Université de Montréal

Avancées théoriques sur la représentation et l'optimisation des réseaux de neurones

par
Nicolas Le Roux

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, 2008

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Avancées théoriques sur la représentation et l'optimisation des réseaux de neurones

présentée par:

Nicolas Le Roux

a été évaluée par un jury composé des personnes suivantes:

Pierre L'Écuyer,	président-rapporteur
Yoshua Bengio,	directeur de recherche
Pascal Vincent,	membre du jury
Yann LeCun,	examinateur externe
Alejandro Murua,	représentant du doyen de la FES

Thèse acceptée le: . . . . . .

RÉSUMÉ

Les réseaux de neurones artificiels ont été abondamment utilisés dans la communauté de l'apprentissage machine depuis les années 80. Bien qu'ils aient été étudiés pour la première fois il y a cinquante ans par Rosenblatt [68], ils ne furent réellement populaires qu'après l'apparition de la rétropropagation du gradient, en 1986 [71].

En 1989, il a été prouvé [44] qu'une classe spécifique de réseaux de neurones (les réseaux de neurones à une couche cachée) était suffisamment puissante pour pouvoir approximer presque n'importe quelle fonction avec une précision arbitraire : le théorème d'approximation universelle. Toutefois, bien que ce théorème eût pour conséquence un intérêt accru pour les réseaux de neurones, il semblerait qu'aucun effort n'ait été fait pour profiter de cette propriété.

En outre, l'optimisation des réseaux de neurones à une couche cachée n'est pas convexe. Cela a détourné une grande partie de la communauté vers d'autres algorithmes, comme par exemple les machines à noyau (machines à vecteurs de support et régression à noyau, entre autres).

La première partie de cette thèse présentera les concepts d'apprentissage machine généraux nécessaires à la compréhension des algorithmes utilisés. La deuxième partie se focalisera plus spécifiquement sur les méthodes à noyau et les réseaux de neurones.

La troisième partie de ce travail visera ensuite à étudier les limitations des machines à noyaux et à comprendre les raisons pour lesquelles elles sont inadaptées à certains problèmes que nous avons à traiter.

La quatrième partie présente une technique permettant d'optimiser les réseaux de neurones à une couche cachée de manière convexe. Bien que cette technique s'avère difficilement exploitable pour des problèmes de grande taille, une version approchée permet d'obtenir une bonne solution dans un temps raisonnable.

La cinquième partie se concentre sur les réseaux de neurones à une couche cachée infinie. Cela leur permet théoriquement d'exploiter la propriété d'approximation universelle et ainsi d'approcher facilement une plus grande classe de fonctions.

Toutefois, si ces deux variations sur les réseaux de neurones à une couche cachée

leur confèrent des propriétés intéressantes, ces derniers ne peuvent extraire plus que des concepts de bas niveau. Les méthodes à noyau souffrant des mêmes limites, aucun de ces deux types d'algorithmes ne peut appréhender des problèmes faisant appel à l'apprentissage de concepts de haut niveau.

Récemment sont apparus les Deep Belief Networks [39] qui sont des réseaux de neurones à plusieurs couches cachées entraînés de manière efficace. Cette profondeur leur permet d'extraire des concepts de haut niveau et donc de réaliser des tâches hors de portée des algorithmes conventionnels. La sixième partie étudie des propriétés de ces réseaux profonds.

Les problèmes que l'on rencontre actuellement nécessitent non seulement des algorithmes capables d'extraire des concepts de haut niveau, mais également des méthodes d'optimisation capables de traiter l'immense quantité de données parfois disponibles, si possible en temps réel. La septième partie est donc la présentation d'une nouvelle technique permettant une optimisation plus rapide.

Mots-clefs : méthodes à noyau, réseaux de neurones, convexité, descente de gradient, réseaux profonds

ABSTRACT

Artificial Neural Networks have been widely used in the machine learning community since the 1980's. First studied more than fifty years ago [68], they became popular with the apparition of gradient backpropagation in 1986 [71].

In 1989, Hornik et al. [44] proved that a specific class of neural networks (neural networks with one hidden layer) was powerful enough to approximate almost any function with arbitrary precision: the universal approximation theorem. However, even though this theorem resulted in increased interest for neural networks, few efforts have been made to take advantage of this property.

Furthermore, the optimization of neural networks with one hidden layer is not convex. This disenchanted a large part of the community which embraced other algorithms, such as kernel machines (support vector machines and kernel regression, amongst others).

The first part of this thesis will introduce the basics of machine learning needed for the understanding of the algorithms which will be used. The second part will focus on kernel machines and neural networks.

The third part will study the limits of kernel machines and try to understand the reasons of their inability to handle some of the more challenging problems we are interested in.

The fourth part introduces a technique allowing a convex optimization of the neural networks with one hidden layer. Even though this technique is impractical for medium and large scale problems, its approximation leads to good solutions achieved in reasonable times.

The fifth part focuses on neural networks with an infinite hidden layer. This allows them to exploit the universal approximation property and therefore to easily model a larger class of functions than standard neural networks.

However, though these two variations on neural networks with one hidden layer show interesting properties, they cannot extract high level concepts any better. Kernel machines suffering from the same curse, none of these families of algorithms can handle

problems requiring the learning of high level concepts.

Introduced in 2006, Deep Belief Networks [39] are neural networks with many hidden layers trained in a greedy way. Their depth allows them to extract high level concepts and thus to perform tasks out of the reach of standard algorithms. The sixth part of this thesis will present some of the properties of these networks.

Problems one has to face today not only require powerful representations, but also efficient optimization techniques able to handle the enormous quantity of data available, in real time if possible. The seventh part is therefore a presentation of a new and efficient online second order gradient descent method.

Keywords: kernel methods, neural networks, convexity, gradient descent, deep networks

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

Je tiens tout d'abord à remercier Yoshua Bengio pour m'avoir accueilli dans son laboratoire, guidé et conseillé tout au long de ces années. Son soutien indéfectible, ses encouragements et la confiance qu'il a placée en moi m'ont donné l'assurance et le courage nécessaire pour poursuivre dans cette voie.

Je tiens aussi à remercier Pierre-Antoine Manzagol pour m'avoir encouragé à remettre en question ce que je croyais savoir, pour les nombreuses discussions qu'on a eues et pour avoir aidé mon insertion à Montréal.

Je remercie encore tous ceux du LISA qui m'ont stimulé intellectuellement, sont venus m'aider quand j'en avais besoin et m'ont forcé à écrire cette thèse.

Enfin, je tiens à remercier Muriel pour avoir supporté mes doutes, ma fatigue et mes cours improvisés sur les réseaux de neurones pendant les repas.

AVANT-PROPOS

« Il avait appris sans effort l'anglais, le français, le portugais, le latin. Je soupçonne cependant qu'il n'était pas très capable de penser. Penser c'est oublier des différences, c'est généraliser, abstraire. Dans le monde surchargé de Funes, il n'y avait que des détails, presque immédiats. »

Jorge Luis Borges, in *Funes ou la mémoire*

À ma famille, pour son soutien indéfectible.. . .

CHAPITRE 1

INTRODUCTION À L'APPRENTISSAGE MACHINE

1.1 L'apprentissage chez les humains

Les humains apprennent de deux manières différentes :

- par règles, c'est-à-dire qu'un élément extérieur (livre, professeur, parent, ...) définit un concept et ce qui le caractérise
- par l'expérience, c'est-à-dire que l'observation du monde qui les entoure leur permet d'affiner leur définition d'un concept[1].

Il est intéressant de remarquer que des concepts qui nous apparaissent élémentaires, comme de savoir ce qu'est un chien, reposent bien plus sur l'expérience que sur des règles définies. Le Trésor de la Langue Française informatisé (`http://atilf.atilf.fr`), par exemple, définit le chien comme un « mammifère carnivore très anciennement domestiqué, dressé à la garde des maisons et des troupeaux, à la chasse ou bien élevé pour l'agrément ». Bien qu'exacte, cette définition nous est inutile lorsqu'il s'agit de différencier un chien d'un chat dans la rue.

Il nous est d'ailleurs quasiment impossible de définir précisément les critères définissant le concept « chien » que nous utilisons. Pour s'en convaincre, voyons les trois exemples suivants :

- un chien à qui on a enfilé un costume d'autruche est-il toujours un chien ?
- un chien à qui on a coupé les cordes vocales et transformé chirurgicalement pour qu'il ressemble à une autruche est-il toujours un chien ?
- un chien qu'on a modifié génétiquement pour qu'il ressemble à une autruche, qu'il puisse se reproduire avec les autruches mais pas avec les chiens est-il toujours un chien ?

Si la réponse à la première de ces questions nous apparaît évidente, il est plus malaisé de répondre aux deux dernières. Nous avons donc, en tant qu'humains, développé une

[1] les animaux n'apprennent que par l'expérience et par imitation, méthode qui peut être assimilée à de l'expérience.

conception très précise de ce qu'est un chien pour faire face aux situations de la vie courante. Les exemples présentés ci-dessus n'étant pas des exemples courants (sauf celui du déguisement dont on connaît bien les conséquences), notre définition est tout à coup prise en défaut.

Cette conception, personne ne nous l'a apprise. La première fois que nous avons vu un chien, un élément extérieur (livre, professeur, parent, ...) nous l'a nommé comme tel puis, découvrant le monde, nous avons vu de nombreux chiens jusqu'à en peaufiner une définition qui est celle que chacun de nous a. Il est d'ailleurs vraisemblable que, dans différentes parties du monde (et même peut-être chez chaque humain), la représentation mentale d'un chien diffère, quand bien même la définition dans les dictionnaires serait la même. **C'est donc bien la succession d'exemples qui nous ont été présentés qui nous a permis de « définir » très précisément ce qu'était un chien**.

De même, un enfant arrivera à créer des nouvelles phrases sans que les notions de sujet, verbe ou complément lui aient été apprises. C'est donc la simple succession d'exemples (les phrases qui ont été prononcées devant lui) qui a amené cette compréhension de la structure syntaxique d'une phrase.

1.2 L'apprentissage machine

L'apprentissage machine est un domaine à la jonction des statistiques et de l'intelligence artificielle qui a pour but de créer des algorithmes capables d'effectuer cette extraction de concepts à partir d'exemples.

Un algorithme d'apprentissage est une fonction mathématique $\mathcal{A}$ qui prend comme argument un ensemble d'exemples $\mathcal{D}$ et qui renvoie une fonction f, ou, plus formellement :

$$\begin{aligned} \mathcal{A} : (\mathcal{X} \times \mathcal{Y})^N &\rightarrow \mathcal{Y}^{\mathcal{X}} \\ \mathcal{D} &\rightarrow \mathcal{A}(\mathcal{D}) = f \end{aligned} \tag{1.1}$$

où $\mathcal{X}$ est l'espace des entrées et $\mathcal{Y}$ l'espace des sorties. Dans l'exemple de la section pré-

cédente, $\mathcal{D}$ représenterait l'ensemble des informations acquises au cours de notre vie (ce qu'on a vu ou entendu, par exemple) et f serait la fonction déterminant si un objet est un chien ou pas. $\mathcal{A}$, au contraire, représente notre patrimoine génétique qui détermine notre réaction à la présentation d'exemples et le spectre des fonctions engendrées possibles. Les règles qui nous ont été apprises requièrent l'extraction de concepts (la transformation de ces règles en données sensorielles) pour les appliquer au monde réel. En ce sens, elles sont modélisées par $\mathcal{D}$. Toutefois, elles peuvent parfois se présenter sous formes de vérités universelles qui contraindront la fonction finale, quels que soient les exemples vus par la suite, et doivent alors être modélisées par $\mathcal{A}$.

1.2.1 Qu'est-ce qu'un bon apprentissage ?

On associe souvent un bon apprentissage avec une bonne mémoire. Cela est valide lorsque la tâche à effectuer suit des règles bien établies et que ces règles nous sont données. En revanche, lorsqu'il s'agit de généraliser une règle à partir d'exemples, cela devient incongru. Si un humain n'a vu que dix chiens dans sa vie, doit-il en conclure que seuls ceux-là sont des chiens et qu'aucun autre élément du monde ne peut être un chien ? Que dire de ces mêmes chiens vus sous un angle différent ? Un ordinateur est excellent pour apprendre par cœur, c'est-à-dire pour réaliser la tâche demandée sur les exemples dont il s'est servi pour apprendre. En revanche, il est bien plus complexe de développer un algorithme capable de généraliser la tâche apprise à de nouveaux exemples.

La phase d'apprentissage d'un algorithme s'appelle aussi la phase d'**entraînement**. Lors de cette phase, l'algorithme se voit présenter des exemples sur lesquels il doit réaliser la tâche demandée. Ainsi, on lui présentera par exemple des images de chiens et d'autruches pour ensuite lui demander de faire la distinction entre ces deux types d'animaux. Lors de la phase de **test**, l'algorithme doit réaliser la tâche demandée sur de nouveaux exemples. S'il lui est présenté un chien d'une race jamais vue auparavant, saura-t-il déterminer correctement sa nature ?

L'idée sous-jacente de la phase d'apprentissage est que, s'il arrive à effectuer correctement la tâche demandée sur les exemples qui lui sont présentés (les exemples d'**apprentissage** ou exemples d'**entraînement**), il en a extrait les concepts importants

et sera à même de l'effectuer sur de nouveaux exemples (les exemples de **test**). En effet, ce qui nous intéresse réellement est la capacité de généralisation d'un algorithme, afin qu'il puisse être utilisé dans des environnements nouveaux. Malheureusement, il arrive parfois qu'un algorithme ne fasse aucune erreur sur les exemples d'apprentissage sans pour autant en avoir extrait aucun concept. Il suffit pour s'en rendre compte d'imaginer un algorithme se contentant d'apprendre par cœur. Si sa performance sur les exemples d'entraînement sera parfaite, rien ne nous permet d'affirmer qu'il sera en mesure d'utiliser ses connaissances sur de nouveaux exemples. Il est donc extrêmement important de garder constamment à l'esprit cette différence entre performance en entraînement et en généralisation. Toute la subtilité de l'apprentissage statistique repose sur ce paradigme : créer des algorithmes ayant une bonne capacité de généralisation alors même que nous n'avons à notre disposition qu'un nombre limité d'exemples d'entraînement.

1.2.2 Et l'apprentissage par règles ?

Une question subsiste : puisque l'apprentissage humain se fait par des règles et par l'expérience, peut-on également utiliser des règles pour l'apprentissage machine ?

Il est en effet tout à fait possible d'introduire des règles dans l'apprentissage machine. Dans ce cas, le concepteur de l'algorithme joue le rôle du professeur et introduit ses connaissances *a priori* en influant sur la structure de l'algorithme. Si cette approche est valide et efficace lorsqu'on veut résoudre un problème en particulier, elle est plus problématique si notre but est de créer une vraie intelligence sans supervision, et ce pour plusieurs raisons. Tout d'abord, le nombre total de règles à définir augmente linéairement avec le nombre de tâches à réaliser, ce qui devient vite irréalisable dès lors qu'on souhaite un algorithme (relativement) général. En outre, les règles sont des principes intangibles, dont la portée est fréquemment limitée par des exceptions. Ces dernières rendent encore plus ardue l'exhaustivité et la cohérence globale de cet ensemble. Enfin, l'évolution de l'humain a été majoritairement guidée par son apprentissage, que celui-là ait été acquis ou transmis par la génération précédente. Il semble donc raisonnable d'espérer parvenir au même résultat en utilisant les mêmes moyens.

