1}^{k} y_{i_j(x)}$$

et en classification il s'agit d'un vote parmi les k voisins :

$$f(x) = \text{argmax}_y \sum_{j=1}^{k} \mathbf{1}_{y=y_{i_j(x)}}$$

où en cas d'égalité parmi les votes le modèle choisit aléatoirement l'une des classes majoritaires. La classification par les k plus proches voisins est illustrée en figure 2.1.

Les propriétés théoriques de l'algorithme des k plus proches voisins sont bien connues, ayant été étudiées depuis longtemps déjà (Cover et al., 1967).



◀ **Fig. 2.1.** *k plus proches voisins ($k = 5$, tâche de classification) : pour classifier un nouvel exemple (le $\times$ blanc) on cherche ses 5 plus proches voisins dans l'ensemble d'entraînement (ce sont ceux à l'intérieur du cercle), et on compte le nombre d'exemples de chaque classe. Il y a ici 4 exemples de la classe + (rouge) et 1 de la classe − (jaune) donc le $\times$ sera classifié comme +.*

Les améliorations récentes de cet algorithme se concentrent sur des techniques visant à le rendre utilisable sur de grands ensembles de données, et à tirer parti de mesures de distances plus évoluées que la simple distance Euclidienne (Shakhnarovich et al., 2006).

2.2 Fenêtres de Parzen

L'algorithme des fenêtres de Parzen a déjà été présenté au chapitre précédent (section 1.4.2) dans le contexte de la régression non paramétrique, où on l'appelle parfois la régression à noyau ou la régression de Nadaraya-Watson (Nadaraya, 1964; Watson, 1964).

On peut également utiliser une approche similaire en apprentissage non supervisé pour l'estimation de densité (Rosenblatt, 1956; Parzen, 1962), en estimant la densité de probabilité au point x par

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} K(x, x_i)$$

ce qui correspond à placer une masse de probabilité "autour" de chaque exemple d'apprentissage x_i, dans un volume défini par le noyau K. Ici, K doit respecter les contraintes

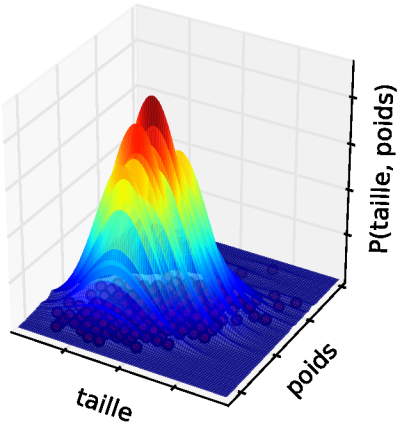
$$K(x, x_i) \geq 0$$
$$\int_x K(x, x_i) dx = 1$$

de manière à ce que f soit une densité de probabilité valide. Le choix le plus répandu pour K est le noyau Gaussien, défini d'une manière similaire

à l'éq. 1.7 mais avec la normalisation appropriée :

$$K(x_i, x_j) = \mathcal{N}(x_i; x_j, \sigma^2 \mathbf{I}) = \frac{1}{(2\pi)^{d/2}\sigma^d} e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

où l'on note $\mathcal{N}(\cdot; \mu, \Sigma)$ la densité de probabilité d'une Gaussienne de moyenne μ et covariance Σ. Un exemple d'estimation de densité par fenêtres de Parzen avec un noyau Gaussien est montré en figure 2.2.



◀ **Fig. 2.2.** *Fenêtres de Parzen pour l'estimation de densité : les données sont les mêmes que dans la figure 1.2. Au lieu d'estimer la densité par une seule Gaussienne, on utilise une Gaussienne centrée sur chaque point (ici, sigma = 2).*

2.3 Mélanges de Gaussiennes

Les mélanges de Gaussiennes généralisent les fenêtres de Parzen (avec noyau Gaussien) pour l'estimation de densité, en écrivant la densité comme une somme pondérée de Gaussiennes :

$$f(x) = \sum_{j=1}^{N} \alpha_j \mathcal{N}(x; \mu_j, \Sigma_j)$$

où N est le nombre de composantes du mélange, et α_j le poids de la j-ème composante (les poids sont tels que $\alpha_j \geq 0$ et $\sum_j \alpha_j = 1$). L'interprétation dite "générative" de cette équation est que le modèle suppose que chaque exemple observé a été généré de la façon suivante :

1. Une composante j est choisie aléatoirement, avec probabilité α_j.
2. Un exemple est généré par une distribution Gaussienne centrée en μ_j avec covariance Σ_j.

Si $N = n$, $\alpha_j = n^{-1}$, $\mu_j = x_j$ et $\Sigma_j = \sigma^2\mathbf{I}$ on retrouve les fenêtres de Parzen non paramétriques vues précédemment. Mais on peut également apprendre un modèle paramétrique de mélange en fixant N et en optimisant les poids α_j, les centres μ_j et les covariances Σ_j de chaque Gaussienne. L'algorithme le plus populaire pour l'apprentissage d'un mélange de Gaussiennes est l'algorithme *Espérance-Maximisation** (Dempster et al., 1977). Il s'agit d'un algorithme itératif où chaque itération comporte deux étapes :

1. **Étape E (Espérance) :** calcul de la probabilité p_{ij} que l'exemple x_i ait été généré par la j-ème composante du mélange. En notant C la variable aléatoire qui représente la composante, et X la variable aléatoire qui représente l'entrée, on a en utilisant la règle de Bayes :

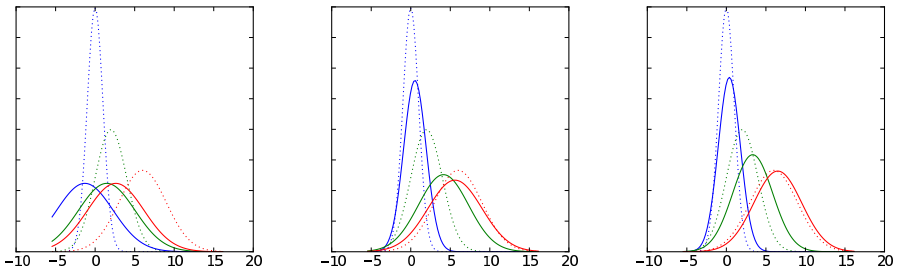
$$\begin{aligned} p_{ij} &\stackrel{\text{def}}{=} P(C = j | X = x_i) \\ &= \frac{P(X = x_i | C = j) P(C = j)}{P(X = x_i)} \\ &= \frac{\mathcal{N}(x_i; \mu_j, \Sigma_j)\alpha_j}{\sum_{k=1}^{N} \mathcal{N}(x_i; \mu_k, \Sigma_k)\alpha_k}. \end{aligned}$$

2. **Étape M (Maximisation) :** ré-estimation des paramètres du modèles α_j, μ_j et Σ_j pour augmenter la log-vraisemblance des données observées. La dérivation exacte est détaillée par exemple dans le livre de Bishop (2006), mais les équations obtenues sont suffisamment intuitives pour comprendre ce que fait l'algorithme. Les paramètres de la j-ème composante sont en effet calculés à partir des exemples x_i pondérés par les poids p_{ij}, c.à.d. que les exemples qui influent le plus sur les paramètres de la j-ème composante sont ceux que l'étape E a considérés comme étant probablement générés par cette composante :

$$\begin{aligned} \mu_j &= \frac{\sum_i p_{ij} x_i}{\sum_i p_{ij}} \\ \Sigma_j &= \frac{\sum_i p_{ij}(x_i - \mu_j)(x_i - \mu_j)^T}{\sum_i p_{ij}} \\ \alpha_j &= \frac{\sum_i p_{ij}}{\sum_{i,k} p_{i,k}}. \end{aligned}$$

L'algorithme EM est garanti de converger vers un maximum local de la vraisemblance (Dempster et al., 1977), mais il est possible qu'il y ait beaucoup de maxima locaux, de qualité très variable. En pratique EM est donc généralement exécuté plusieurs fois en partant de différentes valeurs initiales des paramètres, et on garde la meilleure solution ainsi obtenue. La figure 2.3 montre l'évolution de l'estimation d'un mélange de trois Gaussiennes en 1D, lorsque les composantes du mélange sont optimisées par EM à l'aide des équations ci-dessus.

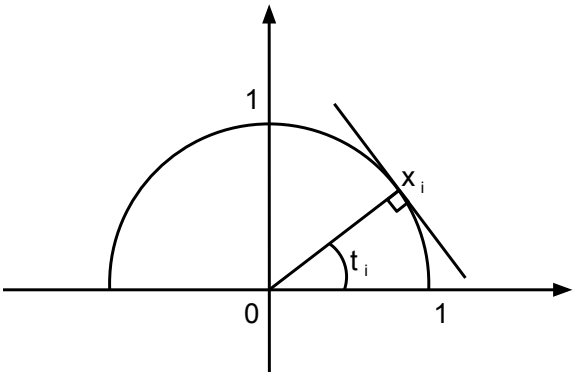
*EM, ou "Expectation-Maximization" en anglais.



◀ **Fig. 2.3.** *Mélange de Gaussiennes optimisé par EM, à partir de 1000 points générés par les trois Gaussiennes en pointillés. Les trois figures montrent les composantes après l'initialisation aléatoire (à gauche), après 10 itérations de EM (au milieu) et après 100 itérations (à droite).*

2.4 Apprentissage de variétés

Sans rentrer dans les détails mathématiques précis qui n'ont que peu d'intérêt dans le cadre de cette thèse, on dit que les données d'entrée $x_i \in \mathbb{R}^d$ appartiennent à une variété ("manifold" en anglais) de dimension $c < d$ si elles sont distribuées sur une surface qui, localement, peut être approximée par un espace linéaire de dimension c. Par exemple, avec $d = 2$, si tous les x_i vérifient $x_{i1}^2 + x_{i2}^2 = 1$, les données sont distribuées sur le cercle de centre 0 et rayon 1. En tout point du cercle, la surface du cercle peut localement être approximée par le plan tangent en ce point, qui est un sous-espace de dimension 1 : il s'agit donc d'une variété de dimension $c = 1$ (voir figure 2.4).

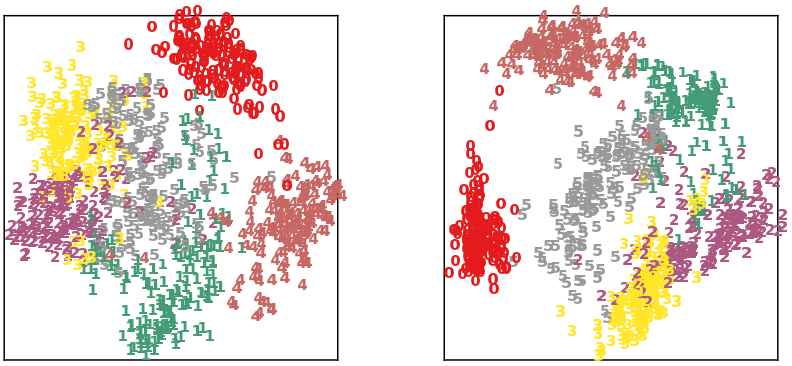


◀ **Fig. 2.4.** *Variété de dimension 1 dans un espace de dimension 2 : le demi-cercle de centre 0 et de rayon 1 peut être localement approximé par son plan tangent. Un point $x_i \in \mathbb{R}^2$ appartenant à cette variété pourrait être identifié par une seule coordonnée, comme l'angle t_i.*

L'idée de base de la plupart des algorithmes d'apprentissage de variétés est de tenter d'assigner à chaque $x_i \in \mathbb{R}^d$ un vecteur $t_i \in \mathbb{R}^c$ qui représente les "coordonnées intrinsèques" de x_i sur la variété. Par exemple, dans l'exemple du cercle présenté précédemment, il pourrait s'agir de l'angle entre x_i et un point référence sur le cercle. Il existe de nombreux algorithmes d'apprentissage de variétés, qui ont tous leurs propres spécificités. Ils sont en général basés sur l'idée que les coordonnées intrinsèques t_i doivent respecter des caractéristiques locales des données dans l'espace d'entrée (par exemple les distances entre voisins), tout en respectant des contraintes globales qui assurent une structure cohérente (par exemple les points éloignés dans l'espace d'entrée $\mathbb{R}^d$ doivent l'être aussi dans l'espace intrinsèque $\mathbb{R}^c$).

La force principale des algorithmes d'apprentissage de variétés est que les variétés ainsi découvertes, bien que contraintes à être localement linéaires, peuvent être globalement non linéaires. Lorsque les données sont effectivement distribuées sur une telle variété non linéaire, ils sont alors bien plus utiles pour réduire la dimensionalité qu'une analyse en composantes principales ("Principal Component Analysis" ou PCA en anglais). La réduction de la dimensionalité a généralement pour but :

- L'extraction de caractéristiques à partir desquelles l'apprentissage sera plus facile qu'à partir des entrées brutes x_i (en particulier cela permet de combattre la malédiction de la dimensionalité vue en section 1.3.3).
- La visualisation des données en deux ou trois dimensions, comme montré en figure 2.5 avec l'algorithme d'apprentissage de variétés Isomap (Tenenbaum et al., 2000).



▶ **Fig. 2.5.** *Projection en 2D de 1000 images des chiffres 0 à 5 (originellement de 8×8 pixels) à l'aide de l'analyse en composantes principales (à gauche) et de Isomap (à droite). Isomap permet de mieux séparer les groupes de chiffres qu'une méthode linéaire comme l'analyse en composantes principales. Aucun des algorithmes n'utilise les étiquettes (en couleur).*

Les méthodes d'apprentissage de variétés ont été popularisées par les articles de Roweis et al. (2000) et Tenenbaum et al. (2000). Ils ont depuis inspiré de nombreux autres algorithmes : le livre récent de Ma et al. (2011) présente l'état de l'art dans ce domaine.

2.5 Apprentissage semi-supervisé à base de graphe

L'apprentissage semi-supervisé (introduit en section 1.2.3) vise à tirer parti à la fois des exemples étiquetés et des exemples non étiquetés disponibles, afin d'améliorer l'apprentissage supervisé. Même s'il est théoriquement possible de faire de la régression semi-supervisée, la plupart des algorithmes sont analysés dans la littérature sur des tâches de classification. Dans le but de simplifier la présentation, l'analyse qui suit se restreint donc

au cas de la classification binaire ($y \in \{0, 1\}$). Plusieurs algorithmes d'apprentissage semi-supervisé ont été développés à partir des mêmes hypothèses de base (Chapelle et al., 2006) :

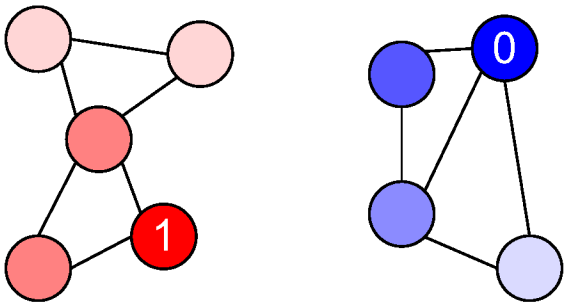
- **L'hypothèse de continuité** ("smoothness assumption" en anglais) stipule que $P(Y|x)$ est continue par rapport à x, c.à.d. si x_1 et x_2 sont proches dans l'espace des entrées, alors les distributions $P(Y|x_1)$ et $P(Y|x_2)$ sont également proches (dans l'espace des distributions). C'est une hypothèse également utilisée par des algorithmes d'apprentissage supervisé, et la version "semi-supervisée" de cette hypothèse consiste à également prendre en compte la densité $P(x)$ en disant que $P(Y|x)$ doit varier d'autant plus lentement que $P(x)$ est élevée (une autre façon de le dire est que les variations de l'étiquette se font surtout dans les régions de faible densité).
- **L'hypothèse de regroupement** ("cluster assumption" en anglais) stipule que l'on peut partitionner les données en groupes $G_1, \ldots, G_k$ selon un critère basé uniquement sur la distribution $P(x)$, de telle manière que tous les exemples d'un même groupe G_j soient de la même classe. Une autre façon de voir cette hypothèse est de la formuler par l'**hypothèse de séparation par régions de faible densité** ("low-density separation assumption" en anglais), qui dit que la surface de décision – définie par $P(Y = 0|x) = P(Y = 1|x)$ – se situe dans les régions de faible densité $P(x)$ (donc "entre" les groupes plutôt qu'à l'intérieur de ces groupes). Par exemple, dans la projection par Isomap de la figure 2.5, on voit que les classes tendent à former des groupes distincts.

Toutes ces hypothèses sont basées sur la même idée de base (utiliser la densité $P(x)$ pour déterminer les régions de variation de $P(Y|x)$), mais ont donné lieu à des algorithmes variés. Parmi ces algorithmes, une classe particulière qui sera étudiée en détails dans cette thèse est celle des algorithmes à base de graphe, où les données sont utilisées pour construire un graphe $\mathbf{g} = (V, E)$ où :

- Les nœuds V sont les exemples x_i.
- Les arêtes E sont les paires (i, j) connectant x_i et x_j s'ils sont considérés comme similaires (typiquement, si x_i est un des k plus proches voisins de x_j, ou vice-versa).

Chaque arête (i, j) est associée à un poids $\mathbf{W}_{ij}$ qui est souvent soit pris égal à 1, soit calculé à l'aide d'un noyau Gaussien comme dans l'éq. 1.7. Ce graphe peut donc être représenté uniquement à l'aide de la matrice $\mathbf{W}$ (avec $\mathbf{W}_{ij} = 0$ si $(i, j) \notin E$), et une formulation mathématique naturelle des hypothèses présentées ci-dessus consiste à dire que y_i et y_j ont d'autant plus de chances d'être égaux que $\mathbf{W}_{ij}$ est grand.

La figure 2.6 illustre une idée sous-jacente à plusieurs de ces algorithmes, qui consiste à propager les étiquettes observées le long des arêtes du graphe. Dans l'exemple de la figure, deux groupes de points sont isolés par une région de l'espace sans aucune donnée : le graphe des plus proches voisins



◀ **Fig. 2.6.** *Apprentissage semi-supervisé, ici en 2D à l'aide d'un graphe des k plus proches voisins (où $k = 2$). L'algorithme propage les étiquettes (0 et 1) par les arêtes du graphe, avec une influence qui décroît avec la distance entre les exemples (l'intensité de la couleur est proportionnelle à la confiance dans l'étiquette prédite).*

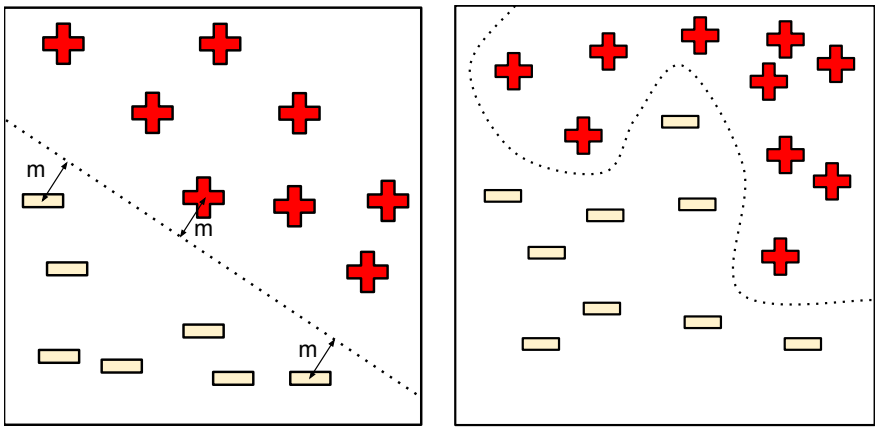
étant scindé en deux sous-graphes déconnectés, cela permet aux étiquettes observées de n'influer que sur leur région respective. Les exemples non étiquetés peuvent être vus ici comme des "relais", permettant de faire circuler l'information sur les étiquettes par les régions de forte densité. Le chapitre 5 étudiera plus en profondeur les propriétés de tels algorithmes.

2.6 Méthodes à noyau

Plusieurs algorithmes mentionnés précédemment utilisent une *fonction noyau* $K(x_i, x_j)$ pour mesurer la similarité entre deux entrées x_i et x_j. Les méthodes dites "à noyau" incluent en particulier une catégorie d'algorithmes basés sur ce que l'on appelle "l'astuce du noyau", qui s'applique à tout algorithme qui peut s'exprimer uniquement à partir de produits scalaires de la forme $x_i^T x_j$. Cette astuce consiste à remplacer $x_i^T x_j$ dans l'algorithme d'origine par une fonction noyau $K(x_i, x_j)$, où K doit satisfaire certaines propriétés mathématiques (on dit que le noyau est défini positif – c'est le cas par exemple du noyau Gaussien que nous avons déjà utilisé). Cela revient à appliquer l'algorithme sur les données transformées implicitement et de manière non linéaire par une fonction ϕ telle que $\phi(x_i)^T \phi(x_j) = K(x_i, x_j)$ (une telle fonction ϕ existe automatiquement si K est défini positif, et en pratique on n'a pas besoin de la calculer explicitement). L'intérêt du noyau est principalement d'effectuer une extraction de caractéristiques non linéaire à partir des entrées, ce qui peut améliorer les performances de l'algorithme.

La plus célèbre méthode à noyau exploitant cette astuce est l'algorithme de la machine à vecteurs de support (SVM, pour "Support Vector Machine" en anglais). Il s'agit d'un algorithme de classification basé sur l'idée de *marge* : pour obtenir une meilleure généralisation, il est préférable de laisser une marge entre les exemples et la surface de décision (Boser et al., 1992; Cortes et al., 1995; Vapnik, 1998). Ce concept de marge est illustré par la figure 2.7 : dans le cas linéaire (c.à.d. sans utiliser l'astuce du noyau) la marge du classifieur est la distance entre l'hyper-plan séparant les deux

classes et les exemples les plus proches. Dans le cas non linéaire, le même principe s'applique dans l'espace des exemples projetés implicitement par ϕ, ce qui se traduit dans l'espace des entrées par une séparation non linéaire des exemples de chaque classe. Depuis les SVMs, l'astuce du noyau et l'idée de marge ont inspiré un grand nombre de nouveaux algorithmes, ainsi que la "noyautisation"* d'algorithmes existants (Schölkopf et al., 2002).



◀ **Fig. 2.7.** *Machine à vecteurs de support, dans le cas linéaire (à gauche) et non linéaire (à droite). La marge est illustrée dans le cas linéaire par la distance m entre la surface de décision (en pointillés) et les exemples les plus proches (elle n'est pas montrée dans le cas non linéaire pour ne pas surcharger la figure).*

Pour la régression, l'astuce du noyau peut directement s'appliquer à la régression linéaire pénalisée ("ridge regression" en anglais, c.f. Hoerl et al., 1970). La prédiction du modèle s'écrivant $f(x) = w^T\phi(x)$, on peut montrer que w peut s'écrire

$$w = \sum_{i=1}^{n} \alpha_i \phi(x_i)$$

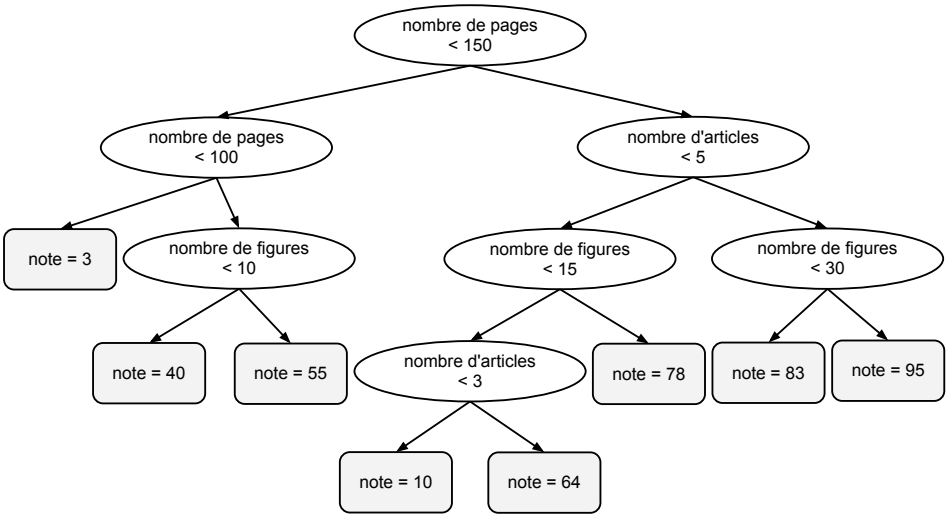
et donc $f(x) = \sum_i \alpha_i K(x, x_i)$ s'exprime en fonction de K uniquement. L'optimisation consiste alors à choisir les α_i de manière à minimiser l'erreur quadratique $\sum_i \|y_i - f(x_i)\|^2$, en rajoutant un terme de régularisation $\lambda\|w\|^2 = \lambda \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$.

Un autre algorithme de régression à noyau est celui des processus Gaussiens ("Gaussian processes" en anglais). Utilisés depuis longtemps en statistiques (O'Hagan et al., 1978), les processus Gaussiens ne sont devenus populaires en apprentissage machine que bien plus tard (Williams et al., 1996; Rasmussen et al., 2006). Dans les processus Gaussiens, le noyau K est interprété comme une sorte de matrice de covariance associée à une distribution Gaussienne dans l'espace des fonctions. L'idée centrale des processus Gaussiens est qu'une valeur élevée de $K(x_i, x_j)$ suppose une grande corrélation entre y_i et y_j. Si K est par exemple notre noyau Gaussien favori, cela implique que lorsque x_i et x_j sont proches, les estimations des distributions $P(Y|x_i)$ et $P(Y|x_j)$ sont similaires.

*On trouve ainsi des versions à noyau de l'analyse en composantes principales, de l'analyse en composantes indépendantes, de l'analyse discriminante, de l'analyse de corrélation canonique, de la régression logistique, de l'algorithme du Perceptron, ...

2.7 Arbres de décision

Les arbres de décision forment une famille d'algorithmes d'apprentissage utilisés pour la classification et la régression (Breiman et al., 1984). Un arbre de décision est un arbre où chaque nœud interne représente un test sur l'entrée x_i. L'exemple typique d'un arbre de décision est un arbre binaire où le test effectué à chaque nœud k est de la forme $x_{ij} < \theta_k$, c.à.d. qu'on compare la j-ème coordonnée de x_i à un seuil θ_k : si elle est plus petite, on continue de parcourir l'arbre en suivant la première branche du nœud, sinon on suit la seconde branche. Lorsqu'on atteint finalement une feuille de l'arbre (un nœud sans enfants), on dit que l'exemple x_i appartient à cette feuille. La figure 2.8 montre un tel arbre de décision.



▶ **Fig. 2.8.** *Arbre de décision typique, où chaque node effectue un test binaire sur l'une des entrées. Cet arbre a été soigneusement construit pour prédire la note d'une thèse de doctorat en fonction de ses attributs principaux : nombre de pages, nombre d'articles et nombre de figures. Si vous utilisez cet arbre, attention à ne pas vous tromper sur l'ordre de parcours des branches !*

L'apprentissage de ce type d'arbre de décision consiste à choisir les variables testées à chaque nœud, les seuils de comparaison, la profondeur de l'arbre, ainsi que la fonction de décision associée à chaque feuille. Il existe plusieurs algorithmes pour cela : leurs détails ne sont pas importants dans le contexte de cette thèse, mais leur principe de base est de faire en sorte que le résultat du test effectué à chaque nœud donne de l'information supplémentaire sur $P(Y|x)$. Idéalement, la distribution $P(Y|x)$ pour un nouvel exemple x doit être bien approximée par la distribution empirique des étiquettes des exemples d'entraînement appartenant à la même feuille que x.

2.8 Réseaux de neurones

Comme leur nom l'indique, les réseaux de neurones sont inspirés de l'architecture du cerveau, étant organisés en couches de neurones connectées

entre elles* (l'équivalent des connexions synaptiques du cerveau). La première couche, appelée la couche d'entrée, est de la même dimension que les entrées x et l'*activation* de son j-ème neurone (c.à.d. la valeur qu'il calcule) est égale à x_{ij} lorsque l'on calcule la prédiction du réseau sur l'exemple x_i. La dernière couche, appelée la couche de sortie, est de la même dimension que l'étiquette dans le cas d'une tâche supervisée. Par exemple, pour la classification, le j-ème neurone de la couche de sortie va calculer $P(Y = j|x_i)$ lorsque x_i est dans la couche d'entrée. Les couches intermédiaires sont appelées les couches cachées. Il existe de nombreuses variations dans les architectures de réseaux de neurones, dont certaines seront présentées dans les sections suivantes. L'architecture la plus connue est celle où il existe une unique couche intermédiaire h qui calcule une transformation non linéaire des entrées, de la forme

$$h(x) = \sigma(W^T x + b)$$

où W est une matrice de *poids*, b est un vecteur de *biais* (de la même taille que le nombre de neurones dans la couche cachée), et σ est une transformation non linéaire élément par élément, dont l'opération sur chaque élément est typiquement la sigmoïde ou la tangente hyperbolique :

$$\begin{aligned} \text{sigmoid}(u) &= \frac{1}{1+e^{-u}} \\ \tanh(u) &= \frac{e^u - e^{-u}}{e^u + e^{-u}}. \end{aligned} \tag{2.1}$$

La fonction donnant la sortie $\hat{y}$ en fonction de h dépend de la tâche ; en classification, on écrit souvent le j-ème neurone de $\hat{y}$, qui estime $P(Y = j|x)$, sous la forme :

$$\frac{e^{V_j^T h + c_j}}{\sum_k e^{V_k^T h + c_k}}$$

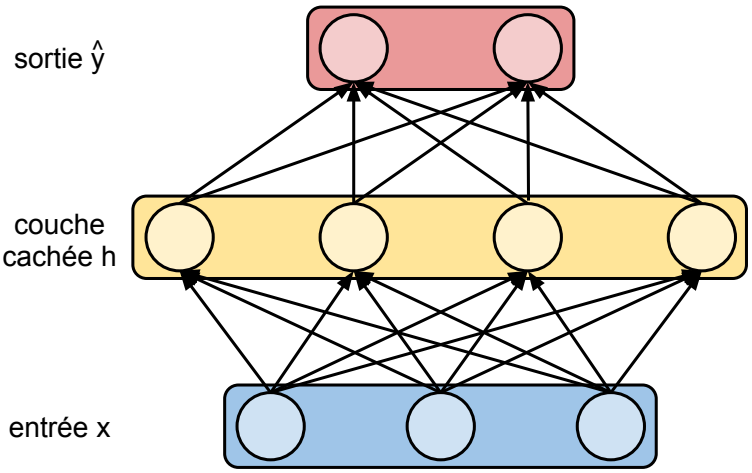
avec V_j la j-ème rangée d'une matrice de poids V, et c_j le biais du j-ème neurone de sortie. La figure 2.9 montre une telle architecture à une couche cachée.

L'apprentissage dans un réseau de neurones consiste à optimiser les poids et biais de façon à minimiser un coût approprié à la tâche. Par exemple, pour la classification, on minimisera la log-vraisemblance négative empirique (éq. 1.2) qui, en notant Θ les paramètres à optimiser, et $f_j(x_i; \Theta)$ la valeur du j-ème neurone de sortie lorsque x_i est en entrée du réseau, se ré-écrit :

$$\hat{C}(\Theta) = -\frac{1}{n} \sum_{i=1}^{n} \ln f_{y_i}(x_i; \Theta).$$

Un autre exemple, en apprentissage non supervisé, est celui des réseaux de neurones *auto-associateurs* dont le but est d'extraire dans la couche cachée

*Nous ne parlerons ici que des réseaux non récurrents, c.à.d. dont le graphe de connectivité ne comporte pas de cycle.



▶ **Fig. 2.9.** *Réseau de neurones à une couche cachée, avec* $x \in \mathbb{R}^3$, $h(x) \in \mathbb{R}^4$, *et* $\hat{y}(h) \in \mathbb{R}^2$. *Une flèche du neurone* i *vers le neurone* j *indique que l'activation de* j *dépend directement de celle de* i.

une représentation qui permet de reconstruire l'entrée x_i (donc l'étiquette y_i est en fait égale à x_i). Dans ce cas, le coût le plus fréquemment utilisé est l'erreur de reconstruction quadratique moyenne

$$\hat{C}(\Theta) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d}(f_j(x_i;\Theta) - x_{ij})^2$$

mais lorsque les entrées $x_{ij} \in \{0,1\}$, on peut également minimiser l'entropie croisée

$$\hat{C}(\Theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d}\left((x_{ij}\ln f_j(x_i;\Theta) + (1-x_{ij})\ln(1-f_j(x_i;\Theta))\right)$$

où le terme dans la somme peut s'interpréter comme la log-vraisemblance des données d'entraînement selon la distribution $P_\Theta(X_{ij} = 1|x_i) = f_j(x_i;\Theta)$.

En pratique, comme indiqué en section 1.3.2, on rajoutera également à $\hat{C}(\Theta)$ un terme qui régularise les paramètres Θ, afin d'obtenir une meilleure généralisation. Les algorithmes d'optimisation les plus souvent utilisés pour minimiser $\hat{C}(\Theta)$ sont des algorithmes d'optimisation locale basés sur l'idée de la *descente de gradient* : le gradient du coût $\hat{C}(\Theta)$ par rapport aux paramètres Θ, noté $\frac{\partial\hat{C}(\Theta)}{\partial\Theta}$, indique la direction dans laquelle le coût augmente le plus lorsque Θ varie. *Descendre* le gradient signifie déplacer les paramètres Θ dans la direction opposée, de manière à diminuer le coût. Le gradient par rapport à tous les paramètres du réseau peut se calculer de manière efficace grâce à l'algorithme de rétro-propagation du gradient (Rumelhart et al., 1986). Pour plus de détails, le livre de Haykin (2008) offre une étude approfondie des réseaux de neurones, tandis qu'une présentation des méthodes classiques d'optimisation (non limitées aux réseaux de neurones) est disponible dans le livre de Nocedal et al. (2006).

2.9 Machines de Boltzmann restreintes

Dans sa version la plus simple, une machine de Boltzmann restreinte ("Restricted Boltzmann Machine" ou RBM en anglais) est un algorithme non supervisé qui modélise la distribution $P(X)$ où $x \in \{0,1\}^d$ comme la marginale d'une distribution jointe

$$P(x,h) = \frac{e^{-\mathcal{E}(x,h)}}{Z} \tag{2.2}$$

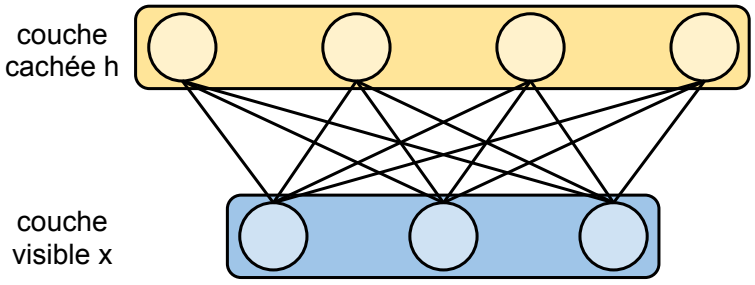
où Z est la *fonction de partition* assurant que cette probabilité est bien normalisée :

$$Z = \sum_{x,h} e^{-\mathcal{E}(x,h)}.$$

Le vecteur $h \in \{0,1\}^k$ représente la *couche cachée*, tandis que le vecteur x est appelée la *couche visible*. Les éléments d'une couche sont généralement appelés *unités* plutôt que neurones. La quantité $\mathcal{E}(x,h)$ est l'*énergie* de la paire (x,h), et l'éq. 2.2 montre que plus l'énergie est faible, plus cette paire est probable. Une forme typique pour $\mathcal{E}$ est :

$$\mathcal{E}(x,h) = -h^T W x - x^T b - h^T c \tag{2.3}$$

où la matrice $W \in \mathbb{R}^{k \times d}$ et les vecteurs $b \in \mathbb{R}^d$ et $c \in \mathbb{R}^k$ sont les paramètres du modèle. Bien que ce modèle ait originellement été introduit sous un autre nom (Smolensky, 1986), il s'agit bien d'une forme particulière de machine de Boltzmann (Hinton et al., 1984, 1986). Comparée à une machine de Boltzmann générique, l'énergie d'une RBM a la propriété que les probabilités conditionnelles $P(x|h)$ et $P(h|x)$ se factorisent, ce qui rend l'entraînement (un peu) plus aisé. Cette propriété se visualise lorsque l'on représente une RBM sous la forme d'un modèle graphique non dirigé (Wainwright et al., 2008), comme dans la figure 2.10 : il y a des connexions entre les unités visibles et les unités cachées, mais aucune connexion de visible à visible, ni de cachée à cachée.



◀ **Fig. 2.10.** *Machine de Boltzmann restreinte, avec $x \in \mathbb{R}^3$ et $h \in \mathbb{R}^4$. Une connexion entre l'unité i et l'unité j indique que la distribution de l'activation de i dépend directement de l'activation de j, et vice-versa.*

Le Roux et al. (2008) ont montré que les RBMs peuvent approximer de manière arbitrairement proche n'importe quelle distribution sur $\{0,1\}^d$

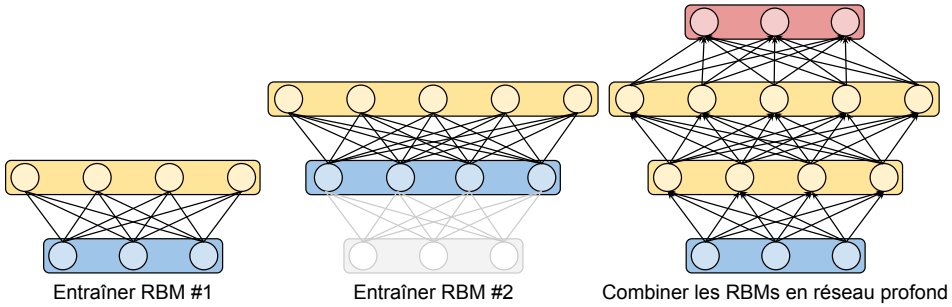
(avec un nombre suffisant k d'unités cachées). Mais l'apprentissage des paramètres d'une RBM à partir des exemples d'entraînement n'en reste pas moins difficile : ceci est dû en particulier au fait que le calcul de la fonction de partition Z n'est pas efficace (car il implique une somme sur un nombre de termes qui augmente exponentiellement avec les dimensions de la couche d'entrée et de la couche cachée). Par conséquent, le gradient de la log-vraisemblance ne peut non plus être calculé efficacement, empêchant la maximisation directe de la log-vraisemblance. L'algorithme de divergence contrastive (Hinton, 2002) introduit une approximation dans le calcul de ce gradient qui le rend bien plus efficace : cette approximation sera étudiée en détails dans le chapitre 7.

2.10 Architectures profondes

Le terme d'*architecture profonde* désigne toute une famille de modèles inspirés des réseaux de neurones, dont le point commun est la composition de transformations successives, permettant de calculer une fonction complexe de l'entrée. Par exemple, un réseau de neurones tel que décrit en section 2.8 peut être considéré comme une architecture profonde si l'on rajoute des couches cachées : chaque couche additionnelle augmente la profondeur du réseau, et déterminer la profondeur idéale en fonction des données fait partie de la problématique des algorithmes d'apprentissage pour architectures profondes. Les réseaux de ce type ont longtemps été ignorés, d'une part parce qu'ils se sont avérés beaucoup plus difficiles à optimiser que les réseaux à une seule couche cachée, d'autre part parce qu'il a été démontré que les réseaux à une seule couche sont des approximateurs universels (Hornik et al., 1989).

L'intérêt pour les réseaux profonds est récemment réapparu après avoir découvert qu'une initialisation non supervisée des poids du réseau peut mener à de bien meilleures performances que l'initialisation aléatoire utilisée jusqu'à présent. Hinton et al. (2006a) ont superposé plusieurs machines de Boltzmann restreintes pour initialiser des réseaux profonds, les rendant capables de généraliser bien mieux que des réseaux à une seule couche cachée sur des tâches de classification (la figure 2.11 montre ce processus pour la superposition de deux RBMs, et il est possible d'en superposer encore plus). Des résultats similaires ont été également obtenus sur d'autres types de tâches et avec d'autres d'architectures profondes (par exemple par Bengio et al., 2007; Hinton et al., 2006b; Vincent et al., 2010; Salakhutdinov et al., 2010). En plus de tenter de comprendre les phénomènes d'optimisation et de régularisation à l'œuvre (Erhan et al., 2010), il est également intéressant d'analyser les architectures profondes sous un autre angle : celui de l'efficacité statistique des architectures profondes, telle que discutée en section 1.5. Bengio (2009) cite en particulier plusieurs résultats suggérant que la profondeur peut aider à représenter de manière beaucoup plus compacte certaines

fonctions. Le chapitre 4 de cette thèse présente de nouveaux résultats dans cette même direction.



◀ **Fig. 2.11.** *Réseau profond construit par superpositions de RBMs : une seconde RBM est entraînée à modéliser la distribution des unités cachées de la première RBM, puis les deux RBMs sont combinées pour initialiser les poids d'un réseau de neurones profond. Ce réseau est ensuite entraîné par descente de gradient à l'aide d'un coût supervisé défini en ajoutant une couche de sortie (en rouge).*

2.11 Méthodes Bayesiennes

Les méthodes Bayesiennes tirent leur nom de la célèbre règle de Bayes :

$$P(\Theta|\mathcal{D}) = \frac{P(\mathcal{D}|\Theta)P(\Theta)}{P(\mathcal{D})} \tag{2.4}$$

qui est ici appliquée avec d'une part les paramètres Θ d'un modèle, et d'autre part les données d'entraînement $\mathcal{D}$ observées. La quantité $P(\mathcal{D}|\Theta)$ est appelée la *vraisemblance* : c'est la probabilité d'observer les données $\mathcal{D}$ en supposant qu'elles ont été générées par notre modèle dont les paramètres sont Θ. $P(\Theta)$ est appelée la distribution a priori : c'est une distribution sur l'espace des paramètres qui reflète notre croyance sur les valeurs possibles des paramètres avant l'observation des données. $P(\Theta|\mathcal{D})$ est alors calculée comme étant proportionnelle au produit de la vraisemblance par la distribution a priori : elle est appelée la distribution *a posteriori*, c.à.d. qu'elle indique la probabilité des paramètres après avoir observé les données (on peut la voir comme une "mise à jour" de notre croyance initiale, grâce aux indices donnés par les données observées). Le terme $P(\mathcal{D})$ est un terme de normalisation qui peut se calculer, si nécessaire, par $P(\mathcal{D}) = \int_\Theta P(\mathcal{D}|\Theta)P(\Theta)d\Theta$. Les méthodes Bayesiennes ne seront mentionnées que superficiellement dans cette thèse, et nous n'entrerons donc pas dans leurs détails ici. Il suffira de garder à l'esprit qu'il s'agit de méthodes probabilistes, qui ont l'avantage en particulier de prendre en compte l'incertitude de manière naturelle : par exemple, la prédiction d'un modèle Bayesien supervisé sur une nouvelle entrée x peut s'écrire comme

$$P(y|x, \mathcal{D}) = \int_\Theta P(y|x, \Theta)P(\Theta|\mathcal{D})d\Theta$$

et la variance de cette distribution peut être interprétée comme l'incertitude sur la prédiction de l'étiquette y. Notons qu'il n'est en général pas trivial de calculer une telle intégrale, et que plusieurs techniques ont été développées spécifiquement dans ce but (Barber, 2011).

2.12 Sélection de modèles

Jusqu'à présent, il n'a été question que de l'apprentissage d'un seul modèle. Mais en pratique, on entraîne plusieurs modèles sur les mêmes données d'entraînement $\mathcal{D}$, et l'on souhaite savoir lequel de ces modèles est le meilleur en terme de généralisation à de nouveaux exemples. Ces modèles peuvent être générés par différents algorithmes d'apprentissage, mais correspondent aussi souvent au même algorithme entraîné avec différentes valeurs d'*hyper-paramètres*. Les hyper-paramètres sont des paramètres de l'algorithme qui influent sur l'entraînement mais ne sont pas modifiés par ce dernier, comme par exemple le nombre de couches cachées d'un réseau de neurones, le nombre de voisins k dans les k plus proches voisins, le coefficient d'un terme de régularisation (comme λ dans l'éq. 1.4), la constante σ du noyau Gaussien dans une méthode à noyau, etc.

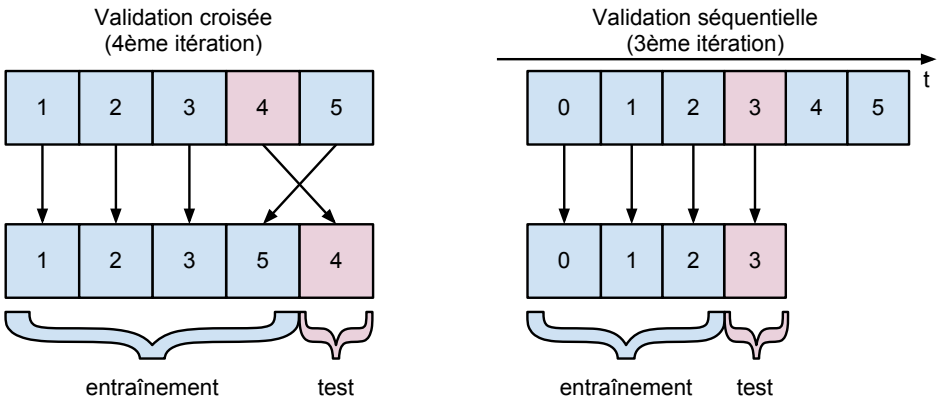
Nous avons déjà utilisé, sans rentrer dans ses détails, une méhode de sélection de modèles dans les exemples du chapitre précédent (figures 1.8 et 1.9). Ces figures montrent la prédiction d'un modèle pour la "meilleure" valeur d'un hyper-paramètre (respectivement σ et λ), où la sélection s'est faite en estimant l'erreur de généralisation sur de nouveaux exemples tirés de la même distribution que ceux de l'ensemble d'entraînement $\mathcal{D}$. C'est la technique la plus basique de sélection de modèles : on sépare les données disponibles en un ensemble d'entraînement $\mathcal{D}$ et un ensemble de test $\mathcal{T}$ (également appelé ensemble de validation*). Le modèle est entraîné pour chaque valeur de ses hyper-paramètres à partir des exemples de $\mathcal{D}$, et l'erreur de généralisation du modèle entraîné est évaluée à l'aide des exemples de $\mathcal{T}$.

La *validation croisée* ("cross-validation" en anglais) est une variante qui répète ce processus afin d'obtenir un estimé plus fiable. Elle consiste à partitionner l'ensemble d'entraînement $\mathcal{D}$ en k sous-ensembles $\mathcal{D}_1, \ldots, \mathcal{D}_k$ (de tailles approximativement égales), et à faire pour chaque $\mathcal{D}_j$:

1. Entraîner le modèle sur $\mathcal{D} \setminus \mathcal{D}_j$.
2. Calculer l'erreur de généralisation du modèle sur $\mathcal{D}_j$.

Le modèle sélectionné comme étant le meilleur est alors celui dont l'erreur calculée au point 2 – moyennée sur tous les $\mathcal{D}_j$ – est la plus faible. La figure 2.12 (à gauche) montre par exemple la définition des ensembles d'entraînement (étape 1) et de test (étape 2) lors de la 4ème itération de validation croisée avec $k = 5$.

*Nous ne rentrerons pas ici dans les subtilités qui justifient la différence de nom selon le contexte.



◀ **Fig. 2.12.** *À gauche : définition des ensembles d'entraînement et de test pour la quatrième itération de validation croisée ($k = 5$). À droite : définition des ensembles d'entraînement et de test pour la troisième itération de validation séquentielle ($k = 5$), sur des données triées chronologiquement selon le temps t.*

Dans le cas où l'ordre des données est important (typiquement lorsqu'elles ont une composante temporelle), la validation croisée n'a pas de sens car on calculerait l'erreur de généralisation sur des exemples qui ont été en fait collectés *avant* certains des exemples de l'ensemble d'entraînement (pour tous les $\mathcal{D}_j$ sauf le dernier, $\mathcal{D}_k$). Afin de respecter cette contrainte temporelle, on peut utiliser la *validation séquentielle*, qui consiste à partitionner $\mathcal{D}$ en $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_k$ tels que tous les exemples de $\mathcal{D}_j$ suivent ceux de $\mathcal{D}_{j-1}$, pour $j \geq 1$. Ensuite, pour chaque $j \geq 1$, on effectue les opérations suivantes :

1. Entraîner le modèle sur $\mathcal{D}_0, \ldots, \mathcal{D}_{j-1}$.
2. Calculer l'erreur de généralisation du modèle sur $\mathcal{D}_j$.

Comme pour la validation croisée, le modèle sélectionné comme étant le meilleur est celui dont l'erreur calculée au point 2 – moyennée sur tous les $\mathcal{D}_j$ – est la plus faible. La figure 2.12 (à droite) illustre ce processus, pour la troisième itération de validation séquentielle avec $k = 5$.

Bibliographie

Barber, D. 2011, *Bayesian Reasoning and Machine Learning*, Cambridge University Press.

Bengio, Y. 2009, "Learning deep architectures for AI", *Foundations and Trends in Machine Learning*, vol. 2, n° 1, p. 1–127. Also published as a book. Now Publishers, 2009.

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information*

Processing Systems 19 (NIPS'06), édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bishop, C. M. 2006, *Pattern Recognition and Machine Learning*, Springer.

Boser, B. E., I. M. Guyon et V. N. Vapnik. 1992, "A training algorithm for optimal margin classifiers", dans *COLT '92 : Proceedings of the fifth annual workshop on Computational learning theory*, ACM, New York, NY, USA, p. 144–152.

Breiman, L., J. H. Friedman, R. A. Olshen et C. J. Stone. 1984, *Classification and Regression Trees*, Wadsworth International Group, Belmont, CA.

Chapelle, O., B. Schölkopf et A. Zien, éd.. 2006, *Semi-Supervised Learning*, MIT Press, Cambridge, MA.

Cortes, C. et V. Vapnik. 1995, "Support vector networks", *Machine Learning*, vol. 20, p. 273–297.

Cover, T. M. et P. E. Hart. 1967, "Nearest neighbor pattern classification", *IEEE Transactions on Information Theory*, vol. 13, n^{o} 1, p. 21–27.

Dempster, A. P., N. M. Laird et D. B. Rubin. 1977, "Maximum-likelihood from incomplete data via the EM algorithm", *Journal of Royal Statistical Society B*, vol. 39, p. 1–38.

Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent et S. Bengio. 2010, "Why does unsupervised pre-training help deep learning ?", *Journal of Machine Learning Research*, vol. 11, p. 625–660.

Haykin, S. 2008, *Neural Networks and Learning Machines (3rd Edition)*, 3^{e} éd., Prentice Hall.

Hinton, G. E. 2002, "Training products of experts by minimizing contrastive divergence", *Neural Computation*, vol. 14, p. 1771–1800.

Hinton, G. E., S. Osindero et Y. Teh. 2006a, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18, p. 1527–1554.

Hinton, G. E. et R. Salakhutdinov. 2006b, "Reducing the dimensionality of data with neural networks", *Science*, vol. 313, n^{o} 5786, p. 504–507.

Hinton, G. E. et T. J. Sejnowski. 1986, "Learning and relearning in Boltzmann machines", dans *Parallel Distributed Processing : Explorations in the Microstructure of Cognition. Volume 1 : Foundations*, édité par D. E. Rumelhart et J. L. McClelland, MIT Press, Cambridge, MA, p. 282–317.

Hinton, G. E., T. J. Sejnowski et D. H. Ackley. 1984, "Boltzmann machines : Constraint satisfaction networks that learn", rapport technique TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science.

Hoerl, A. et R. Kennard. 1970, "Ridge regression : biased estimation for non-orthogonal problems", *Technometrics*, vol. 12, p. 55–67.

Hornik, K., M. Stinchcombe et H. White. 1989, "Multilayer feedforward networks are universal approximators", *Neural Networks*, vol. 2, p. 359–366.

Le Roux, N. et Y. Bengio. 2008, "Representational power of restricted Boltzmann machines and deep belief networks", *Neural Computation*, vol. 20, n° 6, p. 1631–1649.

Ma, Y. et Y. Fu, éd.. 2011, *Manifold Learning Theory and Applications*, Taylor and Francis.

Nadaraya, E. A. 1964, "On estimating regression", *Theory of Probability and its Applications*, vol. 9, p. 141–142.

Nocedal, J. et S. Wright. 2006, *Numerical Optimization*, Springer.

O'Hagan, A. et J. F. C. Kingman. 1978, "Curve fitting and optimal design for prediction", *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 40, n° 1, p. 1–42.

Parzen, E. 1962, "On the estimation of a probability density function and mode", *Annals of Mathematical Statistics*, vol. 33, p. 1064–1076.

Rasmussen, C. E. et C. Williams. 2006, *Gaussian Processes for Machine Learning*, MIT Press.

Rosenblatt, M. 1956, "Remarks on some nonparametric estimates of a density function", *The Annals of Mathematical Statistics*, vol. 27, n° 3, p. 832–837.

Roweis, S. et L. K. Saul. 2000, "Nonlinear dimensionality reduction by locally linear embedding", *Science*, vol. 290, n° 5500, p. 2323–2326.