Si les deux visions coexistent au sein de la communauté, mon souhait étant de dé-

couvrir l'intelligence la plus générale possible, je limiterai au maximum la quantité d'informations manuellement introduite dans mes algorithmes.

1.3 Différents types d'apprentissage

Ce travail portera sur trois catégories de tâches qu'on peut accomplir avec des algorithmes d'apprentissage machine :

- l'apprentissage supervisé
- l'apprentissage non supervisé
- l'apprentissage semi-supervisé

Une présentation de ces trois types d'apprentissage peut-être trouvée dans [20, p.3]. Je ne parlerai volontairement pas de l'apprentissage par renforcement, domaine à la fois plus général et plus difficile, mais dont les trois tâches précitées sont des éléments constitutifs.

1.3.1 Apprentissage supervisé

Dans l'apprentissage supervisé, les données fournies sont des paires : une **entrée** et une **étiquette**. On parle alors d'**entrées étiquetées**. L'entrée est un élément associé à une valeur ou à une classe et l'étiquette est la valeur ou la classe associée. Le but de l'apprentissage est d'inférer la valeur de l'étiquette étant donnée la valeur de l'entrée.

On peut distinguer deux grands types d'apprentissage supervisé : la **classification** et la **régression**.

1.3.1.1 Classification

Lorqu'on fait de la classification, l'entrée est l'instance d'une classe et l'étiquette est la classe correspondante. En reconnaissance de caractères, par exemple, l'entrée serait une suite de pixels représentant une lettre et la classe serait la lettre représentée (ou son index).

La classification consiste donc à apprendre une fonction f_{class} de $\mathbb{R}^d$ dans $\mathbb{N}$ qui associe à un vecteur sa classe. Dans certains cas, on pourra vouloir que la fonction f_{class}

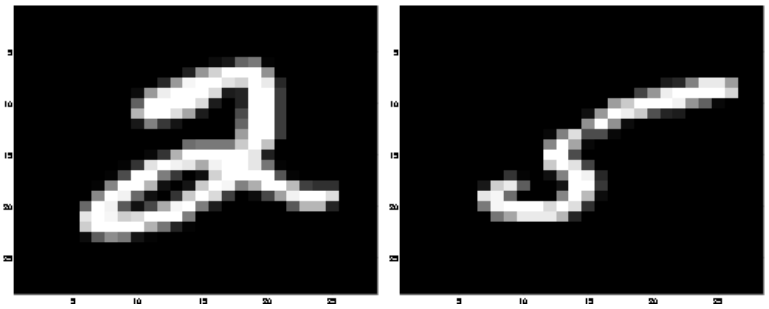


Figure 1.1 – Exemple de problème de classification : que représentent ces deux images ?

soit à valeurs dans $[0, 1]^k$ telle que chaque élément du vecteur de sortie représente la probabilité d'appartenance à une classe (la somme des éléments sera donc 1).

En reprenant la notation de l'équation 1.1, nous avons

$$\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{N}$$

et donc

$$\begin{aligned} f_{\text{class}} : \mathbb{R}^d &\rightarrow \mathbb{N} \\ \text{entrée} &\rightarrow f_{\text{class}}(\text{entrée}) = \text{classe} \end{aligned}$$

1.3.1.2 Régression

Dans le cas de la régression, l'entrée n'est pas associée à une classe mais à une ou plusieurs quantités continues. Ainsi, l'entrée pourrait être les caractéristiques d'une personne (son âge, son sexe, son niveau d'études) et l'étiquette son revenu.

La régression consiste donc à apprendre une fonction f_{regr} de $\mathbb{R}^d$ dans $\mathbb{R}^k$ qui associe à un vecteur sa valeur associée.

En reprenant la notation de l'équation 1.1, nous avons

$$\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{R}^k$$

et donc

$$\begin{aligned} f_{\text{regr}} : \mathbb{R}^d &\rightarrow \mathbb{R}^k \\ \text{entrée} &\rightarrow f_{\text{regr}}(\text{entrée}) = \text{valeur} \end{aligned}$$

1.3.2 Apprentissage non supervisé

Dans l'apprentissage non supervisé, les données sont uniquement constituées d'entrées. Dans ce cas, les tâches à réaliser diffèrent de l'apprentissage supervisé. Bien que de manière plus implicite, ces tâches sont également effectuées par les humains.

1.3.2.1 Estimation de densité

Le but de l'estimation de densité est d'inférer la répartition des données dans l'espace des entrées (ou, plus formellement, leur **distribution**). Par exemple, si on connaît les points d'impact de 500 boulets tirés par un canon dans un champ, peut-on en déduire la probabilité de chute d'un boulet en chaque point du champ (voir figure 1.3.2.1).

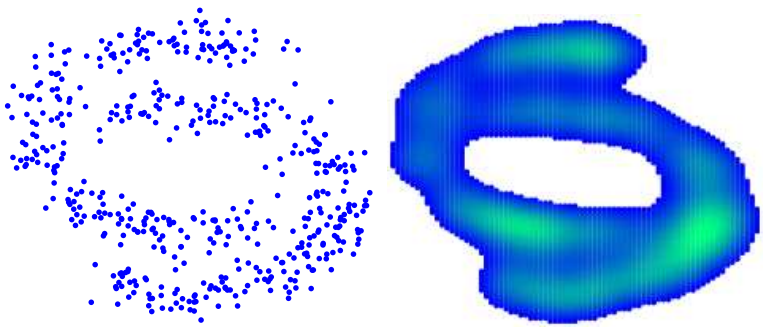


Figure 1.2 – Si un canon tire des boulets sur un champ selon une distribution fixe mais inconnue et qu'on connaît les points d'impact des 500 premiers boulets, peut-on estimer cette distribution ?

L'estimation de densité consiste donc à apprendre une fonction $f_{\text{est-dens}}$ de $\mathcal{X}$ dans $\mathcal{Y}$ telle que $\int_{\mathcal{X}} f_{\text{est-dens}} = 1$ qui associe à un vecteur sa probabilité. Si $\mathcal{X}$ est un espace discret, alors $\mathcal{Y} = [0, 1]$. Si $\mathcal{X}$ est un espace continu, alors $\mathcal{Y} = \overline{\mathbb{R}}_+$.

Supposons que $\mathcal{X}$ est un espace continu. Alors, en reprenant la notation de l'équation 1.1, nous avons

$$\mathcal{Y} = \overline{\mathbb{R}}_+$$

et donc

$$\begin{aligned} f_{\text{est-dens}} : \mathcal{X} &\rightarrow \overline{\mathbb{R}}_+ \\ \text{entrée} &\rightarrow f_{\text{est-dens}}(\text{entrée}) = \text{probabilité} \end{aligned}$$

1.3.2.2 Regroupement (ou « clustering »)

Le regroupement est l'équivalent non supervisé de la classification. Comme son nom l'indique, son but est de regrouper les données en classes en utilisant leurs similarités. La difficulté du regroupement réside dans l'absence de mesure générale de similarité. Celle-là doit donc être définie en fonction du problème à traiter. L'un des algorithmes de regroupement les plus couramment utilisés est l'algorithme des **k-moyennes**.

Le regroupement consiste donc à apprendre une fonction f_{regroup} de $\mathbb{R}^d$ dans $\mathbb{N}$ qui associe à un vecteur son groupe. Contrairement à la classification, le nombre de groupes n'est pas connu a priori.

Tout comme en classification, la fonction f_{regroup} peut parfois renvoyer un « vecteur d'appartenances » dont les éléments sont positifs et somment à 1. Dans ce cas, la i-ème composante représente le degré d'appartenance de l'entrée au groupe i.

En reprenant la notation de l'équation 1.1, nous avons

$$\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{N}$$

et donc

$$\begin{aligned} f_{\text{regroup}} : \mathbb{R}^d &\rightarrow \mathbb{N} \\ \text{entrée} &\rightarrow f_{\text{regroup}}(\text{entrée}) = \text{groupe} \end{aligned}$$

1.3.2.3 Réduction de dimensionnalité

Les algorithmes de réduction de dimensionnalité tentent de trouver une projection des données dans un espace de plus faible dimension, tout en préservant l'information contenue dans celles-là. Cette projection peut être linéaire (comme dans l'analyse en composantes principales ou ACP) ou non (par exemple les algorithmes Locally Linear Embedding (LLE), Isomap ou l'ACP à noyau). Une étude comparative de ces algorithmes peut être trouvée dans [11]. Certains de ces algorithmes ne fournissent que les coordonnées en basse dimension (appelées **coordonnées réduites**) des données d'apprentissage alors que d'autres fournissent explicitement la fonction de projection, ce qui permet de calculer les coordonnées réduites de nouveaux points. Toutefois, Bengio et al. [18] présentent une méthode permettant de calculer les coordonnées réduites de nouveaux points à partir de celles des données d'apprentissage.

La réduction de dimensionnalité consiste donc à apprendre une fonction f de $\mathbb{R}^d$ dans $\mathbb{R}^k$ qui associe à un vecteur sa représentation en faible dimension. Si l'algorithme est linéaire, la fonction f sera représentée par une matrice de taille (k, n).

En reprenant la notation de l'équation 1.1, nous avons

$$\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{R}^k, k < n$$

et donc

$$\begin{aligned} f_{\text{dim-red}} : \mathbb{R}^d &\rightarrow \mathbb{R}^k \\ \text{entrée} &\rightarrow f_{\text{dim-red}}(\text{entrée}) = \text{coordonnées réduites} \end{aligned}$$

1.3.3 Apprentissage semi-supervisé

Comme son nom l'indique, l'apprentissage semi-supervisé se situe entre l'apprentissage supervisé et l'apprentissage non supervisé. Certaines données sont étiquetées (c'est-à-dire sont composées d'une entrée et d'une étiquette) et d'autres ne le sont pas (seule l'entrée est fournie). Les tâches réalisées en apprentissage semi-supervisé sont les mêmes que celles réalisées en apprentissage supervisé (**régression** et **classification**), à la

différence qu'il est fait usage des données non étiquetées. Un exemple où l'apprentissage semi-supervisé est adapté est la reconnaissance d'images. Il est très facile de récolter un grand nombre d'images (sur Internet, par exemple), mais il est plus difficile (que ce soit en termes de coûts ou de temps) de les étiqueter. Dès lors, avoir un algorithme capable de tirer avantages des images non étiquetées pour améliorer ses performances est souhaitable.

1.4 Entraînement, test et coûts

Dans la section 1.2, je mentionnais les étapes d'entraînement et de test associées à un algorithme d'apprentissage. Cette section présente plus en détail les éléments nécessaires à ces étapes.

1.4.1 Algorithmes et paramètres

La section 1.2 définit un algorithme d'apprentissage comme une fonction mathématique $\mathcal{A}$ qui prend comme argument un ensemble d'exemples $\mathcal{D}$ et qui renvoie une fonction f. Cette section explique plus en détail par quels moyens l'ensemble $\mathcal{D}$ influe sur la fonction f.

Un algorithme d'apprentissage est un modèle mathématique composé de **paramètres** θ dont dépend la fonction f finale renvoyée. Le but de l'entraînement est de trouver l'ensemble de paramètres qui permet de réaliser au mieux la tâche sur les exemples d'entraînement. L'entraînement est donc une **optimisation** des paramètres. Il est intéressant de noter qu'il existe une autre vision de l'apprentissage pour lesquels tous les jeux de paramètres sont possibles, mais avec des probabilités différentes. L'entraînement n'est alors plus une optimisation des paramètres mais une **estimation** de cette distribution. Cette approche, appelée **approche bayésienne**, sera brièvement traitée à la section 1.6.2 et le lecteur est renvoyé à [45] pour une introduction complète à ce modèle.

Si tous les algorithmes d'apprentissage dépendent de paramètres, on peut en extraire deux catégories : les **algorithmes paramètriques** et les **algorithmes non-paramétriques**.

1.4.1.1 Algorithmes paramétriques

Un algorithme paramétrique est un algorithme dont le nombre de paramètres et la complexité n'augmentent pas avec la taille de l'ensemble d'entraînement. Par exemple, un classifieur linéaire est un algorithme paramétrique puisque la fonction modélisée sera déterminée par l'hyperplan de séparation, quelle que soit la taille de l'ensemble d'entraînement.

1.4.1.2 Algorithmes non-paramétriques

Un algorithme non-paramétrique est un algorithme dont le nombre de paramètres et la complexité augmentent avec la taille de l'ensemble d'entraînement. Un exemple d'algorithme non-paramétrique est celui des k-plus proches voisins [20, p.125].

1.4.2 Entraînement et test

Lors de la phase d'entraînement, on veut que l'algorithme réalise une tâche prédéfinie sur un ensemble d'exemples (l'**ensemble d'entraînement**), c'est-à-dire qu'il renvoie les bonnes réponses en sortie lorsque les exemples lui sont présentés en entrée.

Il est donc important de définir :

- ce qu'est une réponse correcte
- ce qu'est une réponse incorrecte
- le coût associé à une erreur commise,

définitions qui sont propres au problème à résoudre. Ainsi, dans le cas de la reconnaissance d'images, confondre un 2 avec un 3 est de même importance que de confondre un 4 avec un 7. En revanche, si notre algorithme est utilisé pour des diagnostics médicaux, détecter une tumeur alors qu'il n'y en a pas est bien moins grave que l'inverse.

Pour apprendre à exécuter une tâche, un algorithme d'apprentissage nécessite donc :

- des données d'entraînement qui peuvent être des paires (x_i, y_i) (cas supervisé) ou simplement des entrées $\{x_i\}$ (cas non supervisé)
- un coût qui représente notre définition de l'erreur.

Le principe le plus simple pour entraîner un algorithme d'apprentissage est la minimisation du coût sur les données d'entraînement : on parle d'**erreur d'entraînement**. On teste ensuite notre algorithme sur de nouvelles données en calculant le coût associé : l'**erreur de test** ou **erreur de généralisation**. Cela permet une évaluation réaliste de la qualité de l'algorithme.

Il est important de rappeler qu'on entraîne un algorithme en minimisant l'erreur d'entraînement alors que sa performance se mesure par son erreur de généralisation. Si ces deux erreurs sont généralement liées, il existe des algorithmes qui minimisent parfaitement l'erreur d'apprentissage tout en ayant une erreur de généralisation élevée (on parlera alors de **surapprentissage**). La section 1.5 présente des méthodes pour limiter ce phénomène.

1.4.3 Coûts et erreurs

Il est primordial de bien choisir le coût qu'on utilise. En effet, c'est par son intermédiaire qu'on informe l'algorithme de la tâche qu'on veut réaliser. Si l'algorithme doit faire des compromis entre plusieurs erreurs, c'est le coût qui déterminera le choix final. Les deux sous-sections suivantes présentent les coûts les plus fréquemment utilisés.

Nous appelons :

- $x_i \in \mathcal{X}$ l'entrée de notre i-ème donnée d'entraînement
- $y_i \in \mathcal{Y}$ l'étiquette (ou sortie désirée) de notre i-ème donnée d'entraînement, dans le cas de l'apprentissage supervisé
- f_θ la fonction induite par notre algorithme d'apprentissage avec les paramètres θ.
- $c : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ la fonction de coût.

1.4.3.1 Coût pour l'apprentissage supervisé

Pour chaque exemple d'entraînement x_i, nous avons une sortie désirée y_i et une sortie obtenue $f_\theta(x_i)$. Le but de l'apprentissage est de trouver les paramètres θ de notre algorithme qui minimisent la dissimilarité $c(y_i, f_\theta(x_i))$ entre y_i et $f_\theta(x_i)$. De manière

plus formelle, on cherche

$$\theta^* = \text{argmin}_\theta \sum_i c\left[f_\theta(x_i), y_i\right] \tag{1.2}$$

La mesure de dissimilarité (ou **fonction de coût**) dépend du problème qu'on souhaite résoudre. Les plus répandues sont :

- erreur quadratique : $c(y_i, f_\theta(x_i)) = \|y_i - f_\theta(x_i)\|_2^2$
- log vraisemblance negative : $c(y_i, f_\theta(x_i)) = -\log f_\theta(x_i)_{y_i}$
- erreur de classification : $c(y_i, f_\theta(x_i)) = 1_{y_i \neq f_\theta(x_i)}$ où 1 est la fonction indicatrice.

L'erreur quadratique est utilisée dans les problèmes de **régression**. Sa non-linéarité permet de tolérer les faibles erreurs tout en cherchant activement à limiter les erreurs importantes.

La **log vraisemblance négative (LVN)** est utilisée pour les problèmes de classification. La formule est utilisée pour des algorithmes dont la sortie $f_\theta(x_i)$ est un vecteur dont les composantes sont positives, somment à 1 et sont aussi nombreuses que le nombre de classes. La j-ème composante (c'est-à-dire $f_\theta(x_i)_j$) représente la probabilité que l'entrée x_i appartienne à la classe j. On cherche alors à trouver les paramètres θ^* qui, étant données les entrées $\{x_i\}$, maximisent la probabilité des étiquettes associées y_i :

$$\begin{aligned}
\theta^* &= \text{argmax}_\theta \prod_i P(y_i|x_i) \\
&= \text{argmax}_\theta \sum_i \log P(y_i|x_i) \\
&= \text{argmin}_\theta - \sum_i \log P(y_i|x_i) \\
\theta^* &= \text{argmin}_\theta - \sum_i f_\theta(x_i)_{y_i}
\end{aligned}$$

L'erreur de classification n'est pas utilisée pour l'apprentissage à cause de sa non-dérivabilité par rapport à θ. On l'utilise toutefois en test comme mesure de la qualité d'un algorithme de classification.

1.4.3.2 Coût pour l'apprentissage non-supervisé

Cette section (ainsi que le reste de cette thèse) ne traitera pas de la question du regroupement. Nous allons donc définir un coût pour l'apprentissage non-supervisé dans le seul cadre de l'estimation de densité.

Une supposition commune est que les données sont générées indépendamment et d'une distribution unique (*iid* pour **indépendantes et identiquement distribuées**). Cela est faux dans certains contextes (données temporelles, spatiales, ...) qui ne seront pas traités dans cette thèse. Nous supposerons donc désormais que nos données sont générées de façon **iid**.

Dans ce cas, la vraisemblance d'un ensemble de données est égal au produit des probabilités des éléments de cet ensemble :

$$\text{Si } \mathcal{D} = \{x_1, \ldots, x_N\}, \text{ alors } p(\mathcal{D}) = \prod_i p(x_i) \tag{1.3}$$

Pour des raisons de simplicité de calcul, nous allons minimiser la **log vraisemblance négative (LVN)** au lieu de maximiser la vraisemblance en notant que cela est équivalent.

1.5 Régularisation

Un algorithme d'apprentissage $\mathcal{A}$ prend en entrée un ensemble de N exemples d'entraînement $\mathcal{D}$ et renvoie une fonction $\mathcal{A}(\mathcal{D}) = f$ associée.

Nos exemples d'entraînement sont tirés d'une distribution p sur $\mathcal{X} \times \mathcal{Y}$ dans le cas de l'apprentissage supervisé (et de $\mathcal{X}$ dans le cas de l'apprentissage non supervisé). Comme mentionné à la section 1.4.3.2, cette distribution p induit une distribution sur les ensembles d'entraînement $\mathcal{D}$ de taille N (appartenant à $(\mathcal{X} \times \mathcal{Y})^N$ dans le cas de l'apprentissage supervisé et $\mathcal{X}^N$ dans le cas de l'apprentissage non supervisé) :

$$\text{Si } \mathcal{D} = \{x_1, \ldots, x_N\}, \text{ alors } p(\mathcal{D}) = \prod_i p(x_i)$$

Dans le cas où $\mathcal{A}$ n'est pas un algorithme trivial, deux ensembles d'entraînement diffé-

rents $\mathcal{D}_1$ et $\mathcal{D}_2$ donneront deux fonctions différentes f_1 et f_2.

Appelons f^* la vraie fonction (inconnue) qu'on cherche à approcher le mieux possible. Le choix de notre algorithme d'apprentissage sera guidé par deux questions intimement liées :

- en moyenne sur l'ensemble des ensembles d'entraînement possibles, notre algorithme va-t-il donner la bonne réponse ?
- quelle est la sensibilité de notre algorithme à l'ensemble d'entraînement particulier qui est fourni ?

Mathématiquement, ces deux questions se traduisent par l'évaluation de deux quantités : le biais B et la variance V. Définissons tout d'abord la fonction moyenne résultant de notre algorithme d'apprentissage :

$$\hat{f} = E_{\mathcal{D}}\mathcal{A}(\mathcal{D}) \tag{1.4}$$

On obtient alors les formules du biais et de la variance suivantes :

$$B = E_x\left[\hat{f}(x) - f^*(x)\right] \tag{1.5}$$

$$V = E_{\mathcal{D}}\left[E_x\left[\mathcal{A}(\mathcal{D})(x) - \hat{f}(x)\right]^2\right] \tag{1.6}$$

On peut montrer [20, p.147] que l'erreur de généralisation de notre algorithme est égale à

$$R = B^2 + V + \sigma^2 \tag{1.7}$$

où σ représente le bruit, c'est-à-dire la partie de y qu'il est impossible de prédire, étant donné x.

Pour assurer une bonne généralisation, il est donc important de diminuer à la fois le biais et la variance. Cette équation nous permet de mieux comprendre les défauts de l'apprentissage par coeur. En effet, celui-là aura un biais nul mais une variance très élevée (puisque chaque ensemble d'apprentissage différent donnera lieu à une fonction très différente) avec pour conséquence une grande erreur de généralisation. De la même façon, un algorithme trop simple possèdera une variance faible (ayant peu de paramètres, les

spécificités de chaque ensemble d'entraînement n'auront que peu d'influence) mais un biais élevé (si les fonctions générées sont trop simples, elles ne pourront bien représenter la vraie fonction f^*).

L'idée de la régularisation est donc de réduire la variance de notre algorithme tout en essayant de maintenir un biais le plus faible possible. La méthode utilisée dépendra donc du problème à traiter et la solution finale obtenue sera donc un compromis entre adaptation aux données et simplicité. La figure 1.3 donne deux exemples de l'effet d'une régularisation sur un algorithme.

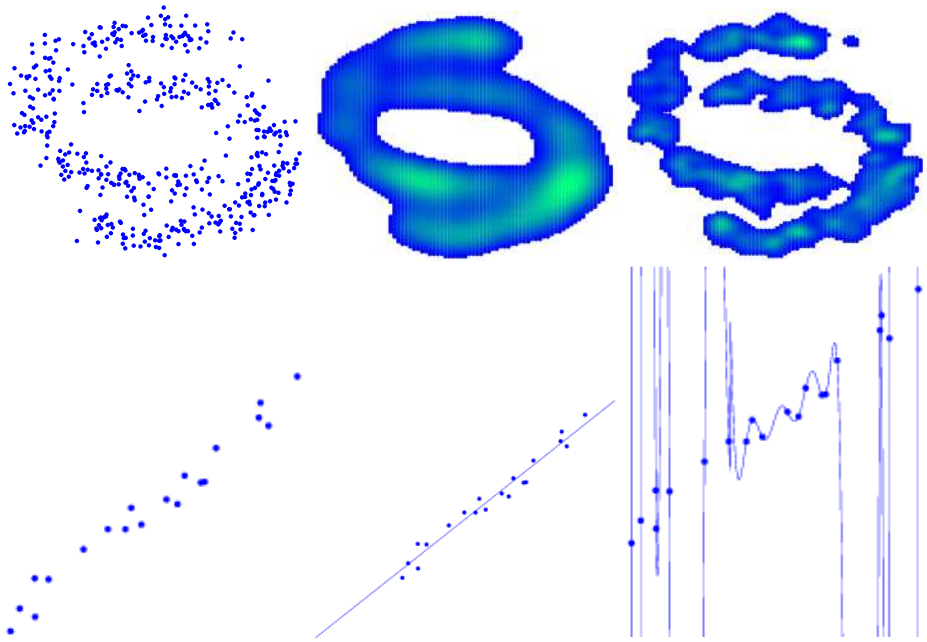


Figure 1.3 – Exemple de solutions de complexités différentes pour deux problèmes d'apprentissage : en haut, de l'estimation de densité et en bas, de la régression. **Gauche** : données d'entraînement. **Milieu** : solution simple (régularisée). **Droite** : solution compliquée (non régularisée).

Cette régularisation peut s'obtenir de plusieurs manières, parmi lesquelles deux seront abordées ici.

1.5.1 Limitation du nombres de paramètres

Les **algorithmes paramétriques** possèdent une structure et un nombre de paramètres fixés et indépendants du nombre d'exemples d'apprentissage. Réduire le nombre de paramètres d'un algorithme paramétrique permet d'en limiter la capacité, c'est-à-dire le nombre de fonctions qu'il peut modéliser. Cela réduira la variance entre les fonctions renvoyées pour des ensembles d'apprentissage différents. Dans un réseau de neurones, cela reviendrait par exemple à avoir un nombre réduit de neurones sur la couche cachée (voir section 2.2).

1.5.2 Pénalisation des poids (« weight decay »)

Plutôt que de supprimer des paramètres afin d'en limiter le nombre total, on pourrait simplement les forcer à prendre la valeur 0. Il en résulte l'intuition qu'un paramètre proche de 0 sera moins « actif » qu'un paramètre dont la valeur est forte. Si cette intuition a des fondements théoriques (voir par exemple [60]), cette section se contente de présenter des exemples de régularisation utilisant ce principe.

Nous allons donc ajouter au coût d'apprentissage une pénalité sur les poids (« weight decay ») que nous allons également chercher à minimiser. Le jeu de paramètres optimal sera donc celui qui assurera un compromis entre une bonne qualité de modélisation des données d'apprentissage (minimisation du coût empirique) et une bonne simplicité (minimisation de la pénalité). On appelle **coût régularisé** de l'algorithme la somme de des deux coûts.

Cette méthode de régularisation présente l'avantage de pouvoir être appliquée tant aux algorithmes paramétriques qu'aux algorithmes non-paramétriques, comme les méthodes à noyau présentées à la section 2.1.

Les sous-sections suivantes présentent les pénalisations les plus fréquemment rencontrées dans la littérature.

1.5.2.1 Norme $\mathcal{L}^1$

Soit θ l'ensemble des paramètres de notre algorithme (ou le sous-ensemble qu'on veut régulariser). La régularisation de la norme $\mathcal{L}^1$ (appelée plus simplement **régularisation** $\mathcal{L}^1$) est de la forme $\lambda\|\theta\|_1$.

λ est appelé coefficient de régularisation (*weight decay coefficient*, en anglais) et sa valeur détermine le compromis entre la simplicité que nous voulons avoir et la qualité de la modélisation des données d'apprentissage.

Modifier un paramètre de la valeur 0 à la valeur ε augmente le coût de régularisation de $\lambda\varepsilon$; cela n'est intéressant que si la dérivée du coût par rapport à ce paramètre est supérieur à λ (en valeur absolue). La régularisation $\mathcal{L}^1$ encourage donc les paramètres à être nuls. C'est pourquoi on utilise ce type de régularisation lorsqu'on veut encourager les solutions possédant peu de paramètres non nuls. Toutefois, sa non-dérivabilité en 0 complique parfois le processus d'optimisation.

1.5.2.2 Norme $\mathcal{L}^2$

La régularisation de la norme $\mathcal{L}^2$ (ou **régularisation** $\mathcal{L}^2$) est de la forme $\frac{\lambda}{2}\|\theta\|_2^2$. La rigueur voudrait qu'elle soit appelé **régularisation de la norme $\mathcal{L}^2$ au carré**, mais comme personne n'utilise la vraie norme $\mathcal{L}^2$, la simplicité y a gagné ce que la précision y a perdu. En outre, on utilise souvent λ à la place de $\frac{\lambda}{2}$.

Modifier un paramètre de la valeur 0 à la valeur ε augmente le coût de régularisation de $\lambda\varepsilon^2$ alors que le coût empirique varie en ε ; c'est la raison pour laquelle la régularisation $\mathcal{L}^2$ tend à fournir des solutions dont aucun paramètre n'est nul. Toutefois, le terme de régularisation étant différentiable, l'optimisation du coût est bien plus facile que dans le cas de la régularisation $\mathcal{L}^1$.

1.5.3 Discussion

Ainsi que nous l'avons dit en préambule de cette section, le but de la régularisation est de favoriser les fonctions simples afin de limiter la variance de notre algorithme.

Toute la difficulté du choix de la régularisation réside donc dans l'analogie entre la méthode choisie et la notion de simplicité induite.

Tout être humain tend également à favoriser les solutions simples, comme en atteste le rasoir d'Occam :

> *Pluralitas non est ponenda sine necessitate*

qui pourrait être traduit par

> *Les multiples ne doivent pas être utilisés sans nécessité*

(http ://fr.wikipedia.org/wiki/Rasoir_d'Occam)

L'une des étapes de notre quête de l'intelligence artificielle est donc de découvrir et de modéliser la notion de simplicité que les humains utilisent. Les deux exemples de la figure 1.3 semblent exhiber des régularités validant nos critères, mais sont-ce les seules ?

Neal [60] montra qu'il n'était pas nécessaire de limiter le nombre de neurones sur la couche cachée d'un réseau de neurones pour se restreindre à des solutions simples, si tant est qu'on pénalise suffisamment les paramètres de notre algorithme. Hinton et al. [39], LeCun et al. [54] proposent des algorithmes possédant un grand nombre de paramètres soumis à des contraintes supplémentaires pour en limiter la capacité. Les résultats spectactulaires obtenus par de tels algorithmes laissent penser que la notion humaine de simplicité n'émane pas d'un nombre limité de paramètres (notre cerveau possède bien plus de neurones que n'importe quel algorithme créé à ce jour) mais plutôt de contraintes supplémentaires restreignant le champ des solutions possibles.

Le travail proposé dans cette thèse sera donc axé autour d'algorithmes possédant un grand nombre de paramètres contraints de manière appropriée.

1.6 Théorème de Bayes

Nous allons maintenant présenter un théorème de grande importance en statistiques et en apprentissage machine. Cela va nous permettre de présenter les concepts de coût et de régularisation sous un nouveau jour. Soit le problème suivant :

> Supposons qu'on connaisse la proportion d'hommes et de femmes au Canada ainsi que la distribution des tailles des individus de chaque sexe. Si un

Canadien choisi au hasard mesure 1m70, quelle est la probabilité que ce soit un homme ?

Le théorème de Bayes permet de répondre à cette question. Plus formellement, il permet de répondre à la question suivante : soit deux événements A et B. Si on connaît a priori la probabilité que l'événement B se produise et la probabilité que l'événement A se produise sachant que B s'est produit, peut-on en déduire la probabilité que B se produise sachant que A s'est produit ?