Rumelhart, D. E., G. E. Hinton et R. J. Williams. 1986, "Learning representations by back-propagating errors", *Nature*, vol. 323, p. 533–536.

Salakhutdinov, R. et H. Larochelle. 2010, "Efficient learning of deep Boltzmann machines", dans *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010), JMLR W&CP*, vol. 9, p. 693–700.

Schölkopf, B. et A. J. Smola. 2002, *Learning with Kernels : Support Vector Machines, Regularization, Optimization and Beyond*, MIT Press, Cambridge, MA.

Shakhnarovich, G., T. Darrell et P. Indyk. 2006, *Nearest-Neighbor Methods in Learning and Vision : Theory and Practice (Neural Information Processing)*, MIT Press.

Smolensky, P. 1986, “Information processing in dynamical systems : Foundations of harmony theory”, dans *Parallel Distributed Processing*, vol. 1, édité par D. E. Rumelhart et J. L. McClelland, chap. 6, MIT Press, Cambridge, p. 194–281.

Tenenbaum, J., V. de Silva et J. C. Langford. 2000, “A global geometric framework for nonlinear dimensionality reduction”, *Science*, vol. 290, n° 5500, p. 2319–2323.

Vapnik, V. 1998, *Statistical Learning Theory*, Wiley, Lecture Notes in Economics and Mathematical Systems, volume 454.

Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio et P.-A. Manzagol. 2010, “Stacked denoising autoencoders : Learning useful representations in a deep network with a local denoising criterion”, *Journal of Machine Learning Research*, vol. 11, n° 3371–3408.

Wainwright, M. J. et M. I. Jordan. 2008, “Graphical models, exponential families, and variational inference”, *Foundations and Trends in Machine Learning*, vol. 1, n° 1-2, p. 1–305.

Watson, G. S. 1964, “Smooth regression analysis”, *Sankhya - The Indian Journal of Statistics*, vol. 26, p. 359–372.

Williams, C. K. I. et C. E. Rasmussen. 1996, “Gaussian processes for regression”, dans *Advances in Neural Information Processing Systems 8 (NIPS’95)*, édité par D. Touretzky, M. Mozer et M. Hasselmo, MIT Press, Cambridge, MA, p. 514–520.

3 Decision trees do not generalize to new variations

Y. Bengio, O. Delalleau et C. Simard
***Computational Intelligence*, vol. 26, n° 4, p. 449–467, 2010**

La malédiction de la dimensionalité introduite en section 1.3.3 indique qu'en général, plus les données occupent un espace en haute dimension, plus l'apprentissage s'avère difficile. Mais chaque algorithme a ses propres particularités, et n'est pas vulnérable aux mêmes difficultés d'apprentissage. Par exemple, dans un article précédant cette thèse (Bengio et al., 2006a) nous avons montré que les méthodes à noyau ont du mal à apprendre des fonctions qui exhibent beaucoup de variations, lorsque le noyau est local – ce qui est par exemple le cas du noyau Gaussien. L'article présenté dans ce chapitre s'inscrit dans la lignée de ces résultats précédents : il s'intéresse cette fois-ci aux limitations des arbres de décision (introduits en section 2.7). Nous y montrons que parce qu'un arbre de décision partitionne l'espace en régions (associées aux feuilles de l'arbre) dans lesquelles l'apprentissage est effectué indépendamment des autres régions, il existe des tâches apparemment simples à apprendre pour lesquelles les arbres de décision ne sont pas du tout efficaces. Il est ici question d'efficacité statistique, c.à.d. du nombre d'exemples nécessaires à l'apprentissage, que nous montrons pouvoir augmenter de manière exponentielle avec la dimension. Ces résultats théoriques permettent de mieux comprendre dans quelle mesure la malédiction de la dimensionalité affecte les arbres de décision, ainsi que d'expliquer pourquoi certaines extensions des arbres de décision peuvent avoir une meilleure capacité de généralisation.

Contribution personnelle Le thème fondateur de l'article (l'idée générale que les arbres de décision sont limités par le fait qu'ils partitionnent l'espace en régions) est à mettre au crédit de Y. Bengio. Je suis intervenu quand est venu le temps de prouver certaines conjectures, et j'ai finalement écrit toutes les preuves non triviales de l'article à l'exception du théorème 3.3.5 (provenant de C. Simard). J'ai également participé aux détails de la définition de la tâche dite du "Checkerboard" afin de pouvoir prouver les résultats souhaités (section 3.3.2). L'introduction et la discussion sont pour l'essentiel dues à Y. Bengio, mais j'ai aidé à formaliser le lien entre arbres et polynômes (section 3.4.3). Au final j'ai rédigé environ 50% de l'article.

Abstract The family of decision tree learning algorithms is among the most widespread and studied. Motivated by the desire to develop learning algorithms that can generalize when learning highly varying functions such as those presumably needed to achieve artificial intelligence, we study some theoretical limitations of decision trees. We demonstrate formally that they can be seriously hurt by the curse of dimensionality in a sense that is a bit different from other non-parametric statistical methods, but most importantly, that they cannot generalize to variations not seen in the training set. This is because a decision tree creates a partition of the input space and needs at least one example in each of the regions associated with a leaf in order to make a sensible prediction in that region. A better understanding of the fundamental reasons for this limitation suggests that one should use *forests* or even deeper architectures instead of trees, which provide a form of distributed representation and can generalize to variations not encountered in the training data.

3.1 Introduction

A long-term goal of machine learning research that remains elusive is to produce methods that will enable artificially intelligent agents. Examples of artificial intelligence (AI) tasks that remain beyond the reach of current algorithms include many tasks involving visual perception, auditory perception, and natural language processing. What we would really like to see are machines that appear to really understand the concepts involved in these tasks. We argue that achieving AI through machine learning involves capturing a good deal of the statistical regularities underlying complex data such as natural text and video. These data objects live in very high-dimensional spaces where the number of possible combinations of values is huge. Yet, we expect these regularities to be representable in a comparatively compact form, simply because they arise from the laws of physics and the organization of our world. Mammals are able to capture a great deal of that structure in a brain that is small in comparison with the set of possible combinations of values of their sensors. Consider as a simple example the variations in pixel intensities one obtains by taking pictures of the same 3D object under different illuminations, in front of different backgrounds, and with different camera geometry, as illustrated in the NORB data set (LeCun et al., 2004). Changing only slightly one of these factors (that we will call *factors of variation* in the remainder of the paper), e.g. rotating the object, gives rise to very different images, when an image is looked at as a vector of pixel intensities, associated with Euclidean distance as a metric. Specifically, Bengio et al. (2006c) illustrate how rotation or translation maps out a manifold in the space of pixel intensities that is highly curved. A function that would be used to really identify an object or estimate density in the space of such data would have to capture most of these variations. If in addition we consider

all the factors that can interact in creating the variations that are observed in natural language text or in video, it becomes clear that modeling such data requires learning *functions with a large number of variations*.

To approach the kind of proficiency that we aim for, it therefore seems plausible to assume that the required learning algorithms will have to learn functions with a large number of variations, which however can be represented compactly (i.e. there exists a small number of factors underlying these variations). These variations correspond to many possible combinations of values for the different factors of variation that underlie the unknown generating process of interest. Note that this large set of variations is not arbitrary. Because these variations arise through complex interactions of real-world factors, these variations in the desired function value are structured: one expects that there exists a reasonably simple* program (such as the one implicitly computed by human brains) that can predict these variations well. Hence, by the theoretical arguments of Kolmogorov complexity (Solomonoff, 1964; Kolmogorov, 1965; Hutter, 2005; Li et al., 2008), one would also expect that some learning algorithms could discover the essential elements of this structure, which would be required to truly generalize in such domains.

A better understanding of the limitations of current algorithms can serve as a guide in moving statistical machine learning towards artificial intelligence. If our goal is to achieve AI through machine learning it is important both to identify the limitations of current learning algorithms with respect to learning highly varying (but structured) functions, and to understand these limitations well enough to work around them.

The study of limitations of particular classes of learning algorithms with respect to learning highly varying functions is not new. This paper is inspired by previous work that has shown such limitations in the case of kernel methods with a *local kernel* (Bengio et al., 2006a) as well as in the case of so-called *shallow function classes* (Bengio et al., 2007b) – which include all fixed-kernel kernel machines such as Support Vector Machines (Boser et al., 1992; Cortes et al., 1995). These papers study in particular the case where the predicted function has the form $f(x) = \sum_i \alpha_i K(x, x_i)$ where x_i are training examples and $K(u, v)$ is a "local" function such as the Gaussian with spread σ. Here, *local* means that a training example x_i has mostly influence on $f(x)$ only for x near x_i. When $\sigma \to 0$ the function is more local and can model more variations (the "bumps" can be distinguished), and when $\sigma \to \infty$ the function becomes quickly very smooth (first a second-order polynomial, then an affine predictor, then a constant predictor). The function can be seen as the addition of local bumps, and it is mostly the training examples in the region around a training example x_i that contribute to the value of the function around x_i. For simple to analyze and simple to express but highly-varying functions such as the parity function (also studied here),

*Simple in the sense that its size is small compared to the number of possible combinations of these underlying factors of variation.

one can show that an exponential number of training examples is necessary to obtain a given level of generalization error.

Here we focus on similar limitations, in the case of decision trees. The theorems are specialized to the most common type of decision trees, which has axis-aligned decision nodes and constant leaves, i.e., where each node partitions the data getting into it according to whether a particular input variable is greater or not than a threshold, and where for the data associated with a leaf, the predicted output is a constant (chosen by learning). However, we believe that similar proof techniques could be extended to wider classes of decision trees. Previous theoretical and empirical studies have already shown that decision trees can be severely limited in their expressive power, unless one allows the number of leaves to be very large (e.g. exponential in the input size). For example Grigoriev et al. (1995) have shown that in order to compute the MAX function (answering whether the j-th input is the maximum over the given n inputs) one would require a tree with a size exponential in n. A related result (which is also closely related to the spirit to this paper) is given by Cucker et al. (1999) and is discussed below. It states a lower bound on the depth of a decision tree when some functions must be approximated with an error less than δ.

Decision trees were introduced by Breiman et al. (1984): a decision tree recursively partitions the input space and assigns an output value for each of the input regions in that partition. Each node of the tree corresponds to a region of the input space and the root is associated with the whole input space. The whole tree corresponds to a piecewise constant function where the pieces are defined by the internal decision nodes, each leaf is associated with one piece, along with a constant to output in the associated region.

In this paper, we study fundamental theoretical limitations of decision trees concerning their inability to *generalize to variations not seen in the training set*. The basic argument is that we need a separate leaf node to properly model each such variation, and at least one training example for each leaf node. Our theoretical analysis is in line with previous empirical results (Pérez et al., 1996; Vilalta et al., 1997) showing that the generalization performance of decision trees degrades when the number of variations in the target function increases. Whereas other non-parametric learning algorithms also suffer from the curse of dimensionality, the way in which the problem appears in the case of decision trees is different and helps to focus on the fundamental difficulty. The general problem is not really dimensionality, nor is it about a predictor that is a sum of purely local terms (like kernel machines). The problem arises from dividing the input space in regions (in a hard way in the case of decision trees) and having separate parameters for each region. Unless the parameters are tied in some way or regularized using strong prior knowledge, the number of available examples thus limits the complexity one can capture, i.e. the number of independent regions that can be distinguished.

3.2 Definitions

An internal node of the tree is associated with a decision function that splits the region (associated with this node) into sub-regions. Each sub-region corresponds to a child of this internal node. Leaf nodes are associated with a function (usually a constant function) that computes the prediction of the tree when the input example falls in the region associated with the leaf. Because the number of possible decision trees is exponential in the size of the tree, the trees are grown greedily, and the size of the tree is selected based on the data, e.g. using cross-validation. A decision tree learning algorithm is thus non-parametric, constructing a more flexible function when more training examples are available. In many implementations (see Breiman et al., 1984) the internal node decision function depends on a single input variable (we call it *axis-aligned*), and for a continuous variable it just splits the space by selecting a threshold value (thus giving rise to a binary tree). It is also possible to use multivariate decision functions, such as a linear classifier (as in Loh et al., 1997), or n-ary splits in the internal nodes.

Definition 3.2.1 (n-ary split function)**.**
Let n be an internal node of a tree and $n_1, \ldots, n_k$ its k child nodes. The n-ary split function S_n of node n is defined on the region R_n of input space $\mathbb{R}^d$ associated with n, and takes values in $\{n_1, \ldots, n_k\}$. It thus defines a region R_{n_i} associated with each child node n_i such that $R_{n_i} = \{x \in R_n | S_n(x) = n_i\}$.

Definition 3.2.2 (Decision Tree)**.**
A decision tree *is the function $T : \mathbb{R}^d \to \mathbb{R}$ resulting from a learning algorithm applied on training data lying in input space $\mathbb{R}^d$, which always has the following form:*

$$T(x) = \sum_{i \in leaves} g_i(x) \mathbf{1}_{x \in R_i} = \sum_{i \in leaves} g_i(x) \prod_{a \in ancestors(i)} \mathbf{1}_{S_a(x) = c_{a,i}} \qquad (3.1)$$

where $R_i \subset \mathbb{R}^d$ is the region associated with leaf i of the tree, $ancestors(i)$ is the set of ancestors of leaf node i, $c_{a,i}$ is the child of node a on the path from a to leaf i, and S_a is the n-ary split function at node a. $g_i(\cdot)$ is the decision function associated with leaf i and is learned only from training examples in R_i. Note that exactly one term of the sum in $T(x)$ can be non-zero (associated with the leaf in which x falls). Learning algorithms for decision trees grow the tree by adding and removing nodes, in such a way that every node has at least one training example falling in it (i.e. no R_i is empty).

Definition 3.2.3 (Piecewise Constant)**.**
We say function $f : R \subseteq \mathbb{R}^d \to \mathbb{R}$ is piecewise constant *if it is of the form $f(x) = \sum_{i=1}^{N} g_i \mathbf{1}_{x \in R_i}$ for some finite N, where $g_i \in \mathbb{R}$ and the R_i's are disjoint subsets of $\mathbb{R}^d$ such that $\cup_{i=1}^{N} R_i = R$.*

Definition 3.2.4 (Piecewise Constant with N Pieces)**.**
We say that function f is piecewise constant with N pieces *if it is piecewise constant and it cannot be represented with less than N pieces.*

Definition 3.2.5 (Constant-Leaves Decision Tree)**.**
A constant-leaves decision tree is a decision tree as in Definition 3.2.2 such that for each leaf node i, the decision function $g_i(\cdot)$ is constant, i.e. $\forall x \in R_i$, $g_i(x) = g_i \in \mathbb{R}$. If it has N leaf nodes, it can thus be written $T_N(x) = \sum_{i=1}^{N} g_i \mathbf{1}_{x \in R_i}$ and it is a piecewise constant function with at most N pieces.

Definition 3.2.6 (Approximation and Error)**.**
We say a function f approximates *a function h with* error ε *if*

$$\sup_x |f(x) - h(x)| \leq \varepsilon.$$

Definition 3.2.7 (ε-variation)**.**
We say that a function h has n ε-variations *if it takes at least n constant pieces for a piecewise constant function to approximate h with an error at most ε over the domain of h.*

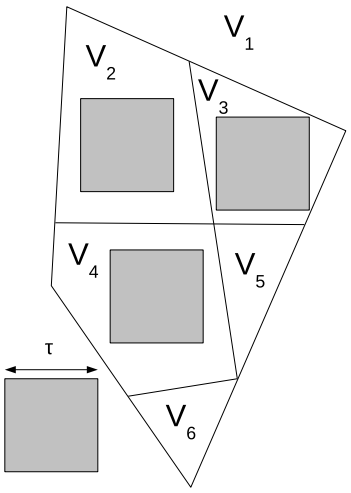
3.3 Inability to generalize to new variations

Cucker et al. (1999) prove a very interesting result on the complexity of function approximation when a round-off error is allowed. We believe that their result is intimately connected to the inability of decision trees to generalize to new variations. Their result, illustrated in figure 3.1, is about trees that define a piecewise-polynomial function T that can approximate another piecewise-polynomial function h. h is associated with regions $V_i \subset \mathbb{R}^d$ that form a partition of the input space $\mathbb{R}^d$, so that h is polynomial in each V_i. The size of these regions is characterized by a quantity $w(\tau)$ that is the number of pieces large enough to contain a d-dimensional cube of side τ. They define a tolerance Γ_τ that depends only on the target function h and the choice of τ, and they prove in their main theorem that if T approximates h with error δ and $\delta < \Gamma_\tau$, then the depth of T cannot be less than $\log_2 w(\tau)$.

The following lemma can be seen as stating for constant-leaves (piecewise constant) decision trees a result that has the same flavor as the above, but in the context of learning and generalization. Instead of characterizing the complexity of the target function by the geometry of regions, we simply count the number of regions N needed to obtain a given accuracy.

Lemma 3.3.1. *Let $\mathcal{F}$ be the set of piecewise constant functions. Consider a target function $h : D \subseteq \mathbb{R}^d \to \mathbb{R}$. For a given representation error level ε, let N be the minimum number of constant pieces required to approximate with a function in $\mathcal{F}$ the target function h with an error less than ε. Then to train a constant-leaves decision tree with error less than ε one requires at least N training examples.*

Proof. This is a direct consequence of the fact that a constant-leaves decision tree with ℓ leaves is piecewise constant with at most ℓ pieces, and each leaf must contain at least one training example. □



◀ **Figure 3.1.** *Example of a partition of $\mathbb{R}^2$ into six regions V_1 to V_6, with the number of regions able to contain a 2-dimensional square of side τ being $w(\tau) = 4$ (V_1 to V_4 can contain such a square, while V_5 and V_6 are too small). A tree approximating a piecewise-polynomial function h defined by these regions must have depth at least $\log_2(w(\tau)) = 2$ when the approximation error is sufficiently small (less than some threshold Γ_τ).*

Since one can easily form an exponential number of distinct regions in $\mathbb{R}^d$ by taking cross-products of one-dimensional partitions, it should now appear clearly that the number of examples required to train a constant-leaves decision tree can grow exponentially with the dimension of the input space $\mathbb{R}^d$.

To illustrate this phenomenon concretely, we prove such exponential growth statements in the special cases of two classes of functions: the parity function and the checkerboard functions (defined below). What is important to note here, is that these functions may otherwise be represented compactly, suggesting that some rather generic learning algorithms could learn them. This is justified by the fact that the Kolmogorov complexity (Kolmogorov, 1965; Li et al., 2008) of these functions could be very low, i.e., one would be able to express them with a small program in any current programming language. Functions with low Kolmogorov complexity can theoretically be learned with few examples (Solomonoff, 1964; Kolmogorov, 1965; Hutter, 2005; Li et al., 2008), but we show in this section that decision trees are unable to do so, regardless of the learning algorithm being used. Keep in mind that although the following classes of functions may be easy to represent compactly in some standard programming language, it does not necessarily mean it is easy to *learn* this representation, because one needs an efficient way to search in the space of programs. The results in this section do not tell us how to solve this computational complexity issue, but by highlighting some fundamental limitations of decision trees, they also give some ideas as to how one may get around them: this will be discussed in section 3.4.

3.3.1 Curse of dimensionality on the parity task

It was already known from empirical evidence that decision trees were not able to learn the parity function, in the sense of not generalizing to regions of the input space that do not correspond to a training example (see e.g. Pérez et al., 1996). The mathematical results in this section show this formally, connecting the number of training examples, input dimension, and generalization error.

Definition 3.3.2 (d-bit Parity Task)**.** *The* d-bit parity task *has as target function the* d*-bit parity function* $p : \{0,1\}^d \subset \mathbb{R}^d \to \{-1,1\}$*,*

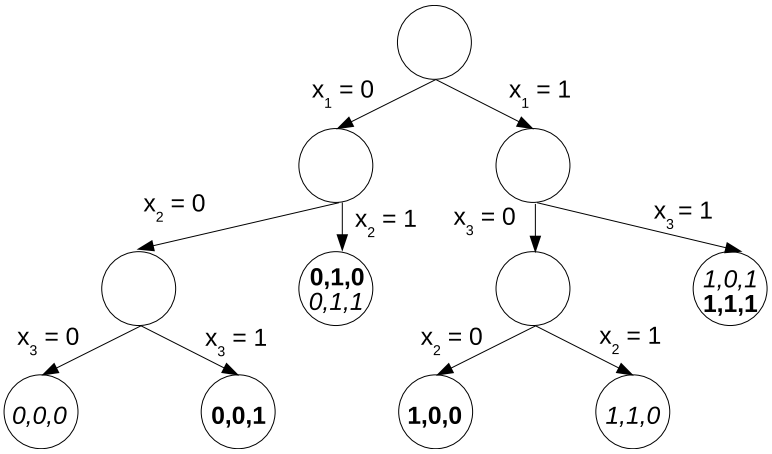
$$p(x) = \begin{cases} -1 & if \quad \sum_{i=1}^{d} x_i \quad is \quad even \\ 1 & if \quad \sum_{i=1}^{d} x_i \quad is \quad odd \end{cases}$$

where x_i *is the* i*-th element of vector* x*. It has a total of* 2^d *possible training examples, sampled with equal probabilities. We thus define the number of generalization errors of a predictor* g *as* $|\{x \in \{0,1\}^d : g(x) \neq p(x)\}|$ *and its generalization error (or error rate) as* $|\{x \in \{0,1\}^d : g(x) \neq p(x)\}|/2^d$*. Similarly we can talk of the generalization error of a node of a decision tree as the average error among the examples falling into that node.*

Definition 3.3.3 (Constant-leaves decision tree with axis-aligned decision nodes)**.** *A* constant-leaves decision tree with axis-aligned decision nodes *is a constant-leaves decision tree whose decision split at internal node* i *is of the form* $S_i(x) = \mathbf{1}_{x_j < \alpha_i}$*, i.e. it splits the current region in two, based on the comparison between the* j*-th coordinate and a single threshold* α_i*.*

Lemma 3.3.4. *As illustrated in figure 3.2, on the task of learning the* d*-bit parity function, a constant-leaves decision tree with axis-aligned decision nodes and output in* $\{-1,1\}$ *will have a generalization error of* $\frac{1}{2}$ *on leaf nodes of depth less than* d*.*

Proof. Let us prove the result by induction on $d \geq 1$, for both the tasks of learning the parity function and its opposite $-p(\cdot)$. For $d = 1$, it is obvious since the only leaf node of depth less than d can be the root node, which contains all 2 possible examples. Let us suppose the result is true for $d = k \geq 1$, and let us consider the case $d = k + 1$. Since no node can be empty, the split function at the root node r must have a threshold $\alpha_r \in (0,1)$. Without loss of generality, suppose the split is performed on the first input coordinate. The two sub-regions thus defined are $R = \{x \in \{0,1\}^{k+1} | x_1 = 0\}$ and $R' = \{x \in \{0,1\}^{k+1} | x_1 = 1\}$. Since the first coordinate is constant in both R and R', the corresponding subtrees cannot perform additional splitting w.r.t. this coordinate (as this would result in empty nodes), and x_1 can be ignored. If the original task was to learn the parity task this implies the subtree trained on R tries to learn the parity task in dimension k, while the subtree trained on R' tries to learn the opposite of the parity task in



◀ **Figure 3.2.** *Illustration of lemma 3.3.4 on a tree with axis-aligned decision nodes in* $\{0,1\}^3$*. Data points are listed inside the leaf node they fall into, in* italics *for points whose parity output would be -1, and in* **bold** *for those whose parity output would be 1. The two leaf nodes with depth less than 3 must have a classification error of* $\frac{1}{2}$*, since they contain one of each kind.*

dimension k (because the target is switched on R', due to $x_1 = 1$). From the induction hypothesis, all leaf nodes of depth less than k in these subtrees (i.e. of depth less than $k+1 = d$ in the full tree) have a generalization error of $\frac{1}{2}$. If the original task was to learn the opposite of the parity task, the same reasoning applies (switching the roles of R and R'). □

Theorem 3.3.5. *On the d-bit parity task, a constant-leaves decision tree with axis-aligned decision nodes trained on n different examples has a generalization error in* $\left[\frac{1}{2} - \frac{n}{2^{d+1}}, 1 - \frac{n}{2^d}\right]$.

Proof. Let k the number of leaf nodes of depth d in a tree trained on n different examples. We will first show that the generalization error is $\varepsilon = \frac{1}{2} - \frac{k}{2^{d+1}}$. On a leaf of depth d, there can be only one training example (because every ancestor of the leaf splits on a different input and divides the input space in 2, and there are only 2^d possible examples). Hence training and generalization error on the k depth d leaves is 0. On the other hand, lemma 3.3.4 shows that the generalization error is $1/2$ on the other leaves. Since there are k examples falling in depth d leaves, and $2^d - k$ examples falling in the others, the total number of generalization errors is $\frac{1}{2}(2^d - k)$ and the error rate is $\varepsilon = \frac{1}{2} - \frac{k}{2^{d+1}}$.

To prove our theorem we now have to find lower and upper bounds for k. Clearly, $k \leq n$, otherwise we could have more leaf nodes of depth d than examples. This proves the first inequality: $\varepsilon \geq \frac{1}{2} - \frac{n}{2^{d+1}}$. Moreover, the worst we can do – in terms of generalization error – before inserting a first leaf node of depth d is to create all the leaf nodes of depth $d-1$ and this requires at least 2^{d-1} examples. Then each additional example will lead to a node of depth $d-1$ being split, creating two nodes of depth d, so that $k \geq 2(n - 2^{d-1})$. Hence $\varepsilon \leq \frac{1}{2} - \frac{2(n-2^{d-1})}{2^{d+1}} = \frac{1}{2} - \frac{2n}{2^{d+1}} + \frac{2^d}{2^{d+1}} = 1 - \frac{n}{2^d}$. □

Corollary 3.3.6. *On the task of learning the d-bit parity function, a constant-leaves decision tree with axis-aligned decision nodes will require at least*

$2^d(1-2\varepsilon)$ *examples in order to achieve a generalization error less than or equal to* ε*.*

Proof. Let $n(\varepsilon)$ be the number of examples to get a generalization error ε. From the lower bound of theorem 3.3.5 we find $n(\varepsilon) \geq 2^d(1-2\varepsilon)$. □

3.3.2 Curse of dimensionality for the checkerboard task

The parity function may look maybe too simple. Can we generalize some of its properties? The checkerboard task defined below is similar in spirit to the parity function, in the sense that it defines a large number of regions such that the target output in each region is different from the target output in neighboring regions, but in such a way that the overall function can be written down with an expression much smaller than the total number of regions, i.e., a learning algorithm approximately minimizing Kolmogorov complexity should be able to discover a good solution without requiring an exponential number of examples.

Definition 3.3.7 (Checkerboard Task)**.** *A Checkerboard task over* $[0,1)^d$ *with* minimum variation δ, minimum mass m per board cell *and interval numbers* $\{n_i\}_{i=1}^d$ *defines:*

- *for each dimension* $i \leq d$*, a partition of* $[0,1)$ *into* n_i *intervals of the form* $\{[\alpha_{i,j}, \alpha_{i,j+1})\}_{j=1}^{n_i}$ *with* $\alpha_{i,1} = 0$*,* $\alpha_{i,n_i+1} = 1$*, and* $\alpha_{i,j} < \alpha_{i,j+1}$
- *a target function* h *constant over each* cell $C_{j_1,\ldots,j_d} = [\alpha_{1,j_1}, \alpha_{1,j_1+1}) \times \ldots \times [\alpha_{d,j_d}, \alpha_{d,j_d+1})$*, such that the constant values of* h *on two neighborhing cells differ at least by* δ*.*

To form a data set, the inputs x *are sampled with a probability distribution* p *such that* $P(x \in C_{j_1,j_2,\ldots,j_d}) = \int_{x \in C_{j_1,j_2,\ldots,j_d}} p(x)dx \geq m$ *for every cell* $C_{j_1,j_2,\ldots,j_d}$ *and* p *is uniform within each cell. The generalization error of a predictor* f *on a checkerboard task is measured as the average squared error* $E = \int (f(x) - h(x))^2 p(x)dx$*.*

Proposition 3.3.8. *To obtain an average generalization error less than* $\frac{m\delta^2}{2}$ *on a checkerboard task over* $[0,1)^d$ *with minimum mass* m *per cell, minimum variation* δ *and interval numbers* $\{n_i\}_{i=1}^d$*, using a constant-leaves decision tree with axis-aligned decision nodes, the tree must be trained with at least* $N = \prod_{i=1}^d n_i$ *different examples.*

Before going into the detailed proof, we first give an intuitive explanation for this result. The idea is that if a tree is trained with less than N different examples, then its output must be constant on two neighboring cells. Because the target values in these cells differ by at least δ, and each cell has a probability mass at least m, then the squared error is related to $m\delta^2$. This reasoning is detailed in the last paragraph of the following proof

(explaining where the $\frac{1}{2}$ factor comes from), while most of this proof is dedicated to showing first that we may consider only trees using cell boundaries as splitting thresholds.

Proof. We will prove that a tree achieving a generalization error less than $\frac{m\delta^2}{2}$ must have at least N leaf nodes, which by Definition 3.2.2 of a decision tree implies it has been trained with at least N examples (since there must be at least one training example falling in each leaf).

We first prove that we can restrict ourselves to decision trees whose splitting functions at each node are of the form

$$S(x) = \mathbf{1}_{x_i < \alpha_{i,j}} \tag{3.2}$$

i.e. whose splitting thresholds on dimension i are constrained to be among the interval boundaries $\{\alpha_{i,j}\}_{j=1}^{n_i}$. To show this, let us consider any constant-leaves decision tree with axis-aligned decision nodes, and let us show its thresholds can be modified to verify constraint (3.2) without adding nodes nor increasing its generalization error on the checkerboard task. Let S be the splitting function of the highest depth node that does not verify (3.2), i.e. is of the form:

$$S(x) = \mathbf{1}_{x_i < \gamma} \tag{3.3}$$

where $\gamma \in (0, 1)$ (otherwise some node would contain no example) and such that $\forall j = 1, \ldots, n_i + 1$ we have $\gamma \neq \alpha_{i,j}$. Since $\alpha_{i,1} = 0$, $\alpha_{i,n_i+1} = 1$ and the $\alpha_{i,j}$ are increasing with j, there must exist $j \in \{1, \ldots, n_i\}$ such that $\gamma \in (\alpha_{i,j}, \alpha_{i,j+1})$. Now consider, among all nodes in the path from the root of the tree to the parent of the node we are considering, those that also make a split based on the same variable x_i, and define $\mathcal{T}$ the set of all their split thresholds. We will focus on the interval defined by the following two real numbers:

$$\begin{aligned} \lambda &= \max(\alpha_{i,j}, \max(\beta \in \mathcal{T} | \beta < \gamma)) \\ \mu &= \min(\alpha_{i,j+1}, \min(\beta \in \mathcal{T} | \beta > \gamma)) \end{aligned}$$

where we take the minimum of an empty set to be $+\infty$ and its maximum to be $-\infty$ (note also that $\gamma \notin \mathcal{T}$ because a tree does not split twice on the same variable with the same threshold in the same branch, otherwise one would get 0 training examples in a node). From their definition, λ and μ verify the following inequality:

$$\alpha_{i,j} \leq \lambda < \gamma < \mu \leq \alpha_{i,j+1}. \tag{3.4}$$

Moreover, for fixed $x_j, j \neq i$, when x_i varies in $[\lambda, \gamma)$ or in $[\gamma, \mu)$ the output of the decision tree does not change, since there exists no split w.r.t. coordinate x_i with a threshold in the interior of these intervals (remember that the node we are considering is the deepest one not verifying (3.2), and thus all its child nodes that may split w.r.t. x_i have a threshold in $(0, \alpha_{i,j}]$ or $[\alpha_{i,j+1}, 1)$).

Let f be the output function of the tree, and E_0 its average error on $D_0 = \{x \in [0,1)^d | x_i \in [\lambda, \gamma)\}$:

$$\begin{aligned} E_0 &= \frac{1}{\int_{x \in D_0} p(x)dx} \int_{x \in D_0} (f(x) - h(x))^2 p(x)dx \\ &\stackrel{def}{=} \frac{1}{p_0} J(D_0) \end{aligned}$$

Similarly, define $E_1 = \frac{1}{p_1} J(D_1)$ its average error on $D_1 = \{x \in [0,1)^d | x_i \in [\gamma, \mu)\}$. Since D_0 and D_1 are disjoint, the overall generalization error of f can be written

$$\begin{aligned} E &= \int_{x \in [0,1)^d} (f(x) - h(x))^2 p(x)dx \\ &= J(D_0) + J(D_1) + J([0,1)^d \setminus (D_0 \cup D_1)) \\ &= p_0 E_0 + p_1 E_1 + J([0,1)^d \setminus (D_0 \cup D_1)). \end{aligned} \tag{3.5}$$

Let us first consider the case $E_0 \leq E_1$. Let f' the output function that would result from replacing threshold γ in (3.3) by μ, and $D' = \{x \in [0,1)^d | x_i \in [\lambda, \mu)\}$ (note that $D' = D_0 \cup D_1$). Denoting by $J'(A)$ the error $\int_{x \in A} (h'(x) - h(x))^2 p(x)dx$, the generalization error E' of f' can be written:

$$E' = J'(D_0) + J'(D_1) + J'([0,1)^d \setminus (D_0 \cup D_1)). \tag{3.6}$$

For $x \in [0,1)^d \setminus D_1$, we have $f'(x) = f(x)$ since for such a x, $x_i < \gamma \Leftrightarrow x_i < \mu$ and $x_i \geq \gamma \Leftrightarrow x_i \geq \mu$. Thus, using (3.5) and (3.6), the difference between generalization errors E and E' reduces to

$$E - E' = p_1 E_1 - J'(D_1). \tag{3.7}$$

Defining $\tilde{x} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d) \in [0,1)^{d-1}$, $J'(D_1)$ can be written

$$\begin{aligned} J'(D_1) &= \int_{x \in D_1} (h'(x) - h(x))^2 p(x)dx \\ &= \int_{\tilde{x} \in [0,1)^{d-1}} \left(\int_{x_i \in [\gamma, \mu)} (h'(x) - h(x))^2 p(x)dx_i \right) d\tilde{x}. \end{aligned} \tag{3.8}$$

In order to simplify (3.8) we observe that, for a fixed $\tilde{x}$, $f'(x)$ is constant w.r.t. $x_i \in [\gamma, \mu)$ because there is no node in the tree performing a split on x_i with a threshold within this interval. Let $y \in [0,1)^d$ the point defined by $y_j = x_j$ for all $j \neq i$, and $y_i = \lambda$. Then $f'(x) = f(y)$, because:

- on the path from the root to the parent of the node we are considering, all splits w.r.t. the i-th coordinate return the same result for all values in $[\lambda, \mu)$ (this is a direct consequence of the definition of λ and μ),
- the split for the node we are considering returns 0 when evaluating $f'(x)$ because $x_i < \mu$, and 0 when evaluating $f(y)$ because $y_i < \gamma$,

- the splits on the i-th coordinate for its child nodes can only involve thresholds equal to some $\alpha_{i,j}$, and thus return the same results for all values in $[\lambda, \mu)$ due to (3.4).

Similarly, we can compute $J(D_0)$ by

$$\begin{aligned} J(D_0) &= \int_{x \in D_0} (f(x) - h(x))^2 p(x) dx \\ &= \int_{\tilde{x} \in [0,1)^{d-1}} \left(\int_{x_i \in [\lambda, \gamma)} (f(x) - h(x))^2 p(x) dx_i \right) d\tilde{x}. \end{aligned} \tag{3.9}$$

For the same fixed $\tilde{x}$ as above, $f(x)$ is constant w.r.t. $x_i \in [\lambda, \gamma)$ since there is no node in the tree performing a split on x_i with a threshold within this interval. Thus $f(x) = f(y)$ (with y defined as above, i.e. with its i-th coordinate set to λ). Finally, we observe that for a fixed $\tilde{x}$, both $p(x)$ and $h(x)$ are also constant w.r.t. $x_i \in [\lambda, \mu)$. Indeed, let C be the cell containing x when $x_i = \lambda \geq \alpha_{i,j}$: as x_i increases, x stays in the same cell as long as $x_i < \alpha_{i,j+1}$, which is true since $\mu < \alpha_{i,j+1}$. The probability distribution being uniform within cell C, $p(x)$ is thus constant and we denote it by $\tilde{p}(\tilde{x})$. Moreover, $h(x)$ is also constant since x stays in the same cell. We denote its value by $\tilde{h}(\tilde{x})$. All these observations allow us to rewrite (3.8) and (3.9) into

$$\begin{aligned} J'(D_1) &= \int_{\tilde{x} \in [0,1)^{d-1}} (\mu - \gamma)(f(y) - \tilde{h}(\tilde{x}))^2 \tilde{p}(\tilde{x}) d\tilde{x} \\ J(D_0) &= \int_{\tilde{x} \in [0,1)^{d-1}} (\gamma - \lambda)(f(y) - \tilde{h}(\tilde{x}))^2 \tilde{p}(\tilde{x}) d\tilde{x} \end{aligned}$$

and consequently

$$J'(D_1) = \frac{\mu - \gamma}{\gamma - \lambda} J(D_0) = \frac{\mu - \gamma}{\gamma - \lambda} p_0 E_0. \tag{3.10}$$

To conclude, we need to express the above ratio in terms of p_0 and p_1. Using the same notations:

$$\begin{aligned} p_0 &= \int_{\tilde{x} \in [0,1)^{d-1}} \left(\int_{x_i \in [\lambda, \gamma)} \tilde{p}(\tilde{x}) dx_i \right) d\tilde{x} = \int_{\tilde{x} \in [0,1)^{d-1}} (\gamma - \lambda) \tilde{p}(\tilde{x}) d\tilde{x} \\ p_1 &= \int_{\tilde{x} \in [0,1)^{d-1}} \left(\int_{x_i \in [\gamma, \mu)} \tilde{p}(\tilde{x}) dx_i \right) d\tilde{x} = \int_{\tilde{x} \in [0,1)^{d-1}} (\mu - \gamma) \tilde{p}(\tilde{x}) d\tilde{x} \end{aligned}$$

which shows that $\frac{\mu - \gamma}{\gamma - \lambda} = \frac{p_1}{p_0}$, and thus, using (3.7) and (3.10):

$$E - E' = p_1 E_1 - \frac{p_1}{p_0} p_0 E_0 = p_1 (E_1 - E_0) \geq 0$$

since we are in the situation where $E_0 \leq E_1$. This means the new tree f' does not degrade the generalization error compared to f. In the case where

$E_0 > E_1$, the same reasoning can be applied when replacing threshold γ with λ instead of μ, in order to obtain a tree f' with a similar (or lower) generalization error. After this step, one of the two following situations can occur:

- Either the new threshold is equal to a threshold of a parent node splitting on the same coordinate. In this case, it is useless and the current node can be deleted (along with one of its subtrees), leading to a smaller tree. The above procedure can then be iterated.
- Or the new threshold is equal to $\alpha_{i,j}$ or $\alpha_{i,j+1}$, and the above procedure can also be iterated (note that if this threshold is equal to 0 or 1, then the tree can also be pruned).

Since at each step we either remove a node or set its threshold to an interval boundary $\alpha_{i,j}$, in the end we obtain a tree that (i) does not contain more nodes than f, (ii) does not have a higher generalization error than f, and (iii) has only thresholds among interval boundaries $\alpha_{i,j}$.

We can now study the case where (3.2) is verified at each node. A direct consequence is that the output function f is constant on all cells of the target function h. Let M the number of pieces in the piecewise constant function f (i.e. the number of leaf nodes in the tree). If $M < N$ pieces, there must be two neighbor cells C and C' on which f assigns the same value t, and on which function h takes values respectively c and c'. The generalization error E of f is then at least $E_{C \cup C'}$, with

$$\begin{aligned} E_{C\cup C'} &= \int_{x\in C\cup C'} (f(x) - h(x))^2 p(x) dx \\ &= \int_{x\in C} (t-c)^2 p(x) dx + \int_{x\in C'} (t-c')^2 p(x) dx \\ &\geq (t-c)^2 m + (t-c')^2 m \end{aligned}$$

with m the minimum mass per cell of the checkerboard task. This quadratic function of t is minimized for $t = \frac{c+c'}{2}$ and is equal to $\frac{m}{2}(c-c')^2$ for this value of t. Since from Definition 3.3.7 we have $|c - c'| \geq \delta$, we can thus conclude that $E \geq \frac{m\delta^2}{2}$. □

3.4 Discussions

Decision trees have been used with great success and have generated an important literature in the statistics, machine learning, and data-mining communities. Even though our results suggest that they are insufficient to learn the type of task involved in AI (with a number of *ε-variations* much greater than the number of examples one could hope to get), they might still be used as useful components. The above results should also help us

gather a better understanding of the limitations of non-parametric learning algorithms by illustrating the differences and common pitfalls of decision trees and other non-parametric learning algorithms.

3.4.1 Trees vs local non-parametric models

Local non-parametric learning methods such as Gaussian Support Vector Machines (SVMs) and nearest-neighbor classifiers or Parzen windows are hurt by the curse of dimensionality because they associate a separate set of parameters to each region, where each region is a kind of blob centered on a training example (like the radially defined Gaussians in Gaussian kernel machines). Instead, decision trees define regions which can extend for arbitrary distances away from a training example. According to the definition of *local* given by Bengio et al. (2006a), decision trees are non-local: they can generalize to a test point arbitrarily far from a training point because they can ignore some dimensions (and do so differently in different parts of the space), and this is true not just for test points that are far from the cloud of training points. Hence the sense in which they are cursed by dimensionality is a bit different from local non-parametric methods such as nearest-neighbor methods, kernel density estimation, or Support Vector Machines. However, we believe that there is a way to view these two effects under a common light, by thinking about the notions (that we have tried to highlight in this paper) of *variations* and *regions*. Both types of methods construct some kind of soft or hard partition of the input space, and consider a simple parametrization inside each region of this partition: constant model in the case of constant-leaves decision trees and nearest-neighbor classifiers or histograms, and something more powerful (close to a low-degree polynomial) in the case of Gaussian SVMs. What hurts generalization in both cases is the need to have examples in each of these regions in order to be able to generalize. What is missing is the ability to learn something about the statistical structure in some region of space that could be somehow applied in other regions of space, besides the immediately neighboring regions. See (Bengio et al., 2006b,c) for discussions of non-local generalization and attempts to transform local kernel methods so as to achieve it.

So in spite of the degree of non-locality which may confer some advantages to decision trees over other non-parametric learning algorithms, decision trees suffer from a very similar limitation arising not so much because of the dimensionality of the input but because of the degree of variability of the target function to be learned.

3.4.2 Forests and boosted trees

Forests, i.e. sums of trees – random forests (Ho, 1995; Breiman, 2001), error-correcting committees of trees (Kong et al., 1995), and boosted trees (Freund et al., 1996) – are known to perform generally better than decision trees. Many empirical results support this statement, and several explanations

have already been proposed, such as variance reduction in forests and margin bounds for boosting. Boosted trees and other forests have the form $f(x) = \sum_{i=1}^{n} \alpha_i T_i(x)$ where $T_i(x)$ is the prediction of the i-th tree. It has been reported often that boosted trees and random forests generalized better than single trees. The theoretical results presented here suggest yet another reason for the better performance of forests and boosted trees over single trees. Indeed, our negative theorems do not apply to sums of trees. In fact we have that:

Proposition 3.4.1. *Let $f(x) = \sum_{i=1}^{n} \alpha_i T_i(x)$ be a forest of constant-leaves decision tree with axis-aligned decision nodes defined on an input space of dimension d, with $n \leq d$. Then the number of different values f can take can grow exponentially with n, even in situations where the number of different values each T_i can take is bounded by a fixed constant.*

Proof. Let $T_i(x) = \mathbf{1}_{x_i < \frac{1}{2}}$ and $\alpha_i = 2^{i-1}$. For any $k \in \{0, 1, 2, \ldots, 2^n - 1\}$ there exists $x \in \mathbb{R}^d$ such that $f(x) = k$: since $n \leq d$ we can simply use the binary representation $b = b_{n-1} \ldots b_1 b_0$ of k, and set $x_i = b_{i-1}$ for $i \leq n$ (and for instance $x_i = 0$ for $i > n$). □

This proof suggests a different way to combine trees. If we consider the output $T(x)$ of a tree to be a discrete variable specifying in which leaf x falls, then we can consider the output of a forest as the encoding of a vector whose elements are these discrete variables, one per tree in the forest. Clearly, this is a form of distributed representation (Hinton, 1986), which can express a number of configurations possibly exponential in the number of trees (even though the model is expressed with a set of numbers of size linear in the number of trees). This expressive power (a small set of numbers saying things about a large set of distinct regions in input space) is also what could buy strong generalization power (for the same reason that a model with a smaller Kolmogorov complexity explaining correctly a much larger data set is likely to generalize well). Note how in error-correcting output coding with one tree for each output bit (Kong et al., 1995), we have a fixed distributed representation (the output code). The work developed by Hinton over the last two decades (e.g. see Hinton, 1986; Hinton et al., 1997; Paccanaro et al., 2000; Hinton et al., 2006) is instead geared towards *learning* internal distributed representations that help capture the main factors of variation in the data.

3.4.3 Architectural depth and distributed representations

Learning algorithms that learn to represent functions with many levels of composition are said to have a *deep architecture* (we are talking here about *architectural depth*, different from tree depth). Bengio et al. (2007b) discuss results in computational theory of circuits that strongly suggest that, compared to their shallow counterparts, deep architectures are much more

efficient in terms of representation, i.e., can require a smaller number of computational elements or of parameters to approximate a target function. In spite of the fact that 2-level architectures (e.g., a one-hidden layer neural network, a kernel machine, or a 2-level digital circuit) are able to represent any function (see for example Hornik et al., 1989), they may need a huge number of elements and, consequently, of training examples. Generally, a function that can be represented efficiently with a circuit of depth k may require an exponentially larger circuit of depth $k-1$. For example, the parity function on d bits can be implemented by a digital circuit of *architectural depth* $\log(d)$ with $O(d)$ elements but requires $O(2^d)$ elements to be represented by a 2-level digital circuit (Ajtai, 1983), e.g., in conjunctive or disjunctive normal form. A similar result was proved for Gaussian kernel machines: they require $O(2^d)$ non-zero coefficients (i.e., support vectors in a Support Vector Machine) to represent such a highly varying function (Bengio et al., 2006a). Note however that parity can be represented efficiently with 2 or 3 levels if the units at each level are slightly more powerful, e.g., with an RBF network that has different spreads in each unit, or with a multi-layer neural network with two hidden layers, so it may not be the best illustration of what requires deeper architectures. Another example, discussed by Bengio et al. (2007b), is that of multiplication of n-bit integers using digital circuits. It can either be achieved with a two-layer architecture that has a number of gates exponential in n, or efficiently with a deep circuit of $O(\log n)$ layers.

What is the architectural depth of decision trees and decision forests? It depends on what elementary units of computation are allowed on each level. By analogy with the disjunctive normal form (which is usually assigned an architectural depth of two) one would assign an architectural depth of two to a decision tree, and of three to decision forests or boosted trees. The top level disjunction computed by a decision tree is a sum over the terms associated with each leaf. A first level conjunctive term is a product of the indicator functions associated with each internal node and with the predicted constant associated with the leaf. With this interpretation, a decision forest has an architectural depth of three. An extra summation layer is added. Note how this summation layer is very different from the top layer of the decision tree architecture. Although both perform a summation, the decision tree top layer sums over mutually exclusive terms, whereas the decision forest sums over terms which are generally non-zero, allowing an exponential number of combinations of leaves (one from each tree) to be added, as discussed above.

It is interesting to pursue the analogy between polynomials and decision trees, in the context of shallow versus deep architectures. For the sake of simplicity, we consider the case of binary inputs $x = (x_1, \ldots, x_d) \in \{0,1\}^d$, and binary functions (i.e. the output is in $\{0,1\}$ as well). We will see how a decision tree corresponds to some kind of (shallow) expansion of a polynomial, while there may exist a kind of (deep) factorization allowing a more compact representation of the same function. In the context of binary

inputs and output, eq. 3.1 becomes of the form

$$T(x) = \sum_{(y,\alpha)\in L} \Pi_{i=1}^{d}(y_i x_i + (1-y_i)(1-x_i))^{\alpha_i} \tag{3.11}$$

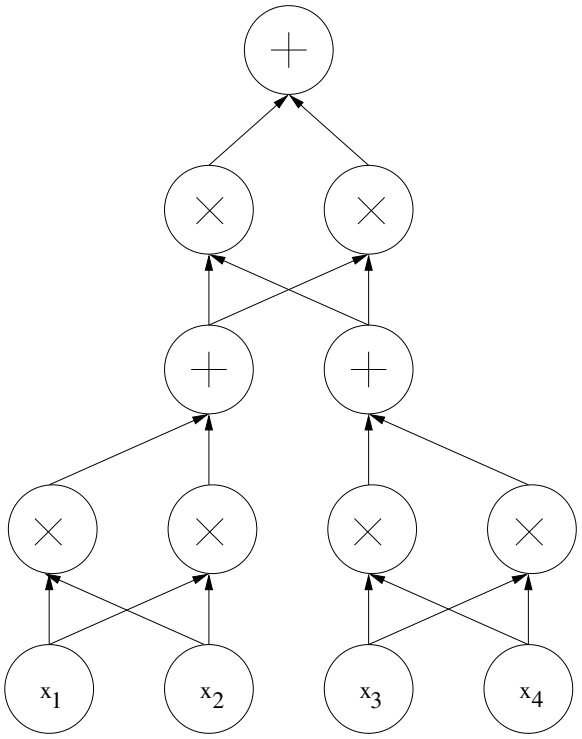
where each $(y, \alpha) \in L$ is associated with a leaf where the output value is 1, and both y and α are d-dimensional binary vectors*: the i-th term of the product is ensuring that x_i and y_i are equal when α_i is 1 (while being indifferent to their values when α_i is 0). The non-zero values of α thus correspond to the variables that are tested on the path from the root of the tree to a leaf (and their number is the length of this path), while y contains the values of these variables that lead to this leaf (note that the value of y_i does not matter when $\alpha_i = 0$, because the variable is not being tested). The resulting function $T(x)$ thus has a polynomial form, and we call its representation by eq. 3.11 its *expansion* (in the form of sums of products). Note that *any* binary function over $\{0,1\}^d$ can be written in this form, because for any f we can write

$$f(x) = \sum_{y\in\{0,1\}^d} f(y)\mathbf{1}_{y=x} = \sum_{y\in\{0,1\}^d \text{ s.t. } f(y)=1} \Pi_i(y_i x_i + (1-y_i)(1-x_i)).$$

This equation shows that we do not need the α's in eq. 3.11 to represent a function. However, they can lead to a more compact representation: for instance the function x_1 *OR* x_2 can be written $x_1 + (1-x_1)x_2$ instead of $x_1x_2 + (1-x_1)x_2 + x_1(1-x_2)$. It remains a shallow representation though, corresponding to a 2-level architecture with a hidden layer containing as many nodes as leaves (each node computing a product $\Pi_{i=1}^{d}(y_i x_i + (1-y_i)(1-x_i))^{\alpha_i}$), and a single output unit that simply computes the sum of all hidden nodes. Consider now a deeper architecture obtained first by allowing multiple output units (each performing a sum over a different subset of hidden nodes), then adding two extra layers similar to our initial hidden and output layers (i.e. the first extra layer's nodes compute products of their inputs, possibly negated, while the second extra layer is a unit computing the sum of all these nodes). This process can be repeated such as to obtain an architecture of depth $2k$, that we call *factorized* (as a sum of products of sums of products of ...). Figure 3.3 shows an example of such an architecture for $k = 2$. When expanding it into an equivalent expansion of the form of eq. 3.11, one may require a number of terms in the sum (i.e. of nodes in the hidden layer of the 2-level corresponding architecture, and of leaves in the corresponding decision tree) potentially exponential in the depth of the tree, requiring many more parameters to be tuned when learning it from data.

As an example, consider representing the parity function in input dimension $d = 2^k$, with its output defined to be 1 iff the number of non-zero bits in the input is odd, and 0 otherwise. It is easy to see that a decision tree of the form of eq. 3.11 requires a sum over 2^{d-1} terms (all possible inputs for

*Here we use the convention $0^0 = 1$.



◀ **Figure 3.3.** *Example of a deep architecture obtained when adding extra layers performing a second "sum of products" operation. Here, the + nodes simply sum their children. If the × nodes compute the product of one of their children by the negation of their other child, then this architecture computes the parity function (see text for details).*

which the output is 1) to perfectly model this function, because no variable can be ignored in the decision (i.e. all α_i's must be 1). On another hand, consider a deep architecture of depth $2k$ where the $(2j-1)$-th layer computes products of pairs of (possibly negated) nodes in its input layer, more precisely by denoting z_i^ℓ the i-th unit in layer ℓ,

$$\begin{aligned} z_{2i-1}^{2j-1} &= z_{2i-1}^{2j-2}(1 - z_{2i}^{2j-2}) \\ z_{2i}^{2j-1} &= (1 - z_{2i-1}^{2j-2}) z_{2i}^{2j-2} \end{aligned}$$

and the layer above is made of half the number of nodes, each computing a sum

$$z_i^{2j} = z_{2i-1}^{2j-1} + z_{2i}^{2j-1}.$$

The resulting architecture for $d = 4$ (i.e. $k = 2$) is illustrated in figure 3.3. The interpretation is that each node of layer $2j$ represents the output of the parity function over two nodes of layer $2(j-1)$, and thus, by recursion, it is also the output of the parity function over a subset of 2^j bits of the input. Consequently, the top layer is a single unit computing the parity function over the whole input. It is easy to see that such an architecture has a total number of units equal to $3(d-1)$, and thus a number of parameters on the order of $O(d)$, which is exponentially less than in the flat expansion of eq. 3.11. Yet, it defines the same function (that varies a lot in input space). This is an example of how the *factorized* form of a polynomial, as represented

by a deep architecture, allows for a more compact representation than its *expanded* form (shallow architecture).

Even though boosted trees and forests have clearly been shown to generalize better than single decision trees in a large number of real world learning tasks (Ho, 1995; Freund et al., 1996; Breiman, 2001), we conjecture that their depth is still too limited to be able to learn highly-varying functions like what is needed for the checkerboard task, in the sense of generalizing to variations not seen in the training set. The conjecture is inspired by the circuit complexity results stating that there are functions computable with a polynomial-size logic gates or threshold circuits of depth k that require exponential size when restricted to depth $k-1$ (Håstad, 1986; Håstad et al., 1991). In other words, the right depth may be data-dependent.

3.5 Conclusion

Inspired by previous work (Bengio et al., 2006a, 2007b) showing the inability of Gaussian kernel machines and more generally of shallow architectures to learn highly varying functions (even when a simple expression for the solution exists), we presented similar negative results for decision trees. We believe that the arguments made in this paper can easily be generalized to the case of decision trees with non-constant leaf predictions (such as linear predictions) and decision nodes that are not axis-aligned. Formal proofs for these more general cases remain to be established, but the crucial ingredients that remain valid from the current analysis are: (i) only the examples falling into a leaf are used to produce the estimator associated with this leaf, and (ii) the leaves are associated with non-intersecting regions.

This analysis helps to understand the old question of the curse of dimensionality by illustrating its effect in the case of decision trees. It clarifies that the central issue is not one of dimensionality, nor purely one of local predictions (like in the case of Gaussian kernels; see Bengio et al., 2006a, 2007b). Instead it is about the limitation of estimators that divide the input space in regions (hard ones in the case of decision trees, soft ones in the case of Gaussian kernel machines), with separate parameters associated with each region. Consider the learning of a highly-varying function, i.e, which requires many such regions to be properly represented. Barring the injection of additional knowledge to guide the estimation of these parameters, the number of examples required to learn such models thus grows linearly with the number of these regions, i.e., the complexity of the functions that can be represented. The analysis also gives an alternative conjecture to explain some of the better generalization abilities of forests and boosted trees. The latter actually exploit a distributed representation of the input space in order to generalize to regions not covered by the training set, giving them a potentially exponentially more efficient representation than single decision

trees. One question raised by this work and inspired by results on complexity theory of circuits is whether even forests and boosted trees could be significantly improved upon by considering yet deeper architectures.

Although previous complexity theory results (Grigoriev et al., 1995; Cucker et al., 1999), and empirical observations (Pérez et al., 1996; Vilalta et al., 1997) have already pointed out limitations of decision trees that originate from the same source that underlies the theorems presented here, we believe that an important contribution of this paper is to connect such results with the question of learning, and in particular of learning complex tasks such as those required for AI. This puts closer to the center stage for AI and machine learning research the question of learning efficient representations of highly-varying but low Kolmogorov complexity functions, such as those one would expect to need to solve AI tasks. This paper adds to previously presented arguments (Bengio et al., 2007b) suggesting that a necessary condition for solving AI tasks is that the learning algorithm should be able to construct a deep architecture for the learned function. Although these results are related to computational complexity theory results, they point to a *statistical* limitation: the need for a large tree implies the need for a large number of examples. Of course, assuming a deeper architecture does not necessarily provide better generalization because to define a learning algorithm we need in addition to a nice function class a way to search in it. The discussion arising here does not address the computational complexity issue due to the difficulty of searching in the space of deep architectures, e.g. optimizing their parameters appears to be a fundamentally difficult challenge. However, new hope has arisen in the form of successful algorithms based on unsupervised learning for particular classes of deep architectures (Hinton et al., 2006; Bengio et al., 2007a; Ranzato et al., 2007; Bengio, 2009).

Acknowledgements

The authors would like to thank Aaron Courville for precious feedback and discussions. This work was supported by CIFAR, NSERC, MITACS and the Canada Research Chairs.

3.6 Commentaires

Si les résultats théoriques démontrés dans cet articles concernant l'efficacité statistique des arbres de décision peuvent sembler intuitifs, leur preuve n'en reste pas moins une contribution significative. D'une part, il est toujours bon de vérifier nos intuitions, qui peuvent parfois s'avérer fausses. D'autre part, développer ces intuitions requiert une familiarité certaine avec les arbres de décision, que n'ont pas nécessairement toutes les personnes utilisant ces algorithmes : en mettant en avant ces limitations, nous espérons aider les utilisateurs d'arbres de décision à mieux comprendre dans quelles situations ils risquent de les voir échouer. Finalement, la formalisation des raisons qui font qu'un arbre de décision peut ne pas être efficace permet également de mieux comprendre pourquoi les ensembles d'arbres (comme les forêts) fonctionnent mieux, et de motiver la recherche d'autres variantes qui repousseront encore ces limites.

Dans la section 3.4.3, nous introduisons une autre direction d'analyse des arbres de décision en suggérant que la structure hiérarchique d'un arbre pourrait permettre de représenter certaines fonctions de manière plus compacte qu'une architecture non factorisée, en faisant un lien avec la factorisation des polynômes. Cette direction de recherche a inspiré une analyse plus approfondie des liens entre arbres et polynômes, qui est justement l'objet du prochain chapitre.

Bibliographie

Ajtai, M. 1983, "$\sum_1^1$-formulae on finite structures", *Annals of Pure and Applied Logic*, vol. 24, n^{o} 1, p. 1–48.

Bengio, Y. 2009, "Learning deep architectures for AI", *Foundations and Trends in Machine Learning*, vol. 2, n^{o} 1, p. 1–127. Also published as a book. Now Publishers, 2009.

Bengio, Y., O. Delalleau et N. Le Roux. 2006a, "The curse of highly variable functions for local kernel machines", dans *Advances in Neural Information Processing Systems 18 (NIPS'05)*, édité par Y. Weiss, B. Schölkopf et J. Platt, MIT Press, Cambridge, MA, p. 107–114.

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007a, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bengio, Y., H. Larochelle et P. Vincent. 2006b, "Non-local manifold Parzen windows", dans *Advances in Neural Information Processing Systems 18*

(NIPS'05), édité par Y. Weiss, B. Schölkopf et J. Platt, MIT Press, p. 115–122.

Bengio, Y. et Y. LeCun. 2007b, "Scaling learning algorithms towards AI", dans *Large Scale Kernel Machines*, édité par L. Bottou, O. Chapelle, D. DeCoste et J. Weston, MIT Press.

Bengio, Y., M. Monperrus et H. Larochelle. 2006c, "Nonlocal estimation of manifold structure", *Neural Computation*, vol. 18, n° 10, p. 2509–2528.

Boser, B. E., I. M. Guyon et V. N. Vapnik. 1992, "A training algorithm for optimal margin classifiers", dans *COLT '92 : Proceedings of the fifth annual workshop on Computational learning theory*, ACM, New York, NY, USA, p. 144–152.

Breiman, L. 2001, "Random forests", *Machine Learning*, vol. 45, n° 1, p. 5–32.

Breiman, L., J. H. Friedman, R. A. Olshen et C. J. Stone. 1984, *Classification and Regression Trees*, Wadsworth International Group, Belmont, CA.

Cortes, C. et V. Vapnik. 1995, "Support vector networks", *Machine Learning*, vol. 20, p. 273–297.

Cucker, F. et D. Grigoriev. 1999, "Complexity lower bounds for approximation algebraic computation trees", *Journal of Complexity*, vol. 15, n° 4, p. 499–512.

Freund, Y. et R. E. Schapire. 1996, "Experiments with a new boosting algorithm", dans *Machine Learning : Proceedings of Thirteenth International Conference*, ACM, USA, p. 148–156.

Grigoriev, D., M. Karpinski et A. C.-C. Yao. 1995, "An exponential lower bound on the size of algebraic decision trees for MAX", *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 2, n° 057.

Håstad, J. 1986, "Almost optimal lower bounds for small depth circuits", dans *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, ACM Press, Berkeley, California, p. 6–20.

Håstad, J. et M. Goldmann. 1991, "On the power of small-depth threshold circuits", *Computational Complexity*, vol. 1, p. 113–129.

Hinton, G. E. 1986, "Learning distributed representations of concepts", dans *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum, Hillsdale, Amherst 1986, p. 1–12.

Hinton, G. E. et Z. Ghahramani. 1997, "Generative models for discovering sparse distributed representations", *Philosophical Transactions of the Royal Society of London*, vol. B, n° 352, p. 1177–1190.

Hinton, G. E., S. Osindero et Y. Teh. 2006, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18, p. 1527–1554.

Ho, T. K. 1995, "Random decision forest", dans *3rd International Conference on Document Analysis and Recognition (ICDAR'95)*, Montreal, Canada, p. 278–282.

Hornik, K., M. Stinchcombe et H. White. 1989, "Multilayer feedforward networks are universal approximators", *Neural Networks*, vol. 2, p. 359–366.

Hutter, M. 2005, *Universal Artificial Intelligence : Sequential Decisions based on Algorithmic Probability*, Springer, Berlin.

Kolmogorov, A. N. 1965, "Three approaches to the quantitative definition of information", *Problems of Information and Transmission*, vol. 1, n° 1, p. 1–7.

Kong, E. B. et T. G. Dietterich. 1995, "Error-correcting output coding corrects bias and variance", dans *International Conference on Machine Learning*, p. 313–321.

LeCun, Y., F.-J. Huang et L. Bottou. 2004, "Learning methods for generic object recognition with invariance to pose and lighting", dans *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'04)*, vol. 2, IEEE Computer Society, Los Alamitos, CA, USA, p. 97–104.

Li, M. et P. Vitányi. 2008, *An Introduction to Kolmogorov Complexity and Its Applications*, 3e éd., Springer, New York, NY.

Loh, W.-Y. et Y.-S. Shih. 1997, "Split selection methods for classification trees", *Statistica Sinica*, vol. 7, p. 815–840.

Paccanaro, A. et G. E. Hinton. 2000, "Extracting distributed representations of concepts and relations from positive and negative propositions", dans *International Joint Conference on Neural Networks (IJCNN)*, IEEE, New York, Como, Italy.

Pérez, E. et L. A. Rendell. 1996, "Learning despite concept variation by finding structure in attribute-based data", dans *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'96)*, édité par L. Saitta, Morgan Kaufmann, p. 391–399.

Ranzato, M., C. Poultney, S. Chopra et Y. LeCun. 2007, "Efficient learning of sparse representations with an energy-based model", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 1137–1144.

Solomonoff, R. J. 1964, "A formal theory of inductive inference", *Information and Control*, vol. 7, p. 1–22, 224–254.

Vilalta, R., G. Blix et L. Rendell. 1997, “Global data analysis and the fragmentation problem in decision tree induction”, dans *Proceedings of the 9th European Conference on Machine Learning (ECML'97)*, Springer-Verlag, p. 312–327.

4 Shallow vs. deep sum-product networks

O. Delalleau et Y. Bengio
***Advances in Neural Information Processing Systems 24*, 2011**

Les architectures profondes sont souvent motivées par l'analogie avec les couches successives de neurones dans le cerveau, en particulier dans le cortex visuel (Bengio, 2009). Les activations de chaque couche étant obtenues par la composition des fonctions calculées par toutes les couches précédentes, on peut observer que les neurones dans les couches supérieures représentent des concepts plus abstraits et plus complexes que dans les couches inférieures. Par exemple dans le système visuel, les premières couches ont des neurones qui réagissent à des propriétés simples et localisées de la scène observée (comme les arêtes entre les objets), tandis que les couches suivantes détectent des formes géométriques simples, et ainsi de suite jusqu'à la détection de concepts beaucoup plus complexes dans les couches supérieures (comme la reconnaissance de visages d'autres êtres humains ; Nelson, 2001). Il est donc naturel que les utilisateurs de réseaux de neurones tentent de reproduire ce type d'architecture profonde : même si on sait qu'un réseau de neurones à une couche cachée est déjà un approximateur universel (Hornik et al., 1989), le phénomène de composition de fonctions dans une architecture profonde semble pouvoir rendre la représentation de fonctions complexes plus aisée. C'est ce que nous montrons dans ce chapitre pour une architecture spécifique de réseaux de neurones appelés les réseaux "sommes-produits" (dû au fait que les seules opérations effectués sont des sommes et des produits), récemment introduits par Poon et al. (2011). L'idée centrale de ce chapitre est qu'un réseau sommes-produits profond calcule un polynôme de manière factorisée, ce qui est beaucoup plus efficace que s'il était développé (ce qui correspondrait à une architecture à une seule couche cachée). Ces résultats formalisent une conjecture de Bengio (2009) ainsi que certaines idées évoquées au chapitre précédent (section 3.4.3).

Contribution personnelle Comme indiqué ci-dessus, cet article est basé sur une conjecture de Y. Bengio, qui a également rédigé l'introduction. De mon côté, j'ai imaginé les énoncés mathématiques (inspirés par cette conjecture) et écrit leurs preuves. Les discussions et la conclusion sont le fruit de notre travail commun.

Abstract We investigate the representational power of sum-product networks (computation networks analogous to neural networks, but whose individual units compute either products or weighted sums), through a theoretical analysis that compares deep (multiple hidden layers) vs. shallow (one hidden layer) architectures. We prove there exist families of functions that can be represented much more efficiently with a deep network than with a shallow one, i.e. with substantially fewer hidden units. Such results were not available until now, and contribute to motivate recent research involving learning of deep sum-product networks, and more generally motivate research in Deep Learning.

4.1 Introduction and prior work

Many learning algorithms are based on searching a family of functions so as to identify one member of said family which minimizes a training criterion. The choice of this family of functions and how members of that family are parameterized can be a crucial one. Although there is no universally optimal choice of parameterization or family of functions (or "architecture"), as demonstrated by the no-free-lunch results (Wolpert, 1996), it may be the case that some architectures are appropriate (or inappropriate) for a large class of learning tasks and data distributions, such as those related to Artificial Intelligence tasks (Bengio et al., 2007b). Different families of functions have different characteristics that can be appropriate or not depending on the learning task of interest. One of the characteristics that has spurred much interest and research in recent years is **depth of the architecture**. In the case of a multi-layer neural network, depth corresponds to the number of (hidden and output) layers. A fixed-kernel Support Vector Machine is considered to have depth 2 (Bengio et al., 2007b) and boosted decision trees to have depth 3 (Bengio et al., 2010). Here we use the word *circuit* or *network* to talk about a directed acyclic graph, where each node is associated with some output value which can be computed based on the values associated with its predecessor nodes. The arguments of the learned function are set at the input nodes of the circuit (which have no predecessor) and the outputs of the function are read off the output nodes of the circuit. Different families of functions correspond to different circuits and allowed choices of computations in each node. Learning can be performed by changing the computation associated with a node, or rewiring the circuit (possibly changing the number of nodes). The depth of the circuit is the length of the longest path in the graph from an input node to an output node.

Deep Learning algorithms (Bengio, 2009) are tailored to learning circuits with variable depth, typically greater than depth 2. They are based on the idea of *multiple levels of representation*, with the intuition that the raw input can be represented at different levels of abstraction, with more abstract features of the input or more abstract explanatory factors represented by

deeper circuits. These algorithms are often based on unsupervised learning, opening the door to semi-supervised learning and efficient use of large quantities of unlabeled data (Bengio, 2009). Analogies with the structure of the cerebral cortex (in particular the visual cortex – see Serre et al., 2007) and similarities between features learned with some Deep Learning algorithms and those hypothesized in the visual cortex (Lee et al., 2008) further motivate investigations into deep architectures. It has been suggested that deep architectures are more powerful in the sense of being able to more efficiently represent highly-varying functions (Bengio et al., 2007b; Bengio, 2009). In this paper, we measure "efficiency" in terms of the number of computational units in the network. An efficient representation is important mainly because: (i) it uses less memory and is faster to compute, and (ii) given a fixed amount of training samples and computational power, better generalization is expected.

The first successful algorithms for training deep architectures appeared in 2006, with efficient training procedures for Deep Belief Networks (Hinton et al., 2006a) and deep auto-encoders (Hinton et al., 2006b; Ranzato et al., 2007; Bengio et al., 2007a), both exploiting the general idea of greedy layer-wise pre-training (Bengio et al., 2007a). Since then, these ideas have been investigated further and applied in many settings, demonstrating state-of-the-art learning performance in object recognition (Larochelle et al., 2007; Ranzato et al., 2008a; Lee et al., 2009a; Kavukcuoglu et al., 2010) and segmentation (Levner, 2008), audio classification (Lee et al., 2009b; Dahl et al., 2010), natural language processing (Collobert et al., 2008; Weston et al., 2008; Mnih et al., 2009; Socher et al., 2011), collaborative filtering (Salakhutdinov et al., 2007b), modeling textures (Osindero et al., 2008), modeling motion (Taylor et al., 2007, 2009), information retrieval (Salakhutdinov et al., 2007a; Ranzato et al., 2008b), and semi-supervised learning (Weston et al., 2008; Mobahi et al., 2009).

Poon et al. (2011) introduced deep **sum-product networks** as a method to compute partition functions of tractable graphical models. These networks are analogous to traditional artificial neural networks but with nodes that compute either products or weighted sums of their inputs. Analogously to neural networks, we define "hidden" nodes as those nodes that are neither input nodes nor output nodes. If the nodes are organized in layers, we define the "hidden" layers to be those that are neither the input layer nor the output layer. Poon et al. (2011) report experiments with networks much deeper (30+ hidden layers) than those typically used until now, e.g. in Deep Belief Networks (Hinton et al., 2006a; Bengio, 2009), where the number of hidden layers is usually on the order of three to five.

Whether such deep architectures have theoretical advantages compared to so-called "shallow" architectures (i.e. those with a single hidden layer) remains an open question. After all, in the case of a sum-product network, the output value can always be written as a sum of products of input variables (possibly raised to some power by allowing multiple connections from

the same input), and consequently it is easily rewritten as a shallow network with a sum output unit and product hidden units. The argument supported by our theoretical analysis is that a deep architecture is able to compute some functions much more efficiently than a shallow one.

Until recently, very few theoretical results supported the idea that deep architectures could present an advantage in terms of representing some functions more efficiently. Most related results originate from the analysis of boolean circuits (see e.g. Allender, 1996, for a review). Well-known results include the proof that solving the n-bit parity task with a depth-2 circuit requires an exponential number of gates (Ajtai, 1983; Yao, 1985), and more generally that there exist functions computable with a polynomial-size depth-k circuit that would require exponential size when restricted to depth $k-1$ (Håstad, 1986). Another recent result on boolean circuits by Braverman (2011) offers proof of a longstanding conjecture, showing that bounded-depth boolean circuits are unable to distinguish some (non-uniform) input distributions from the uniform distribution (i.e. they are "fooled" by such input distributions). In particular, Braverman's result suggests that shallow circuits can in general be fooled more easily than deep ones, i.e., that they would have more difficulty efficiently representing high-order dependencies (those involving many input variables).

It is not obvious that circuit complexity results (that typically consider only boolean or at least discrete nodes) are directly applicable in the context of typical machine learning algorithms such as neural networks (that compute continuous representations of their input). Orponen (1994) surveys theoretical results in computational complexity that are relevant to learning algorithms. For instance, Håstad et al. (1991) extended some results to the case of networks of linear threshold units with positivity constraints on the weights. Bengio et al. (2006, 2010) investigate, respectively, complexity issues in networks of Gaussian radial basis functions and decision trees, showing intrinsic limitations of these architectures e.g. on tasks similar to the parity problem. Utgoff et al. (2002) informally discuss the advantages of depth in boolean circuit in the context of learning architectures. Bengio (2009) suggests that some polynomials could be represented more efficiently by deep sum-product networks, but without providing any formal statement or proofs. This work partly addresses this void by demonstrating families of circuits for which a deep architecture can be exponentially more efficient than a shallow one in the context of real-valued polynomials.

Note that we do not address in this paper the problem of *learning* these parameters: even if an efficient deep representation exists for the function we seek to approximate, in general there is no guarantee for standard optimization algorithms to easily converge to this representation. This paper focuses on the representational power of deep sum-product circuits compared to shallow ones, and studies it by considering particular families of target functions (to be represented by the learner).

We first formally define sum-product networks. We consider two families

of functions represented by deep sum-product networks (families $\mathcal{F}$ and $\mathcal{G}$). For each family, we establish a lower bound on the minimal number of hidden units a depth-2 sum-product network would require to represent a function of this family, showing it is much less efficient than the deep representation.

4.2 Sum-product networks

Definition 4.2.1. *A sum-product network is a network composed of units that either compute the product of their inputs or a weighted sum of their inputs (where weights are strictly positive).*

Here, we restrict our definition of the generic term "sum-product network" to networks whose summation units have positive incoming weights*, while others are called "negative-weight" networks.

Definition 4.2.2. *A "negative-weight" sum-product network may contain summation units whose weights are non-positive (i.e. less than or equal to zero).*

Finally, we formally define what we mean by *deep* vs. *shallow* networks in the rest of the paper.

Definition 4.2.3. *A "shallow" sum-product network contains a single hidden layer (i.e. a total of three layers when counting the input and output layers, and a depth equal to two).*

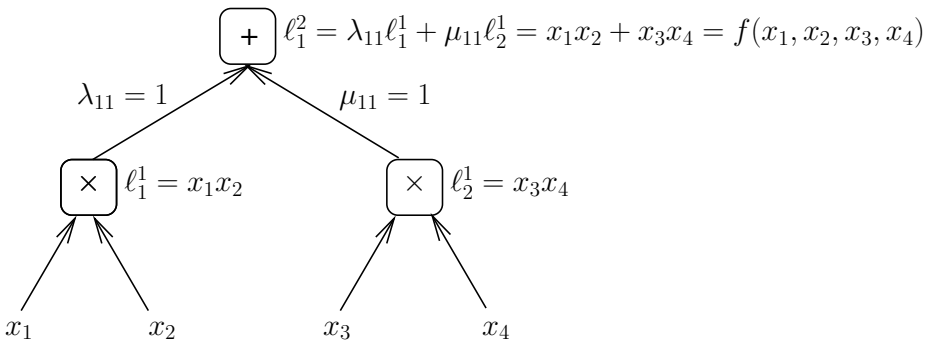
Definition 4.2.4. *A "deep" sum-product network contains more than one hidden layer (i.e. a total of at least four layers, and a depth at least three).*

4.3 The family $\mathcal{F}$

4.3.1 Definition

The first family of functions we study, denoted by $\mathcal{F}$, is made of functions built from deep sum-product networks that alternate layers of product and sum units with two inputs each (details are provided below). The basic idea we use here is that composing layers (i.e. using a deep architecture) is equivalent to using a factorized representation of the polynomial function computed by the network. Such a factorized representation can be exponentially more compact than its expansion as a sum of products (which can be associated to a shallow network with product units in its hidden layer and a sum unit as output). This is what we formally show in what follows.

*This condition is required by some of the proofs presented here.



▶ **Figure 4.1.** *Sum-product network computing the simplest function $f \in \mathcal{F}$ ($i = 1$, and summation weights λ_{11} and μ_{11} equal to 1).*

Let $n = 4^i$, with i a positive integer value. Denote by ℓ^0 the input layer containing scalar variables $\{x_1, \ldots, x_n\}$, such that $\ell_j^0 = x_j$ for $1 \leq j \leq n$. Now define $f \in F$ as any function computed by a sum-product network (deep for $i \geq 2$) composed of alternating product and sum layers:

- $\ell_j^{2k+1} = \ell_{2j-1}^{2k} \cdot \ell_{2j}^{2k}$ for $0 \leq k \leq i-1$ and $1 \leq j \leq 2^{2(i-k)-1}$
- $\ell_j^{2k} = \lambda_{jk}\ell_{2j-1}^{2k-1} + \mu_{jk}\ell_{2j}^{2k-1}$ for $1 \leq k \leq i$ and $1 \leq j \leq 2^{2(i-k)}$

where the weights λ_{jk} and μ_{jk} of the summation units are strictly positive.

The output of the network is given by $f(x_1, \ldots, x_n) = \ell_1^{2i} \in \mathbb{R}$, the unique unit in the last layer. The corresponding (shallow) network for $i = 1$ and additive weights set to one is shown in figure 4.1 (this architecture is also the basic building block of bigger networks for $i > 1$). Note that both the input size $n = 4^i$ and the network's depth $2i$ increase with parameter i.

4.3.2 Theoretical results

The main result of this section is presented below in corollary 4.3.8, providing a lower bound on the minimum number of hidden units required by a shallow sum-product network to represent a function $f \in \mathcal{F}$. The high-level proof sketch consists in the following steps:
(1) Count the number of unique products found in the polynomial representation of f (lemma 4.3.1 and proposition 4.3.2).
(2) Show that the only possible architecture for a shallow sum-product network to compute f is to have a hidden layer made of product units, with a sum unit as output (lemmas 4.3.3 to 4.3.6).
(3) Conclude that the number of hidden units must be at least the number of unique products computed in step 4.3.2 (lemma 4.3.7 and corollary 4.3.8).

Lemma 4.3.1. *Any element ℓ_j^k can be written as a (positively) weighted sum of products of input variables, such that each input variable x_t is used in exactly one unit of ℓ^k. Moreover, the number m_k of products found in the*

sum computed by ℓ_j^k does not depend on j and obeys the following recurrence rule for $k \geq 0$: if $k+1$ is odd, then $m_{k+1} = m_k^2$, otherwise $m_{k+1} = 2m_k$.

Proof. We prove the lemma by induction on k. It is obviously true for $k = 0$ since $\ell_j^0 = x_j$. Assuming this is true for some $k \geq 0$, we consider two cases:

- If $k+1$ is odd, then $\ell_j^{k+1} = \ell_{2j-1}^k \cdot \ell_{2j}^k$. By the inductive hypothesis, it is the product of two (positively) weighted sums of products of input variables, and no input variable can appear in both ℓ_{2j-1}^k and ℓ_{2j}^k, so the result is also a (positively) weighted sum of products of input variables. Additionally, if the number of products in ℓ_{2j-1}^k and ℓ_{2j}^k is m_k, then $m_{k+1} = m_k^2$, since all products involved in the multiplication of the two units are different (since they use disjoint subsets of input variables), and the sums have positive weights.

  Finally, by the induction assumption, an input variable appears in exactly one unit of ℓ^k. This unit is an input to a single unit of ℓ^{k+1}, that will thus be the only unit of ℓ^{k+1} where this input variable appears.

- If $k+1$ is even, then $\ell_j^{k+1} = \lambda_{jk}\ell_{2j-1}^k + \mu_{jk}\ell_{2j}^k$. Again, from the induction assumption, it must be a (positively) weighted sum of products of input variables, but with $m_{k+1} = 2m_k$ such products. As in the previous case, an input variable will appear in the single unit of ℓ^{k+1} that has as input the single unit of ℓ^k in which this variable must appear.

□

Proposition 4.3.2. *The number of products in the sum computed in the output unit l_1^{2i} of a network computing a function in $\mathcal{F}$ is $m_{2i} = 2^{\sqrt{n}-1}$.*

Proof. We first prove by induction on $k \geq 1$ that for odd k, $m_k = 2^{2^{\frac{k+1}{2}}-2}$, and for even k, $m_k = 2^{2^{\frac{k}{2}}-1}$. This is obviously true for $k = 1$ since $2^{2^{\frac{1+1}{2}}-2} = 2^0 = 1$, and all units in ℓ^1 are single products of the form $x_r x_s$. Assuming this is true for some $k \geq 1$, then:

- if $k+1$ is odd, then from lemma 4.3.1 and the induction assumption, we have:

$$m_{k+1} = m_k^2 = \left(2^{2^{\frac{k}{2}}-1}\right)^2 = 2^{2^{\frac{k}{2}+1}-2} = 2^{2^{\frac{(k+1)+1}{2}}-2}$$

- if $k+1$ is even, then instead we have:

$$m_{k+1} = 2m_k = 2 \cdot 2^{2^{\frac{k+1}{2}}-2} = 2^{2^{\frac{(k+1)}{2}}-1}$$

which shows the desired result for $k+1$, and thus concludes the induction proof. Applying this result with $k = 2i$ (which is even) yields

$$m_{2i} = 2^{2^{\frac{2i}{2}}-1} = 2^{\sqrt{2^{2i}}-1} = 2^{\sqrt{n}-1}.$$

□

Lemma 4.3.3. *The products computed in the output unit l_1^{2i} can be split in two groups, one with products containing only variables $x_1, \ldots, x_{\frac{n}{2}}$ and one containing only variables $x_{\frac{n}{2}+1}, \ldots, x_n$.*

Proof. This is obvious since the last unit is a "sum" unit that adds two terms whose inputs are these two groups of variables (see e.g. figure 4.1). □

Lemma 4.3.4. *The products computed in the output unit l_1^{2i} involve more than one input variable.*

Proof. It is straightforward to show by induction on $k \geq 1$ that the products computed by l_j^k all involve more than one input variable, thus it is true in particular for the output layer ($k = 2i$). □

Lemma 4.3.5. *Any shallow sum-product network computing $f \in \mathcal{F}$ must have a "sum" unit as output.*

Proof. By contradiction, suppose the output unit of such a shallow sum-product network is multiplicative. This unit must have more than one input, because in the case that it has only one input, the output would be either a (weighted) sum of input variables (which would violate lemma 4.3.4), or a single product of input variables (which would violate proposition 4.3.2), depending on the type (sum or product) of the single input hidden unit. Thus the last unit must compute a product of two or more hidden units. It can be re-written as a product of two factors, where each factor corresponds to either one hidden unit, or a product of multiple hidden units (it does not matter here which specific factorization is chosen among all possible ones). Regardless of the type (sum or product) of the hidden units involved, those two factors can thus be written as weighted sums of products of variables x_t (with positive weights, and input variables potentially raised to powers above one). From lemma 4.3.1, both x_1 and x_n must be present in the final output, and thus they must appear in at least one of these two factors. Without loss of generality, assume x_1 appears in the first factor. Variables $x_{\frac{n}{2}+1}, \ldots, x_n$ then cannot be present in the second factor, since otherwise one product in the output would contain both x_1 and one of these variables (this product cannot cancel out since weights must be positive), violating lemma 4.3.3. But with a similar reasoning, since as a result x_n must appear in the first factor, variables $x_1, \ldots, x_{\frac{n}{2}}$ cannot be present in the second factor either. Consequently, no input variable can be present in the second factor, leading to the desired contradiction. □

Lemma 4.3.6. *Any shallow sum-product network computing $f \in \mathcal{F}$ must have only multiplicative units in its hidden layer.*

Proof. By contradiction, suppose there exists a "sum" unit in the hidden layer, written $s = \sum_{t \in S} \alpha_t x_t$ with S the set of input indices appearing in this sum, and $\alpha_t > 0$ for all $t \in S$. Since according to lemma 4.3.5 the output unit must also be a sum (and have positive weights according to Definition 4.2.1), then the final output will also contain terms of the form $\beta_t x_t$ for $t \in S$, with $\beta_t > 0$. This violates lemma 4.3.4, establishing the contradiction. □

Lemma 4.3.7. *Any shallow negative-weight sum-product network (see Definition 4.2.2) computing $f \in \mathcal{F}$ must have at least $2^{\sqrt{n}-1}$ hidden units, if its output unit is a sum and its hidden units are products.*

Proof. Such a network computes a weighted sum of its hidden units, where each hidden unit is a product of input variables, i.e. its output can be written as $\Sigma_j w_j \Pi_t x_t^{\gamma_{jt}}$ with $w_j \in \mathbb{R}$ and $\gamma_{jt} \in \{0, 1\}$. In order to compute a function in $\mathcal{F}$, this shallow network thus needs a number of hidden units at least equal to the number of unique products in that function. From proposition 4.3.2, this number is equal to $2^{\sqrt{n}-1}$. □

Corollary 4.3.8. *Any shallow sum-product network computing $f \in \mathcal{F}$ must have at least $2^{\sqrt{n}-1}$ hidden units.*

Proof. This is a direct corollary of lemmas 4.3.5 (showing the output unit is a sum), 4.3.6 (showing that hidden units are products), and 4.3.7 (showing the desired result for any shallow network with this specific structure – regardless of the sign of weights). □

4.3.3 Discussion

Corollary 4.3.8 above shows that in order to compute some function in $\mathcal{F}$ with n inputs, the number of units in a shallow network has to be at least $2^{\sqrt{n}-1}$, (i.e. grows exponentially in $\sqrt{n}$). On another hand, the total number of units in the deep (for $i > 1$) network computing the same function, as described in section 4.3.1, is equal to $1+2+4+8+\ldots+2^{2i-1}$ (since all units are binary), which is also equal to $2^{2i} - 1 = n - 1$ (i.e. grows only quadratically in $\sqrt{n}$). **It shows that some deep sum-product network with n inputs and depth $\mathbf{O}(\log \mathbf{n})$ can represent with $\mathbf{O}(\mathbf{n})$ units what would require $\mathbf{O}(\mathbf{2}^{\sqrt{\mathbf{n}}})$ units for a depth-2 network.** Lemma 4.3.7 also shows a similar result regardless of the sign of the weights in the summation units of the depth-2 network, but assumes a specific architecture for this network (products in the hidden layer with a sum as output).

4.4 The family $\mathcal{G}$

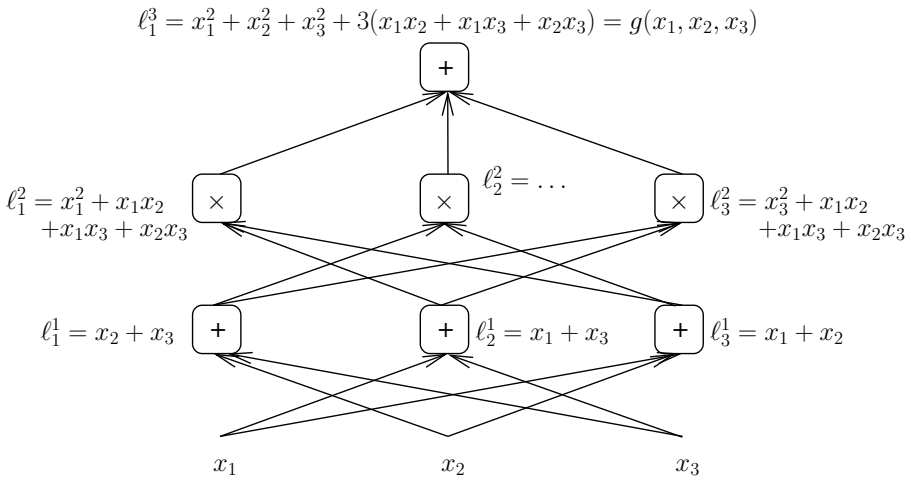
In this section we present similar results with a different family of functions, denoted by $\mathcal{G}$. Compared to $\mathcal{F}$, one important difference of deep sum-product networks built to define functions in $\mathcal{G}$ is that they can vary their input size independently of their depth. Their analysis thus provides additional insight when comparing the representational efficiency of deep vs. shallow sum-product networks in the case of a fixed dataset.

4.4.1 Definition

Networks in family $\mathcal{G}$ also alternate sum and product layers, but their units have as inputs all units from the previous layer *except one*. More formally, define the family $\mathcal{G} = \cup_{n\geq 2, i\geq 0}\mathcal{G}_{in}$ of functions represented by sum-product networks, where the sub-family $\mathcal{G}_{in}$ is made of all sum-product networks with n input variables and $2i+2$ layers (including the input layer ℓ^0), such that:

1. ℓ^1 contains summation units; further layers alternate multiplicative and summation units.
2. Summation units have positive weights.
3. All layers are of size n, except the last layer ℓ^{2i+1} that contains a single sum unit that sums all units in the previous layer ℓ^{2i}.
4. In each layer ℓ^k for $1 \leq k \leq 2i$, each unit ℓ^k_j takes as inputs $\{\ell^{k-1}_m | m \neq j\}$.

An example of a network belonging to $\mathcal{G}_{1,3}$ (i.e. with three layers and three input variables) is shown in figure 4.2.



▶ **Figure 4.2.** *Sum-product network computing a function of $\mathcal{G}_{1,3}$ (summation units' weights are all equal to 1).*

4.4.2 Theoretical results

The main result is stated in proposition 4.4.4 below, establishing a lower bound on the number of hidden units of a shallow sum-product network computing $g \in \mathcal{G}$. The proof sketch is as follows:

1. We show that the polynomial expansion of g must contain a large set of products (proposition 4.4.2 and corollary 4.4.3).

2. We use both the number of products in that set as well as their degree to establish the desired lower bound (proposition 4.4.4).

We will also need the following lemma, which states that when $n-1$ items each belong to $n-1$ sets among a total of n sets, then we can associate to each item one of the sets it belongs to without using the same set for different items.

Lemma 4.4.1. *Let $S_1, \dots, S_n$ be n sets ($n \geq 2$) containing elements of $\{P_1, \dots, P_{n-1}\}$, such that for any q, r, $|\{r | P_q \in S_r\}| \geq n-1$ (i.e. each element P_q belongs to at least $n-1$ sets). Then there exist $r_1, \dots, r_{n-1}$ different indices such that $P_q \in S_{r_q}$ for $1 \leq q \leq n-1$.*

Proof. By iterative construction, starting with $q = 1$, $Q = \emptyset$ and $R = \emptyset$:

1. Since $|R| \leq n-2$, there exists $r_q \notin R$ such that $P_q \in S_{r_q}$ (because P_q cannot be missing in more than one set).

2. Add q to Q and r_q to R.

3. If $|R| \leq n-2$ then increase q by one and iterate from step 1. Otherwise we are done, since all elements of R are different, and $P_q \in S_{r_q}$ for each $r_q \in R$.

□

Proposition 4.4.2. *For any $0 \leq j \leq i$, and any product of variables $P = \Pi_{t=1}^{n} x_t^{\alpha_t}$ such that $\alpha_t \in \mathbb{N}$ and $\sum_t \alpha_t = (n-1)^j$, there exists a unit in ℓ^{2j} whose computed value, when expanded as a weighted sum of products, contains P among these products.*

Proof. We prove this proposition by induction on j.

First, for $j = 0$, this is obvious since any P of this form must be made of a single input variable x_t, that appears in $\ell_t^0 = x_t$.

Suppose now the proposition is true for some $j < i$. Consider a product $P = \Pi_{t=1}^{n} x_t^{\alpha_t}$ such that $\alpha_t \in \mathbb{N}$ and $\sum_t \alpha_t = (n-1)^{j+1}$. P can be factored in $n-1$ sub-products of degree $(n-1)^j$, i.e. written $P = P_1 \dots P_{n-1}$ with $P_q = \Pi_{t=1}^{n} x_t^{\beta_{qt}}$, $\beta_{qt} \in \mathbb{N}$ and $\sum_t \beta_{qt} = (n-1)^j$ for all q. By the induction hypothesis, each P_q can be found in at least one unit $\ell_{k_q}^{2j}$. As a result, by property 4 (in the definition of family $\mathcal{G}$), each P_q will also appear in the

additive layer ℓ^{2j+1}, in at least $n-1$ different units (the only sum unit that may not contain P_q is the one that does not have $\ell^{2j}_{k_q}$ as input).

By lemma 4.4.1, we can thus find a set of units $\ell^{2j+1}_{r_q}$ such that for any $1 \leq q \leq n-1$, the product P_q appears in $\ell^{2j+1}_{r_q}$, with indices r_q being different from each other. Let $1 \leq s \leq n$ be such that $s \neq r_q$ for all q. Then, from property 4 of family $\mathcal{G}$, the multiplicative unit $\ell^{2(j+1)}_s$ computes the product $\Pi_{q=1}^{n-1} \ell^{2j+1}_{r_q}$, and as a result, when expanded as a sum of products, it contains in particular $P_1 \dots P_{n-1} = P$. The proposition is thus true for $j+1$, and by induction, is true for all $j \leq i$. □

Corollary 4.4.3. *The output g_{in} of a sum-product network in $\mathcal{G}_{in}$, when expanded as a sum of products, contains all products of variables of the form $\Pi_{t=1}^{n} x_t^{\alpha_t}$ such that $\alpha_t \in \mathbb{N}$ and $\sum_t \alpha_t = (n-1)^i$.*

Proof. Applying proposition 4.4.2 with $j = i$, we obtain that all products of this form can be found in the multiplicative units of ℓ^{2i}. Since the output unit ℓ^{2i+1}_1 computes a sum of these multiplicative units (weighted with positive weights), those products are also present in the output. □

Proposition 4.4.4. *A shallow negative-weight sum-product network computing $g_{in} \in \mathcal{G}_{in}$ must have at least $(n-1)^i$ hidden units.*

Proof. First suppose the output unit of the shallow network is a sum. Then it may be able to compute g_{in}, assuming we allow multiplicative units in the hidden layer in the hidden layer to use powers of their inputs in the product they compute (which we allow here for the proof to be more generic). However, it will require at least as many of these units as the number of unique products that can be found in the expansion of g_{in}. In particular, from corollary 4.4.3, it will require at least the number of unique tuples of the form $(\alpha_1, \dots, \alpha_n)$ such that $\alpha_t \in \mathbb{N}$ and $\sum_{t=1}^{n} \alpha_t = (n-1)^i$. Denoting $d_{ni} = (n-1)^i$, this number is known to be equal to $\binom{n+d_{ni}-1}{d_{ni}}$, and it is easy to verify it is higher than (or equal to) d_{ni} for any $n \geq 2$ and $i \geq 0$.

Now suppose the output unit is multiplicative. Then there can be no multiplicative hidden unit, otherwise it would mean one could factor some input variable x_t in the computed function output: this is not possible since by corollary 4.4.3, for any variable x_t there exist products in the output function that do not involve x_t. So all hidden units must be additive, and since the computed function contains products of degree d_{ni}, there must be at least d_{ni} such hidden units. □

4.4.3 Discussion

Proposition 4.4.4 shows that in order to compute the same function as $g_{in} \in \mathcal{G}_{in}$, the number of units in the shallow network has to grow exponentially in i, i.e. in the network's depth (while the deep network's size grows linearly in i). The shallow network also needs to grow polynomially in the number

of input variables n (with a degree equal to i), while the deep network grows only linearly in n. **It means that some deep sum-product network with n inputs and depth O(i) can represent with O(ni) units what would require $\mathbf{O((n-1)^i)}$ units for a depth-2 network**.

Note that in the similar results found for family $\mathcal{F}$, the depth-2 network computing the same function as a function in $\mathcal{F}$ had to be constrained to either have a specific combination of sum and hidden units (in lemma 4.3.7) or to have non-negative weights (in corollary 4.3.8). On the contrary, the result presented here for family $\mathcal{G}$ holds without requiring any of these assumptions.

4.5 Conclusion

We compared a deep sum-product network and a shallow sum-product network representing the same function, taken from two families of functions $\mathcal{F}$ and $\mathcal{G}$. For both families, we have shown that the number of units in the shallow network has to grow exponentially, compared to a linear growth in the deep network, so as to represent the same functions. The deep version thus offers a much more compact representation of the same functions.

This work focuses on two specific families of functions: finding more general parameterization of functions leading to similar results would be an interesting topic for future research. Another open question is whether it is possible to represent such functions only approximately (e.g. up to an error bound ε) with a much smaller shallow network. Results by Braverman (2011) on boolean circuits suggest that similar results as those presented in this paper may still hold, but this topic has yet to be formally investigated in the context of sum-product networks. A related problem is also to look into functions defined only on discrete input variables: our proofs do not trivially extend to this situation because we cannot assume anymore that two polynomials yielding the same output values must have the same expansion coefficients (since the number of input combinations becomes finite).

Acknowledgements

The authors would like to thank Razvan Pascanu and David Warde-Farley for their help in improving this manuscript, as well as the anonymous reviewers for their careful reviews. This work was partially funded by NSERC, CIFAR, and the Canada Research Chairs.

4.6 Commentaires

Les résultats de ce chapitre montrent la supériorité d'un certain type d'architecture profonde en terme d'efficacité de la *représentation*, c.à.d. du nombre de paramètres nécessaires pour calculer certaines fonctions. Limiter le nombre de paramètres est intéressant pour plusieurs raisons :

- Cela permet d'utiliser moins de mémoire pour stocker le modèle, et moins de puissance de calcul pour calculer la valeur de la fonction en un point (efficacité *computationnelle*).
- Cela devrait permettre d'avoir besoin de moins d'exemples d'apprentissage pour optimiser les paramètres du modèle, comme expliqué en section 1.5 (efficacité *statistique*).

Une question naturelle qui peut se poser à la lecture de ce chapitre est de savoir si les fonctions des familles $\mathcal{F}$ et $\mathcal{G}$ ont une utilité pratique quelconque. À ma connaissance, ce n'est pas le cas. Une difficulté fondamentale lorsque l'on utilise une famille de fonctions "utiles" – par exemple une famille capable d'approximer n'importe quelle fonction calculable par un réseau sommes-produits – est que parmi ces fonctions utiles, certaines vont s'avérer être très simples et donc calculables de manière efficace par un réseau à une seule couche cachée. On ne pourrait donc pas obtenir un résultat similaire disant que *toutes* les fonctions de cette famille seraient représentables de manière plus efficace par un réseau profond : il faudrait se restreindre à des classes de fonctions définies de manière plus complexe pour obtenir de tels résultats, et par conséquent il serait de nouveau difficile de leur trouver une justification pratique. L'intérêt de nos démonstrations est donc surtout pédagogique : elles illustrent et permettent ainsi de mieux comprendre certains phénomènes reliés à la profondeur dans les réseaux sommes-produits.

Ce que nous disons ici, c'est qu'il semble intéressant de considérer de tels réseaux profonds pour représenter des fonctions complexes des entrées. Ultimement, ce sont les résultats empiriques de l'apprentissage de tels réseaux profonds qui nous renseigneront sur la validité de cette direction de recherche. Il faut d'ailleurs garder à l'esprit que les architectures profondes sont en général difficiles à optimiser, et que l'apprentissage de réseaux sommes-produits profonds est un sujet de recherche d'actualité (Poon et al., 2011).

Bibliographie

Ajtai, M. 1983, "$\sum_1^1$-formulae on finite structures", *Annals of Pure and Applied Logic*, vol. 24, n° 1, p. 1–48.

Allender, E. 1996, "Circuit complexity before the dawn of the new millennium", dans *16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 1180, Springer Verlag, p. 1–18.

Bengio, Y. 2009, "Learning deep architectures for AI", *Foundations and Trends in Machine Learning*, vol. 2, n^{o} 1, p. 1–127. Also published as a book. Now Publishers, 2009.

Bengio, Y., O. Delalleau et N. Le Roux. 2006, "The curse of highly variable functions for local kernel machines", dans *Advances in Neural Information Processing Systems 18 (NIPS'05)*, édité par Y. Weiss, B. Schölkopf et J. Platt, MIT Press, Cambridge, MA, p. 107–114.

Bengio, Y., O. Delalleau et C. Simard. 2010, "Decision trees do not generalize to new variations", *Computational Intelligence*, vol. 26, n^{o} 4, p. 449–467.

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007a, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bengio, Y. et Y. LeCun. 2007b, "Scaling learning algorithms towards AI", dans *Large Scale Kernel Machines*, édité par L. Bottou, O. Chapelle, D. DeCoste et J. Weston, MIT Press.

Braverman, M. 2011, "Poly-logarithmic independence fools bounded-depth boolean circuits", *Communications of the ACM*, vol. 54, n^{o} 4, p. 108–115.

Collobert, R. et J. Weston. 2008, "A unified architecture for natural language processing : Deep neural networks with multitask learning", dans *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, édité par W. W. Cohen, A. McCallum et S. T. Roweis, ACM, p. 160–167.

Dahl, G. E., M. Ranzato, A. Mohamed et G. E. Hinton. 2010, "Phone recognition with the mean-covariance restricted boltzmann machine", dans *Advances in Neural Information Processing Systems (NIPS)*.

Håstad, J. 1986, "Almost optimal lower bounds for small depth circuits", dans *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, ACM Press, Berkeley, California, p. 6–20.

Håstad, J. et M. Goldmann. 1991, "On the power of small-depth threshold circuits", *Computational Complexity*, vol. 1, p. 113–129.

Hinton, G. E., S. Osindero et Y. Teh. 2006a, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18, p. 1527–1554.

Hinton, G. E. et R. Salakhutdinov. 2006b, "Reducing the dimensionality of data with neural networks", *Science*, vol. 313, n^{o} 5786, p. 504–507.

Hornik, K., M. Stinchcombe et H. White. 1989, "Multilayer feedforward networks are universal approximators", *Neural Networks*, vol. 2, p. 359–366.

Kavukcuoglu, K., P. Sermanet, Y.-L. Boureau, K. Gregor, M. Mathieu et Y. LeCun. 2010, "Learning convolutional feature hierarchies for visual recognition", dans *Advances in Neural Information Processing Systems 23 (NIPS'10)*, p. 1090–1098.

Larochelle, H., D. Erhan, A. Courville, J. Bergstra et Y. Bengio. 2007, "An empirical evaluation of deep architectures on problems with many factors of variation", dans *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, édité par Z. Ghahramani, ACM, p. 473–480.

Lee, H., C. Ekanadham et A. Ng. 2008, "Sparse deep belief net model for visual area V2", dans *Advances in Neural Information Processing Systems 20 (NIPS'07)*, édité par J. Platt, D. Koller, Y. Singer et S. Roweis, MIT Press, Cambridge, MA, p. 873–880.

Lee, H., R. Grosse, R. Ranganath et A. Y. Ng. 2009a, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations", dans *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, édité par L. Bottou et M. Littman, ACM, Montreal (Qc), Canada.

Lee, H., P. Pham, Y. Largman et A. Ng. 2009b, "Unsupervised feature learning for audio classification using convolutional deep belief networks", dans *Advances in Neural Information Processing Systems 22 (NIPS'09)*, édité par Y. Bengio, D. Schuurmans, C. Williams, J. Lafferty et A. Culotta, p. 1096–1104.

Levner, I. 2008, *Data Driven Object Segmentation*, thèse de doctorat, Department of Computer Science, University of Alberta.

Mnih, A. et G. E. Hinton. 2009, "A scalable hierarchical distributed language model", dans *Advances in Neural Information Processing Systems 21 (NIPS'08)*, édité par D. Koller, D. Schuurmans, Y. Bengio et L. Bottou, p. 1081–1088.

Mobahi, H., R. Collobert et J. Weston. 2009, "Deep learning from temporal coherence in video", dans *ICML'2009*, p. 737–744.

Nelson, C. A. 2001, "The development and neural bases of face recognition", *Infant and Child Development*, vol. 10, n° 1–2, p. 3–18.

Orponen, P. 1994, "Computational complexity of neural networks : a survey", *Nordic Journal of Computing*, vol. 1, n° 1, p. 94–110.

Osindero, S. et G. E. Hinton. 2008, "Modeling image patches with a directed hierarchy of markov random field", dans *Advances in Neural Information Processing Systems 20 (NIPS'07)*, édité par J. Platt, D. Koller, Y. Singer et S. Roweis, MIT Press, Cambridge, MA, p. 1121–1128.

Poon, H. et P. Domingos. 2011, "Sum-product networks : A new deep architecture", dans *Proceedings of the Twenty-seventh Conference in Uncertainty in Artificial Intelligence (UAI)*, Barcelona, Spain.

Ranzato, M., Y.-L. Boureau et Y. LeCun. 2008a, "Sparse feature learning for deep belief networks", dans *Advances in Neural Information Processing Systems 20 (NIPS'07)*, édité par J. Platt, D. Koller, Y. Singer et S. Roweis, MIT Press, Cambridge, MA, p. 1185–1192.

Ranzato, M., C. Poultney, S. Chopra et Y. LeCun. 2007, "Efficient learning of sparse representations with an energy-based model", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 1137–1144.

Ranzato, M. et M. Szummer. 2008b, "Semi-supervised learning of compact document representations with deep networks", dans *ICML*.

Salakhutdinov, R. et G. E. Hinton. 2007a, "Semantic hashing", dans *Proceedings of the 2007 Workshop on Information Retrieval and applications of Graphical Models (SIGIR'07)*, Elsevier, Amsterdam.

Salakhutdinov, R., A. Mnih et G. E. Hinton. 2007b, "Restricted Boltzmann machines for collaborative filtering", dans *Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML'07)*, édité par Z. Ghahramani, ACM, New York, NY, USA, p. 791–798.

Serre, T., G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich et T. Poggio. 2007, "A quantitative theory of immediate visual recognition", *Progress in Brain Research, Computational Neuroscience : Theoretical Insights into Brain Function*, vol. 165, p. 33–56.