Théorème 1.6.1.

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{1.8}$$

Dans l'exemple ci-dessus, l'événement B est le sexe de l'individu et l'événement A est sa taille.

Réécrivons ce théorème dans le cadre de l'apprentissage machine en précisant ce que sont les événements A et B :

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})} \tag{1.9}$$

Dans ce cas, nous sommes intéressés par la probabilité d'avoir un jeu de paramètres particulier θ étant donné que nous avons à notre disposition l'ensemble d'apprentissage $\mathcal{D}$. Nous voyons que pour calculer cette probabilité, nous avons besoin de calculer trois termes :

- $P(\mathcal{D}|\theta)$ est la probabilité que l'ensemble d'apprentissage $\mathcal{D}$ ait été généré sachant que les paramètres de l'algorithme étaient θ. Cette quantité s'appelle la **vraisemblance** de l'ensemble d'apprentissage. Cette quantité est définie par notre modèle.
- $P(\theta)$ est la probabilité a priori d'avoir comme jeu de paramètres θ. Cette quantité s'appelle l'**a priori** et représente notre connaissance du monde. Elle peut être fixée arbitrairement.
- $P(\mathcal{D})$ est la probabilité d'avoir obtenu cet ensemble d'apprentissage. Elle sert simplement à s'assurer que le terme $P(\theta|\mathcal{D})$ est bien une probabilité et somme à 1. Elle s'appelle la **constante de normalisation** .

Le terme $P(\theta|\mathcal{D})$ est appelé **postérieur** de θ sachant $\mathcal{D}$. Il diffère de l'a priori $P(\theta)$ par l'information apportée par $\mathcal{D}$. On voit donc que, pour qu'un jeu de paramètres θ soit probable sachant qu'on a à notre disposition un ensemble d'apprentissage $\mathcal{D}$, il faut non seulement qu'il ait pu générer cet ensemble d'apprentissage (terme $P(\mathcal{D}|\theta)$), mais également qu'il corresponde à notre vision du monde (terme $P(\theta)$).

À partir de ce postérieur, deux stratégies sont possibles pour calculer la fonction f :

– on peut utiliser le jeu de paramètres θ^* le plus probable selon le postérieur pour faire nos prédictions

$$f(x) = f_{\theta^*}(x) \qquad \theta^* = \text{argmax}_\theta P(\theta|\mathcal{D}) \tag{1.10}$$

– on peut utiliser tous les jeux de paramètres possibles avec la probabilité donnée par leur postérieur pour faire nos prédictions

$$f(x) = \int_\theta f_\theta(x) P(\theta|\mathcal{D})\ d\theta \tag{1.11}$$

La première approche s'appelle **Maximum A Posteriori** et la deuxième approche s'appelle l'**apprentissage bayésien**.

1.6.1 Maximum A Posteriori

L'entraînement d'un algorithme d'apprentissage par Maximum A Posteriori (MAP) est la procédure la plus couramment utilisée dans la communauté, notamment pour sa simplicité. Elle consiste à trouver l'ensemble de paramètres qui est le plus probable a posteriori, c'est-à-dire après avoir vu l'ensemble d'apprentissage $\mathcal{D}$. On veut donc trouver le θ qui maximise

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})} \tag{1.12}$$

En ignorant le dénominateur de l'équation 1.12 (qui ne dépend pas de θ) et en prenant l'opposé du logarithme du numérateur, on obtient que le θ^* cherché est celui qui minimise

$$\mathcal{L}(\theta) = -\log\left[P(\mathcal{D}|\theta)P(\theta)\right] \tag{1.13}$$

par rapport à θ. L'a priori $P(\theta)$ peut être choisi arbitrairement et correspond à notre définition d'un jeu de paramètres probable. Si cela dépend fortement du problème, les a priori les plus fréquemment utilisés seront présentés et analysés à la section 1.6.3. Le terme $P(\mathcal{D}|\theta)$ est la vraisemblance de notre ensemble d'apprentissage étant donnés les paramètres. Les modèles les plus couramment utilisés pour cette distribution sont présentés et analysés dans la section 1.6.3. Si on utilise un a priori uniforme, c'est-à-dire qui ne privilégie pas certaines configurations de paramètres, la procédure MAP revient à choisir le jeu de paramètres pour lequel les données sont les plus vraisemblables : on fait alors du **maximum de vraisemblance**.

Appelons θ^* le jeu de paramètres qui minimise le coût :

$$\theta^* = \text{argmin}_\theta \mathcal{L}(\theta) \tag{1.14}$$

La sortie y pour un exemple x sera alors

$$y = f_{\theta^*}(x) \tag{1.15}$$

1.6.2 Apprentissage bayésien

L'apprentissage bayésien n'essaye pas de trouver le jeu de paramètres le plus probable mais considère l'intégralité du postérieur sur les paramètres. Étant donnée une entrée x, la sortie y vaut :

$$y = \int_\theta f_\theta(x) P(\theta|\mathcal{D}) \, d\theta \tag{1.16}$$

Le but de l'apprentissage bayésien est donc de calculer le postérieur sur les paramètres $P(\theta|\mathcal{D})$ au lieu de simplement considérer le θ^* le plus probable, comme c'est le cas lorsqu'on fait du MAP. On ne parle plus d'**optimisation** des paramètres mais d'**estimation** du postérieur. Cette méthode a été présentée en détail par Neal [60] et, si elle est l'approche correcte pour réaliser des prédictions, la difficulté d'estimation du postérieur nécessite généralement des approximations.

1.6.3 Interprétation bayésienne du coût

Cette section met en correspondance certains des coûts présentés à la section 1.4.3 avec des termes de vraisemblance, ainsi que les régularisations présentées à la section 1.5 avec des termes d'a priori. Pour cela, nous allons tout d'abord considérer le contexte de l'apprentissage par MAP de la section 1.6.1. Nous appelons $\mathcal{L}$ le coût que nous essayons de minimiser

$$\mathcal{L}(\theta) = -\log\left[P(\mathcal{D}|\theta)P(\theta)\right] \tag{1.17}$$

par rapport à θ.

En supposant que les données sont tirées de manière iid, comme à la section 1.4.3.2, et en utilisant l'équation 1.3, on en déduit

$$-\log P(\mathcal{D}|\theta) = -\sum_{i=1}^{n} \log P((x_i, y_i)|\theta) \tag{1.18}$$

et le coût total s'écrit

$$\mathcal{L}(\theta) = -\sum_{i=1}^{n} \log P((x_i, y_i)|\theta) - \log P(\theta) \tag{1.19}$$

Les deux prochaines sections s'intéressent au terme $-\sum_{i=1}^{n} \log P((x_i, y_i)|\theta)$ alors que les deux suivantes s'intéressent au terme $-\log P(\theta)$.

1.6.3.1 Bruit gaussien et erreur quadratique

Soit f^* la vraie fonction qui a généré les données. Supposons que ces données aient été perturbées par un bruit gaussien isotrope de variance $\sigma_d I$, c'est-à-dire que, pour tout couple $\{x, y\}$ appartenant à $(\mathcal{X} \times \mathcal{Y})$, on a

$$y|x \sim \mathbb{N}(f^*(x), \sigma_d) \tag{1.20}$$

En appelant f_θ la fonction générée par notre algorithme, la log-vraisemblance néga-

tive des données sous notre modèle est

$$-\sum_{i=1}^{n} \log P(y_i|x_i,\theta) = \sum_{i=1}^{n} \frac{\|y_i - f_\theta(x_i)\|^2}{2\sigma_d^2} - \log Z \tag{1.21}$$

où Z est la constante de normalisation de la gaussienne et est donc independante de θ. Le lecteur aura remarqué que le terme $P((x_i, y_i)|\theta)$ de l'équation 1.19 a été remplacé par un terme $P(y_i|x_i, \theta)$, c'est-à-dire qu'on a remplacé la modélisation de la densité jointe des données par la modélisation de la densité conditionnelle. Lasserre et al. [50] discutent de la différence entre ces deux modèles et nous invitons le lecteur à s'y référer pour une analyse des implications de cette modification. Nous dirons simplement que celle-là est valide dès lors que nous voulons effectivement modéliser uniquement la probabilité conditionnelle.

Dans ce contexte, la minimisation de la log-vraisemblance négative des données est donc égale à la minimisation de l'erreur quadratique.

1.6.3.2 Interprétation bayésienne de la log-vraisemblance négative

Dans le contexte de la classification, supposer un bruit gaussien sur les étiquettes n'a aucun sens. Toutefois, si nous définissons f_θ comme une fonction à valeurs dans $\mathbb{R}^k$ où k est le nombre de classes telle que pour tout x de $\mathcal{X}$, tous les éléments de $f_\theta(x)$ sont positifs et somment à 1, alors on peut définir

$$P(y = y_i|x = x_i) = \mathrm{softmax}\left(f_\theta(x_i)\right)_{y_i} \tag{1.22}$$

où la fonction $\mathrm{softmax}$ est définie par

$$\mathrm{softmax}(u)_i = \frac{\exp(u_i)}{\sum_j \exp(u_j)}$$

.

On retrouve alors le coût de la log-vraisemblance négative.

1.6.3.3 A priori de Laplace et régularisation $\mathcal{L}^1$

Une **distribution de Laplace** est une distribution de probabilité de la forme

$$P(\theta) = \frac{1}{Z} \exp\left(-\frac{\|\theta - \mu\|}{b}\right)$$

.

Nous allons supposer une moyenne nulle sur nos paramètres d'après l'analyse faite à la section 1.5.2 liant la faible valeur des paramètres à la simplicité de la fonction engendrée.

Ainsi,

$$P(\theta) = \frac{1}{Z} \exp\left(-\frac{\|\theta\|}{b}\right)$$

et donc

$$-\log P(\theta) = \frac{\|\theta\|}{b} \tag{1.23}$$

et on retrouve la régularisation $\mathcal{L}^1$ présentée à la section 1.5.2.1 avec $\lambda = \frac{1}{b}$. Une valeur élevée de λ, pénalisant grandement les paramètres, correspond donc à un a priori sur les paramètres reserré autour de 0. Il est donc normal qu'on cherche à réduire la valeur des paramètres si on croit fermement qu'ils doivent être proches de 0.

1.6.3.4 A priori gaussien et la régularisation $\mathcal{L}^2$

Supposons un a priori gaussien de moyenne 0 sur nos paramètres, c'est-à-dire

$$P(\theta) = \mathbb{N}(\theta, 0, \Sigma) \tag{1.24}$$

Nous allons également considérer une variance isotrope, c'est-à-dire pénalisant de manière égale tous les éléments de θ. Cette supposition est valide dans tous les algorithmes où tous les paramètres jouent un rôle identique, ce qui sera le cas de ceux présentés dans cette thèse. Nous avons donc $\Sigma = \sigma I$. En supposant une moyenne nulle sur les

paramètres, comme à la section précédente, nous avons alors

$$P(\theta) = \mathbb{N}(\theta, 0, \sigma I) \tag{1.25}$$

et donc

$$-\log P(\theta) = \frac{\|\theta\|^2}{2\sigma^2} \tag{1.26}$$

On retrouve la régularisation $\mathcal{L}^2$ présentée à la section 1.5.2.2. Là encore, par analogie, on obtient $\lambda = \frac{1}{\sigma^2}$ et le même raisonnement que précédemment peut être appliqué.

CHAPITRE 2

MÉTHODES À NOYAU ET RÉSEAUX DE NEURONES

Ce chapitre a pour but de présenter deux familles d'algorithmes fréquemment utilisées en apprentissage machine et qui posent les fondations de cette thèse.

2.1 Les méthodes à noyau

Les méthodes à noyau ont été introduites dans les années 90 et largement utilisées depuis lors. Schölkopf et al. [75] passent en revue de manière détaillée ce domaine. Cette dénomination recouvre un large éventail d'algorithmes dont la particularité est d'utiliser une **fonction à noyau** qui permet d'obtenir des versions non-linéaires d'algorithmes autrement linéaires. Les plus populaires d'entre eux sont entre autres [20] :

- la régression linéaire à noyau, une extension de la régression linéaire,
- la régression logistique à noyau, une extension de la régression logistique,
- les machines à vecteurs de support, une extension des classifieurs à marge maximale.

2.1.1 Principe des méthodes à noyau

Les méthodes à noyaux se fondent sur des algorithmes dont la solution peut être écrite en fonction de produits scalaires $< x_i, x_j >$ et dont la valeur en un nouveau point x peut être écrite en fonction des produits scalaires $< x, x_i >$ où les x_i sont des exemples d'apprentissage. Le produit canonique de l'espace euclidien est alors remplacé par un produit scalaire dans un **espace de caractéristiques** (*feature space*) de dimension supérieure (potentiellement infinie). En pratique, les termes $< x_i, x_j >$ sont remplacés par des termes $K(x_i, x_j)$ où K est une fonction symétrique définie positive de $\mathcal{X} \times \mathcal{X}$ dans $\mathbb{R}$ appelée **noyau** (parfois également **fonction à noyau**). Dans de tels algorithmes, le choix de la fonction K est crucial. Les plus populaires sont [77] :

- le **noyau gaussien** : $K(x_i, x_j) = \exp\left(-\frac{(x_i - x_j)^T \Sigma^{-1} (x_i - x_j)}{2}\right)$ où Σ est une

matrice symétrique définie positive appelée **matrice de covariance** choisie manuellement (c'est donc un **hyperparamètre**). La matrice de covariance est souvent un multiple de la matrice identité ($\Sigma = \sigma^2 I$), auquel cas la fonction de noyau s'écrit $K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$.

- le **noyau polynomial inhomogène** : $K(x_i, x_j) = (c + < x_i, x_j >)^D$ où c est dans $\mathbb{R}$ et D est dans $\mathbb{N}$. c et D sont des hyperparamètres. Ce noyau est parfois simplement appelé **noyau polynomial**.
- le **noyau sigmoïdal** : $K(x_i, x_j) = \tanh(\kappa < x_i, x_j > + \nu)$ où κ et ν sont dans $\mathbb{R}$. κ et ν sont des hyperparamètres. Ce noyau n'est pas défini positif.

2.1.2 Espace de Hilbert à noyau reproduisant

Étant donnés un noyau K défini positif et un ensemble $\mathcal{X}$, on peut définir un espace de Hilbert $\mathcal{H}_k$ des fonctions f de la forme

$$f = \sum_{i=1}^{m} \alpha_i K(\cdot, x_i) \tag{2.1}$$

où m est dans $\mathbb{N}$, $(\alpha_1, \ldots, \alpha_m)$ est dans $\mathbb{R}^m$ et $(x_1, \ldots, x_m)$ est dans $\mathcal{X}^m$. Le produit scalaire dans $\mathcal{H}_k$ est défini par

$$\begin{aligned} f &= \sum_{i=1}^{m} \alpha_i K(\cdot, x_i) \\ g &= \sum_{j=1}^{n} \beta_j K(\cdot, y_j) \\ < f, g >_{\mathcal{H}_k} &= \sum_{i=1}^{m} \sum_{j=1}^{n} \alpha_i \beta_j K(x_i, y_j) \end{aligned} \tag{2.2}$$

$\mathcal{H}_K$ est appelé l'**espace de Hilbert à noyau reproduisant** associé au noyau K. Le terme « reproduisant » vient du fait que, pour toute paire (x_i, x_j) de $\mathcal{X}^2$, nous avons

$$< K(\cdot, x_i), K(\cdot, x_j) >_{\mathcal{H}_k} = K(x_i, x_j) \tag{2.3}$$

Une méthode à noyau appliquera d'abord une fonction K_k aux exemples d'apprentissage :

$$\begin{aligned} K_k : \mathcal{X} &\rightarrow \mathcal{H}_k \\ x_i &\rightarrow K(\cdot, x_i) \end{aligned} \tag{2.4}$$

puis entraînera un modèle linéaire sur les exemples transformés, tout en maintenant une régularisation particulière mettant en jeu les termes $K(x_i, x_j)$.