Socher, R., C. Lin, A. Y. Ng et C. Manning. 2011, "Learning continuous phrase representations and syntactic parsing with recursive neural networks", dans *ICML'2011*.

Taylor, G. et G. Hinton. 2009, "Factored conditional restricted Boltzmann machines for modeling motion style", dans *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, édité par L. Bottou et M. Littman, ACM, Montreal, Quebec, Canada, p. 1025–1032.

Taylor, G., G. E. Hinton et S. Roweis. 2007, "Modeling human motion using binary latent variables", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, Cambridge, MA, p. 1345–1352.

Utgoff, P. E. et D. J. Stracuzzi. 2002, "Many-layered learning", *Neural Computation*, vol. 14, p. 2497–2539.

Weston, J., F. Ratle et R. Collobert. 2008, "Deep learning via semi-supervised embedding", dans *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, édité par W. W. Cohen, A. McCallum et S. T. Roweis, ACM, New York, NY, USA, p. 1168–1175.

Wolpert, D. H. 1996, "The lack of a priori distinction between learning algorithms", *Neural Computation*, vol. 8, n° 7, p. 1341–1390.

Yao, A. 1985, "Separating the polynomial-time hierarchy by oracles", dans *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, p. 1–10.

5 Graph-based semi-supervised learning

"Label propagation and quadratic criterion"
Y. Bengio, O. Delalleau et N. Le Roux
***Semi-Supervised Learning*, édité par O. Chapelle, B. Schölkopf et A. Zien, MIT Press, p. 193–216, 2006**

"Large-scale algorithms"
O. Delalleau, Y. Bengio et N. Le Roux
***Semi-Supervised Learning*, édité par O. Chapelle, B. Schölkopf et A. Zien, MIT Press, p. 333–341, 2006**

Les données non étiquetées étant dans certaines applications beaucoup plus abondantes que les données étiquetées (c'est souvent le cas par exemple pour les images, vidéos ou textes), l'apprentissage semi-supervisé a connu un essor fulgurant depuis le début des années 2000. Cette recherche a notamment débouché sur le livre *Semi-Supervised Learning* (Chapelle et al., 2006), dont nous avons écrit deux chapitres se concentrant sur l'analyse des techniques d'apprentissage semi-supervisé à base de graphe. Le premier chapitre (*Label propagation and quadratic criterion*) se veut essentiellement une synthèse des différentes techniques proposées jusque là dans la littérature, apportant de nouveaux liens entre elles et les analysant sous un nouvel angle. Le second chapitre (*Large-scale algorithms*) présente une nouvelle technique d'approximation pour améliorer de manière significative l'efficacité computationnelle de ces algorithmes, basée sur une extension proposée dans le premier chapitre. Le contenu de ces deux chapitres (à l'origine dans deux parties distinctes du livre duquel ils sont issus) a été légèrement remanié dans cette thèse afin d'offrir un ensemble plus cohérent. Ils ont été combinés en un seul chapitre, le second (plus court) correspondant à la section 5.6.

Contribution personnelle Ces travaux tirent leur origine de discussions entre les trois auteurs sur les points communs entre plusieurs algorithmes d'apprentissage semi-supervisé à base de graphe. J'ai fait la majeure partie de la revue de littérature, et établi de nouveaux liens entre les méthodes itératives et les méthodes d'optimisation. L'algorithme d'approximation pour rendre l'optimisation plus efficace (section 5.6) est également à mettre à mon actif. La majorité de la partie concernant la malédiction de la dimensionalité (section 5.7.4) a été écrite par Y. Bengio, mais je suis l'auteur de certains des résultats mathématiques qui y sont énoncés.

Abstract Various graph-based semi-supervised learning algorithms have been proposed in the recent literature. They rely on the idea of building a graph whose nodes are data points (labeled and unlabeled) and edges represent similarities between points. Known labels are used to propagate information through the graph in order to label all nodes. In this chapter, we show how these different algorithms can be cast into a common framework where one minimizes a quadratic cost criterion whose closed-form solution is found by solving a linear system of size n (total number of data points). The cost criterion naturally leads to an extension of such algorithms to the inductive setting, where one obtains test samples one at a time: the derived induction formula can be evaluated in $O(n)$ time, which is much more efficient than solving again exactly the linear system (which in general costs $O(kn^2)$ time for a sparse graph where each data point has k neighbors). This induction formula inspired an approximation scheme that significantly reduces the memory and computational requirements of the original algorithm, thus allowing one to take advantage of significantly bigger unlabeled datasets. We also use our inductive formula to show that when the similarity between points satisfies a locality property, then the algorithms are plagued by the curse of dimensionality, with respect to the dimensionality of an underlying manifold.

5.1 Introduction

Many semi-supervised learning algorithms rely on the geometry of the data induced by both labeled and unlabeled examples to improve on supervised methods that use only the labeled data. This geometry can be naturally represented by an empirical graph $\mathbf{g} = (V, E)$ where nodes $V = \{1, \ldots, n\}$ represent the training data and edges E represent similarities between them. These similarities are given by a weight matrix $\mathbf{W}$: $\mathbf{W}_{ij}$ is non-zero iff x_i and x_j are "neighbors", i.e. the edge (i, j) is in E (weighted by $\mathbf{W}_{ij}$). The weight matrix $\mathbf{W}$ can be for instance the k-nearest neighbor matrix: $\mathbf{W}_{ij} = 1$ iff x_i is among the k nearest neighbors of x_j or vice-versa (and is 0 otherwise). Another typical weight matrix is given by the Gaussian kernel of width σ:

$$\mathbf{W}_{ij} = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}. \tag{5.1}$$

In general, we assume that the weight $\mathbf{W}_{ij}$ is given by a symmetric positive function W_X (possibly dependent on the dataset $X = (x_1, \ldots, x_n)$) by $\mathbf{W}_{ij} = W_X(x_i, x_j) \geq 0$. This functional view will be useful in the inductive setting (section 5.4).

This chapter is organized as follows. In section 5.2 we present algorithms based on the idea of using the graph structure to spread labels from labeled examples to the whole dataset (Szummer et al., 2001; Zhu et al., 2002; Zhou et al., 2004; Zhu et al., 2003). An alternative approach originating from

smoothness considerations yields algorithms based on graph regularization, which naturally leads to a regularization term based on the graph Laplacian (Belkin et al., 2003; Joachims, 2003; Zhou et al., 2004; Zhu et al., 2003; Belkin et al., 2004; Delalleau et al., 2005). This approach, detailed in section 5.3, is then shown to be tightly linked to the previous label propagation algorithms. In sections 5.4 and 5.5 we present two extensions of these algorithms: first a simple way to turn a number of them, originally designed for the transductive setting, into induction algorithms, then a method to better balance classes using prior information about the classes distribution. Section 5.6 presents a new approximation scheme to speed-up learning – inspired by our induction extension – and empirically demonstrates its effectiveness. Finally, section 5.7 explores theoretical limitations of these methods which, being based mostly on the local geometry of the data in small neighborhoods, are subject to the curse of dimensionality when the intrinsic dimension of the underlying distribution (the dimensionality of the manifold near which it concentrates) increases, when this manifold is far from being flat.

5.2 Label propagation on a similarity graph

5.2.1 Iterative algorithms

Given the graph $\mathbf{g}$, a simple idea for semi-supervised learning is to *propagate labels* on the graph. Starting with nodes $1, 2, \ldots, \ell$ labeled* with their known label (1 or -1) and nodes $\ell+1, \ldots, n$ labeled with 0, each node starts to propagate its label to its neighbors, and the process is repeated until convergence.

Algorithm 1 – Label propagation (Zhu et al., 2002)

Compute affinity matrix $\mathbf{W}$ from eq. 5.1
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Initialize $\hat{Y}^{(0)} \leftarrow (y_1, \ldots, y_\ell, 0, 0, \ldots, 0)$
Iterate
1. $\hat{Y}^{(t+1)} \leftarrow \mathbf{D}^{-1}\mathbf{W}\hat{Y}^{(t)}$
2. $\hat{Y}_\ell^{(t+1)} \leftarrow Y_\ell$

until convergence to $\hat{Y}^{(\infty)}$
Label point x_i by the sign of $\hat{y}_i^{(\infty)}$

* If there are $M > 2$ classes, one can label each node i with a M-dimensional vector (one-hot for labeled samples, i.e. with 0 everywhere except a 1 at index y_i = class of x_i), and use the same algorithms in a one-versus-rest fashion. We consider here the classification case, but extension to regression is straightforward for most algorithms, which treat labels as real values.

An algorithm of this kind has been proposed by Zhu et al. (2002), and is described in algorithm 1. Estimated labels on both labeled and unlabeled data are denoted by $\hat{Y} = (\hat{Y}_\ell, \hat{Y}_u)$, where $\hat{Y}_\ell$ may be allowed to differ from the given labels $Y_\ell = (y_1, \dots, y_\ell)$. In this particular algorithm, $\hat{Y}_\ell$ is constrained to be equal to Y_ℓ. We propose in algorithm 2 below a slightly different label propagation scheme (originally inspired from the Jacobi iterative method for linear systems), similar to the previous algorithm except that:

- we advocate forcing $\mathbf{W}_{ii} = 0$, which often works better,
- we allow $\hat{Y}_\ell \neq Y_\ell$ (which may be useful e.g. when classes overlap), and
- we use an additional regularization term ε for better numerical stability.

Algorithm 2 – Label propagation (inspired from Jacobi iterations)

Compute an affinity matrix $\mathbf{W}$ such that $\mathbf{W}_{ii} = 0$
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Choose a parameter $\alpha \in (0, 1)$ and a small $\varepsilon > 0$
$\mu \leftarrow \frac{\alpha}{1-\alpha} \in (0, +\infty)$
Compute the diagonal matrix $\mathbf{A}$ by $\mathbf{A}_{ii} \leftarrow \mathbf{1}_{i \leq \ell} + \mu \mathbf{D}_{ii} + \mu\varepsilon$
Initialize $\hat{Y}^{(0)} \leftarrow (y_1, \dots, y_\ell, 0, 0, \dots, 0)$
Iterate $\hat{Y}^{(t+1)} \leftarrow \mathbf{A}^{-1}(\mu \mathbf{W} \hat{Y}^{(t)} + \hat{Y}^{(0)})$ until convergence to $\hat{Y}^{(\infty)}$
Label point x_i by the sign of $\hat{y}_i^{(\infty)}$

The iteration step of algorithm 2 is written for labeled examples ($i \leq \ell$):

$$\hat{y}_i^{(t+1)} \leftarrow \frac{\sum_j \mathbf{W}_{ij} \hat{y}_j^{(t)} + \frac{1}{\mu} y_i}{\sum_j \mathbf{W}_{ij} + \frac{1}{\mu} + \varepsilon} \tag{5.2}$$

and for unlabeled examples ($\ell + 1 \leq i \leq n$):

$$\hat{y}_i^{(t+1)} \leftarrow \frac{\sum_j \mathbf{W}_{ij} \hat{y}_j^{(t)}}{\sum_j \mathbf{W}_{ij} + \varepsilon}. \tag{5.3}$$

These two equations can be seen as a weighted average of the neighbors' current labels, where for labeled examples we also add the initial label (whose weight is inversely proportional to the parameter μ). The ε parameter is a regularization term to prevent numerical problems when the denominator becomes too small. The convergence of this algorithm follows from the convergence of the Jacobi iteration method for a specific linear system, and will be discussed in section 5.3.3.

Another similar label propagation algorithm was given by Zhou et al. (2004): at each step a node i receives a contribution from its neighbors j (weighted by the normalized weight of the edge (i, j)), and an additional

small contribution given by its initial value. This process is detailed in algorithm 3 below (the name "label spreading" was inspired from the terminology used by Zhou et al. (2004)). Compared to algorithm 2, it corresponds to the minimization of a slightly different cost criterion, maybe not as intuitive: this will be studied later in sections 5.3.2 and 5.3.3.

Algorithm 3 – Label spreading (Zhou et al., 2004)

Compute the affinity matrix $\mathbf{W}$ from eq. 5.1 for $i \neq j$ (and $\mathbf{W}_{ii} \leftarrow 0$)
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Compute the normalized graph Laplacian $\mathcal{L} \leftarrow \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$
Initialize $\hat{Y}^{(0)} \leftarrow (y_1, \ldots, y_\ell, 0, 0, \ldots, 0)$
Choose a parameter $\alpha \in [0, 1)$
Iterate $\hat{Y}^{(t+1)} \leftarrow \alpha\mathcal{L}\hat{Y}^{(t)} + (1-\alpha)\hat{Y}^{(0)}$ until convergence to $\hat{Y}^{(\infty)}$
Label point x_i by the sign of $\hat{y}_i^{(\infty)}$

The proof of convergence of algorithm 3 is simple (Zhou et al., 2004). The iteration equation being $\hat{Y}^{(t+1)} \leftarrow \alpha\mathcal{L}\hat{Y}^{(t)} + (1-\alpha)\hat{Y}^{(0)}$, we have

$$\hat{Y}^{(t+1)} = (\alpha\mathcal{L})^t\hat{Y}^{(0)} + (1-\alpha)\sum_{i=0}^{t}(\alpha\mathcal{L})^i\hat{Y}^{(0)}.$$

The matrix $\mathcal{L}$ being similar to $\mathbf{P} = \mathbf{D}^{-1}\mathbf{W} = \mathbf{D}^{-1/2}\mathcal{L}\mathbf{D}^{1/2}$, it has the same eigenvalues. Since $\mathbf{P}$ is a stochastic matrix by construction, its eigenvalues are in $[-1, 1]$, and consequently the eigenvalues of $\alpha\mathcal{L}$ are in $(-1, 1)$ (remember $\alpha < 1$). It follows that when $t \to \infty$, $(\alpha\mathcal{L})^t \to 0$ and

$$\sum_{i=0}^{t}(\alpha\mathcal{L})^i \to (I - \alpha\mathcal{L})^{-1}$$

so that

$$\hat{Y}^{(t)} \to \hat{Y}^{(\infty)} = (1-\alpha)(I - \alpha\mathcal{L})^{-1}\hat{Y}^{(0)}. \tag{5.4}$$

The convergence rate of these three algorithms depends on specific properties of the graph such as the eigenvalues of its Laplacian. In general, we can expect it to be at worst on the order of $O(kn^2)$, where k is the number of neighbors of a point in the graph. In the case of a dense weight matrix, the computational time is thus cubic in n.

5.2.2 Markov random walks

A different algorithm based on label propagation on the similarity graph was proposed earlier by Szummer et al. (2001). They consider Markov random walks on the graph with transition probabilities from i to j

$$p_{ij} = \frac{\mathbf{W}_{ij}}{\sum_k \mathbf{W}_{ik}} \tag{5.5}$$

in order to estimate probabilities of class labels. Here, $\mathbf{W}_{ij}$ is given by a Gaussian kernel for neighbors and 0 for non-neighbors, and $\mathbf{W}_{ii} = 1$ (but one could also use $\mathbf{W}_{ii} = 0$). Each data point x_i is associated with a probability $P(y = 1|i)$ of being of class 1. Given a point x_k, we can compute the probability $P^{(t)}(y_{start} = 1|k)$ that we started from a point of class $y_{start} = 1$ given that we arrived to x_k after t steps of random walk by

$$P^{(t)}(y_{start} = 1|k) = \sum_{i=1}^{n} P(y = 1|i) P_{0|t}(i|k)$$

where $P_{0|t}(i|k)$ is the probability that we started from x_i given that we arrived to k after t steps of random walk (this probability can be computed from the p_{ij}). x_k is then classified to 1 if $P^{(t)}(y_{start} = 1|k) > 0.5$, and to -1 otherwise. The authors propose two methods to estimate the class probabilities $P(y = 1|i)$. One is based on an iterative EM algorithm, the other on maximizing a margin-based criterion, which leads to a closed-form solution (Szummer et al., 2001).

It turns out that this algorithm's performance depends crucially on the hyper-parameter t (the length of the random walk). This parameter has to be chosen by cross-validation (if enough data is available) or heuristically (it corresponds intuitively to the amount of propagation we allow in the graph, i.e. to the scale of the clusters we are interested in). An alternative way of using random walks on the graph is to assign to point x_i a label depending on the probability of arriving to a positively labeled example when performing a random walk *starting from x_i and until a labeled example is found* (Zhu et al., 2002, 2003). The length of the random walk is not constrained anymore to a fixed value t. In the following, we will show that this probability, denoted by $P(y_{end} = 1|i)$, is equal (up to a shift and scaling) to the label obtained with algorithm 1 (this is similar to the proof by Zhu et al., 2002).

When x_i is a labeled example, $P(y_{end} = 1|i) = \mathbf{1}_{y_i=1}$, and when it is unlabeled we have the relation

$$P(y_{end} = 1|i) = \sum_{j=1}^{n} P(y_{end} = 1|j) p_{ij} \tag{5.6}$$

with the p_{ij} computed as in eq. 5.5. Let us consider the matrix $\mathbf{P} = \mathbf{D}^{-1}\mathbf{W}$, i.e. such that $\mathbf{P}_{ij} = p_{ij}$. We will denote $\hat{z}_i = P(y_{end} = 1|i)$ and $\hat{Z} = (\hat{Z}_\ell, \hat{Z}_u)$ the corresponding vector split into its labeled and unlabeled parts. Similarly, the matrices $\mathbf{D}$ and $\mathbf{W}$ can be split into four parts:

$$\begin{aligned} \mathbf{D} &= \begin{pmatrix} \mathbf{D}_{\ell\ell} & 0 \\ 0 & \mathbf{D}_{uu} \end{pmatrix} \\ \mathbf{W} &= \begin{pmatrix} \mathbf{W}_{\ell\ell} & \mathbf{W}_{\ell u} \\ \mathbf{W}_{\ell\ell} & \mathbf{W}_{uu} \end{pmatrix} \end{aligned}$$

Equation 5.6 can then be written

$$\begin{aligned}\hat{Z}_u &= \left(\mathbf{D}_{uu}^{-1}\mathbf{W}_{\ell\ell} \mid \mathbf{D}_{uu}^{-1}\mathbf{W}_{uu}\right)\begin{pmatrix}\hat{Z}_\ell \\ \hat{Z}_u\end{pmatrix} \\ &= \mathbf{D}_{uu}^{-1}\left(\mathbf{W}_{\ell\ell}\hat{Z}_\ell + \mathbf{W}_{uu}\hat{Z}_u\right)\end{aligned}$$

which leads to the linear system

$$L_{uu}\hat{Z}_u = \mathbf{W}_{\ell\ell}\hat{Z}_\ell \tag{5.7}$$

where $L = \mathbf{D} - \mathbf{W}$ is the un-normalized graph Laplacian. Since $\hat{Z}_\ell$ is known ($\hat{z}_i = 1$ if $y_i = 1$, and 0 otherwise), this linear system can be solved in order to find the probabilities $\hat{Z}_u$ on unlabeled examples. Note that if $(\hat{Z}_u, \hat{Z}_\ell)$ is a solution of eq. 5.7, then $(\hat{Y}_u, \hat{Y}_\ell)$ is also a solution, with

$$\begin{aligned}\hat{Y}_u &= 2\hat{Z}_u - (1, 1, \ldots, 1)^\top \\ \hat{Y}_\ell &= 2\hat{Z}_\ell - (1, 1, \ldots, 1)^\top = Y_\ell\end{aligned}$$

This allows us to rewrite the linear system eq. 5.7 in terms of the vector of original labels Y_ℓ as follows:

$$L_{uu}\hat{Y}_u = \mathbf{W}_{\ell\ell}\hat{Y}_\ell \tag{5.8}$$

with the sign of each element y_i of $\hat{Y}_u$ giving the estimated label of x_i (which is equivalent to comparing $\hat{z}_i$ to a 0.5 threshold).

The solution of this random walk algorithm is thus given in closed-form by a linear system, which turns out to be equivalent to iterative algorithm 1 (or equivalently, algorithm 2 when $\mu \to 0$ and $\varepsilon = 0$), as we will see in section 5.3.4.

5.3 Quadratic cost criterion

In this section, we investigate semi-supervised learning by minimization of a cost function derived from the graph $\mathbf{g}$. Such methods will be shown to be equivalent to label propagation algorithms presented in the previous section.

5.3.1 Regularization on graphs

The problem of semi-supervised learning on the graph $\mathbf{g}$ consists in finding a labeling of the graph that is consistent with both the initial (incomplete) labeling and the geometry of the data induced by the graph structure (edges and weights $\mathbf{W}$). Given a labeling $\hat{Y} = (\hat{Y}_\ell, \hat{Y}_u)$, consistency with the initial labeling can be measured e.g. by

$$\sum_{i=1}^{\ell}(\hat{y}_i - y_i)^2 = \|\hat{Y}_\ell - Y_\ell\|^2. \tag{5.9}$$

On the other hand, consistency with the geometry of the data, which follows from the Smoothness (or Manifold) Assumption discussed in section 2.5, motivates a penalty term of the form

$$\begin{aligned}\frac{1}{2}\sum_{i,j=1}^{n}\mathbf{W}_{ij}(\hat{y}_i-\hat{y}_j)^2 &= \frac{1}{2}\left(2\sum_{i=1}^{n}\hat{y}_i^2\sum_{j=1}^{n}\mathbf{W}_{ij}-2\sum_{i,j=1}^{n}\mathbf{W}_{ij}\hat{y}_i\hat{y}_j\right)\\ &= \hat{Y}^\top(\mathbf{D}-\mathbf{W})\hat{Y}\\ &= \hat{Y}^\top L\hat{Y} \qquad (5.10)\end{aligned}$$

with $L = \mathbf{D} - \mathbf{W}$ the un-normalized graph Laplacian. This means we penalize rapid changes in $\hat{Y}$ between points that are close (as given by the similarity matrix $\mathbf{W}$).

Various algorithms have been proposed based on such considerations. Zhu et al. (2003) force the labels on the labeled data ($\hat{Y}_\ell = Y_\ell$) then minimize eq. 5.10 over $\hat{Y}_u$. However, if there is noise in the available labels, it may be beneficial to allow the algorithm to re-label the labeled data (this could also help generalization in a noise-free setting when classes overlap, and for instance a positive sample may have been drawn from a region of space mainly filled with negative samples). This observation leads to a more general cost criterion involving a trade-off between eq. 5.9 and 5.10 (Belkin et al., 2004; Delalleau et al., 2005). A small regularization term can also be added in order to prevent degenerate situations, for instance when the graph $\mathbf{g}$ has a connected component with no labeled sample. We thus obtain the following general labeling cost*:

$$C(\hat{Y}) = \|\hat{Y}_\ell - Y_\ell\|^2 + \mu\hat{Y}^\top L\hat{Y} + \mu\varepsilon\|\hat{Y}\|^2. \qquad (5.11)$$

Joachims (2003) obtained the same kind of cost criterion from the perspective of spectral clustering. The unsupervised minimization of $\hat{Y}^\top L\hat{Y}$ (under the constraints $\hat{Y}^\top\mathbf{1} = 0$ and $\|\hat{Y}\|^2 = n$, with $\mathbf{1}$ a vector filled with 1) is a relaxation of the NP-hard problem of minimizing the normalized cut of the graph $\mathbf{g}$, i.e. splitting $\mathbf{g}$ into two subsets $\mathbf{g}^+ = (V^+, E^+)$ and $\mathbf{g}^- = (V^-, E^-)$ such as to minimize

$$\frac{\sum_{i\in V^+, j\in V^-} W_{ij}}{|V^+||V^-|}$$

where the normalization by $|V^+||V^-|$ favors balanced splits. Based on this approach, Joachims (2003) introduced an additional cost which corresponds to our part $\|\hat{Y}_\ell - Y_\ell\|^2$ of the cost given by eq. 5.11, in order to turn this unsupervised minimization into a semi-supervised transductive algorithm (called Spectral Graph Transducer). Note however that although very similar, the solution obtained differs from the straighforward minimization of eq. 5.11 since:

*Belkin et al. (2004) first center the vector Y_ℓ and also constrain $\hat{Y}$ to be centered: these restrictions are needed to obtain theoretical bounds on the generalization error, and will not be discussed in this chapter.

- the labels are not necessarily +1 and -1, but depend on the ratio of the number of positive examples over the number of negative examples (this follows from the normalized cut optimization),
- the constraint $\|\hat{Y}\|^2 = n$ used in the unsupervised setting remains, thus leading to an eigenvalue problem instead of the direct quadratic minimization that will be studied in the next section,
- the eigenspectrum of the graph Laplacian is normalized by replacing the ordered Laplacian eigenvalues by a monotonically increasing function, in order to focus on the ranking among the smallest cuts and abstract, for example, from different magnitudes of edge weights.

Belkin et al. (2003) also proposed a semi-supervised algorithm based on the same idea of graph regularization, but using a regularization criterion different from the quadratic penalty term of eq. 5.10. It consists in taking advantage of properties of the graph Laplacian L, which can be seen as an operator on functions defined on nodes of the graph $\mathbf{g}$. The graph Laplacian is closely related to the Laplacian on the manifold, whose eigenfunctions provide a basis for the Hilbert space of $\mathcal{L}^2$ functions on the manifold (Rosenberg, 1997). Eigenvalues of the eigenfunctions provide a measure of their smoothness on the manifold (low eigenvalues correspond to smoother functions, with the eigenvalue 0 being associated with the constant function). Projecting any function in $\mathcal{L}^2$ on the first p eigenfunctions (sorted by order of increasing eigenvalue) is thus a way of smoothing it on the manifold. The same principle can be applied to our graph setting, thus leading to algorithm 4 (Belkin et al., 2003) below.

Algorithm 4 – Laplacian regularization (Belkin et al., 2003)

Compute affinity matrix $\mathbf{W}$ (with $\mathbf{W}_{ii} = 0$)
Compute the diagonal degree matrix $\mathbf{D}$ by $\mathbf{D}_{ii} \leftarrow \sum_j W_{ij}$
Compute the un-normalized graph Laplacian $L = \mathbf{D} - \mathbf{W}$
Compute the p eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_p$ corresponding to the p smallest eigenvalues of L
Minimize over $a_1, \ldots, a_p$ the quadratic criterion $\sum_{i=1}^{\ell} \left(y_i - \sum_{j=1}^{p} a_j \mathbf{e}_{j,i} \right)^2$
Label point x_i $(1 \leq i \leq n)$ by the sign of $\sum_{j=1}^{p} a_j \mathbf{e}_{j,i}$

It consists in computing the first p eigenvectors of the graph Laplacian (each eigenvector can be seen as the corresponding eigenfunction applied on training points), then finding the linear combination of these eigenvectors that best predicts the labels (in the mean-squared sense). The idea is to obtain a smooth function (in the sense that it is a linear combination of the p smoothest eigenfunctions of the Laplacian operator on the manifold) that fits the labeled data. This algorithm does not explicitely correspond to the minimization of a non-parametric quadratic criterion such as eq. 5.11

and thus is not covered by the connection shown in section 5.3.3 with label propagation algorithms, but one must keep in mind that it is based on similar graph regularization considerations and offers competitive classification performance.

5.3.2 Optimization framework

In order to minimize the quadratic criterion of eq. 5.11, we can compute its derivative with respect to $\hat{Y}$. We will denote by $\mathbf{S}$ the diagonal matrix $(n \times n)$ given by $\mathbf{S}_{ii} = \mathbf{1}_{i \leq \ell}$, so that the first part of the cost can be re-written $\|\mathbf{S}\hat{Y} - \mathbf{S}Y\|^2$. The derivative of the criterion is then:

$$\begin{aligned} \frac{1}{2}\frac{\partial C(\hat{Y})}{\partial \hat{Y}} &= \mathbf{S}(\hat{Y} - Y) + \mu L\hat{Y} + \mu\varepsilon\hat{Y} \\ &= (\mathbf{S} + \mu L + \mu\varepsilon I)\,\hat{Y} - \mathbf{S}Y. \end{aligned}$$

The second derivative is:

$$\frac{1}{2}\frac{\partial^2 C(\hat{Y})}{\partial \hat{Y}\partial \hat{Y}^\top} = \mathbf{S} + \mu L + \mu\varepsilon I$$

which is a positive definite matrix when $\varepsilon > 0$ (L is positive semi-definite as shown by eq. 5.10). This ensures the cost is minimized when the derivative is set to 0, i.e.

$$\hat{Y} = (\mathbf{S} + \mu L + \mu\varepsilon I)^{-1}\,\mathbf{S}Y. \tag{5.12}$$

This shows how the new labels can be obtained by a simple matrix inversion. It is interesting to note that this matrix does not depend on the original labels, but only on the graph Laplacian L: the way labels are "propagated" to the rest of the graph is entirely determined by the graph structure.

An alternative (and very similar) criterion was proposed by Zhou et al. (2004), and can be written:

$$\begin{aligned} C'(\hat{Y}) &= \|\hat{Y} - \mathbf{S}Y\|^2 + \frac{\mu}{2}\sum_{i,j}\mathbf{W}_{ij}\left(\frac{\hat{y}_i}{\sqrt{\mathbf{D}_{ii}}} - \frac{\hat{y}_j}{\sqrt{\mathbf{D}_{jj}}}\right)^2 \qquad (5.13) \\ &= \|\hat{Y}_\ell - Y_\ell\|^2 + \|\hat{Y}_u\|^2 + \mu\hat{Y}^\top(I - \mathcal{L})\,\hat{Y} \\ &= \|\hat{Y}_\ell - Y_\ell\|^2 + \|\hat{Y}_u\|^2 + \mu\hat{Y}^\top\mathbf{D}^{-1/2}(\mathbf{D} - \mathbf{W})\,\mathbf{D}^{-1/2}\hat{Y} \\ &= \|\hat{Y}_\ell - Y_\ell\|^2 + \|\hat{Y}_u\|^2 + \mu(\mathbf{D}^{-1/2}\hat{Y})^\top L(\mathbf{D}^{-1/2}\hat{Y}) \end{aligned}$$

This criterion C' has two main differences with C (eq. 5.11):

- the term $\|\hat{Y} - \mathbf{S}Y\|^2 = \|\hat{Y}_\ell - Y_\ell\|^2 + \|\hat{Y}_u\|^2$ not only tries to fit the given labels, but also to pull to 0 labels of unlabeled samples (this is a similar but stronger regularization compared to the term $\mu\varepsilon\|\hat{Y}\|^2$ in the cost C), and

- labels are normalized by the square root of the degree matrix elements $\mathbf{D}_{ii}$ when computing their similarity. This normalization may not be intuitive, but is necessary for the equivalence with the label propagation algorithm 3, as seen below.

5.3.3 Links with label propagation

The optimization algorithms presented above turn out to be equivalent to the label propagation methods from section 5.2. Let us first study the optimization of the cost $C(\hat{Y})$ from eq. 5.11. The optimum $\hat{Y}$ is given by eq. 5.12, but another way to obtain this solution, besides matrix inversion, is to solve the linear system using one of the many standard methods available. We focus here on the simple Jacobi iteration method (Saad, 1996), which consists in solving for each component iteratively. Given the system

$$\mathbf{M}x = b \tag{5.14}$$

the approximate solution at step $t+1$ is

$$x_i^{(t+1)} = \frac{1}{\mathbf{M}_{ii}} \left(b - \sum_{j \neq i} \mathbf{M}_{ij} x_j^{(t)} \right). \tag{5.15}$$

Applying this formula with $x := \hat{Y}$, $b := \mathbf{S}Y$ and $\mathbf{M} := \mathbf{S} + \mu L + \mu\varepsilon I$, we obtain:

$$\hat{y}_i^{(t+1)} = \frac{1}{\mathbf{1}_{i \leq \ell} + \mu \sum_{j \neq i} \mathbf{W}_{ij} + \mu\varepsilon} \left(\mathbf{1}_{i \leq \ell} y_i + \mu \sum_{j \neq i} \mathbf{W}_{ij} \hat{y}_j^{(t)} \right)$$

i.e. exactly the update equations 5.2 and 5.3 used in algorithm 2. Convergence of this iterative algorithm is guaranteed by the following theorem (Saad, 1996): if the matrix $\mathbf{M}$ is strictly diagonally dominant, the Jacobi iteration (eq. 5.15) converges to the solution of the linear system (eq. 5.14). A matrix $\mathbf{M}$ is strictly diagonally dominant iff $|\mathbf{M}_{ii}| > \sum_{j \neq i} |\mathbf{M}_{ij}|$, which is clearly the case for the matrix $\mathbf{S} + \mu L + \mu\varepsilon I$ (remember $L = \mathbf{D} - \mathbf{W}$ with $\mathbf{D}_{ii} = \sum_{i \neq j} \mathbf{W}_{ij}$, and all $\mathbf{W}_{ij} \geq 0$). Note that this condition also guarantees the convergence of the Gauss-Seidel iteration, which is the same as the Jacobi iteration except that updated coordinates $x_i^{(t+1)}$ are used in the computation of $x_j^{(t+1)}$ for $j > i$. This means we can apply equations 5.2 and 5.3 with $\hat{Y}^{(t+1)}$ and $\hat{Y}^{(t)}$ sharing the same storage.

To show the equivalence between algorithm 3 and the minimization of C' given in eq. 5.13, we compute its derivative with respect to $\hat{Y}$:

$$\frac{1}{2} \frac{\partial C'(\hat{Y})}{\partial \hat{Y}} = \hat{Y} - \mathbf{S}Y + \mu \left(\hat{Y} - \mathcal{L}\hat{Y} \right)$$

and is zero iff

$$\hat{Y} = \left((1+\mu)I - \mu\mathcal{L} \right)^{-1} \mathbf{S}Y$$

which is the same equation as eq. 5.4 with $\mu = \alpha/(1-\alpha)$, up to a positive factor (which has no effect on the classification since we use only the sign).

5.3.4 Limit case and analogies

It is interesting to study the limit case when $\mu \to 0$. In this section we will set $\varepsilon = 0$ to simplify notations, but one should keep in mind that it is usually better to use a small positive value for regularization. When $\mu \to 0$, the cost of eq. 5.11 is dominated by $\|\hat{Y}_\ell - Y_\ell\|^2$. Intuitively, this corresponds to

1. forcing $\hat{Y}_\ell = Y_\ell$, then
2. minimizing $\hat{Y}^\top L \hat{Y}$.

Writing $\hat{Y} = (Y_\ell, \hat{Y}_u)$ (i.e. $\hat{Y}_\ell = Y_\ell$) and

$$L = \begin{pmatrix} L_{\ell\ell} & L_{\ell u} \\ L_{\ell\ell} & L_{uu} \end{pmatrix}$$

the minimization of $\hat{Y}^\top L \hat{Y}$ with respect to $\hat{Y}_u$ leads to

$$L_{\ell\ell} Y_\ell + L_{uu} \hat{Y}_u = 0 \Rightarrow \hat{Y}_u = -L_{uu}^{-1} L_{\ell\ell} Y_\ell. \tag{5.16}$$

If we consider now eq. 5.12 where $\hat{Y}_\ell$ is not constrained anymore, when $\varepsilon = 0$ and $\mu \to 0$, using the continuity of the inverse matrix application at I, we obtain that

$$\begin{aligned} \hat{Y}_\ell &\to Y_\ell \\ \hat{Y}_u &= -L_{uu}^{-1} L_{\ell\ell} \hat{Y}_\ell \end{aligned}$$

which, as expected, gives us the same solution as eq. 5.16.

Analogy with Markov random walks

In section 5.2.2, we presented an algorithm of label propagation based on Markov random walks on the graph, leading to the linear system of eq. 5.8. It is immediate to see that this system is exactly the same as the one obtained in eq. 5.16. The equivalence of the solutions discussed in the previous section between the linear system and iterative algorithms thus shows that the random walk algorithm described in section 5.2.2 is equivalent to the iterative algorithm 2 when $\mu \to 0$, i.e. when we keep the original labels instead of iteratively updating them by eq. 5.2.

Analogy with electric networks

Zhu et al. (2003) also link this solution to heat kernels and give an electric network interpretation taken from Doyle et al. (1984), which we will present

here. This analogy is interesting as it gives a physical interpretation to the optimization and label propagation framework studied in this chapter. Let us consider an electric network built from the graph $\mathbf{g}$ by adding resistors with conductance $\mathbf{W}_{ij}$ between nodes i and j (the conductance is the inverse of the resistance). The positive labeled nodes are connected to a positive voltage source ($+1V$), the negative ones to a negative voltage source ($-1V$), and we want to compute the voltage on the *unlabeled* nodes (i.e. their label). Denoting the intensity between i and j by I_{ij}, and the voltage by $V_{ij} = \hat{y}_j - \hat{y}_i$, we will use Ohm's law

$$I_{ij} = \mathbf{W}_{ij} V_{ij} \tag{5.17}$$

and Kirchoff's law on an unlabeled node $i > \ell$:

$$\sum_j I_{ij} = 0. \tag{5.18}$$

Kirchoff's law states that the sum of currents flowing out from i (such that $I_{ij} > 0$) is equal to the sum of currents flowing into i ($I_{ij} < 0$). Here, it is only useful to apply it to unlabeled nodes as the labeled ones are connected to a voltage source, and thus receive some unknown (and uninteresting) current. Using eq. 5.17, we can rewrite eq. 5.18

$$\begin{aligned} 0 &= \sum_j \mathbf{W}_{ij}(\hat{y}_j - \hat{y}_i) \\ &= \sum_j \mathbf{W}_{ij}\hat{y}_j - \hat{y}_i \sum_j \mathbf{W}_{ij} \\ &= (\mathbf{W}\hat{Y} - \mathbf{D}\hat{Y})_i \\ &= -(L\hat{Y})_i \end{aligned}$$

and since this is true for all $i > \ell$, it is equivalent in matrix notations to

$$L_{\ell\ell} Y_\ell + L_{uu} \hat{Y}_u = 0$$

which is exactly eq. 5.16. Thus the solution of the limit case (when labeled examples are forced to keep their given label) is given by the voltage in an electric network where labeled nodes are connected to voltage sources and resistors correspond to weights in the graph $\mathbf{g}$.

5.4 From transduction to induction

The previous algorithms all follow the transduction setting presented in section 1.2.3. However, it could happen that one needs an inductive algorithm, for instance in a situation where new test examples are presented one at a time and solving the linear system turns out to be too expensive. In such

a case, the cost criterion (eq. 5.11) naturally leads to an induction formula that can be computed in $O(n)$ time. Assuming that labels $\hat{y}_1, \ldots, \hat{y}_n$ have already been computed by one of the algorithms above, and we want the label $\hat{y}$ of a new point x: we can minimize $C(\hat{y}_1, \ldots, \hat{y}_n, \hat{y})$ only with respect to this new label $\hat{y}$, i.e. minimize

$$\text{constant} + \mu \left(\sum_j W_X(x, x_j)(\hat{y} - \hat{y}_j)^2 + \varepsilon \hat{y}^2 \right)$$

where W_X is the (possibly data-dependent) function that generated the matrix $\mathbf{W}$ on $X = (x_1, \ldots, x_n)$. Setting to zero the derivative with respect to $\hat{y}$ directly yields

$$\hat{y} = \frac{\sum_j W_X(x, x_j)\hat{y}_j}{\sum_j W_X(x, x_j) + \varepsilon} \tag{5.19}$$

a simple inductive formula whose computational requirements scale linearly with the number of samples already seen.

It is interesting to note that, if W_X is the k-nearest neighbor function, eq. 5.19 reduces to k-nearest neighbor classification. Similarly, if W_X is the Gaussian kernel (eq. 5.1), it is equivalent to the formula for Parzen windows or Nadaraya-Watson non-parametric regression (Nadaraya, 1964; Watson, 1964). However, we use in this formula the *learned* predictions on the labeled and unlabeled examples *as if they were observed training values*, instead of relying only on labeled data.

5.5 Incorporating class prior knowledge

From the beginning of the chapter, we have assumed that the class label is given by the sign of $\hat{y}$. Such a rule works well when classes are well separated and balanced. However, if this is not the case (which is likely to happen with real-world datasets), the classification resulting from the label propagations algorithms studied in this chapter may not reflect the prior class distribution.

A way to solve this problem is to perform *class mass normalization* (Zhu et al., 2003), i.e. to rescale classes so that their respective weights over unlabeled examples match the prior class distribution (estimated from labeled examples). Until now, we had been using a scalar label $\hat{y}_i \in [-1, 1]$, which is handy in the binary case. In this section, for the sake of clarity, we will use a M-dimensional vector (M being the number of classes), with each element $\hat{y}_{i,k}$ between 0 and 1 giving a score (or weight) for class k (see also the footnote at the beginning of this chapter). For instance, in the binary case, a scalar $\hat{y}_i \in [-1, 1]$ would be represented by the vector $\left(\frac{1}{2}(1 + \hat{y}_i), \frac{1}{2}(1 - \hat{y}_i)\right)^\top$, where the second element would be the score for class -1.

Class mass normalization works as follows. Let us denote by p_k the prior probability of class k obtained from the *labeled* examples, i.e.

$$p_k = \frac{1}{\ell} \sum_{i=1}^{\ell} y_{i,k}.$$

The *mass* of class k as given by our algorithm will be the average of *estimated* weights of class k over unlabeled examples, i.e.

$$m_k = \frac{1}{u} \sum_{i=\ell+1}^{n} \hat{y}_{i,k}.$$

Class mass normalization consists in scaling each class k by the factor

$$w_k = \frac{p_k}{m_k}$$

i.e. to classify x_i in the class given by $\operatorname{argmax}_k w_k \hat{y}_{i,k}$ (instead of the simpler decision function $\operatorname{argmax}_k \hat{y}_{i,k}$, equivalent to $\operatorname{sign}(\hat{y}_i)$ in the scalar binary case studied in the previous sections). The goal is to make the scaled masses match the prior class distribution, i.e. after normalization we have that for all k

$$\frac{w_k m_k}{\sum_{j=1}^{M} w_j m_j} = p_k.$$

In general, such a scaling gives a better classification performance *when there are enough labeled data* to accurately estimate the class distribution, and *when the unlabeled data come from the same distribution*. Note also that if there is a m such that each class mass is $m_k = mp_k$, i.e. the masses already reflect the prior class distribution, then the class mass normalization step has no effect, as $w_k = m^{-1}$ for all k.

5.6 Large-scale algorithms

The graph-based semi-supervised algorithms presented previously do not scale well to very large datasets. In this section, we propose an approximation method that significantly reduces the computational and memory requirements of such algorithms.

5.6.1 The scale problem

Let us first recall the main notations we will be using:

- $Y = (Y_\ell, Y_u)$ is the set of "original" labels on labeled and unlabeled points (here, Y_u is filled with 0),
- $\hat{Y} = (\hat{Y}_\ell, \hat{Y}_u)$ is the set of estimated labels on labeled and unlabeled points,

- $\hat{y}$ is the function to learn, which assigns a label to each point of the input space,
- $\hat{y}(x_i) = \hat{y}_i$ is the value of the function $\hat{y}$ on training points (labeled and unlabeled).

In section 5.3, we defined the following quadratic cost (eq. 5.11):

$$C(\hat{Y}) = \|\hat{Y}_\ell - Y_\ell\|^2 + \mu \hat{Y}^\top L \hat{Y} + \mu\varepsilon\|\hat{Y}\|^2. \tag{5.20}$$

Minimizing this cost gives rise to a linear system in $\hat{Y}$ with regularization hyper-parameters μ and ε:

$$(\mathbf{S} + \mu L + \mu\varepsilon I)\,\hat{Y} = \mathbf{S}Y \tag{5.21}$$

where $\mathbf{S}$ is the $(n \times n)$ diagonal matrix defined by $\mathbf{S}_{ii} = \mathbf{1}_{i \leq \ell}$, and $L = \mathbf{D} - \mathbf{W}$ is the un-normalized graph Laplacian. This linear system can be solved to obtain the value of $\hat{y}_i$ on the training points x_i. We can extend the formula to obtain the value of $\hat{y}$ on every point x in the input space as shown in section 5.4:

$$\hat{y} = \frac{\sum_j W_X(x, x_j)\hat{y}_j}{\sum_j W_X(x, x_j) + \varepsilon} \tag{5.22}$$

where W_X is the symmetric data-dependent edge weighting function (e.g. a Gaussian kernel) such that $\mathbf{W}_{ij} = W_X(x_i, x_j)$. However, in case of very large training sets, solving the linear system in eq. 5.21 may be computationally prohibitive, even using iterative techniques such as those described in section 5.2. In this chapter we consider how to approximate the cost using only a subset of the examples, thanks to the induction formula (eq. 5.22). Even though this will not yield an exact solution to the original problem, it will make the computation time much more reasonable.

5.6.2 Cost approximations

Estimating the cost from a subset

A simple way to reduce the $O(kn^2)$ computational requirement and $O(kn)$ memory requirement for training graph-based semi-supervised algorithms is to force the solutions to be expressed in terms of a **subset of the examples**. This idea has already been exploited successfully in a different form for other kernel algorithms, e.g. for Gaussian processes (Williams et al., 2001) or spectral embedding algorithms (Ouimet et al., 2005).

Here we will take advantage of the induction formula (eq. 5.22) to simplify the linear system to $m \ll n$ equations and variables, where m is the size of a subset of examples that will form a basis for expressing all the other function values. Let $S \subset \{1, \ldots, n\}$ be a subset, with $|S| = m$ and $S \supset \{1, \ldots, \ell\}$ (i.e. we take all labeled examples in the subset). Define

$R = \{1, \ldots, n\} \backslash S$ (the rest of the data). In the following, vector and matrices will be split into their S and R parts, e.g. $\hat{Y} = (\hat{Y}_S, \hat{Y}_R)$ and

$$L = \begin{pmatrix} L_{SS} & L_{SR} \\ L_{RS} & L_{RR} \end{pmatrix}.$$

The idea is to force $\hat{y}_i \in \hat{Y}_R$ to be expressed as a linear combination of the $\hat{y}_j \in \hat{Y}_S$ following eq. 5.22:

$$\forall i \in R,\ \hat{y}_i = \frac{\sum_{j \in S} \mathbf{W}_{ij} \hat{y}_j}{\sum_{j \in S} \mathbf{W}_{ij} + \varepsilon} \tag{5.23}$$

or in matrix notation

$$\hat{Y}_R = \overline{\mathbf{W}}_{RS} \hat{Y}_S \tag{5.24}$$

with $\overline{\mathbf{W}}_{RS}$ the matrix of size $((n-m) \times m)$ with entries $\mathbf{W}_{ij} / (\varepsilon + \sum_{k \in S} \mathbf{W}_{ik})$, for $i \in R$ and $j \in S$. We will then split the cost in eq. 5.20 in terms that involve only the subset S or the rest R, or both of them. To do so, we must first split the diagonal matrix $\mathbf{D}$ (whose elements are row sums of $\mathbf{W}$) into $\mathbf{D} = \mathbf{D}^S + \mathbf{D}^R$, with $\mathbf{D}^S$ and $\mathbf{D}^R$ the $(n \times n)$ diagonal matrices whose elements are sums over S and R respectively, i.e.

$$\begin{aligned} \mathbf{D}^S_{ii} &= \sum_{j \in S} \mathbf{W}_{ij} \\ \mathbf{D}^R_{ii} &= \sum_{j \in R} \mathbf{W}_{ij}. \end{aligned}$$

The un-normalized Laplacian $L = \mathbf{D} - \mathbf{W}$ can then be written

$$L = \begin{pmatrix} \mathbf{D}^S_{SS} + \mathbf{D}^R_{SS} - \mathbf{W}_{SS} & -\mathbf{W}_{SR} \\ -\mathbf{W}_{RS} & \mathbf{D}^S_{RR} + \mathbf{D}^R_{RR} - \mathbf{W}_{RR} \end{pmatrix}. \tag{5.25}$$

Using eq. 5.25, the cost in eq. 5.20 can now be expanded as follows:

$$\begin{aligned} C(\hat{Y}) &= \mu \hat{Y}^\top L \hat{Y} + \mu\varepsilon \|\hat{Y}\|^2 + \|\hat{Y}_\ell - Y_\ell\|^2 \\ &= \underbrace{\mu \hat{Y}_S^\top \left(\mathbf{D}^S_{SS} - \mathbf{W}_{SS}\right) \hat{Y}_S + \mu\varepsilon \|\hat{Y}_S\|^2}_{C_{SS}} + \underbrace{\mu \hat{Y}_R^\top \left(\mathbf{D}^R_{RR} - \mathbf{W}_{RR}\right) \hat{Y}_R + \mu\varepsilon \|\hat{Y}_R\|^2}_{C_{RR}} \\ &+ \underbrace{\mu \left(\hat{Y}_S^\top \mathbf{D}^R_{SS} \hat{Y}_S + \hat{Y}_R^\top \mathbf{D}^S_{RR} \hat{Y}_R - \hat{Y}_R^\top \mathbf{W}_{RS} \hat{Y}_S - \hat{Y}_S^\top \mathbf{W}_{SR} \hat{Y}_R \right)}_{C_{RS}} \\ &+ \underbrace{\|\hat{Y}_\ell - Y_\ell\|^2}_{C_\ell} \end{aligned} \tag{5.26}$$

Resolution

Using the approximation $\hat{Y}_R = \overline{\mathbf{W}}_{RS}\hat{Y}_S$ (eq. 5.24), the gradient of the different parts of the above cost with respect to $\hat{Y}_S$ is then

$$\begin{aligned}
\frac{\partial C_{SS}}{\partial \hat{Y}_S} &= \left[2\mu\left(\mathbf{D}^S_{SS} - \mathbf{W}_{SS} + \varepsilon I\right)\right]\hat{Y}_S \\
\frac{\partial C_{RR}}{\partial \hat{Y}_S} &= \left[2\mu\overline{\mathbf{W}}^\top_{RS}\left(\mathbf{D}^R_{RR} - \mathbf{W}_{RR} + \varepsilon I\right)\overline{\mathbf{W}}_{RS}\right]\hat{Y}_S \\
\frac{\partial C_{RS}}{\partial \hat{Y}_S} &= \left[2\mu\left(\mathbf{D}^R_{SS} + \overline{\mathbf{W}}^\top_{RS}\mathbf{D}^S_{RR}\overline{\mathbf{W}}_{RS} - \overline{\mathbf{W}}^\top_{RS}\mathbf{W}_{RS} - \mathbf{W}_{SR}\overline{\mathbf{W}}_{RS}\right)\right]\hat{Y}_S \\
&= \left[2\mu\left(\mathbf{D}^R_{SS} - \mathbf{W}_{SR}\overline{\mathbf{W}}_{RS}\right)\right]\hat{Y}_S \qquad (5.27) \\
\frac{\partial C_L}{\partial \hat{Y}_S} &= 2\mathbf{S}_{SS}(\hat{Y}_S - Y)
\end{aligned}$$

where to obtain eq. 5.27 we have used the equality $\mathbf{D}^S_{RR}\overline{\mathbf{W}}_{RS} = \mathbf{W}_{RS}$, which follows from the definition of $\overline{\mathbf{W}}_{RS}$.

Recall the original linear system in $\hat{Y}$ was $\left(\mathbf{S} + \mu L + \mu\varepsilon I\right)\hat{Y} = \mathbf{S}Y$ (eq. 5.21). Here it is replaced by a new system in $\hat{Y}_S$, written $\mathbf{A}\hat{Y}_S = \mathbf{S}_{SS}Y_S$ with

$$\begin{aligned}
\mathbf{A} = \quad & \mu\left(\mathbf{D}^S_{SS} - \mathbf{W}_{SS} + \varepsilon I + \mathbf{D}^R_{SS} - \mathbf{W}_{SR}\overline{\mathbf{W}}_{RS}\right) \\
+ \quad & \mu\overline{\mathbf{W}}^\top_{RS}\left(\mathbf{D}^R_{RR} - \mathbf{W}_{RR} + \varepsilon I\right)\overline{\mathbf{W}}_{RS} \\
+ \quad & \mathbf{S}_{SS}.
\end{aligned}$$

Since the system's size has been reduced from n to $|S| = m$, it can be solved much faster, even if $\mathbf{A}$ is not guaranteed* to be sparse anymore (we assume $m \ll n$).

Unfortunately, in order to obtain the matrix $\mathbf{A}$, we need to compute $\mathbf{D}^R_{RR}$, which costs $O(n^2)$ in time, as well as products of matrices that cost $O(mn^2)$ if $\mathbf{W}$ is not sparse. A simple way to get rid of the quadratic complexity in n is to ignore C_{RR} in the total cost. If we remember that C_{RR} can be written

$$C_{RR} = \mu\left(\frac{1}{2}\sum_{i,j\in R}\mathbf{W}_{ij}(\hat{y}_i - \hat{y}_j)^2 + \varepsilon\|\hat{Y}_R\|^2\right)$$

this corresponds to ignoring the smoothness assumption between points in R, as well as the regularization term on R. Even if it may look like a bad idea, it turns out it usually preserves (and even improves) the performance of the semi-supervised classifier, for various reasons:

- assuming the subset S is chosen to correctly "fill" the space, smoothness between points in S and points in R (encouraged by the part C_{RS} of the cost) also enforces smoothness between points in R only,

*In practice, if $\mathbf{W}$ is sparse, $\mathbf{A}$ is also likely to be sparse, even if additional assumptions on $\mathbf{W}$ are needed if one wants to prove it.

- when reducing to a subset, the loss in capacity (we can choose only m values instead of n when working with the full set) suggests we should weaken regularization, and the smoothness constraints are a form of regularization, thus dropping some of them is a way to achieve this goal,

- for some points $i \in R$, the approximation from eq. 5.23

$$\hat{y}_i = \frac{\sum_{j \in S} \mathbf{W}_{ij} \hat{y}_j}{\sum_{j \in S} \mathbf{W}_{ij} + \varepsilon}$$

  may be poor (e.g. for a point far from all points in S, i.e. $\sum_{j \in S} \mathbf{W}_{ij}$ very small), thus smoothness constraints between points in R could be noisy and detrimental to the optimization process (this is not a big issue when considering smoothness between a point x_i in R and a point x_j in S as the smoothness penalty is weighted by $\mathbf{W}_{ij}$, which will be small if x_i is far from all points in S).

Given the above considerations, ignoring the part C_{RR} leads to the new system

$$\left(\mathbf{S}_{SS} + \mu \left(\mathbf{D}_{SS} - \mathbf{W}_{SS} - \mathbf{W}_{SR}\overline{\mathbf{W}}_{RS} + \varepsilon I\right)\right) \hat{Y}_S = \mathbf{S}_{SS} Y_S$$

which in general can be solved in $O(m^3)$ time (less if the system matrix is sparse).

5.6.3 Subset selection

Random selection

In general, training using only a subset of $m \ll n$ samples will not perform as well as using the whole dataset. Carefully choosing the subset S can help in limiting this loss in performance. Even if random selection is certainly the easiest way to choose the points in S, it has two main drawbacks:

- It may not pick points in some regions of the space, resulting in the approximation of eq. 5.23 being very poor in these regions.

- It may pick uninteresting points: the region near the decision surface is the one where we are most likely to make mistakes by assigning the wrong label. Therefore, we would like to have as many points as possible in S being in that region, while we do not need points which are far away from that surface.

As a result, it is worthwhile considering more elaborate subset selection schemes, such as the one presented below.

Smart data sampling

There could be many ways of choosing which points to take in the subset. The algorithm described below is one solution, based on the previous considerations about the random selection weaknesses. The first step of the algorithm will be to select points somewhat uniformly in order to get a first estimate of the decision surface, while the second step will consist in the choice of points near that estimated surface.

Recall equation 5.23:

$$\hat{y}_i = \frac{\sum_{j \in S} \mathbf{W}_{ij} \hat{y}_j}{\sum_{j \in S} \mathbf{W}_{ij} + \varepsilon}.$$

This equation suggests that the value of $\hat{y}_i$ is well approximated when there is a point in S near x_i (two points x_i and x_j are nearby if $\mathbf{W}_{ij}$ is high). The idea will therefore be to cover the manifold where the data lie as well as possible, that is to say ensure that every point in R is near a point (or a set of points) in S. There is another issue we should be taking care of: as we discard the part C_{RR} of the cost, we must now be careful not to modify the structure of the manifold. If there are some parts of the manifold without any point of S, then the smoothness of $\hat{y}$ will not be enforced at such parts (and the labels will be poorly estimated).

This suggests to start with $S = \{1, \dots, \ell\}$ and $R = \{\ell + 1, \dots, n\}$, then add samples x_i by iteratively choosing the point farthest from the current subset, i.e. the one that minimizes $\sum_{j \in S} \mathbf{W}_{ij}$. The idea behind this method is that it is useless to have two points nearby each other in S, as this will not give extra information while increasing the cost. However, one can note that this method may tend to select outliers, which are far from all other points (and especially those from S). A way to avoid this is to consider the quantity $\sum_{j \in R \setminus \{i\}} \mathbf{W}_{ij}$ for a given x_i. If x_i is such an outlier, this quantity will be very low (as all $\mathbf{W}_{ij}$ are small). Thus, if it is smaller than a given threshold δ, we do not take x_i in the subset. The cost of this additional check is of $O((m + o)n)$ where o is the number of outliers: assuming there are only a few of them (less than m), it scales as $O(mn)$.

Once this first subset is selected, it can be refined by training the algorithm presented in section 5.3.2 on the subset S, in order to get an approximation of the $\hat{y}_i$ for $i \in S$, and by using the induction formula (eq. 5.23) to get an approximation of the $\hat{y}_j$ for $j \in R$. Samples in S which are far away from the estimated decision surface can then be discarded, as they will be correctly classified no matter whether they belong to S or not, and they are unlikely to give any information on the shape of the decision surface. These discarded samples are then replaced by other samples that are near the decision surface, in order to be able to estimate it more accurately.

The distance from a point x_i to the decision surface is estimated by the confidence we have in its estimated label $\hat{y}_i$. In the binary classification case considered here (with targets -1 and 1), this confidence is given by $|\hat{y}_i|$,

while in a multi-class setting it would be the absolute value of the difference between the predicted scores of the two highest-scoring classes. One should be careful when removing samples, though: we must make sure we do not leave "empty" regions. This can be done by ensuring that $\sum_{j \in S} \mathbf{W}_{ij}$ stays above some threshold for all $i \in R$ after a point has been removed.

Overall, the cost of this selection phase is on the order of $O(mn + m^3)$. It is summarized in algorithm 5.

Algorithm 5 – Subset selection

Choose a small threshold δ (e.g. $\delta \leftarrow 10^{-10}$)
Choose a small regularization parameter ε (e.g. $\varepsilon \leftarrow 10^{-11}$).
(1) Greedy selection
$S \leftarrow \{1, \ldots, \ell\}$ {The subset is initialized with labeled points}
$R \leftarrow \{\ell + 1, \ldots, n\}$ {The rest is initialized with unlabeled points}
while $|S| < m$ **do**
  Find $i \in R$ s.t. $\sum_{j \in R \setminus \{i\}} \mathbf{W}_{ij} \geq \delta$ and $\sum_{j \in S} \mathbf{W}_{ij}$ is minimum
  $S \leftarrow S \cup \{i\}$
  $R \leftarrow R \setminus \{i\}$
end while
(2) Decision surface improvement
Compute an approximate of $\hat{y}_i$ with $i \in S$ by applying the standard semi-supervised minimization of section 5.3.2 with the data set S.
Compute an approximate of $\hat{y}_j$ with $j \in R$ by eq. 5.23
$S_H \leftarrow$ the points in S with highest confidence
$R_L \leftarrow$ the points in R with lowest confidence
for all $i \in S_H$ **do**
  if $\min_{j \in R} \sum_{k \in S \setminus \{i\}} \mathbf{W}_{jk} \geq \delta$ **then**
    {i can be safely removed from S without leaving empty regions}
    {We find the point with low confidence farthest from S}
    $k^* \leftarrow \operatorname{argmin}_{k \in R_L} \sum_{j \in S} \mathbf{W}_{jk}$
    Replace i by k^* in S (and k^* by i in R)
  end if
end for

5.6.4 Computational issues

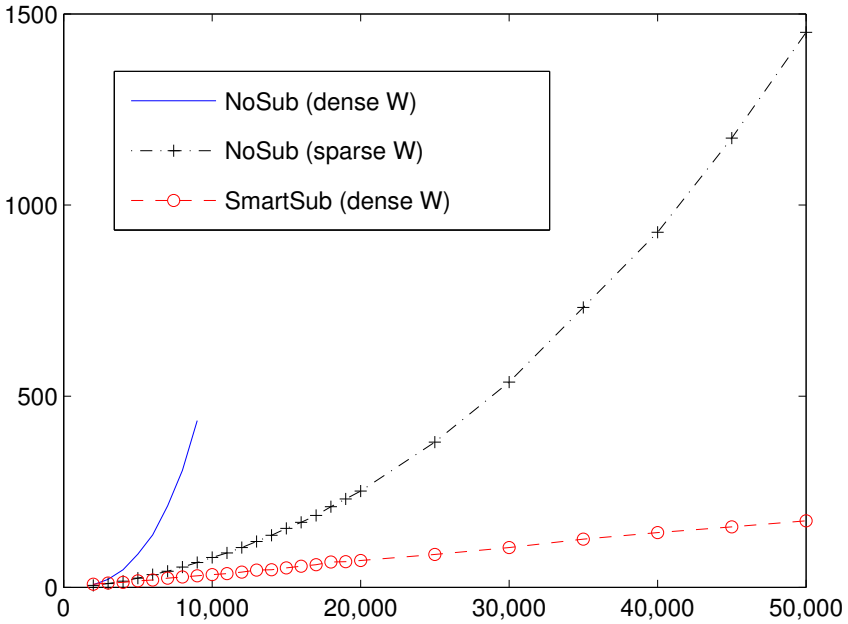
We are now in position to present the overall computational requirements for the different algorithms proposed in this chapter. As before, the subset size m is taken to be much smaller than the total number of points n, and the weight matrix $\mathbf{W}$ may either be dense or sparse (with k non-zero entries in each row or column). Table 5.1 summarizes time and memory requirements for the following algorithms:

- *NoSub*: the original transductive algorithm (using the whole dataset), which solves the system of eq. 5.21, as presented in algorithm 2,

	Time	Memory
NoSub (sparse **W**)	$O(kn^2)$	$O(kn)$
NoSub (dense **W**)	$O(n^3)$	$O(n^2)$
RandSub	$O(m^2n)$	$O(m^2)$
SmartSub	$O(m^2n)$	$O(m^2)$

Table 5.1. *Comparative computational requirements of* NoSub, RandSub *and* SmartSub *(n = number of labeled and unlabeled training data, m = subset size with $m \ll n$, k = number of neighbors for each point in* **W** *when* **W** *is sparse).*

- *RandSub*: the approximation algorithm discussed in section 5.6.2, with the subset S being randomly chosen (section 5.6.3),
- *SmartSub*: the same approximation algorithm as *RandSub*, but with S being chosen as in section 5.6.3.



▶ **Figure 5.1.** *Training time (in seconds) w.r.t. the amount of unlabeled samples on a benchmark dataset. W_X is a Gaussian kernel (combined with an approximate 100-nearest-neighbor kernel in the sparse case). There are $l = 100$ labeled samples, and* SmartSub *selects $m = 500$ unlabeled samples in the subset approximation scheme. Note how the dependence of* SmartSub *in the total number of unlabeled samples $u \in [2000, 50000]$ is only linear.* NoSub *with dense* **W** *fails for $u \geq 10000$ because of memory shortage. Experiments were performed on a 3.2 GHz P4 CPU with 2 Gb of RAM.*

The table shows the approximation method described in this chapter is particularly useful when **W** is dense or n is very large. This is confirmed by empirical experimentation in figure 5.1, which compares the training times on a benchmark dataset (Chapelle et al., 2006) of *NoSub* with a dense kernel, *NoSub* with a sparse kernel, and *SmartSub* with a dense kernel. With a dense kernel, *NoSub* becomes quickly impractical because of the need to store (and solve) a linear system of size $n = \ell + u$, with $l = 100$ and $u \in [2000, 50000]$.

With a sparse kernel (and the iterative version presented in algorithm 2) it scales much better, but still exhibits a quadratic dependency in n. On the other hand, *SmartSub* can handle much more unlabeled data as its training time scales only linearly in n. We have not presented a sparse version of *SmartSub* since our current code cannot take advantage of a sparse weighting function. However, this could be useful to obtain further improvement, especially in terms of memory usage (working with full $m \times m$ matrices can become problematic when $m \geq 10000$).

5.7 Curse of dimensionality for semi-supervised learning

A large number of the semi-supervised learning algorithms proposed in recent years are essentially non-parametric local learning algorithms, relying on a neighborhood graph to approximate manifolds near which the data density is assumed to concentrate. It means that the out-of-sample or transductive prediction at x depends mostly on the unlabeled examples very near x and on the labeled examples that are close in the sense of this graph. In this section, we present theoretical arguments that suggest that such methods are unlikely to scale well (in terms of generalization performance) when the intrinsic dimension of these manifolds becomes large (curse of dimensionality), if these manifolds are sufficiently curved (or the functions to learn vary enough).

5.7.1 The smoothness prior, manifold assumption and non-parametric semi-supervised learning

As discussed in the introduction on semi-supervised learning (section 2.5), the *smoothness assumption* (or its semi-supervised variant) about the underlying target function $y(\cdot)$ (such that $y(x_i) = y_i$) is at the core of many semi-supervised algorithms, along with the *cluster assumption* (or its variant, the *low-density separation assumption*). The former implies that if x_1 is near x_2, then y_1 is expected to be near y_2, and the latter implies that the data density is low near the decision surface. The smoothness assumption is intimately linked to a definition of what it means for x_1 to be near x_2, and that can be embodied in a similarity function on input space, $W_X(\cdot, \cdot)$, which is at the core of the graph-based algorithms reviewed in this chapter, transductive SVMs (where W_X is seen as a kernel), and semi-supervised Gaussian processes (where W_X is seen as the covariance of a prior over functions), as well as the algorithms based on a first unsupervised step to learn a better representation (Chapelle et al., 2006, part IV).

The central claim of this section is that in order to obtain good results with algorithms that rely solely on the smoothness assumption and on the

cluster assumption (or the low-density separation assumption), an acceptable decision surface (in the sense that its error is at an acceptable level) must be "smooth" enough. This can happen if the data for each class lie near a low-dimensional manifold (i.e. the manifold assumption), and these manifolds are smooth enough, i.e., do not have high curvature where it matters, i.e., where a wrong characterization of the manifold would yield to large error rate. This claim is intimately linked to the well known *curse of dimensionality*, so we start the section by reviewing results on generalization error for classical non-parametric learning algorithms as dimension increases. We present theoretical arguments that suggest notions of *locality* of the learning algorithm that make it sensitive to the dimension of the manifold near which data lie. These arguments are not trivial extensions of the arguments for classical non-parametric algorithms, because some semi-supervised algorithms involve expansion coefficients (e.g. the $\hat{y}_j$ in eq. 5.19) that are non-local, i.e., the coefficient associated with the j-th example x_j may depend on inputs x_i that are far from x_j, in the sense of the similarity function or kernel $W_X(x_i, x_j)$. For instance, a labeled point x_i far from an unlabeled point x_j (i.e. $W_X(x_i, x_j)$ is small) may still influence the estimated label of x_j if there exists a path in the neighborhood graph $\mathbf{g}$ that connects x_i to x_j (going through unlabeled examples).

In the last sub-section (5.7.5), we will try to argue that it is possible to build *non-local* learning algorithms, while not using very specific priors about the task to be learned. This goes against common folklore that when there are not enough training examples in a given region, one cannot generalize properly in that region. This would suggest that difficult learning problems such as those encountered in Artificial Intelligence (e.g., vision, language, robotics, etc) would benefit from the development of a larger array of such non-local learning algorithms.

In order to discuss the curse of dimensionality for semi-supervised learning, we introduce a particular notion of locality. It applies to learning algorithms that can be labeled as *kernel machines*, i.e., shown to explicitly or implicitly learn a predictor function of the form

$$f(x) = b + \sum_{i=1}^{n} \alpha_i K_X(x, x_i) \tag{5.28}$$

where i runs over all the examples (labeled and unlabeled), and $K_X(\cdot, \cdot)$ is a symmetric function (kernel) that is either chosen a priori or using the whole data set X (and does not need to be positive semi-definite). The learning algorithm is then allowed to choose the scalars b and α_i.

Most of the decision functions learned by the algorithms discussed in this chapter can be written as in eq. 5.28. In particular, the label propagation

algorithm 2 leads to the induction formula (eq. 5.19) corresponding to

$$\begin{aligned} b &= 0 \\ \alpha_i &= \hat{y}_i \\ K_X(x, x_i) &= \frac{W_X(x, x_i)}{\varepsilon + \sum_j W_X(x, x_j)} \end{aligned} \tag{5.29}$$

The Laplacian regularization algorithm (algorithm 4) from Belkin et al. (2003), which first learns about the shape of the manifold with an embedding based on the principal eigenfunctions of the Laplacian of the neighborhood, also falls into this category. As shown by Bengio et al. (2004), the principal eigenfunctions can be estimated by the Nyström formula:

$$f_k(x) = \frac{\sqrt{n}}{\lambda_k} \sum_{i=1}^{n} v_{k,i} K_X(x, x_i) \tag{5.30}$$

where (λ_k, v_k) is the k-th principal (eigenvalue, eigenvector) pair of the Gram matrix K obtained by $K_{ij} = K_X(x_i, x_j)$, and where $K_X(\cdot, \cdot)$ is a data-dependent equivalent kernel derived from the Laplacian of the neighborhood graph $\mathbf{g}$. Since the resulting decision function is a linear combination of these eigenfunctions, we obtain again a kernel machine (eq. 5.28).

In the following, we say that a kernel function $K_X(\cdot, \cdot)$ is **local** if for all $x \in X$, there exists a neighborhood $\mathcal{N}(x) \subset X$ such that

$$f(x) \simeq b + \sum_{x_i \in \mathcal{N}(x)} \alpha_i K_X(x, x_i). \tag{5.31}$$

Intuitively, this means that only the near neighbors of x have a significant contribution to $f(x)$. For instance, if K_X is the Gaussian kernel, $\mathcal{N}(x)$ is defined as the points in X that are close to x with respect to σ (the width of the kernel). If eq. 5.31 is an equality, we say that K_X is **strictly local**. An example is when W_X is the k-nearest neighbor kernel in algorithm 2. K_X obtained by eq. 5.29 is then also the k-nearest neighbor kernel, and we have $\mathcal{N}(x) = \mathcal{N}_k(x)$ the set of the k nearest neighbors of x, so that

$$f(x) = \sum_{x_i \in \mathcal{N}_k(x)} \frac{\hat{y}_i}{k}.$$

Similarly, we say that K_X is **local-derivative** if there exists another kernel $\tilde{K}_X$ such that for all $x \in X$, there exists a neighborhood $\mathcal{N}(x) \subset X$ such that

$$\frac{\partial f}{\partial x}(x) \simeq \sum_{x_i \in \mathcal{N}(x)} \alpha_i (x - x_i) \tilde{K}_X(x, x_i). \tag{5.32}$$

Intuitively, this means that the derivative of f at point x is a vector contained mostly in the span of the vectors $x - x_i$ with x_i a near neighbor of x. For instance, with the Gaussian kernel, we have $K_X(x, x_i) = e^{-\|x - x_i\|^2 / 2\sigma^2}$ and

$$\frac{\partial K_X(x, x_i)}{\partial x} = -\frac{x - x_i}{\sigma^2} \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2}\right)$$

so that

$$f(x) \simeq b + \sum_{x_i \in \mathcal{N}(x)} \alpha_i (x - x_i) \left(-\frac{1}{\sigma^2} \exp\left(-\frac{\|x - x_i\|^2}{2\sigma^2} \right) \right).$$

Because here $\tilde{K}_X$ is proportional to a Gaussian kernel with width σ, the neighborhood $\mathcal{N}(x)$ is also defined as the points in X which are close to x with respect to σ. Again, we say that K_X is **strictly local-derivative** when eq. 5.32 is an equality (for instance, when K_X is a thresholded Gaussian kernel, i.e. $K_X(x, x_i) = 0$ when $\|x - x_i\| > \delta$).

5.7.2 Curse of dimensionality for classical non-parametric learning

The term **curse of dimensionality** has been coined by Bellman (1961) in the context of control problems, but it has been used rightfully to describe the poor generalization performance of local non-parametric estimators as the dimensionality increases. We define *bias* as the square of the expected difference between the estimator and the true target function, and we refer generically to *variance* as the variance of the estimator, in both cases the expectations being taken with respect to the training set as a random variable. It is well known that classical non-parametric estimators must trade bias and variance of the estimator through a smoothness hyper-parameter, e.g. kernel bandwidth σ for the Nadarya-Watson estimator (Gaussian kernel). As σ increases, bias increases and the predictor becomes less local, but variance decreases, hence the *bias-variance dilemma* (Geman et al., 1992) is also about the *locality* of the estimator.

A nice property of classical non-parametric estimators is that one can prove their convergence to the target function as $n \to \infty$, i.e. these are consistent estimators. One obtains consistency by appropriately varying the hyper-parameter that controls the locality of the estimator as n increases. Basically, the kernel should be allowed to become more and more local, so that bias goes to zero, but the "effective number of examples" involved in the estimator at x,

$$\frac{1}{\sum_{i=1}^{n} K_X(x, x_i)^2}$$

(equal to k for the k-nearest neighbor estimator, with $K_X(x, x_i) = 1/k$ for x_i a neighbor of x) should increase as n increases, so that variance is also driven to 0. For example one obtains this condition with $\lim_{n\to\infty} k = \infty$ and $\lim_{n\to\infty} \frac{k}{n} = 0$ for the k-nearest neighbor. Clearly the first condition is sufficient for variance to go to 0 and the second for the bias to go to 0 (since k/n is proportional to the volume around x containing the k nearest neighbors). Similarly, for the Nadarya-Watson estimator with bandwidth σ, consistency is obtained if $\lim_{n\to\infty} \sigma = 0$ and $\lim_{n\to\infty} n\sigma = \infty$ (in addition to regularity conditions on the kernel). See the book by Härdle et al. (2004) for

a recent and easily accessible exposition (with web version). The bias is due to smoothing the target function over the volume covered by the effective neighbors. As the intrinsic dimensionality of the data increases (the number of dimensions that they actually span locally), bias increases. Since that volume increases exponentially with dimension, the effect of the bias quickly becomes very severe. To see this, consider the classical example of the $[0,1]^d$ hypercube in $\mathbb{R}^d$ with uniformly distributed data in the hypercube. To hold a fraction p of the data in a sub-cube of it, that sub-cube must have sides of length $p^{1/d}$. As $d \to \infty$, $p^{1/d} \to 1$, i.e. we are averaging over distances that cover almost the whole span of the data, just to keep variance constant (by keeping the effective number of neighbors constant).

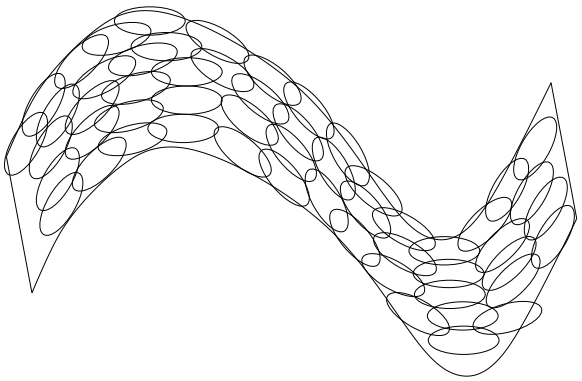
For a wide class of kernel estimators with kernel bandwidth σ, the expected generalization error (bias plus variance, ignoring the noise) can be written as follows (Härdle et al., 2004):

$$\text{expected error} = \frac{C_1}{n\sigma^d} + C_2\sigma^4,$$

with C_1 and C_2 not depending on n nor d. Hence an optimal bandwidth is chosen proportional to $n^{-1/(4+d)}$, and the resulting generalization error converges in $n^{-4/(4+d)}$, which becomes very slow for large d. Consider for example the increase in number of examples required to get the same level of error, in 1 dimension versus d dimensions. If n_1 is the number of examples required to get a level of error e, to get the same level of error in d dimensions requires on the order of $n_1^{(4+d)/5}$ examples, i.e. the **required number of examples is exponential in** d. However, if the data distribution is concentrated on a lower dimensional manifold, it is the **manifold dimension** that matters. Indeed, for data on a smooth lower-dimensional manifold, the only dimension that for instance a k-nearest neighbor classifier sees is the dimension of the manifold, since it only uses the Euclidean distances between the near neighbors, and if they lie on such a manifold then the local Euclidean distances approach the local geodesic distances on the manifold (Tenenbaum et al., 2000). The curse of dimensionality on a manifold (acting with respect to the dimensionality of the manifold) is illustrated in figure 5.2.

5.7.3 Manifold geometry: the curse of dimensionality for local non-parametric manifold learning

Let us first consider how semi-supervised learning algorithms could learn about the shape of the manifolds near which the data concentrate, and how either a high-dimensional manifold or a highly curved manifold could prevent this when the algorithms are local, in the *local-derivative* sense discussed above. As a prototypical example, let us consider the algorithm proposed by Belkin et al. (2003) (algorithm 4). The embedding coordinates are given by the eigenfunctions f_k from eq. 5.30.



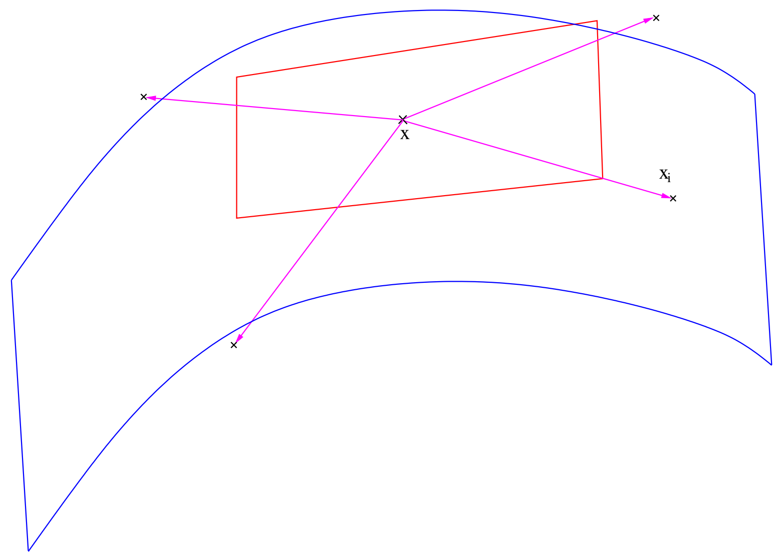
◀ **Figure 5.2.** *Geometric illustration of the effect of the curse of dimensionality on manifolds: the effect depends on the dimension on the manifold, as long as the data are lying strictly on the manifold. In addition to dimensionality, the lack of smoothness (e.g. curvature) of the manifold also has an important influence on the difficulty of generalizing outside of the immediate neighborhood of a training example.*

The first derivative of f_k with respect to x represents the *tangent vector* of the k-th embedding coordinate. Indeed, it is the direction of variation of x that gives rise locally to the maximal increase in the k-th coordinate. Hence the set of manifold tangent vectors $\{\frac{\partial f_1(x)}{\partial x}, \frac{\partial f_2(x)}{\partial x}, \ldots, \frac{\partial f_d(x)}{\partial x}\}$ spans the estimated **tangent plane** of the manifold.

By the local-derivative property (strict or not), each of the tangent vectors at x is constrained to be exactly or approximately in the span of the difference vectors $x - x_i$, where x_i is a neighbor of x. Hence *the tangent plane is constrained to be a subspace of the span of the vectors* $x - x_i$, with x_i neighbors of x. This is illustrated in figure 5.3. In addition to the algorithm of Belkin et al. (2003), a number of non-parametric manifold learning algorithms can be shown (e.g. see Bengio et al., 2006a) to have the local derivative property (or the strictly local derivative property): LLE, Isomap, and spectral clustering with Gaussian or nearest-neighbor kernels.

Hence the local-derivative property gives a strong locality constraint to the tangent plane, in particular when the set of neighbors is small. If the number of neighbors is not large in comparison with the manifold dimension, then the locally estimated shape of the manifold will have *high variance*, i.e., we will have a poor estimator of the manifold structure. If the manifold is approximately flat in a large region, then we could simply increase the number of neighbors. However, if the manifold has high curvature, then we cannot increase the number of neighbors without significantly increasing bias in the estimation of the manifold shape. Bias will restrict us to small regions, and the number of such regions could grow exponentially with the dimension of the manifold (figure 5.2).

A good estimation of the manifold structure – in particular in the region near the decision surface – is crucial for all the graph-based semi-supervised learning algorithms studied in this chapter. It is thanks to a good estimation of the regions in data space where there is high density that we can "propa-



◀ **Figure 5.3.** *Geometric illustration of the effect of the local derivative property shared by semi-supervised graph-based algorithms and spectral manifold learning algorithms. The tangent plane at x is implicitly estimated, and is constrained to be in the span of the vectors $(x_i - x)$, with x_i near neighbors of x. When the number of neighbors is small the estimation of the manifold shape has high variance, but when it is large, the estimation would have high bias unless the true manifold is very flat.*

gate labels" in the right places and obtain an improvement with respect to ordinary supervised learning on the labeled examples. The problems due to high curvature and high dimensionality of the manifold are therefore important to consider when applying these graph-based semi-supervised learning algorithms.

5.7.4 Curse of dimensionality for local non-parametric semi-supervised learning

In this section we focus on graph-based algorithms, using the notation and the induction formula presented in this chapter. We consider here that the ultimate objective is to learn a decision surface, i.e. we have a classification problem, and therefore the region of interest in terms of theoretical analysis is mostly the region near the decision surface. For example, if we do not characterize the manifold structure of the underlying distribution in a region far from the decision surface, it is not important, as long as we get it right near the decision surface. Whereas in the previous section we built an argument based on capturing the shape of the manifold associated with each class, here we focus directly on the discriminant function and on learning the shape of the decision surface.

An intuitive view of label propagation suggests that a region of the manifold around a labeled (e.g. positive) example will be entirely labeled positively, as the example spreads its influence by propagation on the graph representing the underlying manifold. Thus, the number of regions with constant label should be on the same order as (or less than) the number of labeled examples. This is easy to see in the case of a sparse weight matrix $\mathbf{W}$, i.e. when the affinity function is *strictly local*. We define a region with

constant label as a connected subset of the graph $\mathbf{g}$ where all nodes x_i have the same estimated label (sign of $\hat{y}_i$), and such that no other node can be added while keeping these properties. The following proposition then holds (note that it is also true, but trivial, when $\mathbf{W}$ defines a fully connected graph, i.e. $\mathcal{N}(x) = X$ for all x).

Proposition 5.7.1. *After running a label propagation algorithm minimizing a cost of the form of eq. 5.11, the number of regions with constant estimated label is less than (or equal to) the number of labeled examples.*

Proof. By contradiction, if this proposition is false, then there exists a region with constant estimated label that does not contain any labeled example. Without loss of generality, consider the case of a positive constant label, with $x_{\ell+1}, \ldots, x_{\ell+q}$ the q samples in this region. The part of the cost defined in eq. 5.11 depending on their labels is

$$\begin{aligned} C(\hat{y}_{\ell+1}, \ldots, \hat{y}_{\ell+q}) = & \quad \frac{\mu}{2} \sum_{i,j=\ell+1}^{\ell+q} \mathbf{W}_{ij}(\hat{y}_i - \hat{y}_j)^2 \\ & + \mu \sum_{i=\ell+1}^{\ell+q} \left(\sum_{j \notin \{\ell+1,\ldots,\ell+q\}} \mathbf{W}_{ij}(\hat{y}_i - \hat{y}_j)^2 \right) \\ & + \mu\varepsilon \sum_{i=\ell+1}^{\ell+q} \hat{y}_i^2. \end{aligned}$$

The second term is stricly positive, and because the region we consider is maximal (by definition) all samples x_j outside of the region such that $\mathbf{W}_{ij} > 0$ verify $\hat{y}_j < 0$ (for x_i a sample in the region). Since all $\hat{y}_i$ are stricly positive for $i \in \{\ell+1, \ldots, \ell+q\}$, this means this second term can be stricly decreased by setting all $\hat{y}_i$ to 0 for $i \in \{\ell+1, \ldots, \ell+q\}$. This also sets the first and third terms to zero (i.e. their minimum), showing that the set of labels $\hat{y}_i$ are not optimal, which is in contradiction with their definition as the labels that minimize C. □

This means that if the class distributions are such that there are many distinct regions with constant labels (either separated by low-density regions or regions with samples from the other class), we will need at least the same number of labeled samples as there are such regions (assuming we are using a strictly local kernel such as the k-nearest neighbor kernel, or a thresholded Gaussian kernel). But this number could *grow exponentially with the dimension of the manifold(s) on which the data lie*, for instance in the case of a labeling function varying highly along each dimension, *even if the label variations are "simple" in a non-local sense*, e.g. if they alternate in a regular fashion.

When the affinity matrix $\mathbf{W}$ is not sparse (e.g. Gaussian kernel), obtaining such a result is less obvious. However, for local kernels, there often exists

a sparse approximation of $\mathbf{W}$ (for instance, in the case of a Gaussian kernel, one can set to 0 entries below a given threshold or that do not correspond to a k-nearest neighbor relationship). Thus we conjecture the same kind of result holds for such dense weight matrices obtained from a local kernel.

Another indication that highly varying functions are fundamentally hard to learn with graph-based semi-supervised learning algorithms is given by the following theorem (Bengio et al., 2006a):

Theorem 5.7.2. *Suppose that the learning problem is such that in order to achieve a given error level for samples from a distribution P with a Gaussian kernel machine (eq. 5.28), then f must change sign at least $2k$ times along some straight line (i.e., in the case of a classifier, the decision surface must be crossed at least $2k$ times by that straight line). Then the kernel machine must have at least k examples (labeled or unlabeled).*

The theorem is proven for the case where K_X is the Gaussian kernel, but we conjecture that the same result applies to other local kernels, such as the normalized Gaussian or the k-nearest-neighbor kernels implicitly used in graph-based semi-supervised learning algorithms. It is coherent with proposition 5.7.1 since both tell us that we need at least k examples to represent k "variations" in the underlying target classifier, whether along a straight line or as the number of regions of differing class on a manifold.

5.7.5 Outlook: non-local semi-supervised learning

What conclusions should we draw from the previous results? They should help to better circumscribe where the current local semi-supervised learning algorithms are likely to be most effective, and they should also help to suggest directions of research into non-local learning algorithms, either using non-local kernels or similarity functions, or using altogether other principles of generalization.

When applying a local semi-supervised learning algorithm to a new task, one should consider the plausibility of the hypothesis of a low-dimensional manifold near which the distribution concentrates. For some problems this could be very reasonable a priori (e.g. printed digit images vary mostly due to a few geometric and optical effects). For others, however, one would expect tens or hundreds of degrees of freedom (e.g., many Artificial Intelligence problems, such as natural language processing or recognition of complex composite objects).

Concerning new directions of research suggested by these results, several possible approaches can already be mentioned:

- Semi-supervised algorithms that are not based on the neighborhood graph, such as the one presented by Grandvalet et al. (2005), in which a discriminant training criterion for supervised learning is adapted to semi-supervised learning by taking advantage of the *cluster hypothesis*, more precisely, the *low-density separation hypothesis* (see section 2.5),

- Algorithms based on the neighborhood graph but in which the kernel or similarity function (a) is non-isotropic (b) is adapted based on the data (with the spread in different directions being adapted). In that case the predictor will not be either local nor local-derivative. More generally, the structure of the similarity function at x should be inferred based not just on the training data in the close neighborhood of x. For an example of such non-local learning in the unsupervised setting, see Bengio et al. (2005, 2006b).

- Other data-dependent kernels could be investigated, but one should check whether the adaptation allows non-local learning, i.e. that information at x could be used to usefully alter the prediction at a point x' far from x.

- More generally, algorithms that *learn a similarity function* $Sim(x, y)$ in a non-local way (i.e. taking advantage of examples far from x and y) should be good candidates to consider to defeat the curse of dimensionality.

5.8 Discussion

This chapter shows how different graph-based semi-supervised learning algorithms can be cast into a common framework of label propagation and quadratic criterion optimization. They benefit from both points of view: the iterative label propagation methods can provide simple efficient approximate solutions, while the analysis of the quadratic criterion helps to understand what these algorithms really do. The solution can also be linked to physical phenomena such as voltage in an electric network built from the graph, which provides other ways to reason about this problem. In addition, the optimization framework leads to a natural extension of the inductive setting that is closely related to other classical non-parametric learning algorithms such as k-nearest neighbor or Parzen windows.

We showed how graph-based semi-supervised learning algorithms can scale to large amounts of data. The idea is to express the cost to be minimized as a function of only a subset of the unknown labels, in order to reduce the number of free variables: this can be obtained thanks to our induction formula. The form of this formula suggests it is only accurate when the points in the subset cover the whole manifold on which the data lie. This explains why choosing the subset randomly can lead to poor results, while it is possible to design a simple heuristic algorithm (such as algorithm 5) giving much better classification performance. Better selection algorithms (e.g. explicitly optimizing the cost we are interested in) are subject of future research. One must note that the idea of expressing the cost from a subset of the data is not equivalent to training a standard algorithm on the subset

only, before extending to the rest of the data with the induction formula. Here, the rest of the data is explicitly used in the part of the cost enforcing the smoothness between points in the subset and points in the rest (part C_{RS} of the cost), which helps to obtain a smoother labeling function, usually giving better generalization.

Finally, we have shown that the local semi-supervised learning algorithms are likely to be limited to learning smooth functions for data living near low dimensional manifolds. Our approach of locality properties suggests a way to check whether new semi-supervised learning algorithms have a chance to scale to higher dimensional tasks or learning less smooth functions, and motivates further investigation in non-local learning algorithms.

Acknowledgements

The authors would like to thank the editors and anonymous reviewers for their helpful comments and suggestions. This chapter has also greatly benefited from advice from Mikhail Belkin, Dengyong Zhou and Xiaojin Zhu, whose papers first motivated this research (Belkin et al., 2003; Zhou et al., 2004; Zhu et al., 2003). The authors also thank the following funding organizations for their financial support: Canada Research Chair, NSERC and MITACS.

5.9 Commentaires

La faible quantité de résultats présentés dans ce chapitre, qui peut paraître étonnante, est due à la structure du livre pour lequel il a été écrit. Nous avons évidemment évalué les nouveaux algorithmes qui y sont proposés, et parmi les expériences effectuées, il me semble important d'en mentionner brièvement au moins une, qui synthétise les principales nouvelles idées algorithmiques que nous apportons. Nous avons comparé la performance en classification de *NoSub*, *RandSub* et *SmartSub* (c.f. section 5.6.4), en utilisant notre formule d'induction (eq. 5.19) et un noyau Gaussien (dense). Pour trois jeux de données différents*, chaque algorithme a d'abord été entraîné sur 10000 exemples dont seulement 1000 étaient étiquetés, puis sa performance en terme d'erreur de généralisation a été estimée sur 10000 nouveaux exemples jamais vus auparavant. La table 5.2 montre les résultats obtenus. Sans entrer dans les détails de l'expérience, ce qu'il est important de noter c'est que l'algorithme *SmartSub* que nous proposons est compétitif avec l'algorithme original *NoSub* en terme de performance en classification (tout en étant bien plus rapide, comme montré en figure 5.1). La sélection "intelligente" (algorithme 5) des exemples utilisés dans l'approximation permet également de faire mieux qu'une sélection aléatoire, puisque *SmartSub* obtient de meilleures performances que *RandSub*.

Tab. 5.2. *Erreurs de classification en transduction des algorithmes* NoSub, RandSub *et* SmartSub, *sur trois jeux de données différents. Les meilleurs résultats sont en gras :* SmartSub *est comparable à* NoSub, *et toujours meilleur que* RandSub.

Algorithme	Données #1	Données #2	Données #3
NoSub	$\mathbf{18.8} \pm 0.3$	$\mathbf{9.5} \pm 0.1$	34.7 ± 0.1
RandSub	20.3 ± 0.1	9.7 ± 0.1	64.7 ± 3.6
SmartSub	19.8 ± 0.1	$\mathbf{9.5} \pm 0.1$	$\mathbf{33.4} \pm 0.1$

D'autre part, ce chapitre aborde plusieurs points directement reliés au thème central de cette thèse. Tout d'abord, notons que la mise en évidence de liens entre les algorithmes de propagation des étiquettes et la minimisation d'un critère quadratique, établis en section 5.3.3, permet de mieux comprendre l'influence de la connectivité du graphe sur l'efficacité de l'optimisation. En effet, les algorithmes de propagation des étiquettes ont l'avantage d'être beaucoup plus efficaces qu'une optimisation naïve du critère quadratique lorsque la matrice de poids $\mathbf{W}$ est une matrice creuse, ce qui est le cas lorsque l'on considère le graphe des k plus proches voisins. Grâce à l'analyse présentée dans ce chapitre, on peut voir que cela découle naturellement de résultats connus sur l'optimisation de systèmes linéaires creux (Saad, 1996).

La nouvelle technique d'approximation proposée en section 5.6 est une contribution importante pour permettre à de tels algorithmes d'être applicables sur les jeux de données toujours plus imposants qui sont désormais disponibles. Elle permet en particulier d'éviter que le temps de calcul dé-

*Données #1 = "Letter Image Recognition" (Asuncion et al., 2007), données #2 = "MNIST" (LeCun et al., 1998), données #3 = "Forest CoverType" (Asuncion et al., 2007).

pende de manière quadratique de la quantité de données, une dépendance qui limitait auparavant l'intérêt de telles méthodes en présence d'un grand nombre d'exemples non étiquetés.

Outre ces considérations d'efficacité computationnelle, la dernière partie de ce chapitre considère également la question de l'efficacité statistique sous l'angle de la malédiction de la dimensionalité. Nous apportons de nouveaux arguments indiquant que de tels algorithmes, basés sur une notion de similarité locale, ne peuvent apprendre de manière efficace les fonctions qui présentent de nombreuses variations. Dans ce contexte, il est intéressant de noter que les techniques qui ont remis sur le devant de la scène les réseaux de neurones profonds (par exemple Hinton et al., 2006; Bengio et al., 2007; Vincent et al., 2010) sont basées sur l'initialisation des poids du réseau par un apprentissage non supervisé. Ces techniques sont donc applicables dans le cadre de l'apprentissage semi-supervisé, et peuvent tirer parti des capacités de généralisation non locale de tels réseaux. L'un des plus récents algorithmes issus de cette ligne de recherche, dû à Rifai et al. (2011), exhibe ainsi des performances impressionnantes en apprentissage semi-supervisé.

Bibliographie

Asuncion, A. et D. J. Newman. 2007, "UCI machine learning repository", `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Belkin, M., I. Matveeva et P. Niyogi. 2004, "Regularization and semi-supervised learning on large graphs", dans *COLT*.

Belkin, M. et P. Niyogi. 2003, "Using manifold structure for partially labeled classification", dans *Advances in Neural Information Processing Systems 15*, édité par S. Becker, S. Thrun et K. Obermayer, MIT Press, Cambridge, MA.

Bellman, R. 1961, *Adaptive Control Processes : A Guided Tour*, Princeton University Press, New Jersey.

Bengio, Y., O. Delalleau et N. Le Roux. 2006a, "The curse of highly variable functions for local kernel machines", dans *Advances in Neural Information Processing Systems 18*, MIT Press, Cambridge, MA.

Bengio, Y., O. Delalleau, N. Le Roux, J.-F. Paiement, P. Vincent et M. Ouimet. 2004, "Learning eigenfunctions links spectral embedding and kernel PCA", *Neural Computation*, vol. 16, n^{o} 10, p. 2197–2219.

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bengio, Y., H. Larochelle et P. Vincent. 2006b, "Non-local manifold parzen windows", dans *Advances in Neural Information Processing Systems 18*, MIT Press, Cambridge, MA.

Bengio, Y. et M. Monperrus. 2005, "Non-local manifold tangent learning", dans *Advances in Neural Information Processing Systems 17*, édité par L. Saul, Y. Weiss et L. Bottou, MIT Press.

Chapelle, O., B. Schölkopf et A. Zien, éd.. 2006, *Semi-Supervised Learning*, MIT Press, Cambridge, MA.

Delalleau, O., Y. Bengio et N. Le Roux. 2005, "Efficient non-parametric function induction in semi-supervised learning", dans *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*.

Doyle, P. G. et J. L. Snell. 1984, "Random walks and electric networks", *Mathematical Association of America*.

Geman, S., E. Bienenstock et R. Doursat. 1992, "Neural networks and the bias/variance dilemma", *Neural Computation*, vol. 4, n° 1, p. 1–58.

Grandvalet, Y. et Y. Bengio. 2005, "Semi-supervised Learning by Entropy Minimization", dans *Advances in Neural Information Processing Systems 17 (NIPS'04)*, édité par L. Saul, Y. Weiss et L. Bottou, MIT Press, Cambridge, MA.

Härdle, W., M. Müller, S. Sperlich et A. Werwatz. 2004, *Nonparametric and Semiparametric Models*, Springer, http ://www.xplore-stat.de/ebooks/ebooks.html.

Hinton, G. E., S. Osindero et Y. Teh. 2006, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18, p. 1527–1554.

Joachims, T. 2003, "Transductive learning via spectral graph partitioning", dans *ICML*.

LeCun, Y., L. Bottou, Y. Bengio et P. Haffner. 1998, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, n° 11, p. 2278–2324.

Nadaraya, E. A. 1964, "On estimating regression", *Theory of Probability and its Applications*, vol. 9, p. 141–142.

Ouimet, M. et Y. Bengio. 2005, "Greedy spectral embedding", dans *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*.

Rifai, S., Y. Dauphin, P. Vincent, Y. Bengio et X. Muller. 2011, "The manifold tangent classifier", dans *NIPS'2011*.

Rosenberg, S. 1997, *The Laplacian on a Riemannian Manifold*, Cambridge University Press, Cambridge, UK.

Saad, Y. 1996, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA.

Szummer, M. et T. Jaakkola. 2001, "Partially labeled classification with markov random walks", dans *NIPS*, vol. 14.

Tenenbaum, J. B., V. de Silva et J. C. Langford. 2000, "A global geometric framework for nonlinear dimensionality reduction", *Science*, vol. 290, n° 5500, p. 2319–2323.

Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio et P.-A. Manzagol. 2010, "Stacked denoising autoencoders : Learning useful representations in a deep network with a local denoising criterion", *Journal of Machine Learning Research*, vol. 11, n° 3371–3408.

Watson, G. S. 1964, "Smooth regression analysis", *Sankhya - The Indian Journal of Statistics*, vol. 26, p. 359–372.

Williams, C. K. I. et M. Seeger. 2001, "Using the Nyström method to speed up kernel machines", dans *Advances in Neural Information Processing Systems 13*, édité par T. Leen, T. Dietterich et V. Tresp, MIT Press, Cambridge, MA, p. 682–688.

Zhou, D., O. Bousquet, T. Lal, J. Weston et B. Schölkopf. 2004, "Learning with local and global consistency", dans *NIPS*, vol. 16.

Zhu, X. et Z. Ghahramani. 2002, "Learning from labeled and unlabeled data with label propagation", rapport technique CMU-CALD-02-107, Carnegie Mellon University.

Zhu, X., Z. Ghahramani et J. Lafferty. 2003, "Semi-supervised learning using Gaussian fields and harmonic functions", dans *ICML*.

6 Efficient EM Training of Gaussian Mixtures with Missing Data

O. Delalleau, A. Courville et Y. Bengio
Soumis à *IEEE Transactions on Neural Networks and Learning Systems*, 2012

Les valeurs manquantes ne manquent pas dans les applications pratiques de l'apprentissage machine. Malheureusement, beaucoup d'algorithmes d'apprentissage ne peuvent pas être directement appliqués en présence de valeurs manquantes. Une approche classique pour attaquer ce problème consiste à d'abord tenter de "deviner" les variables non observées pour se débarrasser des valeurs manquantes. L'approche la plus basique est de remplacer une variable non observée par la moyenne des valeurs observées pour cette même variable. Mais cette approche ignore entièrement, pour un exemple donné, l'information fournie par les variables observées dans ce même exemple : s'il existe des dépendances entre les variables d'entrée (ce qui est souvent le cas), il devrait être possible de faire mieux en les prenant en compte. L'algorithme des mélanges de Gaussiennes permet justement d'apprendre de telles dépendances et ainsi de prédire les valeurs manquantes. De plus, sa formulation probabiliste s'adapte naturellement à la présence de valeurs manquantes pendant l'apprentissage des paramètres du mélange. Malgré ses attraits, l'utilisation des mélanges de Gaussiennes pour l'estimation des valeurs manquantes est handicapée par le fait que l'algorithme peut être extrêmement inefficace sur de grands ensembles de données en haute dimension. Dans ce chapitre, nous analysons ce problème et proposons un nouvel algorithme beaucoup plus efficace (sans approximation). Nous démontrons aussi l'intérêt pratique d'une telle approche par des résultats empiriques, vérifiant la supériorité de l'estimation par mélange de Gaussiennes sur des méthodes plus basiques.

Contribution personnelle Le point de départ de cet article est une application pratique sur laquelle j'ai dû faire face à des valeurs manquantes. L'utilisation de mélanges de Gaussiennes a été suggérée par Y. Bengio. Je me suis rendu compte de leur inefficacité sur le problème que je tentais de résoudre, j'ai analysé l'origine de cette inefficacité, et imaginé l'algorithme de base (la mise à jour incrémentale des matrices coûteuses à calculer). A. Courville a aidé à améliorer certaines étapes du calcul, notamment par l'utilisation de l'arbre couvrant de poids minimal pour optimiser l'ordonnancement des exemples. J'ai implémenté l'algorithme et réalisé les expériences.

Abstract In data-mining applications, we are frequently faced with a large fraction of missing entries in the data matrix, which is problematic for most discriminant machine learning algorithms. A solution that we explore in this paper is the use of a generative model (a mixture of Gaussians) to compute the conditional expectation of the missing variables given the observed variables. Since training a Gaussian mixture with many different patterns of missing values can be computationally very expensive, we introduce a spanning-tree based algorithm that significantly speeds up training in these conditions. We also observe that good results can be obtained by using the generative model to fill-in the missing values for a separate discriminant learning algorithm.

6.1 Introduction

The presence of missing values in a dataset often makes it difficult to apply a large class of machine learning algorithms. In many real-world data-mining problems, databases may contain missing values due directly to the way the data is obtained (e.g. survey or climate data), or also frequently because the gathering process changes over time (e.g. addition of new variables or merging with other databases). One of the simplest ways to deal with missing data is to discard samples and/or variables that contain missing values. However, such a technique is not suited to datasets with many missing values; these are the focus of this paper.

Here, we propose to use a generative model (a mixture of Gaussians with full covariances) to learn the underlying data distribution and replace missing values by their conditional expectation given the observed variables. A mixture of Gaussians is particularly well suited to generic data-mining problems because:

- By varying the number of mixture components, one can combine the advantages of simple multivariate parametric models (e.g. a single Gaussian), that usually provide good generalization and stability properties, to those of non-parametric density estimators (e.g. Parzen windows, that puts one Gaussian per training sample; Parzen, 1962), that avoid making strong assumptions on the underlying data distribution.

- The Expectation-Maximization (EM) training algorithm (Dempster et al., 1977) naturally handles missing values and provides the missing values imputation mechanism. One should keep in mind that the EM algorithm assumes the missing data are "Missing At Random" (MAR), i.e. that the probability of variables to be missing does not depend on the actual value of missing variables. Even though this assumption will not always hold in practical applications like those mentioned

above, we note that applying the EM algorithm might still yield sensible results in the absence of theoretical justifications.

- Training a mixture of Gaussians scales only linearly with the number of samples, which is attractive for large datasets (at least in low dimension, and we will see in this paper how large-dimensional problems can be tackled).
- Computations with Gaussians often lead to analytical solutions that avoid the use of approximate or sampling methods.
- When confronted with the supervised problem of learning a function $y = f(x)$, a mixture of Gaussians trained to learn a joint density $P(x, y)$ directly provides a least-square estimate of $f(x)$ by $\hat{y} = E[Y|x]$.

Even though such mixtures of Gaussians can indeed be applied directly to supervised problems (Ghahramani et al., 1994), we will see in experiments that using them for missing value imputation before applying a discriminant learning algorithm yields better results. This observation is in line with the common belief that generative models that are trained to learn the global data distribution are not directly competitive with discriminant algorithms for prediction tasks (Bahl et al., 1986): however, they can provide useful information regarding the data that will help such discriminant algorithms to reach better accuracy.

The contributions in this paper are two-fold:

1. We explain why the basic EM training algorithm is not practical in large-dimensional applications in the presence of missing values, and we propose a novel training algorithm that significantly speeds up training by EM.
2. We show, both by visual inspection on image data and by feeding the imputed values to another classification algorithm, how a mixture of Gaussians can model the data distribution so as to provide a valuable tool for missing values imputation.

Note that an extensive study of Gaussian mixture training and missing value imputation algorithms is out of the scope of this paper. Various variants of EM have been proposed in the past (e.g. (Lin et al., 2006)), while we focus here on the "original" EM, showing how it can be solved **exactly** at a significantly lower computational cost. For missing value imputation, statisticians may prefer to draw from the conditional distribution instead of inputing its mean, as the former better preserves data covariance (Zio et al., 2007). In machine learning, the fact that a value is missing may also be by itself a useful piece of information worth taking into account (for instance by adding extra binary inputs to the model, indicating whether each value was observed). All these are important considerations that one should keep in mind, but they will not be addressed here.

In the following section, we present the standard EM algorithm for training mixtures of Gaussians in the presence of missing data. The fast EM algorithm is detailed in section 6.3, while section 6.4 describes experimental results assessing the efficiency of the proposed method for missing value imputation. Finally, section 6.5 concludes this paper and discusses several possible extensions.