2.1.3 Théorème du représentant

Le **théorème du représentant** a été formulé et prouvé dans [47]. Ce théorème lie la forme du terme de régularisation dans la fonction de coût à la forme de la fonction minimisant ce coût régularisé. La version de ce théorème proposée ici est celle trouvée dans [77], mais d'autres variantes existent.

Theorem 2.1.1 (Théorème du représentant). *Soit $\Omega : [0, +\infty[\rightarrow \mathbb{R}$ une fonction strictement croissante, $\mathcal{X}$ un ensemble, $c : (\mathcal{X} \times \mathcal{Y}^2)^m \rightarrow \mathbb{R} \cup \{+\infty\}$[1] une fonction de coût et $\mathcal{H}_k$ un espace de Hilbert à noyau reproduisant.*

Toute fonction f de $\mathcal{H}_k$ minimisant le risque régularisé

$$c((x_1, y_1, f(x_1)), \ldots, (x_m, y_m, f(x_m))) + \Omega(\|f\|_{\mathcal{H}_k})$$

avec $((x_1, y_1), \ldots, (x_m, y_m))$ dans $(\mathcal{X} \times \mathcal{Y})^m$ peut s'écrire sous la forme

$$f = \sum_{i=1}^{m} \alpha_i K(\cdot, x_i) \tag{2.5}$$

On notera que l'équation 2.5 ne représente pas toutes les fonctions appartenant à $\mathcal{H}_k$ puisqu'elle ne fait intervenir que les x_i présents dans la formule du risque.

Ce théorème permet d'écrire facilement la solution d'un problème régularisé en fonc-

[1] On suppose que $f(x)$ et y appartiennent au même espace $\mathcal{Y}$. Si cela n'est pas toujours vrai, par exemple dans le cas de la classification, on peut toujours s'y rapporter de manière simple.

tion des exemples d'apprentissage dès lors qu'on utilise une méthode à noyau. L'équation 2.5 montre que la seule inconnue est l'ensemble des α_i. Même si l'algorithme est non-paramétrique et que les éléments de $\mathcal{H}_K$ sont de dimension infinie, sa solution optimale se trouve et s'exprime facilement. Cette **astuce du noyau** suggère que le point de vue d'un modèle linéaire dans $\mathcal{H}_K$ ne sert que de justification aux méthodes à noyau sans avoir de réel intérêt pratique.

2.1.4 Processus gaussiens

Les **processus gaussiens** ont été abondamment expliqués dans [84] et dans [55]. Ils sont une généralisation de la distribution gaussienne multivariée aux fonctions f de $\mathbb{R}^d$ dans $\mathbb{R}$. Cette généralisation se fait en définissant une distribution sur les vecteurs $\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_p) \end{bmatrix}$ pour tout ensemble $(x_1, \ldots, x_p)$ de $\left(\mathbb{R}^d\right)^p$ de manière cohérente. Plus précisément, pour tout ensemble $(x_1, \ldots, x_p)$ de $\mathbb{R}^p$, nous avons

$$\left(\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_p) \end{bmatrix} \mid \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix} \right) \sim \mathcal{N} \left(\begin{bmatrix} \mu(x_1) \\ \vdots \\ \mu(x_p) \end{bmatrix}, C_{(x_1,\ldots,x_p)} \right) \tag{2.6}$$

où μ est la **fonction moyenne** et l'élément (i, j) de $C_{(x_1,\ldots,x_p)}$ est $K(x_i, x_j)$. K est appelé **fonction de covariance**. Un processus gaussien est donc une distribution sur les fonctions f.

2.2 Réseaux de neurones

Un réseau de neurones artificiel est une approximation informatique d'un réseau de neurones biologique. Il est composé d'unités de calcul élémentaires (les **neurones formels**) échangeant des signaux entre elles.

Un neurone formel (simplement appelé neurone) reçoit un ensemble de m signaux d'entrée, e_1 à e_m. L'entrée totale du neurone est la somme de ces entrées pondérées par

des poids w_1 à w_m ainsi que d'un biais w_0.

Le signal d'entrée E du neurone est donc égal à

$$E = \sum_{i=1}^{m} w_i e_i + w_0 \tag{2.7}$$

Le neurone transforme ensuite cette entrée en appliquant une fonction g, appelée **fonction de transfert**, sur cette entrée. Le résultat est le signal de sortie de ce neurone :

$$S = g(E) = g\left(\sum_{i=1}^{m} w_i e_i + w_0\right) \tag{2.8}$$

Les algorithmes d'apprentissage pour les réseaux de neurones artificiels (simplement appelés « réseaux de neurones ») ont été développés pour la première fois dans [69] sous le terme **perceptrons**. Les neurones utilisés dans ces réseaux étaient ceux définis dans [58] utilisant l'échelon de Heaviside comme fonction de transfert.

Toutefois, l'optimisation de tels réseaux était ardue, et ce n'est que dans les années 80, lorsque Rumelhart et al. [71] découvrirent la rétropropagation du gradient, que la popularité des réseaux de neurones grandit. Ils ont depuis lors été utilisés dans toutes sortes de problèmes, de la reconnaissance de caractères (par exemple dans [51]) au traitement du langage (par exemple [12]) en passant par la chimie.

Au fil des ans, des architectures diverses et variées de réseaux de neurones apparurent. On peut citer en particulier les réseaux à convolution [51], les réseaux récurrents, les réseaux à propagation avant, les machines de Boltzmann et les machines de Boltzmann restreintes [43].

Les sections suivantes seront exclusivement consacrées aux réseaux à propagation avant ; les machines de Boltzmann restreintes seront quant à elles traitées dans le chapitre 9.

2.2.1 Les réseaux de neurones à propagation avant

Les réseaux de neurones à propagation avant (que l'on appellera simplement « réseaux de neurones ») sont une famille de fonctions linéaires et non-linéaires de $\mathbb{R}^d$ dans $\mathbb{R}^k$. Ils sont composés d'un nombre arbitraire de neurones et la connexion entre deux neurones est dirigée.

Dans un réseau de neurones quelconque, toutes les paires de neurones peuvent être connectées. Dans un réseau à propagation avant, les neurones sont répartis en couches ordonnées et les connexions sont dirigées d'une couche inférieure vers une couche supérieure et chaque couche effectue une transformation (possiblement non-linéaire) de la sortie des couches précédentes. Tout au long de cette thèse, nous allons considérer un cas spécial de réseaux à propagation avant où chaque couche effectue une transformation de la sortie de la couche directement inférieure (et non de toutes les couches inférieures, comme dans le cas général). Cette architecture simplifiée étant largement prédominante dans la communauté et la seule étudiée dans cette thèse, nous les appellerons simplement de tels réseaux **réseaux de neurones** sans risque de confusion.

L'entrée d'un réseau de neurones est un vecteur x dans $\mathbb{R}^d$. La sortie de chaque unité est une transformation (potentiellement non-linéaire) des sorties des unités de la couche inférieure ainsi que des poids des connexions entre les deux couches (voir eq. 2.8). Les paramètres d'un réseau de neurones sont les poids associés aux connexions entre les neurones ainsi que les biais associés à chaque neurone. Dans le réseau de la figure 2.1 a donc comme paramètres $(w_1, \ldots, w_n, c_1, \ldots, c_n, a_1, \ldots, a_n, b)$. On appellera également **poids d'entrée** les poids des connexions entre la couche d'entrée et la première couche cachée, **biais d'entrée** les biais des neurones de la première couche cachée, **poids de sortie** les poids des connexions entre la dernière couche cachée et la couche de sortie et **biais de sortie** les biais des neurones de la couche de sortie.

L'optimisation de la transformation effectuée par une couche d'un réseau de neurones se fait par l'apprentissage de ses poids et biais associés.

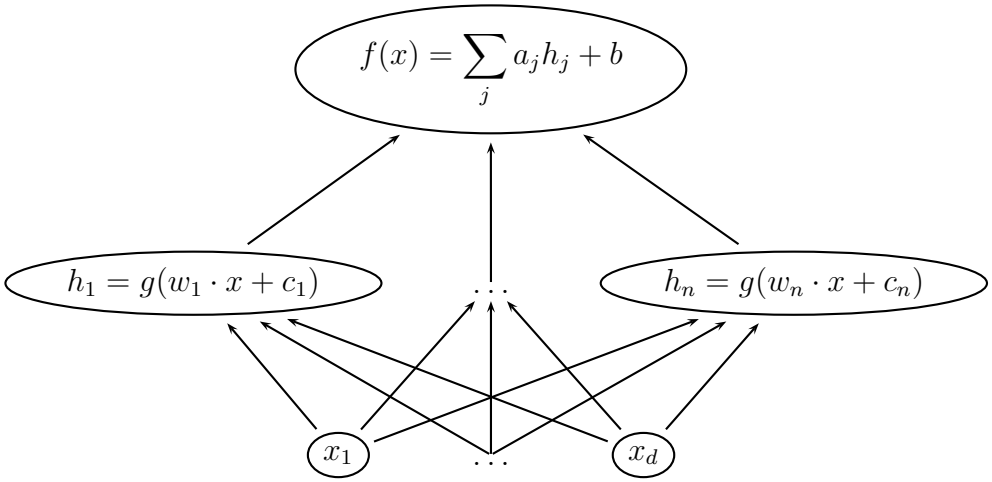


Figure 2.1 – Architecture d'un réseau de neurones à propagation avant à une couche cachée et sans fonction de sortie.

2.2.2 Fonction de transfert

Le choix de la fonction de transfert g est critique dans la modélisation d'un réseau de neurones. La classe de fonctions la plus fréquemment utilisée est la classe des **fonctions écrasantes**. g est une fonction écrasante si :

1. g est une fonction de $\mathbb{R}$ dans $[0, 1]$
2. g est croissante
3. $\lim_{x \to -\infty} g(x) = 0$
4. $\lim_{x \to +\infty} g(x) = 1$.

La fonction écrasante la plus populaire est la **fonction logistique** définie par $g(x) = \frac{1}{1+\exp(-x)}$. On pourra également remarquer que pour toute distribution p, la fonction cumulative de p est une fonction écrasante.

Une classe de fonctions plus large est celle des **fonctions sigmoïdales**. La condition 2 est la même que pour les fonctions écrasantes mais les bornes sont remplacées par deux réels A et B, $A < B$. On peut trouver parmi les fonctions sigmoïdales la fonction logistique, la **tangente hyperbolique** (notée $\tanh$) et l'**arctangente** (notée $\arctan$ ou $\tan^{-1}$).

Pour certains auteurs, les fonctions écrasantes et sigmoïdales doivent être **continues** ou même **différentiables**. Bien que nous n'imposions pas ces restrictions, les fonctions utilisées dans ce travail les respecteront.

2.2.3 Terminologie

Pour éviter toute confusion à la lecture de ce document, nous allons définir ici les différentes terminologies utilisées pour décrire les réseaux de neurones :

- la couche correspondant aux vecteurs d'entrée n'est généralement pas comptée lorsqu'on décrit un réseau de neurones. On considèrera donc que le réseau de neurones de la figure 2.1 n'a que deux couches.
- un réseau de neurones à p couches contient $p - 1$ couches cachées. On pourra donc décrire un réseau de neurones en utilisant son nombre total de couches ou son nombre de couches cachées. Le réseau de la figure 2.1 est donc à la fois un réseau de neurones à deux couches et un réseau de neurones à une couche cachée.

Dans cette thèse, par souci de cohérence, nous décrirons toujours un réseau de neurones en utilisant son nombres de couches cachées.

2.2.3.1 Les réseaux de neurones et la nécessité des couches cachées

Un réseau de neurones sans couche cachée, c'est-à-dire dans lequel les entrées sont directement connectées aux sorties, ne peut discriminer les entrées que de façon linéaire. En effet, la sortie d'un réseau de neurones sans couche cachée et avec un seul neurone de sortie peut s'écrire sous la forme :

$$f(x) = g(b + w \cdot x) \tag{2.9}$$

où b est le biais du neurone de sortie et w le vecteur de poids reliant les neurones d'entrée au neurone de sortie. Dans le cas de la classification, la surface de décision $f(x) = c$ peut donc être réécrite sous la forme $w \cdot x + b = g^{-1}(c)$ (si l'on suppose que c est dans l'image de g et que g est une injection), qui est un hyperplan. Pour cette raison, des fonctions simples comme le « ou exclusif » ne peuvent être représentées. Dans le cas de

la régression, s'il n'y a pas de fonction de sortie (le g de la formule 2.9), alors la sortie est également une fonction affine de l'entrée.

Cela justifie l'utilisation de couches cachées dans les réseaux de neurones classiques. Au vu des limitations des réseaux de neurones sans couches cachées, deux questions se posent alors :

1. Quelles sont les limitations des réseaux de neurones avec k couches cachées ?
2. Si k_0 couches cachées sont suffisantes pour modéliser n'importe quelle fonction, est-il utile d'en avoir plus ?

2.2.3.2 Théorème de Hornik

Il existe une réponse très simple à la première question qui a été formulée et prouvée par Hornik et al. [44] :

Theorem 2.2.1. *Les réseaux de neurones multicouches à propagation avant avec une couche cachée contenant suffisamment de neurones cachés et une fonction de transfert écrasante arbitraire sont capable d'approcher, avec une précision arbitraire, n'importe quelle fonction Borel-mesurable d'un espace de dimension finie dans un espace de dimension finie. Dans ce sens, les réseaux de neurones multicouches à propagation avant sont une classe d'approximateurs universels.*

Les réseaux de neurones classiques avec une ou plusieurs couches cachées peuvent donc approcher arbitrairement bien toute fonction continue (le théorème prouve cette propriété pour la classe des fonctions Borel-mesurables qui contient strictement la classe des fonctions continues).

2.2.3.3 La nécessité de modèles hiérarchiques

Bien qu'une seule couche cachée soit nécessaire pour modéliser n'importe quelle fonction continue, des réseaux de neurones avec plusieurs couches cachées sont également utilisés, réalisant souvent des performances à l'état de l'art sur de nombreux problèmes. Deux exemples de tels modèles (appelés **réseaux profonds**) sont les Deep

Belief Networks [39] et les réseaux à convolutions [51]. Les Deep Belief Networks seront étudiés au chapite 9.

2.2.4 Loi a priori sur les poids et a priori sur les fonctions

La section 1.5 mentionnait le lien entre la présence d'un terme de régularisation et les contraintes de simplicité imposées à la fonction apprise par notre algorithme. Subséquemment, les sections 1.6.3.4 et 1.6.3.3 mettaient en évidence le lien entre le terme de régularisation et l'a priori sur les paramètres d'un modèle.