6.2 EM for Gaussian mixtures with missing data

In this section, we present the EM algorithm for learning a mixture of Gaussians on a dataset with missing values. The notations we use are as follows. The training set is denoted by $\mathcal{D} = \{x^1, \ldots, x^n\}$. Each sample $x^i \in \mathbb{R}^d$ may have different missing variables. A *missing pattern* is a maximal set of variables that are simultaneously missing in at least one training sample. When the input dimension d is high, there may be many different missing patterns, possibly on the same order as the number of samples n (since the number of possible missing patterns is 2^d). For a sample x^i, we denote by x^i_o and x^i_m the vectors corresponding to respectively the observed and missing variables in x^i. Similarly, given x^i, a symmetric $(d \times d)$ matrix M can be split into four parts corresponding to the observed and missing variables in x^i, as follows:

- M_{oo} contains elements M_{kl} where variables k and l are observed in x^i,
- M_{mm} contains elements M_{kl} where both variables k and l are missing in x^i,
- $M_{om} = M_{mo}^T$ contains elements M_{kl} where variable k is observed in x^i, while variable l is missing.

It is important to keep in mind that with these notations, we have for instance $M_{mm}^{-1} \stackrel{\text{def}}{=} (M_{mm})^{-1} \neq (M^{-1})_{mm}$, i.e. the inverse of a sub-matrix is not the sub-matrix of the inverse. Also, although in this paper we always write for instance M_{oo} so as to keep notations simple, the observed part depends on the sample currently being considered: for two different samples, the M_{oo} matrix may represent a different sub-part of M. It should be clear from the context which sample is being considered.

The EM algorithm (Dempster et al., 1977) can be directly applied in the presence of missing values (Little et al., 2002). For a mixture of Gaussians, Ghahramani et al. (1994) have derived the computations for the two steps (Expectation and Maximization) of the algorithm*:

*Although these equations assume constant (and equal) mixing weights, they can trivially be extended to optimize those weights as well.

Expectation Compute p_{ij}, the probability that Gaussian j generated sample x^i. For the sake of clarity in notations, let us denote $\mu = \mu_j^{(t)}$ and $\Sigma = \Sigma_j^{(t)}$ the estimated mean and covariance of Gaussian j at iteration t of the algorithm. To obtain p_{ij} for a given sample x^i and Gaussian j, we first compute the density

$$q_{ij} = \mathcal{N}(x_o^i; \mu_o, \Sigma_{oo}) \tag{6.1}$$

where $\mathcal{N}(\cdot; \mu_o, \Sigma_{oo})$ is the Gaussian distribution of mean μ_o and covariance Σ_{oo}:

$$\mathcal{N}(z; \mu_o, \Sigma_{oo}) = \frac{1}{\sqrt{2\pi|\Sigma_{oo}|^d}} e^{(-\frac{1}{2}(z-\mu_o)^T \Sigma_{oo}^{-1}(z-\mu_o))}. \tag{6.2}$$

p_{ij} is now simply given by

$$p_{ij} = \frac{q_{ij}}{\sum_{\ell=1}^N q_{i\ell}}$$

where N is the total number of Gaussians in the mixture.

Maximization First fill-in missing values, i.e. define, for each Gaussian j, $\hat{x}^{i,j}$ by $\hat{x}_o^{i,j} = x_o^i$ and $\hat{x}_m^{i,j}$ being equal to the expectation of the missing values x_m^i given the observed x_o^i, assuming Gaussian j has generated x^i. Denoting again $\mu = \mu_j^{(t)}$ and $\Sigma = \Sigma_j^{(t)}$, this expectation is equal to

$$\hat{x}_m^{i,j} = \mu_m + \Sigma_{mo}\Sigma_{oo}^{-1}(x_o^i - \mu_o). \tag{6.3}$$

From these $\hat{x}^{i,j}$, the Maximization step of EM yields the new estimates for the mean and covariances of the Gaussians:

$$\mu_j^{(t+1)} = \frac{\sum_{i=1}^n p_{ij}\hat{x}^{i,j}}{\sum_{i=1}^n p_{ij}}$$

and

$$\Sigma_j^{(t+1)} = \frac{\sum_{i=1}^n p_{ij}(\hat{x}^{i,j} - \mu_j^{(t+1)})(\hat{x}^{i,j} - \mu_j^{(t+1)})^T}{\sum_{i=1}^n p_{ij}} + C_j^{(t)}.$$

The additional term $C_j^{(t)}$ results from the imputation of missing values by their conditional expectation, and is computed as follows (for the sake of clarity, we denote $C_j^{(t)}$ by C and $\Sigma_j^{(t)}$ by Σ):

1. $C \leftarrow 0$
2. for each x^i with observed and missing parts x_o^i, x_m^i:

$$C_{mm} \leftarrow C_{mm} + \frac{p_{ij}}{\sum_{k=1}^n p_{kj}} \Sigma_{mm} - \Sigma_{mo}\Sigma_{oo}^{-1}\Sigma_{om} \tag{6.4}$$

The term being added for each sample corresponds to the covariance of the missing values x_m^i.

Regularization can be added into this framework for instance by adding a small value to the diagonal of the covariance matrix $\Sigma_j^{(t)}$, or by keeping only the first k principal components of this covariance matrix (filling the rest of the data space with a constant regularization coefficient).

6.3 Scaling EM to large datasets

While the EM algorithm naturally extends to problems with missing values, doing so comes with a high computational cost. As we will show, the computational burden may be mitigated somewhat by exploiting the similarity between the covariance matrices between "nearby" patterns of missing values. Before we delve into the details of our algorithm, let us first analyze the computational costs of the operations presented in the EM algorithm above, for each individual training sample:

- The evaluation of q_{ij} from eq. 6.1 and 6.2 requires the inversion of Σ_{oo}, which costs $O(n_o^3)$ operations, where n_o is the number of observed variables in the evaluated sample.

- The contribution to $C_j^{(t)}$ for each Gaussian j (eq. 6.4) can be done in $O(n_o^2 n_m + n_o n_m^2)$ (computation of $\Sigma_{mo}\Sigma_{oo}^{-1}\Sigma_{om}$), or in $O(n_m^3)$ if Σ^{-1} is available, due to the equality

$$\Sigma_{mm} - \Sigma_{mo}\Sigma_{oo}^{-1}\Sigma_{om} = (\Sigma^{-1})_{mm}^{-1}. \tag{6.5}$$

Note that for two examples x^i and x^k with exactly the same pattern of missing variables, the two expensive operations above need only be performed once, as they are the same for both x^i and x^k. But in high-dimensional datasets without a clear structure in the missing values, most missing patterns are not shared by many samples. For instance, in a real-world financial dataset we have been working on, the number of missing patterns is about half the total number of samples. Since each iteration of EM has a cost of $O(pNd^3)$, with p unique missing patterns, N components in the mixture, and input dimension d, the EM algorithm as presented in section 6.2 is not computationally feasible for large high-dimensional datasets. The "fast" EM variant proposed by Lin et al. (2006) also suffers from the same bottlenecks, i.e. it is fast only when there are few unique missing patterns.

While the large numbers of unique patterns of missing values typically found in real-world datasets present a barrier to the application of EM to these problems, they also motivate a means of reducing the computational cost. As discussed above, for high-dimensional datasets, the computational

cost is dominated by the determination of the inverse covariance of the observed variables Σ_{oo}^{-1} and of the conditional covariance of the missing data given the observed data (eq. 6.5). However, as we will show, these quantities corresponding to one pattern of the missing values may be determined from those of another pattern of missing values at a cost proportional to the distance between the two missing value patterns (measured as the number of missing and observed variables on which the patterns differ). Thus, for "nearby" patterns of missing values, these covariance computations may be efficiently computed by chaining their computation through the set of patterns of missing values. Furthermore, since the cost of these updates will be smaller when the missing patterns of two consecutive samples are close to each other, we want to **optimize the samples ordering** so as to minimize this cost.

We present the details of the proposed algorithms in the following sections. First, we observe in section 6.3.1 how we can avoid computing Σ_{oo}^{-1} by using the Cholesky decomposition of Σ_{oo}. Then we show in section 6.3.2 how the so-called inverse variance lemma can be used to update the conditional covariance matrix (eq. 6.5) for a missing pattern given the one computed for another missing pattern. As presented in section 6.3.3, these two ideas combined give rise to an objective function with which one can determine an optimal ordering of the missing patterns, minimizing the overall computational cost. The resulting fast EM algorithm is summarized in section 6.3.4.

6.3.1 Cholesky updates

Computing q_{ij} by eq. 6.1 can be done directly from the inverse covariance matrix Σ_{oo}^{-1} as in eq. 6.2, but, as argued by Seeger (2005), it is just as fast, and numerically more stable, to use the Cholesky decomposition of Σ_{oo}. Writing

$$\Sigma_{oo} = QQ^T$$

with Q a lower triangular matrix with positive diagonal elements, we indeed have

$$z^T \Sigma_{oo}^{-1} z = \|Q^{-1} z\|^2$$

where $Q^{-1}z$ can easily be obtained since Q is lower triangular. Assuming Q is computed once for the missing pattern of the first sample in the training set, the question is thus how to update this matrix for the next missing pattern. This reduces to finding how to update Q when adding or removing rows and columns to Σ_{oo} (adding a row and column when a variable that was missing is now observed in the next sample, and removing a row and column when a variable that was observed is now missing).

Algorithms to perform these updates can be found for instance in the book by Stewart (1998). When adding a row and column, we always add it as the last dimension to minimize the computations. These are on the order of $O(n_o^2)$, where n_o is the length and width of Σ_{oo}. Removing a row and

column is also on the order of $O(n_o^2)$, though the exact cost depends on the position of the row / column being removed.

Let us denote by n_d the number of differences between two consecutive missing patterns. Assuming that n_d is small compared to n_o, the above analysis shows that the overall cost is on the order of $O(n_d\, n_o^2)$ computations. How to find an ordering of the patterns such that n_d is small will be discussed in section 6.3.3.

6.3.2 Inverse variance lemma

The second bottleneck of the EM algorithm resides in the computation of eq. 6.5, corresponding to the conditional covariance of the missing part given the observed part. Note that we cannot rely on a Cholesky decomposition here, since we need the full conditional covariance matrix itself. In order to update $(\Sigma^{-1})^{-1}_{mm}$, we will take advantage of the so-called inverse variance lemma (Whittaker, 1990). It states that the inverse of a partitioned covariance matrix

$$\Lambda = \begin{pmatrix} \Lambda_{XX} & \Lambda_{XY} \\ \Lambda_{YX} & \Lambda_{YY} \end{pmatrix} \tag{6.6}$$

can be computed by

$$\Lambda^{-1} = \begin{pmatrix} \Lambda_{XX}^{-1} + B^T \Lambda_{Y|X}^{-1} B & -B^T \Lambda_{Y|X}^{-1} \\ -\Lambda_{Y|X}^{-1} B & \Lambda_{Y|X}^{-1} \end{pmatrix} \tag{6.7}$$

where Λ_{XX} is the covariance of the X part, and the matrix B and the conditional covariance $\Lambda_{Y|X}$ of the Y part given X are obtained by

$$B = \Lambda_{YX} \Lambda_{XX}^{-1} \tag{6.8}$$

$$\Lambda_{Y|X} = \Lambda_{YY} - \Lambda_{YX} \Lambda_{XX}^{-1} \Lambda_{XY} \tag{6.9}$$

Note that eq. 6.9 is similar to eq. 6.5, since the conditional covariance of Y given X verifies

$$\Lambda_{Y|X} = (\Lambda^{-1})^{-1}_{YY} \tag{6.10}$$

where we have also partitioned the inverse covariance matrix as

$$\Lambda^{-1} = \begin{pmatrix} (\Lambda^{-1})_{XX} & (\Lambda^{-1})_{XY} \\ (\Lambda^{-1})_{YX} & (\Lambda^{-1})_{YY} \end{pmatrix}. \tag{6.11}$$

These equations can be used to update the conditional covariance matrix of the missing variables given the observed variables when going from one missing pattern to the next one, so that it does not need to be re-computed from scratch. Let us first consider the case of going from sample x^i to x^j, where we only add missing values (i.e. all variables that are missing in x^i are also missing in x^j). We can apply the inverse variance lemma (eq. 6.7) with the following quantities:

- Λ^{-1} is the conditional covariance* of the missing variables in x^j given the observed variables, i.e. $(\Sigma^{-1})^{-1}_{mm}$ in eq. 6.5, the quantity we want to compute,
- X are the missing variables in sample x^i,
- Y are the missing variables in sample x^j that were not missing in x^i,
- Λ^{-1}_{XX} is the conditional covariance of the missing variables in x^i given the observed variables, that would have been computed previously,
- since $\Lambda = (\Sigma^{-1})_{mm}$, then Λ_{YX} and Λ_{YY} are simply sub-matrices of the global inverse covariance matrix, that only needs to be computed once (per iteration of EM).

Let us denote by n_d the number of missing values added when going from x^i to x^j, and by n_m the number of missing values in x^i. Assuming n_d is small compared to n_m, then the computational cost of eq. 6.7 is dominated by the cost $O(n_d\ n_m^2)$ for the computation of B by eq. 6.8 and of the upper-left term in eq. 6.7 (the inversion of $\Lambda_{Y|X}$ is only in $O(n_d^3)$).

In the case where we remove missing values instead of adding some, this corresponds to computing Λ^{-1}_{XX} from Λ^{-1} using eq. 6.7. This can be done from the partition of Λ^{-1} since, by identifying eq. 6.7 and 6.11, we have:

$$\begin{aligned} & (\Lambda^{-1})_{XX} - (\Lambda^{-1})_{XY}(\Lambda^{-1})^{-1}_{YY}(\Lambda^{-1})_{YX} \\ = \ & \Lambda^{-1}_{XX} + B^T\Lambda^{-1}_{Y|X}B - B^T\Lambda^{-1}_{Y|X}(\Lambda^{-1})^{-1}_{YY}\Lambda^{-1}_{Y|X}B \\ = \ & \Lambda^{-1}_{XX} \end{aligned}$$

where we have used eq. 6.10 to obtain the final result. Once again the cost of this computation is dominated by a term of the form $O(n_d\ n_m^2)$, where this time n_d denotes the number of missing values that are removed when going from x^i to x^j and n_m the number of missing values in x^j.

Thus, in the general case where we both remove and add missing values, the cost of the update is on the order of $O(n_d\ n_m^2)$, if we denote by n_d the total number of differences in the missing patterns, and by n_m the average number of missing values in x^i and x^j (which are assumed to be close, since n_d is supposed to be small). The speed-up is on the order of $O(n_m/n_d)$ compared to the "naive" algorithm that would re-compute the conditional covariance matrix for each missing pattern.

6.3.3 Optimal ordering from the minimum spanning tree

Given an ordering $\{m^1, m^2, \ldots, m^p\}$ of the p missing patterns present in the training set, during an iteration of the EM algorithm we have to:

*Note that in the original formulation of the inverse variance lemma Λ is a covariance matrix, while we use it here as an inverse covariance: since the inverse of a symmetric positive definite matrix is also symmetric positive definite, it is possible to apply eq. 6.7 to an inverse covariance.

1. Compute the Cholesky decomposition of Σ_{oo} and the conditional covariance $(\Sigma^{-1})^{-1}_{mm}$ for the first missing pattern m^1.

2. For each subsequent missing pattern m^i, find the missing pattern in $\{m^1, m^2, \ldots, m^{i-1}\}$ that allows the fastest computation of the same matrices, from the update methods presented in sections 6.3.1 and 6.3.2.

Since each missing pattern but the first one has a "parent" (the missing pattern from which we update the desired matrices), we visit the missing patterns in a tree-like fashion: *the optimal tree is thus the minimum spanning tree of the fully-connected graph whose nodes are the missing patterns and the weight between nodes m^i and m^j is the cost of computing matrices for m^j given matrices for m^i*. Note that the spanning tree obtained this way is the exact optimal solution and not an approximation (assuming we constrain ourselves to visiting only observed missing patterns and we can compute the true cost of updating matrices).

For the sake of simplicity, we used the number of differences n_d between missing pattterns m^i and m^j as the weights between two nodes. Finding the "true" cost is difficult because (1) it is implementation-dependent and (2) it depends on the ordering of the columns for the Cholesky updates, and this ordering varies depending on previous updates due to the fact it is more efficient to add new dimensions as the last ones, as argued in section 6.3.1. We tried more sophisticated variants of the cost function, but they did not decrease significantly the overall computation time.

Note it would be possible to allow the creation of "virtual" missing patterns, whose corresponding matrices could be used by multiple observed missing patterns in order to speed-up updates even further. Finding the optimal tree in this setting corresponds to finding the optimal Steiner tree (Hwang et al., 1992), which is known to be NP-hard. Since we do not expect the available approximate solution schemes to provide a huge speed-up, we did not explore this approach further.

Finally, one may have concerns about the numerical stability of this approach, since computations are incremental and thus numerical errors will be accumulated. The number of incremental steps is directly linked to the depth of the minimum spanning tree, which will often be logarithmic in the number of training samples, but may grow linearly in the worst case. Although we did not face this problem in our own experiments (where the accumulation of errors never led to results significantly different from the exact solution), the following heuristics can be used to solve it: the matrices of interest can be re-computed "from scratch" at each node of the tree whose depth is a multiple of k, with k a hyper-parameter trading accuracy for speed.

6.3.4 Fast EM algorithm overview

We summarize here the previous sections by giving a sketch of the resulting fast EM algorithm for Gaussian mixtures:

1. Find all unique missing patterns in the dataset.

2. Compute the minimum spanning tree* of the corresponding graph of missing patterns (see section 6.3.3).

3. Deduce from the minimum spanning tree on missing patterns an ordering of the training samples (since each missing pattern may be common to many samples).

4. Initialize the means of the mixture components by the K-means clustering algorithm (Lloyd, 1982), and their covariances from the empirical covariances in each cluster (either imputing missing values with the cluster means or just ignoring them, which is what we did in our experiments).

5. Iterate through EM steps (as described in section 6.2) until convergence (or until a validation error increases). At each step, the expensive matrix computations highlighted in section 6.3 are sped-up by using iterative updates, following the ordering obtained in step 3.

6.4 Experiments

6.4.1 Learning to model images

To assess the speed improvement of our proposed algorithm over the "naive" EM algorithm, we trained mixtures of Gaussians on the MNIST dataset of handwritten digits. For each class of digits (from 0 to 9), we optimized an individual mixture of Gaussians in order to model the class distribution. We manually added missing values by removing the pixel information in each image from a randomly chosen square of 5x5 pixels (the images are 28x28 pixels, i.e. in dimension 784). The mixtures were first trained efficiently on the first 4500 samples of each class, while the rest of the samples were used to select the hyperparameters, namely the number of Gaussians (from 1 to 10), the fraction of principal components kept (75%, 90% or 100%), and the random number generator seed used in the mean initialization (chosen between 5 different values). The best model was chosen based on the average negative log-likelihood. It was then re-trained using the "naive" version of

*Since the cost of this computation is in $O(p^2)$, with p the number of missing patterns, if p is too large it is possible to perform an initial basic clustering of the missing patterns and compute the minimum spanning tree independently in each cluster.

the EM algorithm, in order to compare execution time and also ensure the same results were obtained.

On average, the speed-up on our cluster computers (32 bit P4 3.2 Ghz with 2 Gb of memory) was on the order of 8. We also observed a larger speed-up improvement (on the order of 20) on another architecture (64 bit Athlon 2.2 Ghz with 2 Gb of memory): the difference seemed to be due to implementations of the BLAS and LAPACK linear algebra libraries.

▶ **Figure 6.1.** *Imputation of missing pixels in images. Each pair of images includes an image with missing pixels (the grey square), followed by the image with these same pixels imputed by the mixture of Gaussians.*

We display in figure 6.1 the imputation of missing values realized by the trained mixture when provided with sample test images. On each row, images with grey squares have missing values (identified by these squares), while images next to them show the result of the missing value imputation. Although the imputed pixels are somewhat fuzzy, the figure shows the mixture was able to capture meaningful correlations between pixels, and to impute sensible missing values.

6.4.2 Combining generative and discriminative models

The Abalone dataset from the UCI Machine Learning Repository is a standard benchmark regression task. The official training set (3133 samples) is divided into a training (2000) and validation set (1133), while we use the official test set (1044). This dataset does not contain any missing data, which allows us to see how the algorithms behave as we add more missing values. We systematically preprocess the dataset after inserting missing values, by normalizing all variables (including the target) so that the mean and standard deviation on the training set are respectively 0 and 1 (note that we do not introduce missing values in the target, so that mean squared errors can be compared).

We compare three different missing values imputation mechanisms:

1. Imputation by the conditional expectation of the missing values as computed by a mixture of Gaussians learnt on the joint distribution of the input and target (the algorithm proposed in this paper)

2. Imputation by the global empirical mean (on the training set)

3. Imputation by the value found in the nearest neighbor that has a non missing value for this variable (or, alternatively, by the mean of the 10 such nearest neighbors). Because there is no obvious way to compute the nearest neighbors in the presence of missing values (see e.g. Caruana, 2001), we allow this algorithm to compute the neighborhoods based on the original dataset with no missing value: it is thus expected to give the optimal performance that one could obtain with such a nearest-neighbor algorithm.

One one hand, we report the performance of the mixture of Gaussian used directly as a predictor for regression. On another hand, the imputed values are also fed to the two following discriminant algorithms, whose hyper-parameters are optimized on the validation set:

1. A one-hidden-layer feedforward neural network trained by stochastic gradient descent, with hyper-parameters the number of hidden units (among 5, 10, 15, 20, 30, 50, 100, 200), the quadratic weight decay (among 0, 10^{-6}, 10^{-4}, 10^{-2}), the initial learning rate (among 10^{-2}, 10^{-3}, 10^{-4}) and its decrease constant* (among 0, 10^{-2}, 10^{-4}, 10^{-6}).

2. A kernel ridge regressor, with hyper-parameters the weight decay (in 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}, 1) and the kernel: either the linear kernel $x_i^T x_j$, the Gaussian kernel

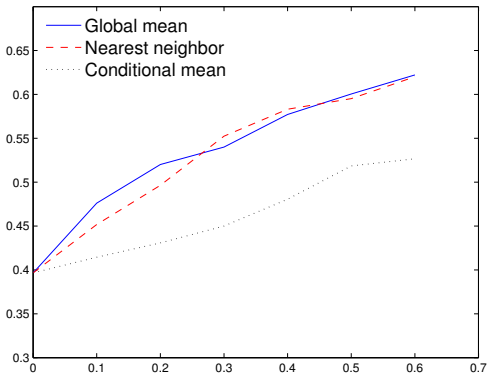
$$K_\sigma(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

(with bandwidth σ in 100, 50, 10, 5, 1, 0.5, 0.1, 0.05, 0.01), or the polynomial kernel $K_{\delta,c}(x_i, x_j) = (1 + c x_i^T x_j)^\delta$ (with degree δ in 1, 2, 3, 4, 5 and dot product scaling coefficient c in 0.01, 0.05, 0.1, 0.5, 1, 5, 10).

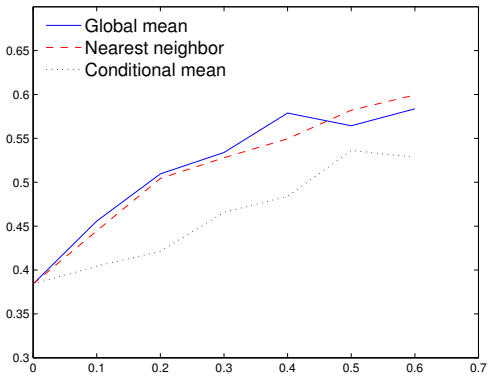
Figures 6.2 and 6.3 compare the three missing values imputation mechanisms when using a neural network and kernel ridge regression. It can be seen that the conditional mean imputation obtained by the Gaussian mixture significantly outperforms the global mean imputation and nearest neighbor imputation (which is tried with both 1 and 10 neighbors, keeping the best on the validation set). The latter seems to be reliable only when there are few missing values in the dataset: this is expected, as when the number of missing values increases one has to go further in space to find neighbors that contain non-missing values for the desired variables.

Figure 6.4 illustrates the gain of combining the generative model (the mixture of Gaussian) with the discriminant learning algorithms: even though the mixture can be used directly as a regressor (as argued in the introduction), its prediction accuracy can be greatly improved by a supervised learning step.

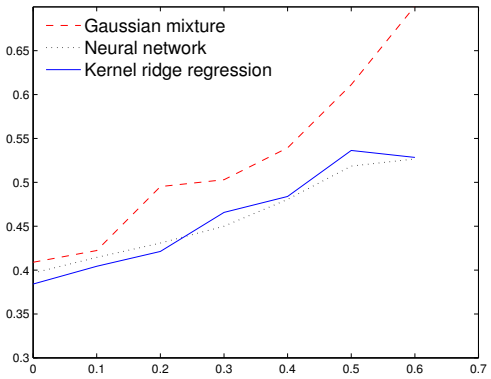
*The learning rate after seeing t samples is equal to $\mu(t) = \frac{\mu(0)}{1+\lambda t}$, where $\mu(0)$ is the initial learning rate and λ the decrease constant.



▶ **Figure 6.2.** *For neural network: test mean-squared error (y axis) on Abalone when the proportion of missing values increases (x axis), for the three missing values imputation mechanisms.*



▶ **Figure 6.3.** *For kernel ridge regression: test mean-squared error (y axis) on Abalone when the proportion of missing values increases (x axis), for the three missing values imputation mechanisms.*



▶ **Figure 6.4.** *Test mean-squared error (y axis) on Abalone when the proportion of missing values increases (x axis). Combining a discriminant algorithm with the generative Gaussian mixture model works better than the Gaussian mixture alone (both the neural network and the kernel ridge regressor use here the conditional mean imputation of missing values provided by the Gaussian mixture).*

6.5 Conclusion

In this paper, we considered the problem of training Gaussian mixtures in the context of large high-dimensional datasets with a significant fraction of the data matrix missing. In such situations, the application of EM to the imputation of missing values results in expensive matrix computations. We have proposed a more efficient algorithm that uses matrix updates over a minimum spanning tree of missing patterns to speed-up these matrix computations, by an order of magnitude.

We also explored the application of a hybrid scheme where a mixture of Gaussians generative model, trained with EM, is used to impute the missing values with their conditional means. These imputed datasets were then used in a discriminant learning model (neural networks and kernel ridge regression) where they were shown to provide significant improvement over more basic missing value imputation methods.

6.6 Commentaires

Notons que l'amélioration apportée par l'algorithme présenté dans ce chapitre, en terme de rapidité de calcul, est d'autant plus significative si :

- Il y a beaucoup de motifs différents de valeurs manquantes ("missing patterns"), puisque le problème de l'algorithme d'origine est qu'il doit effectuer un calcul coûteux pour chaque motif.
- La distribution des motifs de valeurs manquantes (dans $\{0,1\}^d$) est "concentrée", dans le sens que la distance entre un motif et son plus proche voisin est petite. Ceci est dû au fait que le coût de mise à jour d'un motif à un autre dépend de la distance entre les motifs.
- La dimension est élevée : en effet, en faible dimension, les librairies d'algèbre linéaire peuvent généralement calculer une décomposition en valeurs singulières suffisamment rapidement pour que le gain apporté par notre algorithme soit minime.

Il est donc important de bien analyser les données disponibles avant de décider s'il est pertinent d'implémenter cette approche.

Il est intéressant de remarquer que le remplacement des valeurs manquantes est un sujet souvent abordé en statistiques, par exemple pour remplacer les réponses manquantes dans les sondages (Zio et al., 2007). Mais souvent, les statisticiens souhaitent préserver les propriétés statistiques des variables, en particulier la variance : remplacer les valeurs manquantes de manière déterministe (dans notre cas, par une espérance conditionnelle) a typiquement tendance à faire diminuer la variance. Une approche classique consiste alors à plutôt faire de l'échantillonnage des valeurs manquantes (à partir de leur distribution conditionnelle – qui ici serait Gaussienne). Il serait intéressant de voir si en apprentissage machine, une telle technique pourrait aider : le fait de rajouter ainsi du bruit dans les données peut d'un côté rendre l'apprentissage plus difficile, mais, d'un autre côté, aussi jouer un rôle de régularisation qui pourrait aider à combattre le sur-apprentissage. Il est donc probable que la réponse à cette question dépende des données et de la tâche à résoudre.

Finalement, le but de ce chapitre étant d'améliorer un algorithme existant, nous n'avons pas passé en revue tous les algorithmes possibles pour le remplacement de valeurs manquantes. Dans le contexte des algorithmes mentionnés dans cette thèse, les réseaux de neurones auto-encodeurs débruitants proposés par Vincent et al. (2010) semblent être des candidats naturels pour résoudre cette tâche, même si ce n'est pas leur but premier. Ces réseaux sont entraînés de manière à pouvoir reconstruire certaines variables bruitées à partir des variables observées : dans l'algorithme d'origine ces variables sont bruitées de manière artificielle, mais il devrait être possible – moyennant des modifications mineures – d'appliquer cet algorithme en considérant que les variables manquantes font partie des variables bruitées. Il serait intéressant de comparer une telle approche aux mélanges de Gaussiennes étudiées dans ce chapitre.

Bibliographie

Bahl, L., P. Brown, P. deSouza et R. Mercer. 1986, "Maximum mutual information estimation of hidden markov parameters for speech recognition", dans *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Tokyo, Japan, p. 49–52.

Caruana, R. 2001, "A non-parametric EM-style algorithm for imputing missing values", dans *Proceedings of the Eigth International Workshop on Artificial Intelligence and Statistics (AISTATS'01)*, Society for Artificial Intelligence and Statistics.

Dempster, A. P., N. M. Laird et D. B. Rubin. 1977, "Maximum-likelihood from incomplete data via the EM algorithm", *Journal of Royal Statistical Society B*, vol. 39, p. 1–38.

Ghahramani, Z. et M. I. Jordan. 1994, "Supervised learning from incomplete data via an EM approach", dans *Advances in Neural Information Processing Systems 6 (NIPS'93)*, édité par D. Cowan, G. Tesauro et J. Alspector, Morgan Kaufmann, San Mateo, CA.

Hwang, F. K., D. Richards et P. Winter. 1992, "The Steiner tree problem", *Annals of Discrete Mathematics*, vol. 53.

Lin, T. I., J. C. Lee et H. J. Ho. 2006, "On fast supervised learning for normal mixture models with missing information", *Pattern Recognition*, vol. 39, n^{o} 6, p. 1177–1187.

Little, R. J. A. et D. B. Rubin. 2002, *Statistical Analysis with Missing Data*, 2^{e} éd., Wiley, New York.

Lloyd, S. P. 1982, "Least squares quantization in PCM", *IEEE Transactions on Information Theory*, vol. 28, n^{o} 2, p. 129–137.

Parzen, E. 1962, "On the estimation of a probability density function and mode", *Annals of Mathematical Statistics*, vol. 33, p. 1064–1076.

Seeger, M. 2005, "Low rank updates for the Cholesky decomposition", rapport technique, Department of EECS, University of California at Berkeley.

Stewart, G. W. 1998, *Matrix Algorithms, Volume I : Basic Decompositions*, SIAM, Philadelphia.

Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio et P.-A. Manzagol. 2010, "Stacked denoising autoencoders : Learning useful representations in a deep network with a local denoising criterion", *Journal of Machine Learning Research*, vol. 11, n^{o} 3371–3408.

Whittaker, J. 1990, *Graphical Models in Applied Multivariate Statistics*, Wiley, Chichester.

Zio, M. D., U. Guarnera et O. Luzi. 2007, "Imputation through finite Gaussian mixture models", *Computational Statistics & Data Analysis*, vol. 51, n° 11, p. 5305–5316.

7 Justifying and generalizing contrastive divergence

Y. Bengio et O. Delalleau
***Neural Computation*, vol. 21, n° 6, p. 1601–1621, 2009**

L'algorithme de divergence contrastive a originellement été introduit par Hinton (2002) pour l'apprentissage de modèles probabilistes appelés les *produits d'experts*, mais est devenu populaire surtout lorsque Hinton et al. (2006a) ont montré son utilité pour l'apprentissage de réseaux de neurones profonds. Il s'agit en effet d'un algorithme efficace pour l'entraînement de machines de Boltzmann restreintes (RBMs, introduites en section 2.9), qui peuvent être superposées pour initialiser les poids d'un réseau profond et ainsi obtenir de bien meilleures performances qu'avec une initialisation aléatoire (Hinton et al., 2006a; Bengio et al., 2007a). Mais malgré ces succès empiriques incontestables, les propriétés théoriques de la divergence contrastive restent mal comprises. Dans la mesure où cet algorithme peut s'interpréter comme une approximation de la descente de gradient sur la log-vraisemblance négative des données observées, il est naturel de s'interroger sur les conséquences de cette approximation. Dans ce chapitre, nous analysons de manière théorique les liens entre la divergence contrastive et la descente de gradient, et illustrons ces résultats théoriques par des observations empiriques. L'analyse est basée sur l'interprétation de la divergence contrastive comme une troncation de l'expansion du gradient de la log-vraisemblance. Cette analyse permet de mieux comprendre pourquoi la divergence contrastive fonctionne, de tisser d'autres liens avec par exemple l'erreur de reconstruction utilisée dans les réseaux auto-encodeurs, et suggère que d'autres variantes pourraient être développées, basées sur les mêmes principes.

Contribution personnelle L'idée d'analyser la formule de la divergence contrastive comme une troncation de l'expansion du gradient de la log-vraisemblance vient de Y. Bengio. J'ai aidé à formaliser et simplifier une première analyse de cette troncation, ainsi que le lien avec l'erreur de reconstruction, que Y. Bengio avait commencés. J'ai dérivé la borne de convergence et réalisé les expériences. Ma contribution à la rédaction totale de l'article est d'environ 50% (Y. Bengio ayant écrit la majeure partie de l'introduction et de la description de l'algorithme de divergence contrastive).

Abstract We study an expansion of the log-likelihood in undirected graphical models such as the restricted Boltzmann machine (RBM), where each term in the expansion is associated with a sample in a Gibbs chain alternating between two random variables (the visible vector and the hidden vector, in RBMs). We are particularly interested in estimators of the gradient of the log-likelihood obtained through this expansion. We show that its residual term converges to zero, justifying the use of a truncation, i.e. running only a short Gibbs chain, which is the main idea behind the contrastive divergence (CD) estimator of the log-likelihood gradient. By truncating even more, we obtain a stochastic reconstruction error, related through a mean-field approximation to the reconstruction error often used to train autoassociators and stacked auto-associators. The derivation is not specific to the particular parametric forms used in RBMs, and only requires convergence of the Gibbs chain. We present theoretical and empirical evidence linking the number of Gibbs steps k and the magnitude of the RBM parameters to the bias in the CD estimator. These experiments also suggest that the sign of the CD estimator is correct most of the time, even when the bias is large, so that CD-k is a good descent direction even for small k.

7.1 Introduction

Motivated by the theoretical limitations of a large class of non-parametric learning algorithms (Bengio et al., 2007b), recent research has been focusing on learning algorithms for so-called **deep architectures** (Hinton et al., 2006a,b; Bengio et al., 2007a; Salakhutdinov et al., 2007; Ranzato et al., 2007; Larochelle et al., 2007). These represent the learned function through many levels of composition of elements taken in a small or parametric set. The most common element type found in the above papers is the soft or hard linear threshold unit, or **artificial neuron**

$$\text{output(input)} = \sigma(w^T \text{input} + b) \tag{7.1}$$

with parameters w (vector) and b (scalar), and where $\sigma(a)$ could be $\mathbf{1}_{a>0}$, $\tanh(a)$, or $\text{sigmoid}(a) = \frac{1}{1+e^{-a}}$, for example.

Here, we are particularly interested in the restricted Boltzmann machine (Smolensky, 1986; Freund et al., 1994; Hinton, 2002; Welling et al., 2005; Carreira-Perpiñan et al., 2005), a family of bipartite graphical models with hidden variables (the hidden layer) which are used as components in building Deep Belief Networks (Hinton et al., 2006a; Bengio et al., 2007a; Salakhutdinov et al., 2007; Larochelle et al., 2007). Deep Belief Networks have yielded impressive performance on several benchmarks, clearly beating the state-of-the-art and other non-parametric learning algorithms in several cases. A very successful learning algorithm for training a restricted Boltzmann machine (RBM) is the contrastive divergence (CD) algorithm. An

RBM represents the joint distribution between a **visible** vector X which is the random variable observed in the data, and a **hidden** random variable H. There is no tractable representation of $P(X, H)$ but conditional distributions $P(H|X)$ and $P(X|H)$ can easily be computed and sampled from. CD-k is based on a Gibbs Monte-Carlo Markov Chain (MCMC) starting at an example $X = x_1$ from the empirical distribution and converging to the RBM's generative distribution $P(X)$. CD-k relies on a biased estimator obtained after a small number k of Gibbs steps (often only 1 step). Each Gibbs step is composed of two alternating sub-steps: sampling $h_t \sim P(H|X = x_t)$ and sampling $x_{t+1} \sim P(X|H = h_t)$, starting at $t = 1$.

The surprising empirical result is that even $k = 1$ (CD-1) often gives good results. An extensive numerical comparison of training with CD-k versus exact log-likelihood gradient has been presented by Carreira-Perpiñan et al. (2005). In these experiments, taking k larger than 1 gives more precise results, although very good approximations of the solution can be obtained even with $k = 1$. Here we present a follow-up to the work of Carreira-Perpiñan et al. (2005) that brings further theoretical and empirical support to CD-k, even for small k.

CD-1 has originally been justified (Hinton, 2002) as an approximation of the gradient of

$$D_{KL}(P(X_2 = \cdot\,|x_1)\|P(X = \cdot)) - D_{KL}(\hat{P}(X = \cdot)\|P(X = \cdot))$$

where D_{KL} is the Kullback-Leibler (KL) divergence, $\hat{P}$ is the empirical distribution of the training data, and $P(X_2 = \cdot\,|x_1)$ denotes the distribution of the chain after one step. The term left out in the approximation of the gradient of the KL difference is (Hinton, 2002)

$$\sum_x \frac{\partial D_{KL}(P(X_2 = \cdot\,|x_1)||P(X = \cdot)}{\partial P(X_2 = x|x_1)} \frac{\partial P(X_2 = x|x_1)}{\partial \theta} \tag{7.2}$$

which was empirically found to be small. On the one hand it is not clear how aligned are the log-likelihood gradient and the gradient with respect to the above D_{KL} difference. On the other hand it would be nice to prove that left-out terms are small in some sense. One of the motivations for this paper is to obtain the contrastive divergence algorithm from a different route, by which we can prove that the term left-out with respect to the *log-likelihood gradient* is small and converging to zero, as we take k larger.

We show that the log-likelihood and its gradient can be expanded by considering samples in a Gibbs chain. We show that when truncating the gradient expansion to k steps, the remainder converges to zero at a rate that depends on the mixing rate of the chain. The inspiration for this derivation comes from Hinton et al. (2006a): first the idea that the Gibbs chain can be associated with an infinite directed graphical model (which here we associate to an expansion of the log-likelihood and of its gradient), and second that the convergence of the chain justifies contrastive divergence

(since the k-th sample from the Gibbs chain becomes equivalent to a model sample). However, our empirical results also show that the convergence of the chain alone cannot explain the good results obtained by contrastive divergence, because this convergence becomes too slow as weights increase during training. It turns out that even when k is not large enough for the chain to converge (e.g. the typical value $k = 1$), the CD-k rule remains a good update direction to increase the log-likelihood of the training data.

Finally, we show that when truncating the series to a single sub-step we obtain the gradient of a stochastic reconstruction error. A mean-field approximation of that error is the reconstruction error often used to train autoassociators (Rumelhart et al., 1986; Bourlard et al., 1988; Hinton et al., 1994; Schwenk et al., 1995; Japkowicz et al., 2000). Auto-associators can be stacked using the same principle used to stack RBMs into a Deep Belief Network in order to train deeep neural networks (Bengio et al., 2007a; Ranzato et al., 2007; Larochelle et al., 2007). Reconstruction error has also been used to monitor progress in training RBMs by CD (Taylor et al., 2007; Bengio et al., 2007a), because it can be computed tractably and analytically, without sampling noise.

In the following we drop the $X = x$ notation and use shorthands such as $P(x|h)$ instead of $P(X = x|H = h)$. The t index is used to denote position in the Markov chain, whereas indices i or j denote an element of the hidden or visible vector respectively.

7.2 Restricted Boltzmann machines and contrastive divergence

7.2.1 Boltzmann machines

A Boltzmann machine (Hinton et al., 1984, 1986) is a probabilistic model of the joint distribution between **visible units** x, marginalizing over the values of **hidden units** h,

$$P(x) = \sum_h P(x, h) \tag{7.3}$$

and where the joint distribution between hidden and visible units is associated with a *quadratic* **energy function**

$$\mathcal{E}(x, h) = -b'x - c'h - h'Wx - x'Ux - h'Vh \tag{7.4}$$

such that

$$P(x, h) = \frac{e^{-\mathcal{E}(x,h)}}{Z} \tag{7.5}$$

where $Z = \sum_{x,h} e^{-\mathcal{E}(x,h)}$ is a normalization constant (called the partition function) and (b, c, W, U, V) are parameters of the model. b_j is called the

bias of visible unit x_j, c_i is the bias of visible unit h_i, and the matrices W, U, and V represent **interaction terms** between units*. Note that non-zero U and V mean that there are interactions between units belonging to the same layer (hidden layer or visible layer). Marginalizing over h at the level of the energy yields the so-called **free energy**:

$$\mathcal{F}(x) = -\log \sum_h e^{-\mathcal{E}(x,h)}. \tag{7.6}$$

We can rewrite the log-likelihood accordingly

$$\log P(x) = \log \sum_h e^{-\mathcal{E}(x,h)} - \log \sum_{\tilde{x},\tilde{h}} e^{-\mathcal{E}(\tilde{x},\tilde{h})} = -\mathcal{F}(x) - \log \sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})}. \tag{7.7}$$

Differentiating the above, the gradient of the log-likelihood with respect to some model parameter θ can be written as follows:

$$\begin{aligned} \frac{\partial \log P(x)}{\partial \theta} &= -\frac{\sum_h e^{-\mathcal{E}(x,h)} \frac{\partial \mathcal{E}(x,h)}{\partial \theta}}{\sum_h e^{-\mathcal{E}(x,h)}} + \frac{\sum_{\tilde{x},\tilde{h}} e^{-\mathcal{E}(\tilde{x},\tilde{h})} \frac{\partial \mathcal{E}(\tilde{x},\tilde{h})}{\partial \theta}}{\sum_{\tilde{x},\tilde{h}} e^{-\mathcal{E}(\tilde{x},\tilde{h})}} \\ &= -\sum_h P(h|x) \frac{\partial \mathcal{E}(x,h)}{\partial \theta} + \sum_{\tilde{x},\tilde{h}} P(\tilde{x},\tilde{h}) \frac{\partial \mathcal{E}(\tilde{x},\tilde{h})}{\partial \theta}. \end{aligned} \tag{7.8}$$

Computing $\frac{\partial \mathcal{E}(x,h)}{\partial \theta}$ is straightforward. Therefore, if sampling from the model was possible, one could obtain a stochastic gradient for use in training the model, as follows. Two samples are necessary: h given x for the first term, which is called the **positive phase**, and an $(\tilde{x},\tilde{h})$ pair from $P(\tilde{x},\tilde{h})$ in what is called the **negative phase**. Note how the resulting stochastic gradient estimator

$$-\frac{\partial \mathcal{E}(x,h)}{\partial \theta} + \frac{\partial \mathcal{E}(\tilde{x},\tilde{h})}{\partial \theta} \tag{7.9}$$

has one term for each of the positive phase and negative phase, with the same form but opposite signs. Let $u = (x,h)$ be a vector with all the unit values. In a general Boltzmann machine, one can compute and sample from $P(u_i|u_{-i})$, where u_{-i} is the vector with all the unit values except the i-th. Gibbs sampling with as many sub-steps as units in the model has been used to train Boltzmann machines in the past, with very long chains, yielding correspondingly long training times.

7.2.2 Restricted Boltzmann machines

In a restricted Boltzmann machine (RBM), $U = 0$ and $V = 0$ in eq. 7.4, i.e. the only interaction terms are between a hidden unit and a visible unit,

*Although through most of the paper we will denote by $x_1, \ldots, x_t, \ldots$ and $h_1, \ldots, h_t, \ldots$ vectors sampled from a Gibbs chain, we will also occasionally denote by x_j the j-th coordinate of a visible vector x, and by h_i the i-th coordinate of a hidden vector h.

but not between units of the same layer. This form of model was first introduced under the name of **Harmonium** (Smolensky, 1986). Because of this restriction, $P(h|x)$ and $P(x|h)$ factorize and can be computed and sampled from easily. This enables the use of a 2-step Gibbs sampling alternating between $h \sim P(H|X = x)$ and $x \sim P(X|H = h)$. In addition, the positive phase gradient can be obtained exactly and efficiently because the free energy factorizes:

$$\begin{aligned} e^{-\mathcal{F}(x)} = \sum_h e^{b'x+c'h+h'Wx} &= e^{b'x} \sum_{h_1} \sum_{h_2} \cdots \sum_{h_{d_h}} \prod_{i=1}^{d_h} e^{c_i h_i + (Wx)_i h_i} \\ &= e^{b'x} \sum_{h_1} e^{h_1(c_1+W_1x)} \cdots \sum_{h_{d_h}} e^{h_{d_h}(c_{d_h}+W_{d_h}x)} \\ &= e^{b'x} \prod_{i=1}^{d_h} \sum_{h_i} e^{h_i(c_i+W_ix)} \end{aligned}$$

where W_i is the i-th row of W and d_h the dimension of h. Using the same type of factorization, one obtains for example in the most common case where h_i is binary

$$-\sum_h P(h|x) \frac{\partial \mathcal{E}(x,h)}{\partial W_{ij}} = E[H_i|x] \cdot x_j, \tag{7.10}$$

where

$$E[H_i|x] = P(H_i = 1|X = x) = \text{sigmoid}(c_i + W_i x). \tag{7.11}$$

The log-likelihood gradient for W_{ij} thus has the form

$$\frac{\partial \log P(x)}{\partial W_{ij}} = P(H_i = 1|X = x) \cdot x_j - E_X[P(H_i = 1|X) \cdot X_j] \tag{7.12}$$

where E_X is an expectation over $P(X)$. Samples from $P(X)$ can be approximated by running an alternating Gibbs chain $x_1 \Rightarrow h_1 \Rightarrow x_2 \Rightarrow h_2 \Rightarrow \ldots$ Since the model P is trying to imitate the empirical distribution $\hat{P}$, it is a good idea to start the chain with a sample from $\hat{P}$, so that we start the chain from a distribution close to the asymptotic one.

In most uses of RBMs (Hinton, 2002; Carreira-Perpiñan et al., 2005; Hinton et al., 2006a; Bengio et al., 2007a) both h_i and x_j are binary, but many extensions are possible and have been studied, including cases where hidden and/or visible units are continuous-valued (Freund et al., 1994; Welling et al., 2005; Bengio et al., 2007a).

7.2.3 Contrastive divergence

The k-step contrastive divergence (CD-k, Hinton, 1999, 2002) involves a second approximation besides the use of MCMC to sample from P. This

additional approximation introduces some bias in the gradient: we run the MCMC chain for only k steps, starting from the observed example x. Using the same technique as in eq. 7.8 to express the log-likelihood gradient, but keeping the sums over h inside the free energy, we obtain

$$\begin{aligned}\frac{\partial \log P(x)}{\partial \theta} &= \frac{\partial(-\mathcal{F}(x) - \log \sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})})}{\partial \theta} \\ &= -\frac{\partial \mathcal{F}(x)}{\partial \theta} + \frac{\sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}}{\sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})}} \\ &= -\frac{\partial \mathcal{F}(x)}{\partial \theta} + \sum_{\tilde{x}} P(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}. \end{aligned} \tag{7.13}$$

The CD-k update after seeing example x is taken proportional to

$$\Delta\theta = -\frac{\partial \mathcal{F}(x)}{\partial \theta} + \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \tag{7.14}$$

where $\tilde{x}$ is a sample from our Markov chain after k steps. We know that when $k \to \infty$, the samples from the Markov chain converge to samples from P, and the bias goes away. We also know that when the model distribution is very close to the empirical distribution, i.e., $P \approx \hat{P}$, then when we start the chain from x (a sample from $\hat{P}$) the MCMC samples have already converged to P, and we need less sampling steps to obtain an unbiased (albeit correlated) sample from P.

7.3 Log-likelihood expansion via Gibbs chain

In the following we consider the case where both h and x can only take a finite number of values. We also assume that there is no pair (x, h) such that $P(x|h) = 0$ or $P(h|x) = 0$. This ensures the Markov chain associated with Gibbs sampling is **irreducible** (one can go from any state to any other state), and there exists a unique stationary distribution $P(x, h)$ the chain converges to.

Lemma 7.3.1. *Consider the irreducible Gibbs chain $x_1 \Rightarrow h_1 \Rightarrow x_2 \Rightarrow h_2 \ldots$ starting at data point x_1. The log-likelihood can be written as follows at any step t of the chain*

$$\log P(x_1) = \log \frac{P(x_1)}{P(x_t)} + \log P(x_t) \tag{7.15}$$

and since this is true for any path:

$$\log P(x_1) = E_{X_t}\left[\log \frac{P(x_1)}{P(X_t)} \middle| x_1\right] + E_{X_t}[\log P(X_t)|x_1] \tag{7.16}$$

where expectations are over Markov chain sample paths, conditioned on the starting sample x_1.

Proof. Eq. 7.15 is obvious, while eq. 7.16 is obtained by writing

$$\log P(x_1) = \sum_{x_t} P(x_t|x_1) \log P(x_1)$$

and substituting eq. 7.15. □

Note that $E_{X_t}[\log P(X_t)|x_1]$ is the negative entropy of the t-th visible sample of the chain, and it does not become smaller as $t \to \infty$. Therefore it does not seem reasonable to truncate this expansion. However, the gradient of the log-likelihood is more interesting. But first we need a simple lemma.

Lemma 7.3.2. *For any model $P(Y)$ with parameters θ,*

$$E\left[\frac{\partial \log P(Y)}{\partial \theta}\right] = 0$$

when the expected value is taken according to $P(Y)$.

Proof.

$$\begin{aligned} E\left[\frac{\partial \log P(Y)}{\partial \theta}\right] &= \sum_Y P(Y) \frac{\partial \log P(Y)}{\partial \theta} \\ &= \sum_Y \frac{P(Y)}{P(Y)} \frac{\partial P(Y)}{\partial \theta} \\ &= \frac{\partial \sum_Y P(Y)}{\partial \theta} \\ &= \frac{\partial 1}{\partial \theta} = 0. \end{aligned}$$

□

The lemma is clearly also true for conditional distributions with corresponding conditional expectations.

Theorem 7.3.3. *Consider the converging Gibbs chain $x_1 \Rightarrow h_1 \Rightarrow x_2 \Rightarrow h_2 \ldots$ starting at data point x_1. The log-likelihood gradient can be written*

$$\begin{aligned} &\frac{\partial \log P(x_1)}{\partial \theta} = \\ &-\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + E_{X_t}\left[\frac{\partial \mathcal{F}(X_t)}{\partial \theta}\middle| x_1\right] + E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right] \end{aligned} \quad (7.17)$$

and the final term (which will be shown later to be the bias of the CD estimator) converges to zero as t goes to infinity.

Proof. We take derivatives with respect to a parameter θ in the log-likelihood expansion in eq. 7.15 of lemma 7.3.1:

$$\begin{aligned}\frac{\partial \log P(x_1)}{\partial \theta} &= \frac{\partial}{\partial \theta} \log \frac{P(x_1)}{P(x_t)} + \frac{\partial \log P(x_t)}{\partial \theta} \\ &= \frac{\partial}{\partial \theta} \log e^{-\mathcal{F}(x_1)+\mathcal{F}(x_t)} + \frac{\partial \log P(x_t)}{\partial \theta} \\ &= -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + \frac{\partial \mathcal{F}(x_t)}{\partial \theta} + \frac{\partial \log P(x_t)}{\partial \theta}.\end{aligned}$$

Then we take expectations with respect to the Markov chain conditional on x_1, getting

$$\frac{\partial \log P(x_1)}{\partial \theta} = -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + E_{X_t}\left[\frac{\partial \mathcal{F}(X_t)}{\partial \theta}\middle| x_1\right] + E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right].$$

In order to prove the convergence of the CD bias towards zero, we will use the assumed convergence of the chain, which can be written

$$P(X_t = x | X_1 = x_1) = P(x) + \varepsilon_t(x) \tag{7.18}$$

with $\sum_x \varepsilon_t(x) = 0$ and $\lim_{t\to+\infty} \varepsilon_t(x) = 0$ for all x. Since x is discrete, $\varepsilon_t \stackrel{def}{=} \max_x |\varepsilon_t(x)|$ also verifies $\lim_{t\to+\infty} \varepsilon_t = 0$. Then we can rewrite the last expectation as follows:

$$\begin{aligned}E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right] &= \sum_{x_t} P(x_t|x_1) \frac{\partial \log P(x_t)}{\partial \theta} \\ &= \sum_{x_t} \left(P(x_t) + \varepsilon_t(x_t)\right) \frac{\partial \log P(x_t)}{\partial \theta} \\ &= \sum_{x_t} P(x_t) \frac{\partial \log P(x_t)}{\partial \theta} + \sum_{x_t} \varepsilon_t(x_t) \frac{\partial \log P(x_t)}{\partial \theta}.\end{aligned}$$

Using lemma 7.3.2, the first sum is equal to zero. Thus we can bound this expectation by

$$\begin{aligned}\left|E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right]\right| &\leq \sum_{x_t} |\varepsilon_t(x_t)| \left|\frac{\partial \log P(x_t)}{\partial \theta}\right| \\ &\leq \left(N_x \max_x \left|\frac{\partial \log P(x)}{\partial \theta}\right|\right) \varepsilon_t \qquad (7.19)\end{aligned}$$

where N_x is the number of discrete configurations for the random variable X. This proves the expectation converges to zero as $t \to +\infty$, since $\lim_{t\to+\infty} \varepsilon_t = 0$. □

One may wonder to what extent the above results still hold in the situation where x and h are not discrete anymore, but instead may take values in infinite (possibly uncountable) sets. We assume $P(x|h)$ and $P(h|x)$ are such

that there still exists a unique stationary distribution $P(x, h)$. Lemma 7.3.1 and its proof remain unchanged. On another hand, lemma 7.3.2 is only true for distributions P such that

$$\int_y \frac{\partial P(y)}{\partial \theta} dy = \frac{\partial}{\partial \theta} \int_y P(y) dy. \tag{7.20}$$

This equation can be guaranteed to be verified under additional "niceness" assumptions on P, and we assume it is the case for distributions $P(x)$, $P(x|h)$ and $P(h|x)$. Consequently, the gradient expansion (eq. 7.17) in theorem 7.3.3 can be obtained in the same way as before. The key point to justify further truncation of this expansion is the convergence towards zero of the bias

$$E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right]. \tag{7.21}$$

This convergence is not necessarily guaranteed unless we have convergence of $P(X_t|x_1)$ to $P(X_t)$ in the sense that

$$\lim_{t\to\infty} E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right] = E_X\left[\frac{\partial \log P(X)}{\partial \theta}\right], \tag{7.22}$$

where the second expectation is over the stationary distribution P. If the distributions $P(x|h)$ and $P(h|x)$ are such that eq. 7.22 is verified, then this limit is also zero according to lemma 7.3.2, and it makes sense to truncate eq. 7.17. Note however than eq. 7.22 does not necessarily hold in the most general case (Hernández-Lerma et al., 2003).