Cela suggère qu'il existe un lien fort entre l'a priori sur les paramètres d'un réseau de neurones et l'a priori sur la fonction engendrée par ce réseau. Toutefois, ce lien n'est pas évident et [60] a étudié le cas des réseaux de neurones avec une couche cachée et une sortie linéaire. Tous les paramètres du neurones (poids d'entrée et de sortie, biais d'entrée et de sortie) ont un a priori gaussien.

Pour clarifier le propos, il est nécessaire de rappeler qu'une architecture de réseau de neurones définit une équivalence entre un jeu de paramètres et une fonction (celle induite par un réseau de neurones possédant ces paramètres). Les sections suivantes étudient un réseau de neurones avec un seul neurone d'entrée et un seul neurone de sortie. La fonction induite par ce réseau sera donc de $\mathbb{R}$ dans $\mathbb{R}$.

2.2.4.1 Loi a priori sur les poids et les biais de sortie

Il y a peu à dire sur le lien entre l'a priori sur les poids de sortie et l'a priori sur la fonction induite. En effet, celui-là contrôle simplement l'amplitude de la sortie. Toutefois, il est important de remarquer que, pour maintenir l'amplitude de la sortie constante lorsque le nombre d'unités cachées tend vers l'infini, la variance de l'a priori gaussien sur les poids de sortie doit varier selon la racine carrée de l'inverse du nombre d'unités cachées et la variance de l'a priori sur le biais de sortie doit rester inchangée [60].

2.2.4.2 A priori sur les poids et les biais d'entrée

L'a priori sur les poids d'entrée a une influence bien différente sur la fonction induite par le réseau de neurones. En effet, [60] montra qu'un a priori fort (c'est-à-dire avec une faible variance) sur les poids d'entrée engendrait des fonctions très lisses tandis qu'un a priori faible (permettant aux poids de prendre des valeurs très élévées) donnait lieu à des fonctions accidentées (voir figure 2.2).

2.2.4.3 Résumé et exemples

Le lien entre l'a priori sur les paramètres d'un réseau de neurones à une couche cachée avec une sortie linéaire et la fonction induite par ce réseau peuvent donc être résumés ainsi :

1. l'a priori sur les poids de sortie contrôle l'amplitude générale de la fonction induite. Un a priori fort entraînera des valeurs de sortie proches de 0 tandis qu'un a priori faible entraînera de valeurs de sorties élevées
2. l'a priori sur les poids d'entrée contrôle la rugosité de la fonction induite. Un a priori fort entraînera des fonctions lisses tandis qu'un a priori faible entraînera des fonctions accidentées.

La figure 2.2 présente des fonctions induites par un réseau de neurones avec un neurone d'entrée, 10000 neurones cachés et un neurone de sortie dont les paramètres ont été tirés selon une loi gaussienne.

- L'a priori sur les poids et le biais de sortie est une gaussienne de variance fixe.
- L'a priori sur les poids et biais d'entrée est une gaussienne de variances $\sigma = 1, 5, 20$ and 100.

Chaque graphique contient quatre fonctions induites pour chaque valeur de σ.

2.2.5 Liens entre les machines à noyau et les réseaux de neurones

Neal [60] prouva qu'un réseau de neurones à une couche cachée dont le nombre de neurones cachés tend vers l'infini converge vers un processus gaussien dont la fonction

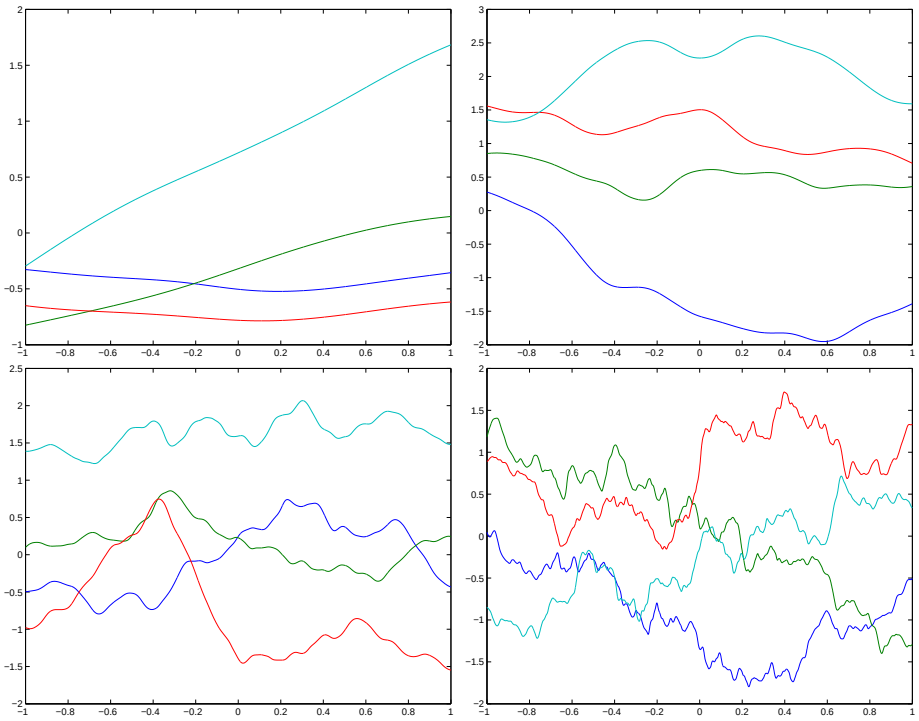


Figure 2.2 – En haut à gauche : $\sigma = 1$. En haut à droite : $\sigma = 5$. En bas à gauche : $\sigma = 20$. En bas à droite : $\sigma = 100$.

de covariance dépend de la fonction de transfert du réseau de neurones. Toutefois, il ne calcula aucune de ces fonctions de covariance explicitement. Williams [83] quant à lui calcula analytiquement les fonctions de covariance associées à plusieurs fonctions de transfert couramment utilisées.

2.3 Descente de gradient

Comme mentionné à la section 1.4.2, l'entraînement d'un algorithme d'apprentissage consiste en la minimisation d'un coût. La **descente de gradient** [20, p.240] effectue cette minimisation en calculant le gradient de ce coût par rapport aux paramètres de l'algorithme. Une légère modification des paramètres dans la direction opposée à ce gradient assurant une diminution du coût, la descente de gradient consiste en des mises

à jour successives des paramètres de la forme

$$\theta_{t+1} = \theta_t - \varepsilon_t \frac{\partial \mathcal{L}}{\partial \theta} \tag{2.10}$$

où ε est le taux d'apprentissage. Une diminution progressive de ce taux [22, 23] assure la convergence vers un minimum **local**.

Les méthodes de second ordre utilisent une formule de mise à jour impliquant une matrice M :

$$\theta_{t+1} = \theta_t - \varepsilon_t M \frac{\partial \mathcal{L}}{\partial \theta} \tag{2.11}$$

Les matrices M les plus fréquemment utilisées sont l'inverse de la hessienne (méthode de **Gauss-Newton**) ou la matrice de covariance des gradients (gradient naturel).

L'apprentissage par descente de gradient pour les réseaux de neurones a été rendu possible grâce à la découverte de la rétropropagation du gradient [52, 71] qui permet de calculer aisément le gradient du coût par rapport aux paramètres d'une couche du réseau en fonction du gradient du coût par rapport aux paramètres de la couche supérieure.

CHAPITRE 3

PRÉSENTATION DU PREMIER ARTICLE

3.1 Détails de l'article

The Curse of Highly Variable Functions for Local Kernel Machines
Y. Bengio, O. Delalleau et N. Le Roux
Publié dans *Advances in Neural Information Processing Systems 18, MIT Press, Cambridge, MA* en 2006.

3.2 Contexte

Les méthodes à noyau ont joui d'une grande popularité depuis l'introduction des machines à vecteurs de support dans les années 90 [21, 29, 75]. Cette popularité s'explique par deux facteurs :

- la convexité de leur optimisation
- leur capacité à modéliser n'importe quelle fonction (propriété d'approximation universelle).

Malheureusement, ces propriétés désirables ont un coût. Ainsi, toutes les méthodes à noyau prédéfinissent une mesure de similarité (implicitement déterminée par la fonction à noyau choisie, voir section 2.1.1) qu'ils utilisent pour comparer les exemples de test avec ceux d'apprentissage. Ce sont donc des « détecteurs de motif » améliorés qui ne font que chercher dans une base de données (l'ensemble d'entraînement) les exemples similaires (selon la mesure définie) à celui qui est testé.

Il serait toutefois inapproprié de négliger les méthodes à noyau pour cette raison. En effet, cette limitation ne les empêche pas d'obtenir des résultats plus qu'honorables sur de nombreux problèmes. Toutefois, dès lors que le problème possède de nombreux degrés de variation, le nombre d'exemples d'apprentissage nécessaires pour que la détection de motifs soit efficace devient prohibitif.

3.3 Commentaires

Ce sont ces problèmes que l'article présenté ici formalise de plusieurs façons :

- l'erreur de généralisation de telles méthodes peut converger aussi lentement que $n^{-\frac{4}{4+d}}$ (avec n le nombre d'exemples et d la dimension de la variété à laquelle ils appartiennent)
- le nombre de gaussiennes (de même variance) nécessaires pour apprendre une surface de décision augmente linéairement avec le nombre de régions de même signe
- le plan tangent à la variété apprise par une méthode à noyau utilisant un noyau local n'est défini que comme une combinaison linéaire d'un faible nombre de points voisins.

Cet article permet donc de mieux comprendre les raisons des faiblesses inhérentes aux méthodes à noyau couramment utilisées ainsi que de démontrer la nécessité de l'extraction de concepts dès lors que la dimensionnalité des données devient importante et que ces dernières n'occupent qu'une partie extrêmement réduite de l'espace total.

Toutefois, notre argumentaire a ses limites. Nous nous sommes limités aux méthodes utilisant des noyaux locaux, excluant notamment le noyau polynomial. De plus, le théorème 4.5.1 se limite à des ensembles de gaussiennes de même variance, ce qui n'est pas toujours le cas. Cependant, ces cas simplifiés sont ceux fréquemment utilisés dans la communauté, accréditant les arguments avancés.

CHAPITRE 4

THE CURSE OF HIGHLY VARIABLE FUNCTIONS FOR LOCAL KERNEL MACHINES

4.1 Abstract

We present a series of theoretical arguments supporting the claim that a large class of modern learning algorithms based on local kernels are sensitive to the curse of dimensionality. These include local manifold learning algorithms such as Isomap and LLE, support vector classifiers with Gaussian or other local kernels, and graph-based semi-supervised learning algorithms using a local similarity function. These algorithms are shown to be local in the sense that crucial properties of the learned function at x depend mostly on the neighbors of x in the training set. This makes them sensitive to the curse of dimensionality, well studied for classical non-parametric statistical learning. There is a large class of data distributions for which non-local solutions could be expressed compactly and potentially be learned with few examples, but which will require a large number of local bases and therefore a large number of training examples when using a local learning algorithm.

4.2 Introduction

A very large fraction of the recent work in statistical machine learning has been focused on so-called kernel machines, which are non-parametric learning algorithms in which the learned function is expressed in terms of a linear combination of kernel functions applied on the training examples :

$$f(x) = b + \sum_{i=1}^{n} \alpha_i K(x, x_i) \tag{4.1}$$

where optionally a bias term b is added, $D = \{x_1, \ldots, x_n\}$ are training examples (without the labels, in the case of supervised learning), the α_i's are scalars chosen by the

learning algorithm using labels $\{y_1, \ldots, y_n\}$, and $K(\cdot, \cdot)$ is the kernel function, a symmetric function (generally expected to be positive definite). We refer to the $n \times n$ matrix M with entries $M_{ij} = K(x_i, x_j)$ as the Gram matrix. A typical kernel function is the Gaussian kernel,

$$K(u, v) = e^{-\frac{1}{\sigma^2}||u-v||^2}, \tag{4.2}$$

with hyper-parameter σ (the width) controlling how local the kernel is. The Gaussian kernel is part of a larger family of kernels which we call local kernels and discuss in section 4.6.

Examples of kernel-based nonlinear manifold learning algorithms include Locally Linear Embedding (LLE) [70], Isomap [79], kernel PCA (kPCA) [76], Laplacian Eigenmaps [8], and Manifold Charting [24]. This framework also includes spectral clustering algorithms (see [81] for more references). For these algorithms, f represents the embedding function, which maps an example x to one of the coordinates of its low-dimensional representation embedding (and a separate set of α_i's is used for each dimension of the embedding). As shown in [11], the α_i correspond to entries in an eigenvector of the Gram matrix, divided by the corresponding eigenvalue (following the Nyström formula [6, 63, 85]). Note that these algorithms employ a special kind of kernel which is data-dependent [11], but the arguments presented in sections 4.6, 4.7 and 4.9 also apply.

Among the statistical classifiers most widely studied in the recent years is the Support Vector Machine (SVM) [21, 29, 75], in which f above is the discriminant function of a binary classifier, i.e. the decision function is given by the sign of f. Since this is a supervised learning algorithm, the training data includes a set of +1 or -1 class labels $\{y_1, \ldots, y_n\}$ for training examples. Results in section 4.5 apply to SVMs and other kernel machines with Gaussian kernels, and a more general conjecture is developed in section 4.9 based on the *local-derivative* notion introduced in section 4.7.

Another class of kernel algorithms that is discussed here are the non-parametric graph-based semi-supervised algorithms of the type described in recently proposed papers [7, 30, 89, 90]. They can be intuitively understood as performing some kind of smoothing or label propagation on the empirical graph defined by the examples (nodes)

and a similarity function (e.g. a kernel) between pairs of examples. The results in section 4.8 are specifically tailored to such algorithms, and show that the number of required labeled examples grows linearly with a measure of the amount of local variation of the predictor required to reach a given error level.

The basic ideas behind the arguments presented in this paper are simple. One class of arguments relies on showing that some important property of $f(x)$ is mostly determined by the neighbors of x in the training set. If one imagines tiling the space with such neighborhoods, the required number of neighborhoods (hence of training examples) could grow exponentially with the dimensionality of the data (or of the manifold on which they live). One issue in this respect is the size of these neighborhoods, and we get inspiration from the classical bias-variance trade-off argument for classical non-parametric models : if we make the regions smaller, bias is reduced (more complex functions can be represented) but variance is increased (not enough data are used to determine the value of $f(x)$ around x, so $f(x)$ becomes less stable).

Another class of arguments considers "apparently complicated" target functions, in the sense that they vary a lot across space, although there might exist simple and compact representations for them that could not be discovered using a purely local representation (eq. 4.1). For these arguments one attempts to find lower bounds on the number of examples required in order to learn the target function. These arguments show that whatever the method used to estimate the α_i's, one must have a large number of them in order to approximate the function at a given set of points (larger than the size of the training set), i.e. in order to get meaningful generalization.

In general, what one should keep in mind is that what matters is not the dimension of the data or of the manifold near which they are found, but rather the "apparent complexity" of the function that we are trying to learn (or one that would yield an acceptable error level). By "apparent complexity" we mean a measure of the number of "variations" of that function. One way to formalize this notion, used in Proposition 4.8.1, is the number of regions with constant sign of the predictor.

4.3 The Curse of Dimensionality for Classical Non-Parametric Models

The **curse of dimensionality** has been coined by Bellman [9] 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.

4.3.1 The Bias-Variance Dilemma

This problem has often been studied by considering the classical bias-variance analyses of statistical estimators. To keep the analysis simple, one usually only computes the conditional bias and conditional variance, in the case of supervised learning, i.e. given the x_i's of the training set, and integrating only over the y_i's. **Bias** of the estimator is the expected difference between the estimator and the target function, while **variance** of the estimator is the expected squared difference between the estimator and its expected value (in both cases the expectations are over the training set or over the y_i's only). The expected mean squared error of the estimator can be shown to be equal to the sum of variance and squared bias (plus the irreducible component of variation of Y given X, i.e. noise).

Two classical non-parametric estimators are worth discussing here. The first one is the k-nearest-neighbors estimator. It can be cast as a kernel machine (eq. 4.1) when K is allowed to be data-dependent : $\alpha_i = y_i$ and $K(x, x_i)$ is $1/k$ if x_i is one of the k nearest neigbors of x in D, 0 otherwise. The second one is the Nadaraya-Watson estimator (also known as the Parzen windows estimator), in which again $\alpha_i = y_i$ and $K(x, x_i)$ is a normalized kernel (i.e. also data-dependent), e.g.

$$K(x, x_i) = \frac{\mathcal{K}(x, x_i)}{\sum_{j=1}^{n} \mathcal{K}(x, x_j)}.$$

Since both estimators have the same form, much of the analysis can be shared. Let t be the target function that we are trying to learn, with $E[Y|X = x] = t(x)$. The

conditional bias is simply

$$\text{conditional bias}(x) = E[Y|X = x] - \sum_{i=1}^{n} E[Y|X = x_i]K(x, x_i). \qquad (4.3)$$

Clearly, the more local the kernel (i.e. $K(x, x_i)$ is nearly 0 except for x_i very close to x), the smaller the bias (note that we consider here kernels K such that $\sum_i K(x, x_i) = 1$). Assuming that $Var[y_i|x_i] = v$ does not depend on x_i the conditional variance of the estimator is

$$\text{conditional variance}(x) = v \sum_{i=1}^{n} K(x, x_i)^2.$$

For example, with the k-nearest neighbors estimator,

$$\text{conditional variance}(x) = \frac{v}{k}.$$

Clearly, the way to reduce variance is to decrease the kernel's locality (e.g. increase σ in a Gaussian kernel or k in a k-nearest neighbors estimator), which increases the effective number of examples used to obtain a prediction at x. But this also increases bias, by making the prediction smoother (possibly too smooth). Since total error involves the sum of squared bias and variance, one should choose the kernel hyper-parameter (e.g. k or σ) to strike the best balance (hence the *bias-variance dilemma* [35]).

4.3.2 Dimensionality and Rate of Convergence

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. Considering the above simplified exposition on bias and variance, 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,

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

(equal to k for the k-nearest neighbors estimator) 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 is sufficient for the bias to go to 0 (since $\frac{k}{n}$ is proportional to the volume of space around x which contains 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 [36] for a recent and easily accessible exposition (web version available).

The bias is due to smoothing the target function over the volume covered by the effective neighbors (consider eq. 4.3). 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).

When the input examples are not considered fixed the calculations of bias and variance are more complex, but similar conclusions are reached. For example, for a wide class of such kernel estimators, the unconditional variance and squared bias can be shown to be written as follows [36] :

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

(where the expected error is the sum of the variance and the squared bias of the estimator) with C_1 and C_2 not depending on n nor d. Hence an optimal bandwidth is chosen proportional to $n^{\frac{-1}{4+d}}$, and the resulting generalization error (not counting the noise) 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, say, a k-nearest neighbors 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 [79].

4.4 Summary of the results

In the following sections, we will see that :

- *In the case of a Gaussian kernel machine classifier, if there exists a line in $\mathbb{R}^d$ that intersects m times with the decision surface S (and is not included in S), one needs at least $\lceil \frac{m}{2} \rceil$ Gaussians (of same width) to learn S.*
- *At least 2^{d-1} examples are required to represent the d-bit parity (the function from $\{0,1\}^d$ to $\{-1,1\}$ which is 1 iff the sum of the bits is even), when using Gaussians with fixed σ centered on data points.*
- *Many graph-based semi-supervised learning algorithms cannot learn a number of regions with constant label higher than the number of labeled examples.*
- *When the test example x is far from all the x_i, a predictor given by eq. 4.1 with a local kernel either converges to a constant or a nearest neighbors classifier.* Neither of these are good in high dimension.
- *When using so-called local-derivative kernels, $\frac{\partial f(x)}{\partial x}$ is constrained to be approximately a linear combination of the vectors $(x - x_i)$ with x_i a near neighbor of x.* In high dimension (when the number of effective neighbors is significantly smaller than the dimension), this is a very strong constraint (either on the shape of the manifold or of the decision surface).
- *When there are examples with $||x - x_i||$ near σ (which is likely to happen for "good" values of σ, i.e. neither too small nor too big), with x on the decision*

surface, small changes in x w.r.t. σ yield only small changes in the normal vector of the decision surface. The above statement is one about bias : within a ball of radius σ, the decision surface is constrained to be smooth (small changes in x yield small changes in the shape of the surface).

– *In the case of a Gaussian kernel classifier, when σ increases the decision surface becomes subject to specific smoothness constraints.* More generally, we present an argument supporting the conjecture that any learning algorithm with a local property, such as the *local-derivative property* ($\frac{\partial f}{\partial x}$ depends mostly on examples in a ball around x) and local smoothness of the learned function (e.g., within that ball) will be subject to the curse of dimensionality.

These statements highlight some important limitations of kernel methods, that one must keep in mind when applying such learning algorithms.

4.5 Minimum Number of Bases Required for Complex Functions

4.5.1 Limitations of Learning with Gaussians

In [74] tight bounds are established for the number of zeros of univariate radial basis function networks under certain parameter conditions. In particular, it is shown that for a fixed Gaussian width σ, a function

$$f : \mathbb{R} \to \mathbb{R}$$

of the form

$$f(x) = b + \sum_{i=1}^{k} \alpha_i \exp\left(-\frac{\|x - x_i\|^2}{\sigma^2}\right) \tag{4.4}$$

cannot have more than $2k$ zeros (if there is at least one non-zero α_i). Consider now the more general case of a multivariate decision function $f : \mathbb{R}^d \to \mathbb{R}$ written as in eq. 4.4. For any $u \in \mathbb{R}^d$ and any $w \in \mathbb{R}^d$ such that $\|w\| = 1$, the function $g : \mathbb{R} \to \mathbb{R}$ defined by

$$g(t) = f(u + tw)$$

can be written in the form

$$g(t) = b + \sum_{i=1}^{k} \beta_i \exp\left(-\frac{|t-\alpha_i|^2}{\sigma^2}\right)$$

where $u+\alpha_i w$ is the projection of x_i on the line $D_{u,w} = \{u+\alpha w, \alpha \in \mathbb{R}\}$. Thanks to the above result from [74] we can then conclude that g has at most $2k$ zeros, i.e. that $D_{u,w}$ crosses the decision surface at most $2k$ times (as long as there is a non-zero coefficient in the resulting Gaussian expansion, otherwise g may be constant and equal to 0).

Corollary 4.5.1. *Let S be a decision surface in $\mathbb{R}^d$ to be learned with an affine combination of Gaussians with unique width as in eq. 4.4. If there exists a line in $\mathbb{R}^d$ that intersects m times with S (and is not included in S), then one needs at least $\lceil \frac{m}{2} \rceil$ Gaussians to learn S.*

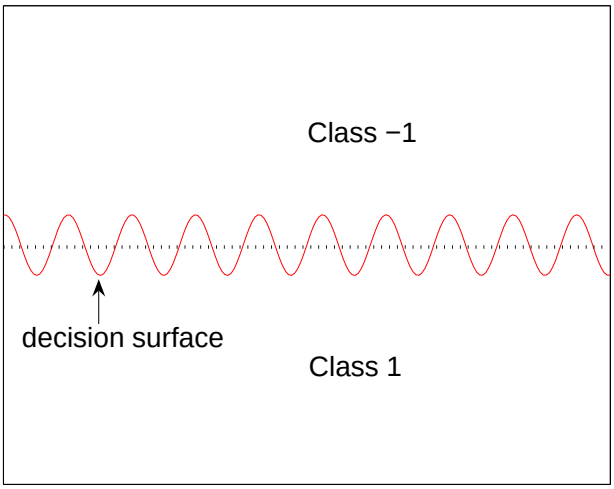


Figure 4.1 – The dotted line crosses the decision surface 19 times : one thus needs at least 10 Gaussians to learn it with an affine combination of Gaussians with same width.

Example 4.5.2. *Consider the decision surface shown in figure 4.1, which is a sinusoidal function. One may take advantage of the global regularity to learn it with few*

parameters (thus requiring few examples), but with an affine combination of Gaussians, corollary 4.5.1 implies one would need at least $\lceil \frac{m}{2} \rceil = 10$ *Gaussians. For more complex tasks in higher dimension, the "complexity" of the decision surface could quickly make learning impractical when using such a local kernel method.*

Remark 4.5.3. *Of course, one only seeks to approximate the decision surface* S*, and does not necessarily need to learn it perfectly : corollary 4.5.1 says nothing about the existence of an easier-to-learn decision surface approximating* S*. For instance, in the example of figure 4.1, the dotted line could turn out to be a good enough estimated decision surface if most samples were far from the true decision surface, and this line can be obtained with only two Gaussians.*

4.5.2 Learning the d-Bits Parity Function

The d-bits parity function is the function

$$\text{parity} : \ (b_1, \ldots, b_d) \in \{0,1\}^d \mapsto \begin{cases} 1 \text{ if } \sum_{i=1}^{d} b_i \text{ is even} \\ -1 \text{ otherwise} \end{cases}$$

We will show that learning this apparently simple function with Gaussians centered on points in $\{0,1\}^d$ is difficult, in the sense that it requires a number of Gaussians exponential in d (for a fixed Gaussian width). We will use the following notations :

$$\begin{aligned} X_d &= \{0,1\}^d = \{x_1, x_2, \ldots, x_{2^d}\} \\ H_d^0 &= \{(b_1, \ldots, b_d) \in X_d \mid b_d = 0\} \\ H_d^1 &= \{(b_1, \ldots, b_d) \in X_d \mid b_d = 1\} \\ K_\sigma(x,y) &= \exp\left(-\frac{\|x-y\|^2}{\sigma^2}\right) \end{aligned} \tag{4.5}$$

We say that a decision function $f : \mathbb{R}^d \to \mathbb{R}$ solves the parity problem if $\text{sign}(f(x_i)) = \text{parity}(x_i)$ for all i in $\{1, \ldots, 2^d\}$.

Lemma 4.5.4. *Let* $f(x) = \sum_{i=1}^{2^d} \alpha_i K_\sigma(x_i, x)$ *be a linear combination of Gaussians*

with same width σ centered on points $x_i \in X_d$. If f solves the parity problem, then $\alpha_i \text{parity}(x_i) > 0$ for all i.

Démonstration. We prove this lemma by induction on d. If $d = 1$ there are only 2 points. Obviously one Gaussian is not enough to classify correctly x_1 and x_2, so both α_1 and α_2 are non-zero, and $\alpha_1\alpha_2 < 0$ (otherwise f is of constant sign). Without loss of generality, assume $\text{parity}(x_1) = 1$ and $\text{parity}(x_2) = -1$. Then $f(x_1) > 0 > f(x_2)$, which implies $\alpha_1(1 - K_\sigma(x_1, x_2)) > \alpha_2(1 - K_\sigma(x_1, x_2))$ and $\alpha_1 > \alpha_2$ since $K_\sigma(x_1, x_2)) < 1$. Thus $\alpha_1 > 0$ and $\alpha_2 < 0$, i.e. $\alpha_i \text{parity}(x_i) > 0$ for $i \in \{1, 2\}$.
Suppose now lemma 4.5.4 is true for $d = d' - 1$, and consider the case $d = d'$. We denote by x_i^0 the points in H_d^0 and by α_i^0 their coefficient in the expansion of f (see eq. 4.5 for the definition of H_d^0). For $x_i^0 \in H_d^0$, we denote by $x_i^1 \in H_d^1$ its projection on H_d^1 (obtained by setting its last bit to 1), whose coefficient in f is α_i^1. For any $x \in H_d^0$ and $x_j^1 \in H_d^1$ we have :

$$\begin{aligned} K_\sigma(x_j^1, x) &= \exp\left(-\frac{\|x_j^1 - x\|^2}{\sigma^2}\right) \\ &= \exp\left(-\frac{1}{\sigma^2}\right) \exp\left(-\frac{\|x_j^0 - x\|^2}{\sigma^2}\right) \\ &= \gamma K_\sigma(x_j^0, x) \end{aligned}$$

where $\gamma = \exp\left(-\frac{1}{\sigma^2}\right) \in (0, 1)$. Thus $f(x)$ for $x \in H_d^0$ can be written

$$\begin{aligned} f(x) &= \sum_{x_i^0 \in H_d^0} \alpha_i^0 K_\sigma(x_i^0, x) + \sum_{x_j^1 \in H_d^1} \alpha_j^1 \gamma K_\sigma(x_j^0, x) \\ &= \sum_{x_i^0 \in H_d^0} \left(\alpha_i^0 + \gamma\alpha_i^1\right) K_\sigma(x_i^0, x) \end{aligned}$$

Since H_d^0 is isomorphic to X_{d-1}, the restriction of f to H_d^0 implicitely defines a function over X_{d-1} that solves the parity problem (because the last bit in H_d^0 is 0, the parity is not modified). Using our induction hypothesis, we have that for all $x_i^0 \in H_d^0$:

$$\left(\alpha_i^0 + \gamma\alpha_i^1\right) \text{parity}(x_i^0) > 0. \tag{4.6}$$

A similar reasoning can be made if we switch the roles of H_d^0 and H_d^1. One has to be careful that the parity is modified between H_d^1 and its mapping to X_{d-1} (because the last bit in H_d^1 is 1). Thus we obtain that the restriction of $(-f)$ to H_d^1 defines a function over X_{d-1} that solves the parity problem, and the induction hypothesis tells us that for all $x_j^1 \in H_d^1$:

$$\left(-\left(\alpha_j^1 + \gamma\alpha_j^0\right)\right)\left(-\text{parity}(x_j^1)\right) > 0. \tag{4.7}$$

and the two negative signs cancel out. Now consider any $x_i^0 \in H_d^0$ and its projection $x_i^1 \in H_d^1$. Without loss of generality, assume $\text{parity}(x_i^0) = 1$ (and thus $\text{parity}(x_i^1) = -1$). Using eq. 4.6 and 4.7 we obtain :

$$\begin{aligned} \alpha_i^0 + \gamma\alpha_i^1 &> 0 \\ \alpha_i^1 + \gamma\alpha_i^0 &< 0 \end{aligned}$$

It is obvious that for these two equations to be simultaneously verified, we need α_i^0 and α_i^1 to be non-zero and of opposite sign. Moreover, $\alpha_i^0 + \gamma\alpha_i^1 > 0 > \alpha_i^1 + \gamma\alpha_i^0 \Rightarrow \alpha_i^0 > \alpha_i^1$, which implies $\alpha_i^0 > 0$ and $\alpha_i^1 < 0$, i.e. $\alpha_i^0 \text{parity}(x_i^0) > 0$ and $\alpha_i^1 \text{parity}(x_i^1) > 0$. Since this is true for all x_i^0 in H_d^0, we have proved lemma 4.5.4. □

Theorem 4.5.5. *Let $f(x) = b + \sum_{i=1}^{2^d} \alpha_i K_\sigma(x_i, x)$ be an affine combination of Gaussians with same width σ centered on points $x_i \in X_d$. If f solves the parity problem, then there are at least 2^{d-1} non-zero coefficients α_i.*

Démonstration. We begin with two preliminary results. First, given any $x_i \in X_d$, the number of points in X_d that differ from x_i by exactly k bits is $\binom{d}{k}$. Thus,

$$\sum_{x_j \in X_d} K_\sigma(x_i, x_j) = \sum_{k=0}^{d} \binom{d}{k} \exp\left(-\frac{k^2}{\sigma^2}\right) = c_\sigma. \tag{4.8}$$

Second, it is possible to find a linear combination (i.e. without bias) of Gaussians g such

that $g(x_i) = f(x_i)$ for all $x_i \in X_d$. Indeed, let

$$g(x) = f(x) - b + \sum_{x_j \in X_d} \beta_j K_\sigma(x_j, x). \tag{4.9}$$

g verifies $g(x_i) = f(x_i)$ iff $\sum_{x_j \in X_d} \beta_j K_\sigma(x_j, x_i) = b$, i.e. the vector β satisfies the linear system $M_\sigma \beta = b\mathbf{1}$, where M_σ is the kernel matrix whose element (i, j) is $K_\sigma(x_i, x_j)$ and $\mathbf{1}$ is a vector of ones. It is well known that M_σ is invertible as long as the x_i are all different, which is the case here [59]. Thus $\beta = bM_\sigma^{-1}\mathbf{1}$ is the only solution to the system.