7.4 Connection with contrastive divergence

7.4.1 Theoretical analysis

Theorem 7.3.3 justifies truncating the series after t steps, i.e. ignoring $E_{X_t}\left[\frac{\partial \log P(X_t)}{\partial \theta}\middle| x_1\right]$, yielding the approximation

$$\frac{\partial \log P(x_1)}{\partial \theta} \simeq -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + E_{X_t}\left[\frac{\partial \mathcal{F}(X_t)}{\partial \theta}\middle| x_1\right]. \tag{7.23}$$

Note how the expectation can be readily replaced by sampling $x_t \sim P(X_t|x_1)$, giving rise to the stochastic update

$$\Delta\theta = -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + \frac{\partial \mathcal{F}(x_t)}{\partial \theta}$$

whose expected value is the above approximation. This is also exactly the CD-$(t-1)$ update (eq. 7.14).

The idea that faster mixing yields better approximation by CD-k was already introduced earlier (Carreira-Perpiñan et al., 2005; Hinton et al.,

2006a). The bound in eq. 7.19 explicitly relates the convergence of the chain – through the convergence of error ε_t in estimating $P(x)$ with $P(X_{k+1} = x|x_1)$ – to the approximation error of the CD-k gradient estimator. When the RBM weights are large it is plausible that the chain will mix more slowly because there is less randomness in each sampling step. Hence it might be advisable to use larger values of k as the weights become larger during training. It is thus interesting to study how fast the bias converges to zero as t increases, depending on the magnitude of the weights in an RBM. Markov chain theory (Schmidt, 2006) ensures that, in the discrete case,

$$\varepsilon_t = \max_x |\varepsilon_t(x)| \leq (1 - N_x a)^{t-1} \tag{7.24}$$

where N_x is the number of possible configurations for x, and a is the smallest element in the transition matrix of the Markov chain. In order to obtain a meaningful bound on eq. 7.19 we also need to bound the gradient of the log-likelihood. In the following we will thus consider the typical case of a binomial RBM, with θ being a weight W_{ij} between hidden unit i and visible unit j. Recall eq. 7.12:

$$\frac{\partial \log P(x)}{\partial W_{ij}} = P(H_i = 1|X = x) \cdot x_j - E_X[P(H_i = 1|X) \cdot X_j].$$

For any x, both $P(H_i = 1|X = x)$ and x_j are in $(0, 1)$. Consequently, the expectation above is also in $(0, 1)$ and thus

$$\left| \frac{\partial \log P(x)}{\partial W_{ij}} \right| \leq 1.$$

Combining this inequality with eq. 7.24, we obtain from eq. 7.19 that

$$\left| E_{X_t} \left[\frac{\partial \log P(X_t)}{\partial \theta} \middle| x_1 \right] \right| \leq N_x (1 - N_x a)^{t-1}. \tag{7.25}$$

It remains to quantify a, the smallest term in the Markov chain transition matrix. Each element of this matrix is of the form

$$\begin{aligned} P(x_2|x_1) &= \sum_h P(x_2|h) P(h|x_1) \\ &= \sum_h \Pi_j P(x_{2,j}|h) \Pi_i P(h_i|x_1). \end{aligned}$$

Since $1 - \text{sigmoid}(z) = \text{sigmoid}(-z)$, we have:

$$\begin{aligned} P(x_{2,j}|h) &= \begin{cases} \text{sigmoid}(h' W_{\cdot j} + b_j) \text{ if } x_{2,j} = 1 \\ \text{sigmoid}(-h' W_{\cdot j} - b_j) \text{ if } x_{2,j} = 0 \end{cases} \\ &\geq \text{sigmoid}\left(-\left|h' W_{\cdot j} + b_j\right|\right) \\ &\geq \text{sigmoid}\left(-\left(\sum_i h_i |W_{ij}| + |b_j|\right)\right) \\ &\geq \text{sigmoid}\left(-\left(\sum_i |W_{ij}| + |b_j|\right)\right). \end{aligned}$$

Let us denote $\alpha_j = \sum_i |W_{ij}| + |b_j|$, and $\beta_i = \sum_j |W_{ij}| + |c_i|$. We can obtain in a similar way that $P(h_i|x_1) \geq \text{sigmoid}(-\beta_i)$. As a result, we have that

$$\begin{aligned} a &\geq \sum_h \Pi_j \text{sigmoid}(-\alpha_j) \Pi_i \text{sigmoid}(-\beta_i) \\ &\geq N_h \Pi_j \text{sigmoid}(-\alpha_j) \Pi_i \text{sigmoid}(-\beta_i). \end{aligned} \tag{7.26}$$

In order to simplify notations (at the cost of a looser bound), let us denote

$$\alpha = \max_j \alpha_j \tag{7.27}$$

$$\beta = \max_i \beta_i. \tag{7.28}$$

Then, by combining equations 7.25 and 7.26, we finally obtain:

$$\left| E_{X_t} \left[\frac{\partial \log P(X_t)}{\partial \theta} \middle| x_1 \right] \right| \leq N_x \left(1 - N_x N_h \text{sigmoid}(-\alpha)^{d_x} \text{sigmoid}(-\beta)^{d_h} \right)^{t-1} \tag{7.29}$$

where $N_x = 2^{d_x}$ and $N_h = 2^{d_h}$. Note that although this bound is tight (and equal to zero) for any $t \geq 2$ when weights and biases are set to zero (since mixing is immediate), the bound is likely to be loose in practical cases. Indeed, the bound approaches N_x fast, as the two sigmoids decrease towards zero. However, the bound clarifies the importance of weight size in the bias of the CD approximation. It is also interesting to note that this bound on the bias decreases exponentially with the number of steps performed in the CD update, even though this decrease may become linear when the bound is loose (which is usually the case in practice): in such cases, it can be written $N_x(1-\gamma)^{t-1}$ with a small γ, and thus is close to $N_x(1-\gamma(t-1))$, which is a linear decrease in t.

If the contrastive divergence update is considered like a biased and noisy estimator of the true log-likelihood gradient, it can be shown that stochastic gradient descent converges (to a local minimum), provided that the bias is not too large (Yuille, 2005). On the other hand, one should keep in mind that for small k, there is no guarantee that contrastive divergence converges near the maximum likelihood solution (MacKay, 2001). The experiments below confirm the above theoretical results and suggest that even when the bias is large and the weights are large, the sign of the CD estimator may be generally correct.

7.4.2 Experiments

In the following series of experiments, we study empirically how the CD-k update relates to the gradient of the log-likelihood. More specifically, in order to remove variance caused by sampling noise, we are interested in

comparing two quantities:

$$\begin{aligned} \Delta_k(x_1) &= -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + E_{X_{k+1}}\left[\left.\frac{\partial \mathcal{F}(X_{k+1})}{\partial \theta}\right| x_1\right] \\ \Delta(x_1) &= -\frac{\partial \mathcal{F}(x_1)}{\partial \theta} + E_X\left[\frac{\partial \mathcal{F}(X)}{\partial \theta}\right] \end{aligned} \tag{7.30}$$

where $\Delta(x_1)$ is the gradient of the likelihood (eq. 7.13) and $\Delta_k(x_1)$ its average approximation by CD-k (eq. 7.23). The difference between these two terms is the bias $\delta_k(x_1)$, i.e., according to eq. 7.17:

$$\delta_k(x_1) = \Delta(x_1) - \Delta_k(x_1) = E_{X_{k+1}}\left[\left.\frac{\partial \log P(X_{k+1})}{\partial \theta}\right| x_1\right]$$

and, as shown in section 7.4.1, we have

$$\lim_{k \to +\infty} \delta_k(x_1) = 0.$$

Note that our analysis is different from the one by Carreira-Perpiñan et al. (2005), where the solutions (after convergence) found by CD-k and gradient descent on the negative log-likelihood were compared, while we focus on the updates themselves.

In these experiments, we use two manually generated binary datasets:

1. *Diag*$_d$ is a d-dimensional dataset containing $d+1$ samples as follows:

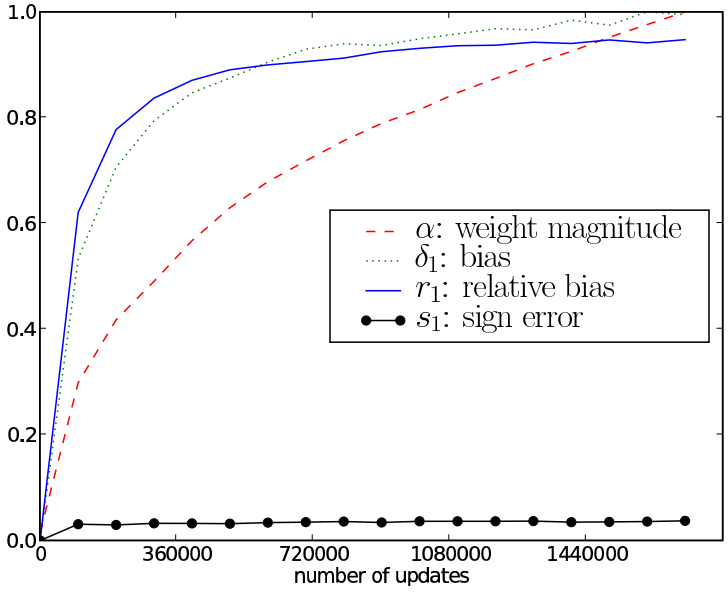
$$\begin{array}{c} \overbrace{000\ldots000}^{d \text{ bits}} \\ 100\ldots000 \\ 110\ldots000 \\ 111\ldots000 \\ \ldots \\ 111\ldots100 \\ 111\ldots110 \\ 111\ldots111 \end{array}$$

2. *1DBall*$_d$ is a d-dimensional dataset containing $2d\left\lfloor \frac{d-1}{2} \right\rfloor$ samples, representing "balls" on a one-dimensional discrete line with d pixels. Half of the data examples are generated by first picking the position b of the beginning of the ball (among d possibilities), then its width w (amongh $\left\lfloor \frac{d-1}{2} \right\rfloor$ possibilities). Pixels from b to $b+w-1$ (modulo d) are then set to 1 while the rest of the pixels are set to 0. The second half of the dataset is generated by simply "reverting" its first half (switching zeros and ones).

In order to be able to compute $\delta_k(x_1)$ exactly, only RBMs with a small (less than 10) number of visible and hidden units are used. We compute quantities for all $\theta = W_{ij}$ (the weights of the RBM connections between visible and input units). The following statistics are then computed over all weights W_{ij} and all training examples x_1:

- the weight magnitude indicators α and β, as defined in eq. 7.27 and 7.28,
- the mean of the gradient bias $|\delta_k(x_1)|$, denoted by δ_k and called the absolute bias,
- the median of $\left|\frac{\delta_k(x_1)}{\Delta(x_1)}\right|$, i.e. the relative difference between the CD-k update and the log-likelihood gradient* (we use the median to avoid numerical issues for small gradients), denoted by r_k and called the relative bias,
- the sign error s_k, i.e., the fraction of updates for which $\Delta_k(x_1)$ and $\Delta(x_1)$ have different signs.

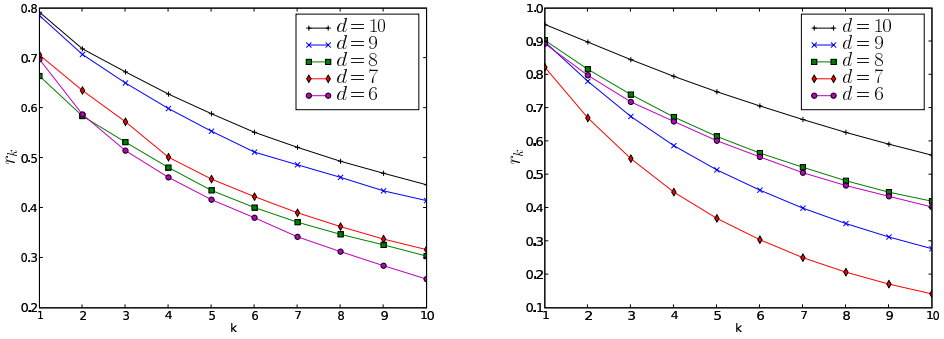
The RBM is initialized with zero biases and small weights uniformly sampled in $\left[-\frac{1}{d}, \frac{1}{d}\right]$ where d is the number of visible units. Note that even with such small weights, the bound from eq. 7.29 is already close to its maximum value N_x, so that it is not interesting to plot it on the figures. The number of hidden units is also set to d for the sake of simplicity. The RBM weights and biases are trained by CD-1 with a learning rate set to 10^{-3}: keep in mind that we are not interested in comparing the learning process itself, but rather how the quantities above evolve for different kinds of RBMs, in particular as weights become larger during training. Training is stopped once the average negative log-likelihood over training samples has less than 5% relative difference compared to its lower bound, which here is $\log(N)$, where N is the number of training samples (which are all unique).



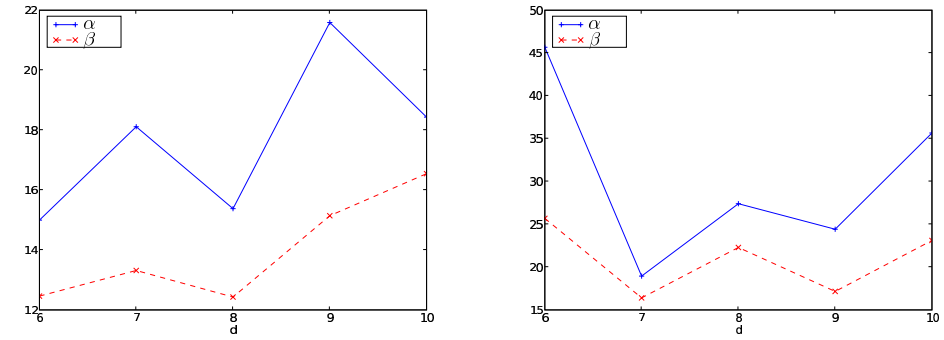
▶ **Figure 7.1.** *Typical evolution of weight magnitude α, gradient absolute bias δ_1, relative bias r_1 and sign error s_1 as the RBM is being trained by CD-1 on* 1DBall_{10}. *The size of weights α and the absolute bias δ_1 are rescaled so that their maximum value is 1, while the relative bias r_1 and the sign disagreement s_1 naturally fall within $[0, 1]$.*

*This quantity is more interesting than the absolute bias because it tells us what proportion of the true gradient of the log-likelihood is "lost" by using the CD-k update.

Figure 7.1 shows a typical example of how the quantities defined above evolve during training (β is not plotted as it exhibits the same behavior as α). As the weights increase (as shown by α), so does the absolute value of the left out term in CD-1 (δ_1), and its relative magnitude compared to the log-likelihood (r_1). In particular, we observe that most of the log-likelihood gradient is quickly lost in CD-1 (here after only 80000 updates), so that CD-1 is not anymore a good approximation of negative log-likelihood gradient descent. However, the RBM is still able to learn its input distribution, which can be explained by the fact that the "sign disagreement" s_1 between CD-1 and the log-likelihood gradient remains small (less than 5% for the whole training period).

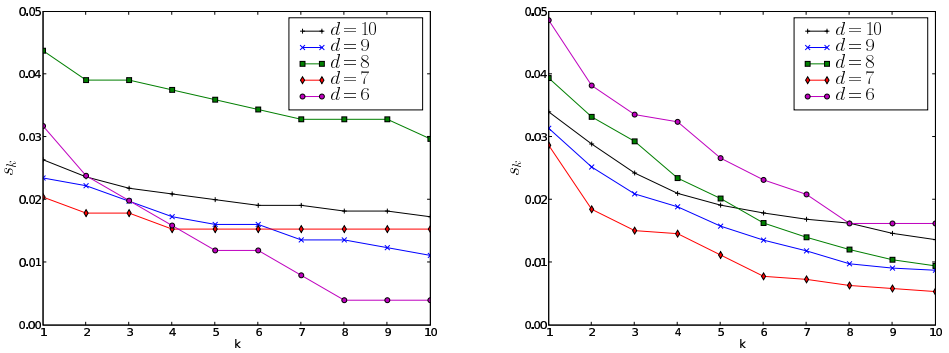


◀ **Figure 7.2.** *Median relative bias r_k between the CD-k update and the gradient of the log-likelihood, for k from 1 to 10, with input dimension $d \in \{6, 7, 8, 9, 10\}$, when the stopping criterion is reached. Left: on datasets* Diag_d. *Right: on datasets* 1DBall_d.

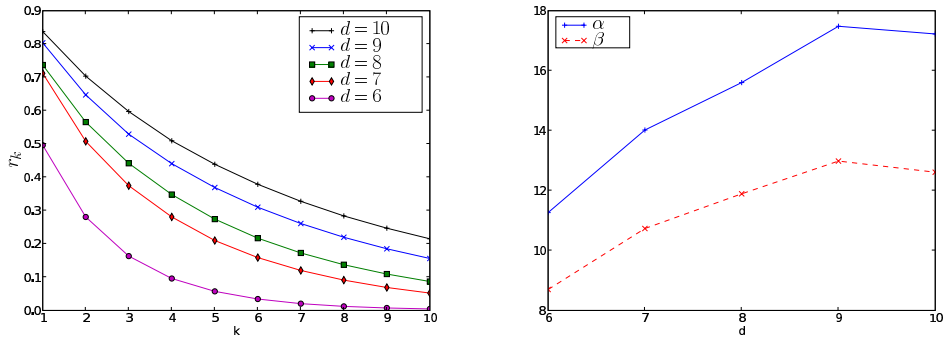


◀ **Figure 7.3.** *Measures of weight magnitude α and β as the input dimension d varies from 6 to 10, when the stopping criterion is reached. Left: on datasets* Diag_d. *Right: on datasets* 1DBall_d.

Figures 7.2 and 7.4 show how r_k and s_k respectively vary depending on the number of steps k performed in CD, on the $Diag_d$ (left) and $1DBall_d$ (right) datasets, for $d \in \{6, 7, 8, 9, 10\}$. All these values are taken when our stopping criterion is reached (i.e. we are close enough to the empirical distribution). It may seem surprising that r_k does not systematically increase with d, but remember that each RBM may be trained for a different number of iterations, leading to potentially very different weight magnitude. Figure 7.3 shows the corresponding values for α and β (which reflect the magnitude of weights): we can see for instance that α and β for dataset $1DBall_6$ are



▶ **Figure 7.4.** *Average disagreement s_k between the CD-k update and negative log-likelihood gradient descent, for k from 1 to 10, with input dimension $d \in \{6, 7, 8, 9, 10\}$, when the stopping criterion is reached. Left: on datasets* Diag_d. *Right: on datasets* 1DBall_d.



▶ **Figure 7.5.** *r_k (left) and α and β (right) on datasets* 1DBall_d, *after only 300000 training iterations: r_k systematically increases with d when weights are small (compared to figures 7.2 and 7.3).*

larger than for dataset *1DBall*$_7$, which explains why r_k is also larger, as shown in figure 7.2 (right). Figure 7.5 shows a "smoother" behavior of r_k w.r.t. d when all RBMs are trained for a fixed (small) number of iterations, illustrating how the quality of CD-k decreases in higher dimension (as an approximation to negative log-likelihood gradient descent).

We observe on figure 7.2 that the relative bias r_k becomes large not only for small k (which means the CD-k update is a poor approximation of the true log-likelihood gradient), but also for larger k in higher dimensions. As a result, increasing k moderately (from 1 to 10) still leaves a large approximation error (e.g. from 80% to 50% with $d = 10$ in figure 7.2) in spite of a 10-fold increase in computation time. This suggests that when trying to obtain a more precise estimator of the gradient, alternatives to CD-k such as persistent CD (Tieleman, 2008) may be more appropriate. On another hand, we notice from figure 7.4 that the disagreement s_k between the two updates remains low even for small k in larger dimensions (in our experiments it always remains below 5%). This may explain why CD-1 can successfully train RMBs even when connection weights become larger and the Markov chain does not mix fast anymore. An intuitive explanation for this empirical observation is the popular view of CD-k as a process that, on one hand, decreases the energy of a training sample x_1 (first term in eq. 7.30), and on

another hand increases the energy of other nearby input examples (second term), thus leading to an overall increase of $P(x_1)$.

7.5 Connection with autoassociator reconstruction error

In this section, we relate the autoassociator reconstruction error criterion (an alternative to contrastive divergence learning) to another similar truncation of the log-likelihood expansion. We can use the same approach as in theorem 7.3.3 to introduce the first hidden sample h_1 as follows:

$$\begin{aligned}\frac{\partial \log P(x_1)}{\partial \theta} &= \frac{\partial}{\partial \theta}\left(\log \frac{P(x_1)}{P(h_1)} + \log P(h_1)\right) \\ &= \frac{\partial}{\partial \theta} \log \frac{P(x_1|h_1)}{P(h_1|x_1)} + \frac{\partial \log P(h_1)}{\partial \theta}.\end{aligned}$$

Taking the expectation with respect to H_1 conditioned on x_1 yields

$$\begin{aligned}\frac{\partial \log P(x_1)}{\partial \theta} = & \quad E_{H_1}\left[\frac{\partial \log P(x_1|H_1)}{\partial \theta}\middle| x_1\right] - E_{H_1}\left[\frac{\partial \log P(H_1|x_1)}{\partial \theta}\middle| x_1\right] \\ + & \quad E_{H_1}\left[\frac{\partial \log P(H_1)}{\partial \theta}\middle| x_1\right] \end{aligned} \tag{7.31}$$

Using lemma 7.3.2, the second term is equal to zero. If we truncate this expansion by removing the last term (as is done in CD) we thus obtain:

$$\sum_{h_1} P(h_1|x_1) \frac{\partial \log P(x_1|h_1)}{\partial \theta} \tag{7.32}$$

which is an average over $P(h_1|x_1)$, that could be approximated by sampling. Note that this is not quite the negated gradient of the **stochastic reconstruction error**

$$\text{SRE} = -\sum_{h_1} P(h_1|x_1) \log P(x_1|h_1). \tag{7.33}$$

Let us consider a notion of **mean-field approximation** by which an average $E_X[f(X)]$ over configurations of a random variable X is approximated by $f(E[X])$, i.e., using the mean configuration. Applying such an approximation to SRE (eq. 7.33) gives the **reconstruction error** typically used in training autoassociators (Rumelhart et al., 1986; Bourlard et al., 1988; Hinton et al., 1994; Schwenk et al., 1995; Japkowicz et al., 2000; Bengio et al., 2007a; Ranzato et al., 2007; Larochelle et al., 2007),

$$\text{RE} = -\log P(x_1|\hat{h}_1) \tag{7.34}$$

where $\hat{h}_1 = E[H_1|x_1]$ is the mean-field output of the hidden units given the observed input x_1. If we apply the mean-field approximation to the truncation of the log-likelihood given in eq. 7.32, we obtain

$$\frac{\partial \log P(x_1)}{\partial \theta} \simeq \frac{\partial \log P(x_1|\hat{h}_1)}{\partial \theta}.$$

It is arguable whether the mean-field approximation per se gives us license to include in $\frac{\partial \log P(x_1|\hat{h}_1)}{\partial \theta}$ the effect of θ on $\hat{h}_1$, but if we do so then we obtain the gradient of the reconstruction error (eq. 7.34), up to the sign (since the log-likelihood is maximized while the reconstruction error is minimized).

As a result, whereas CD-1 truncates the chain expansion at x_2 (as seen in section 7.2.3), ignoring

$$E_{X_2}\left[\left.\frac{\partial \log P(X_2)}{\partial \theta}\right| x_1\right],$$

we see (using the fact that the second term of 7.31 is zero) that reconstruction update truncates the chain expansion one step earlier (at h_1), ignoring

$$E_{H_1}\left[\left.\frac{\partial \log P(H_1)}{\partial \theta}\right| x_1\right]$$

and working on a mean-field approximation instead of a stochastic approximation. The reconstruction error gradient can thus be seen as a more biased approximation of the log-likelihood gradient than CD-1. Comparative experiments between reconstruction error training and CD-1 training confirm this view (Bengio et al., 2007a; Larochelle et al., 2007): CD-1 updating generally has a slight advantage over reconstruction error gradient.

However, reconstruction error can be computed deterministically and has been used as an easy method to monitor the progress of training RBMs with CD, whereas the CD-k itself is generally not the gradient of anything and is stochastic.

7.6 Conclusion

This paper provides a theoretical and empirical analysis of the log-likelihood gradient in graphical models involving a hidden variable h in addition to the observed variable x, and where conditionals $P(h|x)$ and $P(x|h)$ are easy to compute and sample from. That includes the case of contrastive divergence for restricted Boltzmann machines (RBM). The analysis justifies the use of a short Gibbs chain of length k to obtain a biased estimator of the log-likelihood gradient. Even though our results do not guarantee that the bias decreases monotically with k, we prove a bound that does, and observe this decrease experimentally. Moreover, although this bias may be large when using only few steps in the Gibbs chain (as is usually done in practice), our

empirical analysis indicates this estimator remains a good update direction compared to the true (but intractable) log-likelihood gradient.

The analysis also shows a connection between reconstruction error, log-likelihood and contrastive divergence (CD), which helps understand the better results generally obtained with CD and justify the use of reconstruction error as a monitoring device when training an RBM by CD. The generality of the analysis also opens the door to other learning algorithms in which $P(h|x)$ and $P(x|h)$ do not have the parametric forms of RBMs.

7.7 Commentaires

Une des raisons pour lesquelles nous nous sommes penchés sur l'algorithme de divergence contrastive est qu'il s'agit d'un algorithme d'apprentissage qui s'est avéré très utile en pratique, étant très efficace d'un point de vue computationnel. Avant cet algorithme, il semblait vain de tenter d'optimiser les paramètres d'un tel modèle à l'aide du gradient de la log-vraisemblance, dans la mesure où ce gradient ne peut être calculé de manière efficace. Il est donc très intéressant d'analyser comment la divergence contrastive peut rapidement obtenir une approximation de ce gradient qui donne de bons résultats en terme de qualité d'optimisation. Nous montrons dans ce chapitre que même si la divergence contrastive ne peut être considérée en pratique comme une bonne approximation du gradient – à moins de faire un grand nombre de pas d'échantillonnage dans la chaîne de Gibbs, ce qui ne serait pas efficace – elle fournit tout de même une bonne direction d'optimisation.

Notons que par la suite, plusieurs variantes de la divergence contrastive ont été proposées dans le but d'obtenir une meilleure approximation du gradient (Tieleman, 2008; Desjardins et al., 2010; Salakhutdinov, 2010). Ces variantes visent à améliorer l'échantillonnage des exemples x_t pour que leur distribution soit plus proche de la distribution du modèle. Une telle amélioration se traduit généralement par un apprentissage de meilleure qualité (c.à.d. une meilleure modélisation de la distribution des exemples observés). Il reste toutefois difficile d'évaluer objectivement différents algorithmes d'apprentissage pour les RBMs ayant beaucoup d'unités cachées et visibles, qui sont généralement les plus intéressantes en pratique. Le calcul de leur fonction de partition ne pouvant se faire efficacement de manière exacte, des approximations sont nécessaires (Neal, 2001; Salakhutdinov et al., 2008; Desjardins et al., 2011), et ces approximations rajoutent de l'incertitude sur la fiabilité des résultats obtenus.

Bibliographie

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007a, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bengio, Y. et Y. LeCun. 2007b, "Scaling learning algorithms towards AI", dans *Large Scale Kernel Machines*, édité par L. Bottou, O. Chapelle, D. DeCoste et J. Weston, MIT Press.

Bourlard, H. et Y. Kamp. 1988, "Auto-association by multilayer perceptrons and singular value decomposition", *Biological Cybernetics*, vol. 59, p. 291–294.

Carreira-Perpiñan, M. A. et G. E. Hinton. 2005, "On contrastive divergence learning", dans *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, édité par R. G. Cowell et Z. Ghahramani, Society for Artificial Intelligence and Statistics, p. 33–40.

Desjardins, G., A. Courville et Y. Bengio. 2011, "On tracking the partition function", dans *NIPS'2011*.

Desjardins, G., A. Courville, Y. Bengio, P. Vincent et O. Delalleau. 2010, "Tempered Markov chain monte carlo for training of restricted Boltzmann machine", dans *JMLR W&CP : Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, vol. 9, p. 145–152.

Freund, Y. et D. Haussler. 1994, "Unsupervised learning of distributions on binary vectors using two layer networks", rapport technique UCSC-CRL-94-25, University of California, Santa Cruz.

Hernández-Lerma, O. et J. B. Lasserre. 2003, *Markov Chains and Invariant Probabilities*, Birkhäuser Verlag.

Hinton, G. E. 1999, "Products of experts", dans *Proceedings of the Ninth International Conference on Artificial Neural Networks (ICANN)*, vol. 1, IEE, Edinburgh, Scotland, p. 1–6.

Hinton, G. E. 2002, "Training products of experts by minimizing contrastive divergence", *Neural Computation*, vol. 14, p. 1771–1800.

Hinton, G. E., S. Osindero et Y. Teh. 2006a, "A fast learning algorithm for deep belief nets", *Neural Computation*, vol. 18, p. 1527–1554.

Hinton, G. E. et R. Salakhutdinov. 2006b, "Reducing the Dimensionality of Data with Neural Networks", *Science*, vol. 313, p. 504–507.

Hinton, G. E. et T. J. Sejnowski. 1986, "Learning and relearning in Boltzmann machines", dans *Parallel Distributed Processing : Explorations in the Microstructure of Cognition. Volume 1 : Foundations*, édité par D. E. Rumelhart et J. L. McClelland, MIT Press, Cambridge, MA, p. 282–317.

Hinton, G. E., T. J. Sejnowski et D. H. Ackley. 1984, "Boltzmann machines : Constraint satisfaction networks that learn", rapport technique TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science.

Hinton, G. E. et R. S. Zemel. 1994, "Autoencoders, minimum description length, and helmholtz free energy", dans *Advances in Neural Information Processing Systems 6 (NIPS'93)*, édité par D. Cowan, G. Tesauro et J. Alspector, Morgan Kaufmann Publishers, Inc., p. 3–10.

Japkowicz, N., S. J. Hanson et M. A. Gluck. 2000, "Nonlinear autoassociation is not equivalent to PCA", *Neural Computation*, vol. 12, n^{o} 3, p. 531–545.

Larochelle, H., D. Erhan, A. Courville, J. Bergstra et Y. Bengio. 2007, "An empirical evaluation of deep architectures on problems with many factors of variation", dans *Proceedings of the 24th International Conference on Machine Learning (ICML'07)*, édité par Z. Ghahramani, ACM, p. 473–480.

MacKay, D. 2001, "Failures of the one-step learning algorithm", Unpublished report.

Neal, R. M. 2001, "Annealed importance sampling", *Statistics and Computing*, vol. 11, n° 2, p. 125–139.

Ranzato, M., C. Poultney, S. Chopra et Y. LeCun. 2007, "Efficient learning of sparse representations with an energy-based model", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 1137–1144.

Rumelhart, D. E., G. E. Hinton et R. J. Williams. 1986, "Learning representations by back-propagating errors", *Nature*, vol. 323, p. 533–536.

Salakhutdinov, R. 2010, "Learning in Markov random fields using tempered transitions", dans *Advances in Neural Information Processing Systems 22 (NIPS'09)*, édité par Y. Bengio, D. Schuurmans, C. Williams, J. Lafferty et A. Culotta.

Salakhutdinov, R. et G. E. Hinton. 2007, "Semantic hashing", dans *Proceedings of the 2007 Workshop on Information Retrieval and applications of Graphical Models (SIGIR'07)*, Elsevier, Amsterdam.

Salakhutdinov, R. et I. Murray. 2008, "On the quantitative analysis of deep belief networks", dans *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, vol. 25, édité par W. W. Cohen, A. McCallum et S. T. Roweis, ACM, p. 872–879.

Schmidt, V. 2006, "Markov chains and monte-carlo simulation", dans *Lecture Notes, Summer 2006*, Ulm University, Department of Stochastics.

Schwenk, H. et M. Milgram. 1995, "Transformation invariant autoassociation with application to handwritten character recognition", dans *Advances in Neural Information Processing Systems 7 (NIPS'94)*, édité par G. Tesauro, D. Touretzky et T. Leen, MIT Press, p. 991–998.

Smolensky, P. 1986, "Information processing in dynamical systems : Foundations of harmony theory", dans *Parallel Distributed Processing*, vol. 1, édité par D. E. Rumelhart et J. L. McClelland, chap. 6, MIT Press, Cambridge, p. 194–281.

Taylor, G., G. E. Hinton et S. Roweis. 2007, "Modeling human motion using binary latent variables", dans *Advances in Neural Information Processing*

Systems 19 (NIPS'06), édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, Cambridge, MA, p. 1345–1352.

Tieleman, T. 2008, "Training restricted Boltzmann machines using approximations to the likelihood gradient", dans *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, édité par W. W. Cohen, A. McCallum et S. T. Roweis, ACM, p. 1064–1071.

Welling, M., M. Rosen-Zvi et G. E. Hinton. 2005, "Exponential family harmoniums with an application to information retrieval", dans *Advances in Neural Information Processing Systems 17 (NIPS'04)*, édité par L. Saul, Y. Weiss et L. Bottou, MIT Press, Cambridge, MA, p. 1481–1488.

Yuille, A. L. 2005, "The convergence of contrastive divergences", dans *Advances in Neural Information Processing Systems 17 (NIPS'04)*, édité par L. Saul, Y. Weiss et L. Bottou, MIT Press, p. 1593–1600.

8 Beyond skill rating: advanced matchmaking in Ghost Recon Online

O. Delalleau, E. Contal, E. Laufer, R. Chandias Ferrari, Y. Bengio et F. Zhang
Accepté pour publication dans *IEEE Transactions on Computational Intelligence and AI in Games*, 2012

Si la recherche en apprentissage machine est un domaine passionnant en soi, aboutir à des applications pratiques motive et valorise ces recherches. Ce chapitre présente une telle application pratique de techniques d'apprentissage machine appliquées aux jeux vidéo, dans le cadre d'une chaire de recherche industrielle du CRSNG – établissant un partenariat entre l'Université de Montréal et Ubisoft. Le but de cette application est d'améliorer la qualité de l'appariement de joueurs ("matchmaking") dans les jeux multi-joueurs en ligne, spécifiquement ici pour le jeu *Ghost Recon Online*. Dans ce jeu, les joueurs sont groupés en deux équipes qui s'affrontent (chaque joueur incarnant un soldat sur le champ de bataille), et le problème de l'appariement de joueurs est de trouver les compositions d'équipes qui rendront la partie la plus intéressante possible pour les joueurs impliqués. Il est bien connu qu'une sélection aléatoire des équipes risque de nuire au plaisir des joueurs, en particulier à cause des différences de niveau et de style de jeu entre joueurs. Les approches classiques utilisées jusqu'à présent dans les jeux se basent sur une simple estimation du niveau de chaque joueur, et dans ce chapitre nous montrons qu'il est possible de faire encore mieux avec des techniques d'apprentissage machine plus sophistiquées. Bien que la question de l'efficacité ne soit pas directement traitée dans l'article – qui insiste plus sur les nouveaux algorithmes proposés – elle est cruciale au succès de ce projet, et sera abordée plus en détails dans les commentaires de fin de chapitre.

Contribution personnelle L'algorithme de base utilisé pour prédire le vainqueur d'un match a été imaginé par Y. Bengio. Je l'ai ensuite étendu pour prédire la satisfaction du joueur. J'ai supervisé l'écriture du code des algorithmes (et j'en ai écrit moi-même environ 20%), tandis que j'ai implémenté la quasi-totalité du code nécessaire à l'intégration de ces algorithmes dans l'environnement du jeu (ce qui représente encore plus de code que la partie "algorithmes purs"). J'ai réalisé la grande majorité des expériences rapportées dans l'article, que j'ai rédigé dans son intégralité.

Abstract Player satisfaction is particularly difficult to ensure in online games, due to interactions with other players. In adversarial multiplayer games, matchmaking typically consists in trying to match together players of similar skill level. However, this is usually based on a single skill value, and assumes the only factor of "fun" is the game balance. We present a more advanced matchmaking strategy developed for *Ghost Recon Online*, an upcoming team-focused First Person Shooter from Ubisoft. We first show how incorporating more information about players than their raw skill can lead to more balanced matches. We also argue that balance is not the only factor that matters, and present a strategy to explicitly maximize the players' fun, taking advantage of a rich player profile that includes information about player behavior and personal preferences. Ultimately, our goal is to ask players to provide direct feedback on match quality through an in-game survey. However, because such data was not available for this study, we rely here on heuristics tailored to this specific game. Experiments on data collected during *Ghost Recon Online*'s beta tests show that neural networks can effectively be used to predict both balance and player enjoyment.

8.1 Introduction

Making games appealing to a wide audience is a core objective of modern video games (Bateman et al., 2005). This objective has been driving a significant amount on research on "player-centered" game design (Charles et al., 2005). Most of this research has been focused on adapting games to players individually, e.g. by dynamically generating quests in an online Role-Playing-Game (Shi et al., 2004), adapting tracks in a racing game (Togelius et al., 2007), or dynamically adjusting the difficulty of an arcade game (Missura et al., 2009). However, making the game enjoyable for all players in a multiplayer games requires taking into account player interactions. Those are difficult to control, but a good matchmaking process can increase the chance of players having fun with each other, thus improving player retention (Butcher, 2008).

◀ **Figure 8.1.** *In* Ghost Recon Online, *players have access to various character classes, with unique powers, weapons and high-tech gear. This opens up a wide array of potential playstyles, that basic skill rating algorithms are unable to fully capture.*

The algorithms described in this paper have been designed for the matchmaking system of *Ghost Recon Online*, an online First Person Shooter (FPS) currently being developed by Ubisoft. In this game players control elite soldiers with modern weapons and high-tech equipment (figure 8.1), and teams fight against each other in various game modes. These modes include for instance "Capture" (teams fight to capture and control a given number of points on the map) and "Assault" (one team is defending a position which the other team is attacking). The matchmaking task consists in building teams from a pool of players willing to join an adversarial multiplayer match, in a way that maximizes players' enjoyment*. This challenge is traditionally solved by assigning a skill rating to each player (inferred from his previous match results), deriving team ratings from individual skills of all players in a team, then having teams of similar strengths fight each other. This is for instance the basic idea behind the TrueSkill matchmaking system developed by Microsoft for their Xbox Live online gaming service (Herbrich et al., 2007). The motivation is that the game is not fun if a match is unbalanced, as weaker players get frustrated while experienced players get bored (even though getting easy kills may initially be fun).

The research we present here aims to address two limitations of such skill-based matchmaking systems:

- Because skill ratings are often used for player ranking (e.g. online leaderboards) in addition to matchmaking, such ratings are usually uni-dimensional (they result in a single number representing a player's overall proficiency in the game). However, complex games like modern FPS require skills in multiple areas like reflex, planning, tactical analysis or teamwork. The relative importance of these skills depends on the map, game mode (e.g. Deathmatch vs. Capture the Flag), player roles, team compositions, etc. Since in this work our goal is to match players together rather than rank them, we can take advantage of a richer player profile and additional contextual information to predict the game balance, instead of relying on a single skill number.

- Although it seems safe to assume that an unbalanced match is not fun (at least for the weaker team), skill-based matchmaking systems implicitly assume the reverse is also true ("a balanced match is fun"), which is not as obvious. For instance in an FPS, having two teams of campers† will most likely lead to a boring match where no action ever happens, even if the match is perfectly balanced.

Our methodology to tackle these challenges consists in using machine learning algorithms (more specifically neural networks) to predict the match

*Note that here we focus on situations where only two teams face each other and the game is balanced for teams of equal size, but our approach could be generalized to more generic settings as well.

† *Campers* are players who tend to stay still, waiting to ambush enemies.

winner and a measure of individual player enjoyment. These predictions are based on information about players involved in the match as well as on the match's specific settings. The information on players is derived from historical data, taking into account both previous match results and player attributes collected by tracking player behavior over time. Defining what makes a game fun is an interesting but challenging task that has been a research topic for a long time (Malone, 1981; Yannakakis et al., 2007), and we do not intend to solve it here. Instead, we plan to rely on user input, by asking players to regularly provide feedback on their online gaming experience through in-game surveys. However, such survey data was not available yet for this study, so instead we handcrafted a "fun formula" that we used to validate our approach. This formula is based on events tracked during each match (like kills and deaths in an FPS). We will show in our experimental results that our neural network model for fun prediction outperforms skill-based systems like TrueSkill and our own match balance predictor, on the task of finding the matches most likely to be fun for all players involved.

8.2 Neural network models

In this section we describe the neural network architectures we have been using. The two opposing teams are denoted by team A and team B respectively. Note that team order is not random: it is arbitrarily fixed by the map settings, for instance on a given map the team starting from the South area would always be team A, while the team starting from the North area would always be team B. This allows the network to take into account the fact that maps may not be symmetric.

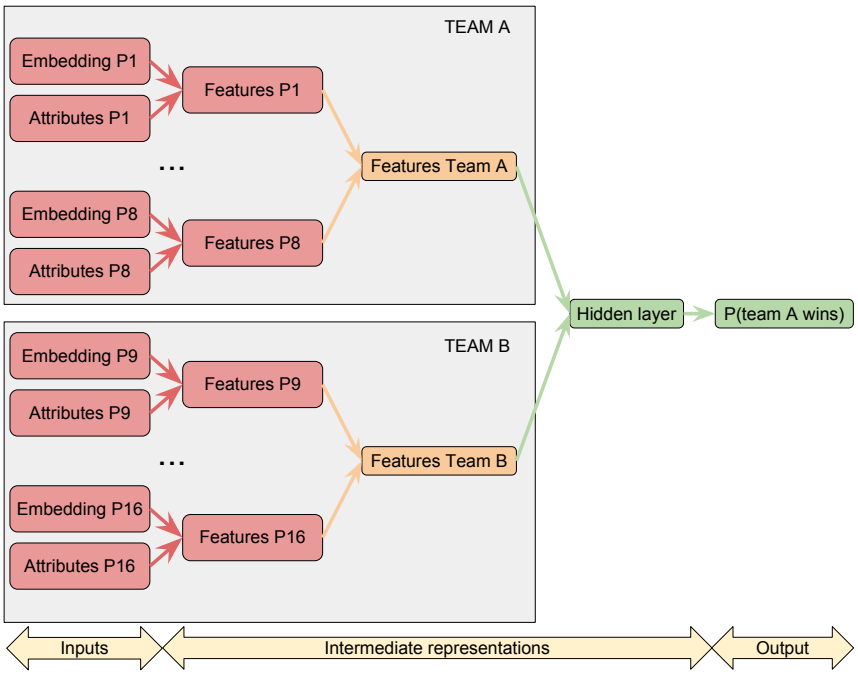
In the following we assume that each team can have up to eight players to keep notations simple, but in general the maximum number of players per team depends on the game mode. Note also that although the matchmaking system (which we will describe in section 8.3) attempts to find matches where teams are balanced and at full size, it may sometimes be forced to start a match with fewer players when not enough players are available.

8.2.1 Predicting match balance

In order to estimate match balance, we train a neural network whose output is the probability that team A wins over team B (the idea is that a match is balanced when this probability is close to 50%). The network's architecture is shown in figure 8.2.

The network's inputs are the player *profiles*. A profile is the combination of an *embedding* and an *attributes* vectors:

- The n_e-dimensional *embedding* vector $\mathbf{e}_i$ of player i is automatically learned during the training phase, and can be seen as a set of num-



◀ **Figure 8.2.** *Neural network computing the probability that team A wins, based on the profiles (embeddings and attributes) of all players involved in the match (players P1 to P8 in team A, and players P9 to P16 in team B).*

bers that summarize previous matches in which a player participated. These numbers cannot be easily interpreted by a human, but the neural network can use them to tweak its predictions so that they better match the players' individual playstyles. For instance, in this architecture, the embedding vector is expected to contain information about the player skill in various aspects of the game ("good sniper", "poor assault", etc.).

- The n_a-dimensional *attributes* vector $\mathbf{a}_i$ of player i is a set of normalized statistics that are extracted from the game logs. It contains for instance the average kill / death ratio of the player, the number of matches he played, his firing accuracy, etc.

The input profiles are successively transformed as follows:

1. The profile information (embedding and attributes) are linearly combined into a single vector of *player features*

$$\mathbf{p}_i = \mathbf{e}_i + \mathbf{W}\mathbf{a}_i \tag{8.1}$$

with $\mathbf{W}$ an $(n_e \times n_a)$ matrix. One may think of these features as a summary of the profile, containing an estimate of a player's skills in various areas of the game, given his history.

2. For each team $j \in \{A, B\}$, *team features* $\mathbf{t}_j$ are computed as the sum

of all player features in the team:

$$\mathbf{t}_j = \sum_{i \in \text{team } j} \mathbf{p}_i. \tag{8.2}$$

3. Team features are compared and summarized by a non-linear transformation into the *hidden layer* $\mathbf{h}$ defined as

$$\mathbf{h} = \tanh\left(\mathbf{b} + \sum_{j \in \{A,B\}} \mathbf{V}_j \mathbf{t}_j\right) \tag{8.3}$$

with $\mathbf{b}$ an n_h-dimensional vector and $\mathbf{V}_A$ and $\mathbf{V}_B$ two $(n_h \times n_e)$ matrices. Note that here, the tanh function is applied on a vector: this is a shortcut notation to represent an element-wise tanh operation on each element of this vector.

4. The last step of the computation is a single sigmoid unit computing the probability α that team A wins by

$$\alpha = \text{sigmoid}(\mathbf{u} \cdot \mathbf{h} + c) \tag{8.4}$$

where $\mathbf{u}$ is an n_h-dimensional vector, c is a scalar, and $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$.

The model described above is one of the most basic that fits our approach, but we will see in experiments that it can already yield a significant improvement when compared to a rating-based system like TrueSkill (Herbrich et al., 2007). It is likely that more complex architectures will be able to reach even higher accuracy. Potential improvements include:

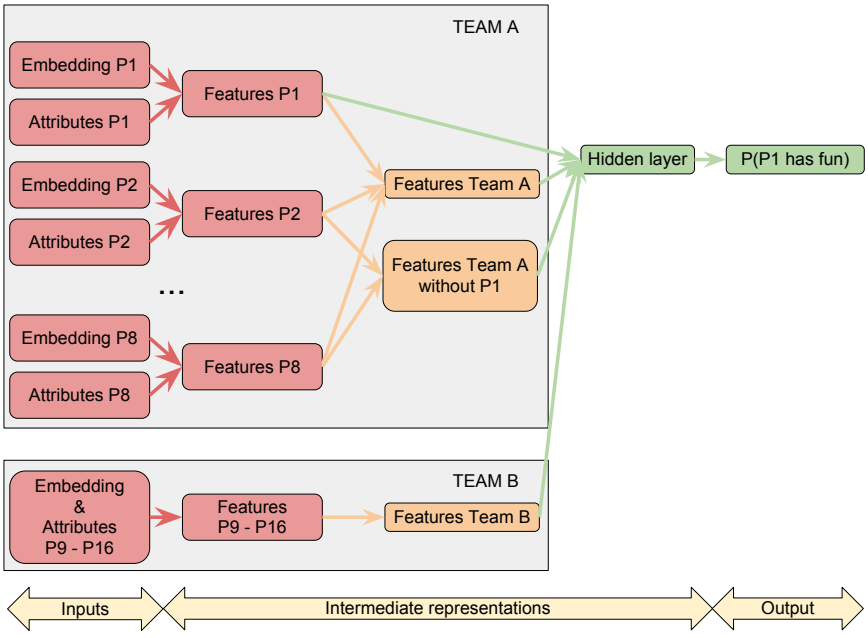
- Performing feature extraction on player attributes to extract high-level information on playstyle, e.g. through unsupervised pre-training of deep neural networks (Bengio et al., 2007; Hinton et al., 2006). This may be especially useful as more player attributes are added to the player profile.
- Trying more pooling operations than the sum performed in eq. 8.2, e.g. also concatenating the mean, standard deviation, (soft)min, (soft)max, etc.
- Adding additional hidden layers to learn a decision function more complex than eqs. 8.3 and 8.4. Recent work on discriminative deep networks may be useful in this regard (Rifai et al., 2011).

Note also that in order to take map and game mode into account, we propose to learn different matrices $\mathbf{V}_A$ and $\mathbf{V}_B$ (see eq. 8.3) for each map and game mode. This will allow different maps / modes to favor specific skills, as well as to weigh differently the contributions of the two teams (which

may be important in unbalanced maps, or in an asymmetric game mode like "Assault"). However, this strategy could not be evaluated yet because our current dataset is limited to a single map and game mode (see experiments in section 8.4.1).

8.2.2 Predicting player enjoyment

For the purpose of fun prediction, we assume that each match in the training set is labeled with a target vector whose i-th element is 1 if player i had fun during the match, and 0 otherwise. This label may come from an in-game player survey, or could be computed from prior knowledge on what makes the game fun. Note that some elements of this vector may be missing, either because some players skipped the survey in the first case, or because we did not have enough confidence in our "fun estimator" in the second. The neural network architecture we use, depicted in figure 8.3, differs from the one used for balance (figure 8.2) in that it predicts a player-dependent output (the probability that a specific player has fun in the match), instead of a single global value.



◀ **Figure 8.3.** *Neural network computing the probability that player P1 has fun, based on the profiles (embeddings and attributes) of all players involved in the match (players P1 to P8 in team A, and players P9 to P16 in team B). Team B computations are identical to those in figure 8.2 and are not shown in details here. Note that similar networks are defined to compute the probabilities that players P2 to P16 have fun (and all these networks share the same weight parameters).*

In particular, in this architecture the hidden layer takes as input the feature vector of the player whose fun is being estimated and the feature vector of the rest of his team, in addition to the feature vectors of both (full) teams. The motivation behind this specific connectivity pattern is that in order to compute in the hidden layer useful information about how likely a player is to have fun, we would like to take into account (i) the player's individual profile, (ii) the profiles of his teammates, and (iii) the

global profiles of the two teams.

The inputs (profiles $\mathbf{e}_i$ and attributes $\mathbf{a}_i$) are the same as in section 8.2.1, and player features are also computed by eq. 8.1. However, if for instance we want to estimate the fun of player i in team A, the hidden layer $\mathbf{h}$ is now computed by

$$\mathbf{h} = \tanh\left(\mathbf{b} + \mathbf{Y}\mathbf{p}_i + \mathbf{V}_1\mathbf{t}_A + \mathbf{V}_2\mathbf{t}_B + U \sum_{k\in\text{team j},k\neq i} p_k\right).$$

Note that if player i was in team B, this formula would instead use $\mathbf{V}_1\mathbf{t}_B + \mathbf{V}_2\mathbf{t}_A$. Finally, the output probability $P(\text{Player } i \text{ has fun}) = \alpha$ is given again by eq. 8.4. The extensions of the balance predictor mentioned at the end of section 8.2.1 can also be considered for this model, in particular matrices $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{Y}$, $\mathbf{U}$ may be learned independently for each unique map and game mode.

8.3 Architecture

Although the main focus of this paper is on the new machine learning models described above, it is also important to understand how they are integrated into the game. In this section we briefly answer the three following questions:

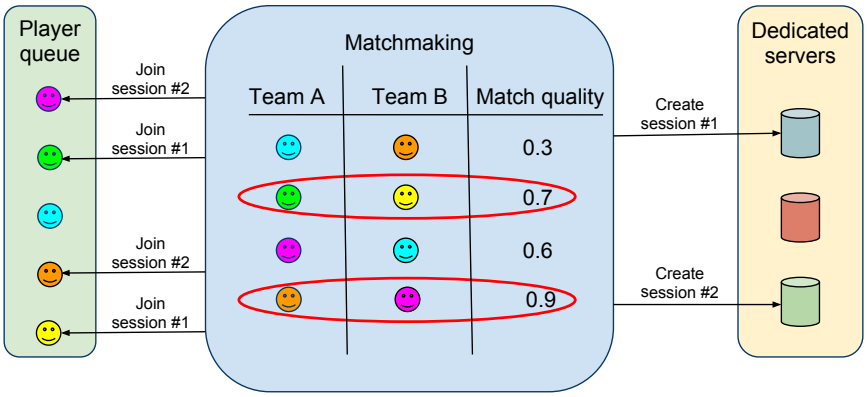
- How are players matched together?
- Where does the training data come from?
- How are the model parameters optimized?

8.3.1 Matchmaking

Once the models described above have been trained, how should they be used in the matchmaking process? Figure 8.4 gives a simplified view of the global architecture (with only 1v1 matches for the sake of clarity). Players who want to join an online match are placed in a queue from which the matchmaking algorithm randomly samples to try various team combinations. Match candidates are scored by one of the neural network models described in section 8.2 to obtain an estimate of match quality. The matchmaking server then launches those with highest scores.

Without going too deep into the details, the following points are worth pointing out:

- The sampling strategy has an important role: in particular it can ensure all players are in a compatible skill range and have a good connection with the same dedicated server. It should also favor players who have been waiting for a longer time, to minimize the wait.



◀ **Figure 8.4.** *Sketch of the matchmaking process: various match compositions are evaluated by random sampling from the player queue. The best matches are launched on dedicated servers that chosen players are instructed to join.*

- There is a trade-off between sampling time and match quality: we sample as many matches as possible while maintaining matchmaking wait below a given threshold.

- This sampling scheme is well suited to distributed computations, so the number of match candidates that can be evaluated mostly depends on the amount of processing power available.

- A similar approach is used for "hot-join" situations, i.e. when selecting players best fit to fill open slots in ongoing matches.

8.3.2 Data collection

After each match, the game saves into a database all relevant information like which team won, which objectives were completed, who were the players in each team, when they joined and left the game, how many kills they got, how many deaths, etc. These statistics are accompanied by a "snapshot" of the players' state (for all players involved in the game that just ended), which includes additional data like current gear, level, special abilities, etc. Then, a parser reads these logs from the database and generates the corresponding match results and player attributes, which form the basis of the training data.

Another source of data collection is the in-game player survey (which had not yet been activated at the time of writing this paper). This survey pops up after every match (or less frequently if needed), and asks in particular whether (i) the player had fun, and (ii) he thought the match was balanced. The first answer will be used to train and evaluate our model for player enjoyment, while the second one will provide us with another way to compare game balance models. Additional survey questions may also be used for the purpose of player modeling (see section 8.5.3).

8.3.3 Model optimization

Training is split in two phases, which we call respectively *offline training* and *online update*. Offline training may be slow, and is meant to periodically provide a "fresh" model optimized on a large amount of data, to be deployed for instance during a weekly maintenance window. On the contrary, the online update needs to be fast enough to update the model in real time from the results of matches being played online.

The offline training phase consists in learning two kinds of parameters:

- The parameters governing the network transformations. For instance for the winner prediction model described in section 8.2.1, this set of parameters is $\{\mathbf{W}, \mathbf{V}_A, \mathbf{V}_B, \mathbf{b}, \mathbf{u}, c\}$.

- The players' embeddings.

We optimize our models by stochastic gradient descent, minimizing the Negative Log-Likelihood (NLL, see e.g. Bishop, 2006) of the model's prediction (in the fun prediction task, missing targets are ignored). The evolution of player embeddings through time (modeling the fact that we expect players to evolve as they play more matches) is currently considered linear in the number of matches that have been played, i.e.

$$\mathbf{e}_{ik} = \mathbf{e}_i^0 + k\mathbf{e}_i^1$$

where $\mathbf{e}_{ik}$ is the embedding of player i after he has played k matches, and the embedding parameters $\mathbf{e}_i^0$ and $\mathbf{e}_i^1$ are optimized by the gradient descent algorithm. Note that a linear evolution is most likely sub-optimal, and we plan to experiment with other variants in future work.

The online update phase takes place once the model is deployed and new matches are being played. At this point, the network's transformation parameters are kept fixed, but we use the information available from new matches to update the players' embeddings. Whenever a match ends, we recover from the database the composition of the last few* matches of all players involved in the match that just ended. The prediction error is then minimized on this small subset of the data, by a fast *conjugate gradient descent* optimization algorithm (Shewchuk, 1994) optimizing only the players' embeddings. This ensures that embeddings always reflect the recent matches of the players (since if they were kept fixed, or optimized with a slow optimization method like stochastic gradient descent, they would become outdated after a while).

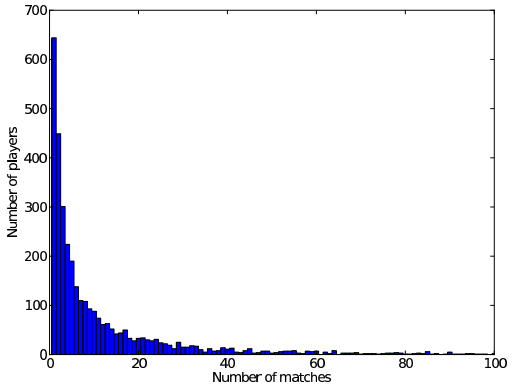
*The number of matches to recover needs to be validated to obtain good performance, both in terms of prediction accuracy and speed.

8.4 Experiments

8.4.1 Dataset

The data at our disposal for this study consists in matches played during an early *Ghost Recon Online* beta-test. All matches were played on the same map and the same multiplayer mode ("Capture"), with up to eight players in each team. After filtering out uninteresting matches (those that are too short or involve less than two players per team), the dataset contains 3937 matches involving 3444 unique players. The histogram of the number of matches per player is shown in figure 8.5. For each player we use the following attributes:

- number of matches played
- sum and mean of kills and deaths
- average kill / death ratio
- sum and mean of number of captures
- TrueSkill skill estimate
- mean and standard deviation of firing accuracy and headshot percentage



◀ **Figure 8.5.** *Distribution of the number of matches per player (truncated to a maximum of 100 matches to keep the figure easy to read – very few players played above 100 matches).*

8.4.2 Algorithms

In the experimental results that follow, we call *BalanceNet* the neural network model that is trained to predict the probability that team A wins (section 8.2.1) and *FunNet* the one that predicts the probabilities that players have fun in the match (section 8.2.2). We compare them to two variants of the TrueSkill algorithm (Herbrich et al., 2007):

- *TrueSkill-Team* only takes match results into account (i.e. "vanilla" TrueSkill).

- *TrueSkill-Player* actually ignores the winning team, focusing instead of individual player performance by ranking players according to their in-game score (a function of their achievements during the match, i.e. for instance kills and captures). This amounts to pretending that an 8v8 match is actually a free-for-all (each player is a team by himself)*.

Each algorithm has a number of hyper-parameters that need to be set carefully. For instance, in TrueSkill the β parameter (that gives the expected variability in a player's performance) and the dynamic factor τ (that ensures skills can evolve over time) can make a significant difference in terms of performance. Our algorithms' most important parameters are the learning rate in stochastic gradient optimization, and capacity-related quantities that help fight overfitting: sizes of the embeddings (n_e) and of the hidden layer (n_h), and weight decay coefficients (we use ℓ_2 regularization on the network matrices and on embeddings, with a separate regularization coefficient for the online update phase). We use a "brute-force" approach to model selection that consists in training a large amount of model variants with randomly chosen hyper-parameters (after running preliminary experiments to define sensible ranges). We ensure we are not overfitting on these hyper-parameters by a rigorous sequential validation setup described below. To give a rough idea, the optimal network sizes in these experiments are on the order of 10 for the embedding size, and on the order of 100 for the hidden layer size. The β parameter in TrueSkill had to be set around 10-20, and τ around 5.

Our neural network models are implemented in Python, using the Theano library (Bergstra et al., 2010) for efficient computations and gradient-based optimization. For the TrueSkill models, we use pure Python code based on a publicly available C# implementation (Moser, 2010).

8.4.3 Experimental setup

Most previous work on matchmaking and skill rating systems usually evaluate algorithms in either an "online" (Herbrich et al., 2007) or a "batch" (Stănescu, 2011) setting:

- In an online setting, the algorithm starts from scratch and is immediately evaluated in the prediction task (also updating its parameters at the same time after each match result).

- In a batch setting, parameters are first optimized on a training set, then performance is evaluated on a disjoint test dataset, keeping parameters fixed.

*Note that although we will show it helps getting better performance for the purpose of matchmaking, we also argue in section 8.5 that this approach should be avoided for public player rankings, as it would promote selfish play.

These settings do not reflect the way our algorithm is meant to be used, so we rely on a different setup that generalizes them. The algorithm is first trained on a training set (this is the "offline training" phase described in section 8.3.3), then its performance is evaluated on a disjoint test dataset while also updating parameters after each test match result ("online update" phase). Note that other algorithms can also be evaluated in this setup, for instance the "offline training" step of a purely online learning algorithm like TrueSkill simply consists in updating player skills by going through all matches in the training set*.

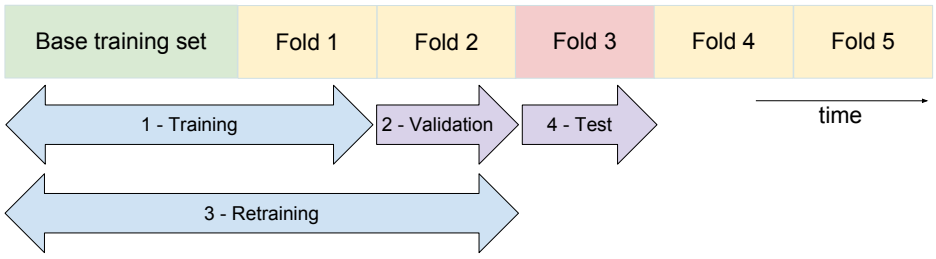
In order to obtain an unbiased estimate of the generalization ability of the algorithms being compared, we use sequential validation with model selection. The first 25% of the dataset (ordered chronologically) is isolated as a base training set which is used to "seed" all algorithms. The remainder of the data is split into five folds, and generalization error is estimated by averaging the test error over folds 2 to 5. The test error on fold $k \in \{2, 3, 4, 5\}$ is computed as follows (figure 8.6 illustrates the process for $k = 3$):

1. Training: train multiple variants of the model ("offline training") on the concatenation of the base training set and folds $1, \ldots, k-2$. Each variant corresponds to a different random choice of hyper-parameters (e.g. learning rate, number of hidden units, embedding size).

2. Validation: evaluate each variant by "online update" on fold $k-1$.

3. Re-training: re-train the best variant ("offline training") after adding fold $k-1$ to the train set.

4. Test: evaluate the re-trained model by "online update" on fold k.



◀ **Figure 8.6.** *Five-fold sequential validation with model selection (here, computation of the third fold's test error). Model variants are compared according to their validation error on the previous fold. The best one is evaluated on the test fold after re-training on all data available before this point. Double arrows indicate "offline training", while single arrows represent "online update".*

8.4.4 Game balance

We first evaluate the model presented in section 8.2.1, that predicts the probability that team A wins. In our matchmaking framework, this model

*For the sake of fairness, we added a new hyper-parameter to TrueSkill which is the number of times it iterates on the "offline training" set, in order to potentially let it refine its skill estimates. This did not appear to help much.

is used by preferring matches for which this probability is close to 50%. It is thus important to predict an accurate probability, not just to predict which team will win the match. For this reason, our main criterion for comparison is the Negative Log-Likelihood (NLL). Following Stănescu (2011), we truncate the model outputs so that they remain within the $(0.01, 0.99)$ range: this prevents a model's NLL from growing arbitrarily large when it is too confident in its prediction (which often happened for TrueSkill in our experiments). We also compute the classification error on the winner prediction task since it is easier to interpret, and provides insight on the overall balance of matches in our dataset (note in particular that if all matches were perfectly balanced, then all algorithms would have 50% error).

Table 8.1. *Winner prediction task (will team A win this match?)*

	NLL	**Class. Error (%)**
TrueSkill-Team	0.547 ± 0.019	26.6 ± 1.9
TrueSkill-Player	0.4785 ± 0.022	21.5 ± 1.8
BalanceNet	$\mathbf{0.457 \pm 0.020}$	$\mathbf{19.2 \pm 1.7}$

Table 8.1 presents the results, with 95% confidence intervals (two standard errors). The best results are shown in bold (we performed a paired t-test to verify that *BalanceNet* is significantly better than *TrueSkill-Player* in terms of both NLL and classification error – with p-value of respectively 0.004 and 0.003). It is obvious that *TrueSkill-Team* is much worse than the other two models. This is because it takes longer for skills to converge when they are based only on the match results, compared to *TrueSkill-Player* that has access to direct player rankings through in-game scores. This supports our argument that for the purpose of matchmaking, it is advised to take advantage of individual statistics on players beyond the global result of their teams.

BalanceNet outperforms *TrueSkill-Player*, but the difference is not as striking. We believe the main reasons are related to the current dataset we are experimenting with:

- As can be seen from the low error rate that can be achieved (under 20%), the matches in the dataset are not properly balanced. This is because the matchmaking algorithm in this beta-test was not trying to match players according to their skill. Thus the majority of the matches are significantly unbalanced, making the task easy to solve with simple algorithms*.
- Data comes from an early beta-test, where most players are still learning the game on their own, without caring much about teamwork. Consequently, there is not much benefit to gain from a model that can deal with team interactions.

*As a baseline, a model that simply predicts the winning team as the team with most players achieves 35% classification error.

- All the data comes from a single map and game mode, and the map is symmetric: although our model is meant to be able to take into account map and game mode specificities, it is not needed here.

- Obviously, 3937 matches is quite small: it is not possible to take full advantage of a high capacity model with this amount of data.

As a result, we expect that as we collect more data (with more variety), our neural network model's advantage over *TrueSkill-Player* will become even more significant.

8.4.5 Player fun

Ultimately, we intend to make player enjoyment the main criterion used in *Ghost Recon Online*'s matchmaking. Note however that we believe a match balance predictor like the one we discussed in the previous experiments would still remain useful, to (i) act as a safeguard against the fun predictor's mistakes, and (ii) possibly speed up computations (the sampling phase described in section 8.3.1) by pre-filtering matches in order to fully evaluate only reasonable candidates. Thus we envision a matchmaking system that would take advantage of the unique benefits brought by both approaches.

As mentioned in the introduction, we are setting up an in-game survey to gather player feedback about their gaming experience. While waiting for this data to be collected, we ran experiments by "simulating" a survey, based on our assumptions on what makes the game fun. We did not put a lot of efforts in designing the ultimate "fun formula" since our goal is to replace it eventually, so we used the following process:

- We started by defining 11 features such that we expect a player to have more fun when these features increase. Some of these features are local to the player (e.g. average life span, number of bullets fired, whether he finished the match or disconnected while in progress), some are team-based (e.g. the ratio of our teammates kill/death ratio compared to our own kill/death ratio, capped to 1 in order to mostly catch frustrating situations where we are matched with less skilled players), and finally some are global to a match (e.g. the match duration, and the total number of kills in the match).

- These features were normalized between 0 and 1 so as to obtain a uniform distribution in the (0, 1) range (this basically amounts to using the rank of the value in the sorted list of all observed values for the same feature).

- For each unique player in the dataset, we randomly picked 4 out of these 11 features as those he actually cares about. Then we uniformly sampled 4 weights (rescaled so that they sum to 1) to weigh these

features differently. This way, each player has his own individual criteria to evaluate fun in a match (although many of these criteria are correlated).

- The fun factor of each player in each match was computed as this weighted sum of features, then all these fun values were normalized into (0, 1) like we did for individual features.
- Finally, we assumed the player answered "Yes" to the question "Did you have fun in the match" when his fun factor was above 0.7, "No" when it was below 0.3, and skipped the survey otherwise.

Note that although our *FunNet* model predicts individual probabilities for each player to have fun, in the end we need a global match quality score. Based on the assumption that we want everyone to have fun in the match, we interpret our normalized "fun factor" score as the ground truth probability that a player has fun, assume independence among players, and define the match quality by

$$\Pi_{\text{Player } i \in \text{ match}} P(\text{Player } i \text{ has fun}).$$

We actually take the logarithm of this score for convenience, and average the resulting sum to make it independent of the number of players involved, yielding the final formula

$$Score(\text{match}) = \frac{1}{n} \sum_{\text{Player } i \in \text{ match}} \log P(\text{Player } i \text{ has fun}). \tag{8.5}$$

where n is the number of players in the match.

We first validate that our model is able to predict the survey answers by looking at the NLL and classification error on this target. Table 8.2 shows that our model can reach under 30% error, which tells us that it was able to capture at least some of the underlying fun patterns we made up.

Table 8.2. *Survey prediction task (will player i have fun in this match?)*

	NLL	**Class. Error (%)**
FunNet	0.571 ± 0.008	29.2 ± 0.7

Then, we turn to the main question this research is concerned about, which is: can such a model select fun matches better than balance-based models like TrueSkill and *BalanceNet*? To answer it, we use a ranking measure, which has the advantage of being independent of the scale of the models' scores, and of reflecting our real-world application where the goal is to rank candidate matches to find the best ones. Specifically, we compute the Kendall's tau rank correlation coefficient (Wikipedia, 2011), denoted by τ, between the ground truth match score (eq. 8.5) and the models' rankings on test matches. For the balance-based models (*TrueSkill-Team*, *TrueSkill-Player* and *BalanceNet*) the matches are ranked by increasing value of $|P(\text{team A wins}) - 0.5|$. The *FunNet* model uses eq. 8.5 with

its own estimated probabilities of each player having fun. A perfect ranking would achieve $\tau = 1$, while random ranking corresponds to $\tau = 0$ (and $\tau = -1$ corresponds to ranking in the exact opposite order of the ground truth).

	τ **(Kendall's tau)**
TrueSkill-Team	0.11
TrueSkill-Player	0.17
BalanceNet	0.20
FunNet	**0.23**

Table 8.3. *Match ranking task (which matches will be most fun?)*

We see from table 8.3 that all models achieve a significantly positive τ, which is not surprising since many of our features that define fun correlate with match balance. We also recover the same ordering w.r.t. performance as in the balance task (table 8.1) i.e. *TrueSkill-Team* < *TrueSkill-Player* < *BalanceNet*. However, as we expected, there is a benefit in designing a specific "fun predictor" like the *FunNet* neural network, since it is the one that achieves best performance. This is a promising result, but of course it remains to be validated on "true" player feedback, which will be the topic of our future research.

8.5 Related work

Our work puts together ideas originating from several fields of research: matchmaking, skill rating and player modeling. We describe below previous work in those areas that is most relevant in the context of the proposed methodology.

8.5.1 Matchmaking

The primary concern for matchmaking in an action game is often the network connection quality. This is especially true in an FPS where accurate aiming is key: being able to reliably estimate latency between players is thus very important, and is a topic of ongoing research (Agarwal et al., 2009). This challenge is made easier in our situation because games are run on dedicated servers, so all we need to do is ensure that we only match together players who have a good connection to the same dedicated server.

From a high-level point of view, our matchmaking architecture is in the same spirit as the one described by Tobias Fritsch (2008), but with a more complex match selection process. In that work, it is suggested to divide players among "bins" (based on their skill) in order to ensure match balance, and it is not said how to pick players from a bin to obtain the final

team composition. This kind of skill-based strategy is used by many games, that do not attempt to globally optimize team compositions. Instead, they match together players of similar skill, then distribute players in teams either randomly or so as to achieve teams of equal strength (Butcher, 2008; League of Legends, 2010).

The idea that players should be matched based on their gaming profile is not new: Riegelsberger et al. (2007) showed empirically that different types of players do not share the same preferences with respect to who they enjoy playing with. However, they did not actually propose a specific matchmaking algorithm based on these considerations. Jimenez-Rodriguez et al. (2011) describe such a matchmaking system, where they call "role" a player's individual type. In this system, examples of "good" matches are first memorized (where good matches are found for instance by asking feedback from players like we intend to, or by human experts who observe matches). These good matches are analyzed in terms of the roles played by the players involved in them, where roles are manually defined in a subjective manner and may correspond to various traits of players that are considered important for matchmaking purpose (e.g. "sniper", "power gamer", "socializer"). A specific algorithm to infer player role from tracked player behavior is not detailed, but several player modeling techniques have been developed in the past years and may be used for this purpose (van den Herik et al., 2005; Tychsen et al., 2008; Thawonmas et al., 2008b). When a match needs to be created from a pool of players waiting in the matchmaking queue, candidate matches are then evaluated by being compared to the set of good matches (in terms of similarity in their role compositions). This approach is thus similar to ours, but replacing our neural network evaluation system with a memory-based algorithm and using only roles as input. Although we believe this is a sensible idea worth experimenting with, it has not been actually implemented yet. One difficulty is that it is not obvious which roles are to be defined (one can think of our algorithm as a way to learn roles automatically within the player embeddings). Also, their proposed algorithm only keeps "good" examples, while also taking into account examples of "bad" situations is probably important as well. Finally, they mention the problem of the combinatorial cost of trying all player combinations to find the best match, but do not propose a solution to this issue: we suggest here to solve it by random sampling.

8.5.2 Skill rating

The problem of assigning skills to players or teams has a long history in both games and sports, mostly for the tasks of ranking, matchmaking and outcome prediction. Although all these tasks may be tackled independently, a skill rating system is very appealing as it can provide a statistically motivated answer to all of them. The ranking task, however, imposes some specific constraints that may hurt performance for matchmaking and out-

come prediction. Besides the fact that a uni-dimensional skill is needed to easily make comparisons, the competitive nature of rankings also makes them a favorite target of players trying to "exploit" the system (Butcher, 2008). This is one important reason why most skill rating systems only consider match results to compute the skill: accounting for extra information like the attributes we feed to our neural network might be abused by players. This could be very detrimental to team-based games, where players would try to maximize statistics that boost their skill (e.g. their own number of kills or captures in an FPS) instead of doing what is best for their team to win.

A skill rating algorithm meant to be used for matchmaking in a multiplayer game like *Ghost Recon Online* needs to be able to assign individual ratings to players, then to derive ratings for arbitrary teams from these player ratings. This rules out most algorithms used in sports, where typically either only global team ratings are considered (Park et al., 2005), or, if individual ratings are sought, players are assumed to play in the same team for a long enough period of time to estimate meaningful correlations (Piette et al., 2011).

The large majority of skill rating systems developed for games take their root from the Bradley-Terry model (Bradley et al., 1952), that in its basic formulation models the probability that team A wins over team B by

$$\alpha = P(\text{team A wins}) = \frac{s_A}{s_A + s_B}$$

with s_j the skill of team j. If we write $s_j = e^{t_j}$ this becomes

$$\alpha = \frac{e^{t_A}}{e^{t_A} + e^{t_B}} = \frac{1}{1 + e^{t_B - t_A}} = \text{sigmoid}(t_A - t_B). \tag{8.6}$$

Note that if we assume

$$t_j = \sum_{i \in \text{team } j} p_i \tag{8.7}$$

with p_i the individual skill of player i, then this is a variant of our neural network model described in section 8.2.1. This can be seen when:

- A player's embedding is made of a single scalar (his skill) and there are no player attributes, so that eq. 8.1 becomes $p_i = e_i$, and thus team features (eq. 8.2) are scalars computed as in eq. 8.7.
- The hidden layer $\mathbf{h}$ (eq. 8.3) is simplified to be the concatenation of the team features, i.e. $\mathbf{h} = (t_A, t_B)^T$.
- The parameters of the output probability α (eq. 8.4) are $\mathbf{u} = (1, -1)^T$ and $c = 0$, making it equivalent to eq. 8.6.

The Bradley-Terry model has already been presented in such a neural network form (Menke et al., 2008), but it was generalized in a different way than

in our model. The application the authors were interested in was in a game where teams were expected to have a significant imbalance (in terms of the number of players facing each other), which led them to model differently the combined player strengths to better account for such large differences. In particular, they incorporate this difference into a so-called "home field advantage" that can also model imbalances resulting from asymmetric maps, and can have a stronger influence on the predicted result when there is a high uncertainty on the player skills (i.e. when many players are new to the game). They also take time into account by weighting a player's contribution with the time he spent in the match, and using the elapsed time as input so that the winning probability evolves as time elapses. Such an extension would be interesting to incorporate in our model to better evaluate "hot-join" situations. Compared to their formulation, the novelty of our approach lies in using a multi-dimensional embedding rather than a single skill value, adding additional player attributes as input, and having more parameters to the neural network transformations in order to potentially learn more complex functions.

As discussed in section 8.3.3, the ability to update player ratings after each match efficiently is important for online rating systems, that need to update ratings in real time. The Elo rating system (Elo, 1978), adopted by the World Chess Federation, is very close to the Bradley-Terry model described above and is based on an efficient online update algorithm. The Elo rating was later extended, in particular to model uncertainty (Glickman, 1999), eventually leading to the fully Bayesian TrueSkill system that is also able to infer individual skills from team results (Herbrich et al., 2007). Various improvements and variants of TrueSkill have been proposed since then (see e.g. Weng et al., 2011; Nikolenko et al., 2011; Huang et al., 2011). Such methods differ significantly from ours: they are probabilistic algorithms that model a player's skill as a scalar random variable, and perform inference based only on match results (in particular they ignore player attributes). One research direction that bears resemblance to our work is the idea of computing skill ratings in a "batch" setting, i.e. instead of only updating current ratings incrementally after each match, both past and future ratings are optimized to globally fit all match results available (Dangauthier et al., 2007; Coulom, 2008)*. This is also what our offline training phase (described in section 8.3.3) is meant to achieve: it can "revisit the past", while our online update phase is currently a more myopic (but faster) incremental procedure.

Although using a single scalar to represent player skill is convenient, it has been recently noted that increased performance on the outcome prediction task can be obtained when using additional factors. A first idea, explored by Zhang et al. (2010) and Usami (2010), consists in adding the concept of "contexts" associated to vectors $\boldsymbol{\theta}_k$, such that the skill of player i in context k is given by the dot product $\mathbf{p}_i \cdot \boldsymbol{\theta}_k$. In our FPS application, a

*Note that the batch approach from Coulom (2008) can actually be made fast enough for real-time use (with some approximations to speed up computations).

context would be for instance a specific map and game mode: we proposed a similar idea in section 8.2.1 by learning context-dependent weight matrices $\mathbf{V}_A$ and $\mathbf{V}_B$ (used in eq. 8.3). Note that in our model we use a context matrix rather than a vector because we want to extract multiple features rather than a single skill value. Another way to use a multi-dimensional skill vector in a Bayesian setting was presented by Stănescu (2011), whose idea consists in modeling the fact that a player may have strengths and weaknesses in different areas. Our neural network approach is also be able to model such strengths and weaknesses in the embedding vector $\mathbf{p}_i$, which the hidden layer transformation (eq. 8.3) can combine optimally for outcome prediction. This is all done implicitly here, while in a Bayesian setting the relations between elements of $\mathbf{p}_i$ are explicitly defined by the graphical model architecture.

To conclude the comparison with Bayesian skill rating models, we should emphasize that such models naturally handle uncertainty, since they are fully probabilistic. For instance, evaluating balance with TrueSkill is not usually done by simply looking at $|P(\text{team A wins}) - 0.5|$. Instead, the balance is computed from the asymptotic probability that the two teams perform equally well (i.e. a draw), which depends on the uncertainty on players' skill and performance (Herbrich et al., 2007). On another hand, our current model ignores uncertainty: player embeddings are fixed and the network transformations are deterministic. However, we expect the addition of player attributes (that contain for instance the number of matches already played) to help by indirectly taking into account uncertainty about new players' embeddings.

8.5.3 Player modeling

The basic idea of *player modeling* (Charles et al., 2005) is to extract information about players, to eventually provide them with an improved gaming experience (either directly – e.g. tuning the game to better suit the player's playstyle – or indirectly – e.g. collecting data to help later improve the game or its sequel). Note that here we only consider models based on players' actions within the game: more intrusive systems based for instance on heart rate monitoring may also bring useful insight into the way players experience video games (Drachen et al., 2010), but are out of the scope of our present research.

Our matchmaking application can be seen as a kind of "game adaptation" mechanism in the context of matchmaking, where the game parameters being tuned are those of the matchmaking decision function. A special case of game adaptation consists in dynamically adjusting the game difficulty to better suit the player's individual skill level (Jimenez, 2009). In single player games, dynamic difficulty adjustment is usually based on the analysis of relevant statistics (e.g. number of successes / failures, rate of damage) to adjust game settings during gameplay (Hunicke et al., 2004; Spronck et al., 2004; Harward

et al., 2007; Missura et al., 2011). Other game adaptation techniques to maximize player enjoyment have been proposed before in other contexts like game content generation and adaptation: Pedersen et al. (2010) provide a good overview of previous work in this area. Although many of these methods share goals similar to our work (they aim at making the game more balanced and more fun), they cannot be readily applied to matchmaking. The main reason is they are designed for single-player games and thus are not meant to simultaneously optimize the game experience of many players at the same time (especially because of player interactions).

Another link with player modeling consists in the addition of player attributes. In the present research we use simple statistics extracted from the game logs, but in the future we intend to add more high-level information about the players' profiles. Such information could for instance be whether the player is more interested in the competitive aspects of the game, in its social interactions, in having casual fun shooting random people, etc. If such "classes" of players can be defined beforehand from a priori knowledge, a survey could be sent to players asking them to identify which class they belong to, or human experts could watch some players and manually label their playstyle. Once a number of such "prototypical" players are available, supervised learning methods can be applied to profile the whole player-base (Tychsen et al., 2008; Thawonmas et al., 2007, 2008a). Alternatively, unsupervised clustering methods can also be used to discover typical classes of player behavior without much prior knowledge (Thawonmas et al., 2008b; Ramirez-Cano et al., 2010). We expect that adding such high-level profiling of players into their attributes vector will help our predictive models achieve better accuracy.

8.6 Conclusion and future directions

Our main contributions are as follows:

- We demonstrated that in order to evaluate match balance in a multiplayer game, using a skill value is not enough. There is much to be gained from a richer player profile, in particular by adding player statistics collected within the game.

- We argued that fun is more important than balance, and showed it is possible to use fun as the main criterion in a matchmaking system (to the best of our knowledge, this is the first implementation of this idea).

- We proposed an implementation based on neural networks, which makes it easy to include additional parameters and to design architecture variants able to better suit a game's specific needs.

- We showed how to integrate these neural networks within an online game's matchmaking system, providing solutions to the problems of (i) finding the best team combinations from a pool of players waiting for a match, and (ii) continuously updating the model in real time as new data is being collected.

Our experimental results, although promising, remain preliminary: as more data is being collected during *Ghost Recon Online*'s beta tests, we will be able to better evaluate the proposed models, and experiment with more variants. The main directions we plan to investigate are the following:

- Once enough data from the in-game player survey has been collected, it will be interesting to compare our handcrafted formula of fun with actual player feedback. One question is also how often the survey should be presented to players after launch: we may not even need it if it proves possible to learn a reliable enough predictive model of fun.

- The current set of attributes we are using is very limited. We will augment it with more statistics, as well as with more high level information derived from player modeling.

- With more data, we may be able to take advantage of more elaborate neural network architectures so as to better learn complex statistical dependencies.

Acknowledgements

We would like to thank the *Ghost Recon Online* development team for their support throughout this project. We are also thankful to Frédéric Bernard, Myriam Côté, Aaron Courville and Steven Pigeon for the many fruitful discussions on various aspects of this research. In addition, this work was made possible thanks to the research funding and computing support from the following agencies: NSERC, FQRNT, Calcul Québec and CIFAR.

8.7 Commentaires

Comme indiqué au début de ce chapitre, l'article lui-même n'aborde pas directement la question de l'efficacité, ce qui ne veut pas dire qu'elle ne soit pas importante ici, bien au contraire.

8.7.1 Efficacité statistique

Dans la mesure où nous devons être capables de démontrer que nos algorithmes apportent une plus-value pendant le développement du jeu (sinon l'équipe du jeu refuserait de s'impliquer dans un tel projet), il faut pouvoir travailler avec la faible quantité de données collectées au cours de tests à petite échelle. Il est donc important que les algorithmes utilisés soient capables d'obtenir une bonne capacité de généralisation avec un nombre limité d'exemples d'entraînement.

Un avantage de l'architecture des réseaux de neurones que nous utilisons est justement qu'elle peut s'adapter à la quantité de données disponibles, en contrôlant plusieurs paramètres de régularisation qui permettent de combattre le sur-apprentissage. Notons en particulier :

- La taille des vecteurs ("embeddings") associés à chaque joueur.
- La taille de la couche cachée (ainsi que le nombre de couches cachées, car même si pour l'instant nous n'en utilisons qu'une, il pourrait s'avérer utile d'utiliser des réseaux plus profonds lorsque plus de données seront collectées).
- Les coefficients de pénalités ℓ_1 et ℓ_2 sur d'une part les matrices de poids du réseau, et d'autre part les "embeddings" eux-mêmes (l'article ne mentionne que la régularisation ℓ_2, mais nous avons depuis rajouté une régularisation ℓ_1 qui peut parfois donner de meilleurs résultats).
- Le nombre d'attributs utilisés.

Les réseaux que nous utilisons actuellement sont donc très régularisés, car nous avons encore peu de données. Mais nous espérons qu'après le lancement du jeu, nous pourrons tirer parti de bien plus grands ensembles d'entraînement avec des réseaux plus complexes – tout en gardant la même architecture de base.

8.7.2 Efficacité computationnelle

Un tel jeu devrait attirer en moyenne plusieurs dizaines de milliers de joueurs connectés simultanément. Avec un tel nombre de joueurs en ligne, il est crucial de pouvoir les apparier rapidement et l'efficacité computationnelle de nos algorithmes devient primordiale. Dans les précédents chapitres de cette thèse, le problème de l'efficacité computationnelle a été abordé du point de vue algorithmique. Ici, nous nous concentrons sur des questions d'ingénierie, qui sont souvent passées sous silence dans la recherche en apprentissage machine, mais qui ne peuvent être ignorées dans des applications

pratiques. Voici donc quelques leçons qui se sont avérées utiles pour ce projet.

32 bits, ce n'est pas beaucoup

Une contrainte à laquelle nous avons dû nous soumettre est que le processus principal faisant tourner nos algorithmes est un processus 32 bits, ce qui limite en particulier la quantité de mémoire utilisée (à environ 2 Go). Cela nous empêche notamment de garder en mémoire les "embeddings" de tous les joueurs, ce qui rend leur indexage plus complexe et nous oblige à une gestion intelligente des "embeddings" stockés en mémoire. D'autre part, chaque match est divisé en rounds (typiquement deux ou trois), et les informations associées à chaque round sont reçues à la fin du round, tandis que pour la prédiction de la satisfaction, l'optimisation du réseau ne peut se faire qu'à la fin du match (lorsque le joueur indique s'il s'est amusé ou non). Cela implique qu'il faut stocker les informations de chaque round tant que le match n'est pas fini, mais il n'est pas possible de les garder en mémoire : pour régler ce problème, nous avons dû utiliser un système de serveur ("memcache") qui ralentit et complexifie le stockage de ces données.

Le langage de programmation doit permettre une exécution rapide

Le langage de programmation utilisé sur les serveurs du jeu est Python. C'est un langage de script très pratique, mais dont un inconvénient majeur est la lenteur d'exécution. Heureusement, nous avons pu utiliser la librairie Theano (Bergstra et al., 2010), qui permet de coder en Python des expressions qui sont ensuite traduites en langage C et compilées dynamiquement sur le serveur. Sans Theano, nos réseaux de neurones seraient bien trop lents pour pouvoir gérer des milliers de joueurs simultanément.

Les mini-batchs, c'est fantastique

Si théoriquement parlant, le temps de calcul de k multiplications matrice-vecteur est du même ordre de grandeur qu'une multiplication matrice-matrice où la seconde matrice contiendrait les k vecteurs, en pratique la seconde opération s'avère bien plus rapide. Cela est dû en particulier aux effets de cache au niveau du processeur, ainsi qu'au fait que les processeurs modernes offrent des instructions optimisées permettant de paralléliser certaines opérations. La boucle externe (itérant sur les k vecteurs) dans le premier cas peut également introduire des délais non négligeables (surtout si elle est en Python). Dans notre implémentation des algorithmes décrits dans ce chapitre, nous essayons donc autant que possible d'utiliser ce que l'on appelle des "mini-batchs", c.à.d. des exemples groupés ensemble dans une seule matrice, plutôt que de les séparer en vecteurs indépendants. C'est le cas par exemple lorsque l'on veut évaluer la qualité de plusieurs matchs candidats générés aléatoirement (pour choisir le prochain match) – on évalue tous les

candidats simultanément au lieu d'utiliser une boucle. Cela rend l'implémentation des réseaux de neurones présentés ici un peu plus complexe, mais l'effort en vaut la peine.

Les calculs doivent être distribués

Une architecture de calcul centralisée est mal adaptée à l'environnement d'un jeu en ligne, dont le nombre de joueurs peut varier de manière significative au cours du temps, selon le succès du jeu. Il est alors préférable d'utiliser une architecture distribuée, où les calculs sont partagés entre différentes unités (machines indépendantes ou processeurs au sein de la même machine), et le nombre d'unités peut facilement être contrôlé en fonction de la popularité du jeu. Notre approche se prête bien à une telle architecture distribuée puisque les deux opérations les plus coûteuses à effectuer en temps réel – l'évaluation des matchs générés aléatoirement et la mise à jour des "embeddings" après chaque match – peuvent être effectuées de manière indépendante par différentes unités. Nous travaillons actuellement à l'implémentation d'une telle architecture, car notre première version était centralisée (par souci de simplicité).

Les threads permettent d'accélérer les opérations d'entrée-sortie

En Python, l'utilisation de threads est souvent déconseillée car l'implémentation du langage empêche généralement l'exécution véritablement parallèle du code. Une exception qu'il est malgré tout très utile de garder à l'esprit est que si la vitesse d'exécution du code est limitée par des opérations d'entrée-sortie, comme la lecture ou l'écriture dans une base de données (plutôt que par des calculs intensifs sur le processeur), alors déplacer ces opérations dans des threads séparés permet au programme principal de ne pas être bloqué par ces opérations. Nous avons ainsi observé un gain significatif en performances en déplaçant systématiquement les accès à la base de données et au serveur de mémoire ("memcache") dans leur propre thread.

Bibliographie

Agarwal, S. et J. R. Lorch. 2009, "Matchmaking for online games and other latency-sensitive P2P systems", dans *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, ACM, New York, NY, USA, p. 315–326.

Bateman, C. et R. Boon. 2005, *21st Century Game Design (Game Development Series)*, Charles River Media, Inc., Rockland, MA, USA, ISBN 1584504293.

Bengio, Y., P. Lamblin, D. Popovici et H. Larochelle. 2007, "Greedy layer-wise training of deep networks", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 153–160.

Bergstra, J., O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley et Y. Bengio. 2010, "Theano : a CPU and GPU math expression compiler", dans *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

Bishop, C. M. 2006, *Pattern Recognition and Machine Learning*, Springer.

Bradley, R. A. et M. E. Terry. 1952, "The rank analysis of incomplete block designs — I. The Method of Paired Comparisons", *Biometrika*, vol. 39, p. 324–345.

Butcher, C. 2008, "E pluribus unum : Matchmaking in Halo 3", dans *Game Developers Conference (GDC 2008)*.

Charles, D., M. McNeill, M. McAlister, M. Black, A. Moore, K. Stringer, J. Kücklich et A. Kerr. 2005, *Player-Centred Game Design : Player Modelling and Adaptive Digital Games*.

Coulom, R. 2008, "Whole-history rating : A Bayesian rating system for players of time-varying strength", dans *Proceedings of the 6th International Conference on Computers and Games*, CG '08, Springer-Verlag, Berlin, Heidelberg, p. 113–124.

Dangauthier, P., R. Herbrich, T. Minka et T. Graepel. 2007, "TrueSkill through time : Revisiting the history of chess", dans *Advances in Neural Information Processing Systems*, édité par M. Press, Vancouver, Canada.

Drachen, A., L. E. Nacke, G. Yannakakis et A. L. Pedersen. 2010, "Correlation between heart rate, electrodermal activity and player experience in first-person shooter games", dans *Proceedings of the 5th ACM SIGGRAPH Symposium on Video Games*, ACM, New York, NY, USA, p. 49–54.

Elo, A. E. 1978, *The rating of chessplayers, past and present*, Batsford.

Glickman, M. E. 1999, "Parameter estimation in large dynamic paired comparison experiments", *Applied Statistics*, vol. 48, n° 3, p. 377–394.

Harward, K. et A. Cole. 2007, "Challenging everyone : Dynamic difficulty deconstructed", dans *Game Developers Conference (GDC 2007)*.

Herbrich, R., T. Minka et T. Graepel. 2007, "TrueSkill™ : A Bayesian skill rating system", dans *Advances in Neural Information Processing Systems 19 (NIPS'06)*, édité par B. Schölkopf, J. Platt et T. Hoffman, MIT Press, p. 569–576.

van den Herik, H. J., H. H. L. M. Donkers et P. H. M. Spronck. 2005, "Opponent modelling and commercial games", dans *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games CIG'05*, édité par G. Kendall et S. Lucas, p. 15–25.

Hinton, G. E. et R. Salakhutdinov. 2006, "Reducing the dimensionality of data with neural networks", *Science*, vol. 313, n° 5786, p. 504–507.

Huang, J. C. et B. J. Frey. 2011, "Cumulative distribution networks and the derivative-sum-product algorithm : Models and inference for cumulative distribution functions on graphs", *Journal of Machine Learning Research*, vol. 12, p. 301–348.

Hunicke, R. et V. Chapman. 2004, "AI for dynamic difficulty adjustment in games", dans *Proceedings of AIIDE 2004*.

Jimenez, E. 2009, "The Pure advantage : Advanced racing game AI", `http://www.gamasutra.com`.

Jimenez-Rodriguez, J., G. Jimenez-Diaz et B. Diaz-Agudo. 2011, "Matchmaking and case-based recommendations", dans *Workshop on Case-Based Reasoning for Computer Games, 19th International Conference on Case Based Reasoning*.

League of Legends. 2010, "League of Legends matchmaking", `http://na.leagueoflegends.com/learn/gameplay/matchmaking`.

Malone, T. 1981, "What makes computer games fun ?", *SIGSOC Bull.*, vol. 13, p. 143–, ISSN 0163-5794.

Menke, J. E. et T. R. Martinez. 2008, "A Bradley-Terry artificial neural network model for individual ratings in group competitions.", *Neural Computing and Applications*, vol. 17, p. 175–186.

Missura, O. et T. Gärtner. 2009, "Player modeling for intelligent difficulty adjustment", dans *Discovery Science*, *Lecture Notes in Computer Science*, vol. 5808, édité par J. Gama, V. Costa, A. Jorge et P. Brazdil, Springer Berlin / Heidelberg, ISBN 978-3-642-04746-6, p. 197–211.

Missura, O. et T. Gärtner. 2011, "Predicting dynamic difficulty", dans *Ninth Workshop on Mining and Learning with Graphs*.

Moser, J. 2010, "Computing your skill", `http://www.moserware.com/2010/03/computing-your-skill.html`.

Nikolenko, S. et A. Sirotkin. 2011, "A new Bayesian rating system for team competitions", dans *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, édité par L. Getoor et T. Scheffer, ACM, New York, NY, USA, p. 601–608.

Park, J. et M. E. J. Newman. 2005, "A network-based ranking system for US college football", *Journal of Statistical Mechanics : Theory and Experiment*, vol. 2005, n° 10.

Pedersen, C., J. Togelius et G. N. Yannakakis. 2010, "Modeling player experience for content creation", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, n° 1, p. 54–67.

Piette, J., S. Anand et L. Pham. 2011, "Evaluating basketball player performance via statistical network modeling", dans *MIT Sloan Sports Analytics Conference*.

Ramirez-Cano, D., S. Colton et R. Baumgarten. 2010, "Player classification using a meta-clustering approach", dans *Proceedings of the International Conference on Computer Games, Multimedia and Allied Technology*.

Riegelsberger, J., S. Counts, S. Farnham et B. C. Philips. 2007, "Personality matters : Incorporating detailed user attributes and preferences into the matchmaking process", dans *HICSS*, IEEE Computer Society, p. 87.

Rifai, S., Y. Dauphin, P. Vincent, Y. Bengio et X. Muller. 2011, "The manifold tangent classifier", dans *NIPS'2011*.

Shewchuk, J. R. 1994, "An introduction to the conjugate gradient method without the agonizing pain", rapport technique, Pittsburgh, PA, USA.

Shi, L. et W. Huang. 2004, "Apply social network analysis and data mining to dynamic task synthesis for persistent MMORPG virtual world", dans *Proceedings of the Third International Conference on Entertainment Computing, Eindhoven, The Netherlands, 2004*, *Lecture Notes in Computer Science*, vol. 3166, édité par M. Rauterberg, Springer, p. 204–215.

Spronck, P., S. I. Kuyper et E. Postma. 2004, "Difficulty scaling of game AI", dans *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)*, p. 33–37.

Stănescu, M. 2011, *Rating systems with multiple factors*, mémoire de maîtrise.

Thawonmas, R. et J.-Y. Ho. 2007, "Classification of online game players using action transition probabilities and Kullback Leibler entropy", *Journal of Advanced Computational Intelligence and Intelligent Informatics, Special issue on Advances in Intelligent Data Processing*, vol. 11, n° 3, p. 319–326.

Thawonmas, R. et K. Iizuka. 2008a, "Haar wavelets for online-game player classification with dynamic time warping", *Journal of Advanced Computational Intelligence and Intelligent Informatics, Special issue on Intelligence Techniques in Computer Games and Simulations*, vol. 12, n° 2, p. 150–155.

Thawonmas, R. et K. Iizuka. 2008b, "Visualization of online-game players based on their action behaviors.", *Int. J. Computer Games Technology*.

Tobias Fritsch, J. S., Benjamin Voigt. 2008, "The next generation of competitive online game organization", dans *Netgames 2008*.

Togelius, J., R. De Nardi et S. M. Lucas. 2007, "Towards automatic personalised content creation for racing games", dans *IEEE Symposium on Computational Intelligence and Games*.

Tychsen, A. et A. Canossa. 2008, "Defining personas in games using metrics", dans *Proceedings of the 2008 Conference on Future Play : Research, Play, Share*, Future Play '08, New York, NY, USA, p. 73–80.

Usami, S. 2010, "Individual differences multidimensional Bradley-Terry model using reversible jump Markov chain monte carlo algorithm", *Behaviormetrika*, vol. 37, n° 2, p. 135–155.

Weng, R. C. et C.-J. Lin. 2011, "A Bayesian approximation method for online ranking", *Journal of Machine Learning Research*, vol. 12, p. 267–300.

Wikipedia. 2011, "Kendall tau rank correlation coefficient", `http://en.wikipedia.org/wiki/Kendall_tau_rank_correlation_coefficient`.

Yannakakis, G. et J. Hallam. 2007, "Towards optimizing entertainment in computer games", *Applied Artificial Intelligence*, vol. 21, n° 10, p. 933–971.

Zhang, L., J. Wu, Z.-C. Wang et C.-J. Wang. 2010, "A factor-based model for context-sensitive skill rating systems", dans *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence – Volume 02*, ICTAI '10, IEEE Computer Society, Washington, DC, USA, p. 249–255.

9 Conclusion

Le thème de l'efficacité a été abordé sous différents angles dans les chapitres précédents. Il est possible de considérer les questions de l'efficacité computationnelle et de l'efficacité statistique comme deux problèmes distincts :

- D'un côté, l'efficacité computationnelle est importante à cause des limites de nos ordinateurs en termes de mémoire disponible et de puissance des processeurs. Bien qu'un ordinateur soit généralement considéré comme bien meilleur qu'un humain pour la puissance brute de calcul, il ne faut pas oublier que notre cerveau contient de l'ordre de 10^{11} neurones (Williams et al., 1988), tandis que les réseaux de neurones utilisés en apprentissage machine ont souvent moins de 10^4 neurones – en particulier afin de garder le temps de calcul raisonnable. Pour combler ce retard, nous pouvons bien sûr compter sur les progrès de la technologie, mais des améliorations algorithmiques (chapitres 5, 6, 7) et une utilisation judicieuse des ressources matérielles et logicielles disponibles (chapitre 8) peuvent offrir un gain immédiat à coût minimal.
- D'un autre côté, la question de l'efficacité statistique d'un algorithme est incontournable si l'on est vraiment en quête de l'intelligence artificielle, c.à.d. d'être capable de reproduire le comportement humain à l'aide d'ordinateurs. Un des arguments importants de cette thèse (chapitres 3 et 5) est qu'en particulier les algorithmes incapables de généraliser de manière non locale sont inappropriés, étant victimes de la malédiction de la dimensionalité. Et ce, même si ces algorithmes peuvent donner de très bons résultats sur certaines applications pratiques, où leurs limitations ne sont pas un handicap majeur.

Malgré tout, comme indiqué en introduction, le problème de l'efficacité statistique reste intimement lié à celui de l'efficacité computationnelle, puisqu'une puissance de calcul limitée implique l'impossibilité de tirer parti d'un nombre arbitrairement grand d'exemples. C'est pourquoi, outre la propriété de généralisation non locale, une autre caractéristique qui semble importante pour un algorithme d'apprentissage "humainement plausible" est une représentation efficace des fonctions apprises, capable de recombiner des concepts déjà assimilés pour en générer de nouveaux sans repartir de zéro (chapitre 4). Il faut donc *partager* de l'information non seulement entre les exemples, mais également entre les tâches. On parle alors d'apprentissage par *transfert* ("transfer learning" en anglais) : une compétition récente a d'ailleurs per-

mis aux architectures profondes de se distinguer dans ce domaine (Bengio, 2011).

Au final, ces deux concepts d'efficacité reviennent à la même problématique, celle de l'apprentissage à partir de ressources limitées. Chez l'être humain, ces ressources sont d'une part le cerveau (ressources computationnelles), et d'autre part les observations disponibles sur le monde qui nous entoure (ressources statistiques). Même s'il est plausible que l'évolution génétique améliore avec le temps nos ressources computationnelles et les mécanismes de base de l'apprentissage*, les progrès dans ce domaine sont évidemment très lents. Alors si l'intelligence humaine augmente de manière significative sur une courte période de temps, comme l'*effet de Flynn* (Flynn, 2009) le suggère†, la conclusion logique est que la qualité et / ou la quantité de ressources statistiques tend à augmenter. Il semble peu probable que la quantité soit le facteur principal, étant essentiellement liée à notre espérance de vie (qui n'augmente pas si vite). Cela nous amène à un troisième type d'efficacité dont il n'a pas été question dans cette thèse : une mesure d'efficacité en terme de *qualité* des exemples d'apprentissage, que ce soit en tant qu'exemples pris individuellement ou en considérant une séquence d'observations dans son ensemble. Ce thème est notamment relié à la stratégie de *curriculum* consistant à choisir l'ordre des exemples pour faciliter l'apprentissage (Bengio et al., 2009; Kumar et al., 2010; Khan et al., 2011), ainsi qu'aux théories récentes de Bengio (2012) sur la manière dont l'intelligence individuelle peut bénéficier de l'optimisation collective s'effectuant à l'échelle de sociétés humaines.

Bibliographie

Bengio, Y. 2011, "Deep learning of representations for unsupervised and transfer learning", dans *JMLR W& CP : Proc. Unsupervised and Transfer Learning challenge and workshop.*

Bengio, Y. 2012, "Evolving culture vs local minima", rapport technique, Université de Montréal.

Bengio, Y., J. Louradour, R. Collobert et J. Weston. 2009, "Curriculum learning", dans *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, édité par L. Bottou et M. Littman, ACM.

Flynn, J. R. 2009, *What is Intelligence : Beyond the Flynn Effect*, Cambridge University Press.

*Je parle ici de l'apprentissage au sens de l'adaptation des connexions synaptiques dans le cerveau.

†Il faut garder à l'esprit qu'il s'agit d'une observation controversée.

Khan, F., X. Zhu et B. Mutlu. 2011, "How do humans teach : On curriculum learning and teaching dimension", dans *Advances in Neural Information Processing Systems 24 (NIPS'11)*, p. 1449–1457.

Kumar, M. P., B. Packer et D. Koller. 2010, "Self-paced learning for latent variable models", dans *Advances in Neural Information Processing Systems 23*, édité par J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel et A. Culotta, p. 1189–1197.

Williams, R. W. et K. Herrup. 1988, "The control of neuron number", *Annual Review of Neuroscience*, vol. 11, p. 423–453.