We now proceed to the proof of the theorem. By contradiction, suppose $f(x) = b + \sum_{i=1}^{2^d} \alpha_i K_\sigma(x_i, x)$ solves the parity problem with less than 2^{d-1} non-zero coefficients α_i. Then there exist two points x_s and x_t in X_d such that $\alpha_s = \alpha_t = 0$ and $\mathrm{parity}(x_s) = 1 = -\mathrm{parity}(x_t)$. Consider the function g defined as in eq. 4.9 with $\beta = bM_\sigma^{-1}\mathbf{1}$. Since $g(x_i) = f(x_i)$ for all $x_i \in X_d$, g solves the parity problem with a linear combination of Gaussians centered points in X_d. Thus, applying lemma 4.5.4, we have in particular that $\beta_s \mathrm{parity}(x_s) > 0$ and $\beta_t \mathrm{parity}(x_t) > 0$ (because $\alpha_s = \alpha_t = 0$), so that $\beta_s \beta_t < 0$. But, because of eq. 4.8, $M_\sigma \mathbf{1} = c_\sigma \mathbf{1}$, which means $\mathbf{1}$ is an eigenvector of M_σ with eigenvalue $c_\sigma > 0$. Consequently, $\mathbf{1}$ is also an eigenvector of M_σ^{-1} with eigenvalue $c_\sigma^{-1} > 0$, and $\beta = bM_\sigma^{-1}\mathbf{1} = bc_\sigma^{-1}\mathbf{1}$, which is in contradiction with $\beta_s \beta_t < 0$: f must have at least 2^{d-1} non-zero coefficients. □

Remark 4.5.6. *The bound in theorem 4.5.5 is tight, since it is possible to solve the parity problem with exactly 2^{d-1} Gaussians and a bias, for instance by using a negative bias and putting a positive weight on each example satisfying* $\mathrm{parity}(x) = 1$.

Remark 4.5.7. *When trained to learn the parity function, a SVM may learn a function that looks like the opposite of the parity on test points (while still performing optimally on training points). For instance, a SVM trained with 4000 unique points from X_{15} achieves a 90% error rate on 20000 different test samples (with $\sigma = 0.5$). This is because a bit less than 90% of these test samples are at distance 1 from their nearest neighbor in the training set. With this value of σ, the SVM output is similar to nearest neighbor*

classification, which is wrong when the nearest neighbor is at distance 1. However, this observation cannot in general be used to build a model for parity with Gaussians (by taking the opposite of the SVM output) : indeed, this approach would work here because the training samples are dense enough in X_{15}, but this will not be the case with fewer training data or in higher dimensions. Note that in all our experiments performed with parity datasets, a SVM had a 50% or more error rate on new points, which illustrates its inability to generalize for this problem.

Remark 4.5.8. *If the centers of the Gaussians are not restricted anymore to be points in X_d, it is possible to solve the parity problem with only $d+1$ Gaussians and no bias. Indeed, consider f defined by*

$$f(x) = \sum_{i=0}^{d} (-1)^i K_\sigma(y_i, x)$$

with

$$y_i = \frac{i}{d}\mathbf{1}.$$

For σ small enough, $\operatorname{sign}(f(y_i)) = (-1)^i$. In addition, for any $x \in X_d$, $f(x) = \gamma f(\hat{x})$, where $\hat{x}$ is the projection of x on the diagonal (the line spanned by $\mathbf{1}$) and $\gamma = \exp\left(-\frac{\|x-\hat{x}\|^2}{\sigma^2}\right) \in (0,1)$. Let $x = (b_1, \ldots, b_d)$: its projection $\hat{x}$ is given by

$$\hat{x} = (x \cdot \mathbf{1})\frac{\mathbf{1}}{\|\mathbf{1}\|^2} = \frac{1}{d}\left(\sum_i b_i\right)\mathbf{1} = y_{(\sum_i b_i)}$$

and therefore $\operatorname{sign}(f(x)) = \operatorname{sign}(f(\hat{x})) = \operatorname{sign}\left(f\left(y_{(\sum_i b_i)}\right)\right) = (-1)^{(\sum_i b_i)} = \operatorname{parity}(x)$: f solves the parity problem with a linear combination of $d+1$ Gaussians.

4.6 When a Test Example is Far from Training Examples

The argument presented in this section is mathematically trivial but nonetheless very powerful, especially in high-dimensional spaces. We consider here functions f as in eq. 4.1 where K is a **local kernel**, i.e. is such that for x a test point and x_i a training

point

$$\lim_{||x-x_i||\to\infty} K(x, x_i) \to c_i \tag{4.10}$$

where c_i is a constant that does not depend on x. For instance, this is true for the Gaussian kernel (eq. 4.2) with $c_i = 0$. This is also true for the data-dependent kernel obtained by the centering step in kernel PCA [76] when the original kernel is the Gaussian kernel, with

$$c_i = -\frac{1}{n}\sum_k K_\sigma(x_i, x_k) + \frac{1}{n^2}\sum_{k,l} K_\sigma(x_k, x_l).$$

Clearly, from inspection of equations 4.1 and 4.10, when x goes farther from the training set, i.e. when

$$d(D, x) = \min_i ||x - x_i||$$

goes to ∞, then $f(x) \to b + \sum_i \alpha_i c_i$. In the case of regression the prediction converges to a constant. In the case of manifold learning with $f(x)$ representing an embedding function, it means all points far from the training set are mapped to the same low-dimensional coordinates. Note that this convergence to a constant does not hold for non-local kernels such as those of Isomap or LLE [11].

In the case of classification (e.g. SVMs), there are two distinct cases. If $b + \sum_i \alpha_i c_i \neq 0$, $\mathrm{sign}(f(x))$ converges to a constant classifier. Otherwise, the result is less obvious and will depend on the kernel used. For instance, with a Gaussian kernel, it is easy to see that the classification will only depend on the nearest neighbor as long as x is not too close to a nearest neighbor classifier decision surface (which is more and more likely as x goes farther from the training set). In all cases we clearly get either a poor high-bias prediction (a constant), or a highly local one (the nearest neighbor rule, which is also likely to have a high bias when x is far from its nearest neighbor) that suffers from the curse of dimensionality. Note that when x is a high-dimensional vector, the nearest neighbor is not much closer than the other examples (the ratio of the distance between the nearest and the farthest converges to 1 [19]), hence it is not very informative. A random test point is therefore not unlikely to be relatively far from its nearest neighbor in the training set, compared to σ, when σ is chosen to be smaller than the scale of the

training data, so this situation is not a rare one for high-dimensional data.

4.7 Locality of the Estimator and its Tangent

In this section we consider how the derivative of $f(x)$ w.r.t x (i.e. its "shape") is influenced by the positions of training examples x_i. We say a kernel is **local-derivative** if its derivative can be written

$$\frac{\partial K(x, x_i)}{\partial x} = \sum_j \beta_{ij}(x - x_j)K'(x, x_j) \tag{4.11}$$

where K' is either a local kernel verifying eq. 4.10 with $c_i = 0$ or any kernel that is 0 when x_j is not a k nearest neighbor of x in the training set. From equations 4.1 and 4.11, we will see that $\partial f/\partial x$ is contained (possibly approximately) in the span of the vectors $(x - x_j)$ with x_j a neighbor of x (the notion of neighborhood being defined by k or by how fast K' converges to 0 when $\|x - x_j\|$ increases).

Examples of local-derivative kernels include the Gaussian kernel as well as the kernels for Isomap or LLE. In the case of the Gaussian kernel, we have

$$\frac{\partial K_\sigma(x, x_i)}{\partial x} = -\frac{2\alpha_i}{\sigma^2}(x - x_i)K_\sigma(x, x_i). \tag{4.12}$$

As shown in [17], in the case of Isomap we obtain a linear combination of vectors $(x-x_j)$ with x_j one of the k nearest neighbors of x, where k is an hyper-parameter for the Isomap kernel. Thus it verifies eq. 4.11 with $K'(x, x_j) = 0$ when x_j is not a k nearest neighbor of x. The same property holds for the LLE kernel, where $K_{LLE}(x, x_i)$ is the weight of x_i in the reconstruction of x by its k nearest neighbors [11]. Indeed, this weight is obtained by the following equation [73] :

$$K_{LLE}(x, x_i) = \frac{\sum_{j=1}^{k} G_{ij}^{-1}}{\sum_{l,m=1}^{k} G_{lm}^{-1}} \tag{4.13}$$

with G^{-1} the inverse of the local Gram matrix G

$$G_{lm} = (x - x_l) \cdot (x - x_m)$$

for all pairs (x_l, x_m) of k nearest neighbors of x in the training set. Because $G^{-1} = |G|^{-1}C^T$ with C the cofactor matrix of G, eq. 4.13 can be rewritten as

$$K_{LLE}(x, x_i) = \frac{\sum_j s_j \prod_{l,m} (G_{lm})^{t_{jlm}}}{\sum_j u_j \prod_{l,m} (G_{lm})^{v_{jlm}}}$$

and consequently, thanks to the usual derivation rules, its derivative is a linear combination of derivatives of terms of the form $(G_{lm})^t$. But

$$\begin{aligned} \frac{\partial (G_{lm})^t}{\partial x} &= \frac{\partial ((x - x_l) \cdot (x - x_m))^t}{\partial x} \\ &= t(G_{lm})^{t-1}(x - x_l + x - x_m) \end{aligned}$$

which implies that the derivative of $K_{LLE}(x, x_i)$ w.r.t x is in the span of the vectors $(x - x_j)$ with x_j a k nearest neighbor of x, as for Isomap : eq. 4.11 is verified.

From equations 4.1 and 4.11, we obtain that

$$\frac{\partial f(x)}{\partial x} = \sum_i \gamma_i K'(x, x_i)(x - x_i). \tag{4.14}$$

This equation helps us to understand how changes in x yield changes in $f(x)$ through the different training examples. In particular, because of the properties of K', **only the terms involving near neighbors of x have a significant influence on the result.** These neighbors are either defined by the hyper-parameter k or by how fast K' converges to 0 (i.e. σ for a Gaussian kernel, see section 4.7.2 for an example). Note that in the second case, we will also need the γ_i to be bounded in order to ensure such a locality property.

To better understand this, we will consider specifically the geometry of manifold learning and the geometry of decision surfaces.

4.7.1 Geometry of Tangent Planes

For manifold learning, with $f_k(x)$ the k-th embedding coordinate of x, these derivatives together (for $k = 1, 2, \ldots$) span the tangent plane of the manifold at x. Indeed, $\frac{\partial f_k(x)}{\partial x}$ is the direction of variation of x which corresponds to the largest increase in the k-th embedding coordinate. Inspection of eq. 4.14 shows that $\frac{\partial f_k(x)}{\partial x}$ is a linear combination of the vectors $(x - x_i)$, with x_i a near neighbor of x in the training set, since the other training examples contribute very little to the sum.

Consequently, the **tangent plane of the manifold is approximately in the span of the vectors** $(x - x_i)$ **with** x_i **a near neighbor of** x. If the number of data dimensions is large compared to the number of near neighbors, then this is a very strong constraint, irrespective of the learning method. Consider for example a distribution whose samples have an intrinsic dimension of about 100, but where the algorithm uses 5 significant neighbors. Constraining the tangent plane to be in the span of these 5 near neighbors is a very strong constraint in the local $\mathbb{R}^{100}$ subspace, which will result in a **high-variance estimator** for the manifold.

4.7.2 Geometry of Decision Surfaces

A similar argument can be made for the decision surfaces of kernel classifiers. Consider a point x on the decision surface, i.e. $f(x) = 0$. The normal of the decision surface is the vector $\frac{\partial f(x)}{\partial x}$ evaluated at x. From eq. 4.14 we see that **the decision surface normal vector is a linear combination of the vectors** $(x - x_i)$**, with** x_i **a near neighbor of** x. Again, if the intrinsic dimension is large compared to the number of near neighbors, then this is a very strong constraint, irrespective of the learning method. Like in the unsupervised case, this is likely to yield a **high-variance estimator** since only a few points determine a crucial property of the decision surface, i.e. its shape.

Note that in the above statements, depending on the kernel K', we may need to require the γ_i to be bounded. Take for instance the classical 1-norm soft margin SVM.

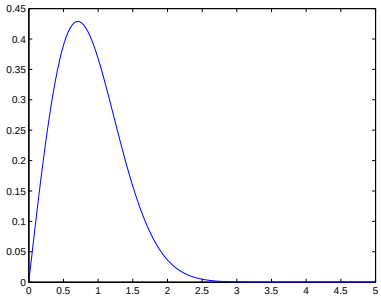


Figure 4.2 – Plot of ze^{-z^2}.

The decision surface normal vector can be obtained from eq. 4.1 and 4.12 :

$$\frac{\partial f(x)}{\partial x} = -\sum_i \frac{2\alpha_i}{\sigma} K_\sigma(x, x_i) \frac{\|x_i - x\|}{\sigma} \frac{(x_i - x)}{\|x_i - x\|} \tag{4.15}$$

i.e. as a linear combination of vectors of norm 1 whose weights depend notably on ze^{-z^2} with $z = \|x_i - x\|/\sigma$. This function is plotted in figure 4.2, showing that if there are training examples x_i such that $\|x_i - x\|/\sigma$ is less than 2, they will be the main contributors in eq. 4.15, as long as the coefficients α_i are bounded (which is the case for the 1-norm SVM). Consider a change in x small with respect to σ : this change does not affect much the direction nor the weight of these main contributors, which means the normal vector is almost unchanged. The bottom line is that **the decision surface is constrained to be smooth**, in the sense that the normal vector is almost constant in a ball whose radius is small with respect to σ. This very qualitative statement helps us understand why one usually needs a small σ to learn complex decision functions while smooth functions can be learned with a bigger σ.

In this specific example of the 1-norm soft margin SVM, it is important to note that we do not want training examples to be all close to x or all far from x (w.r.t. σ), i.e. σ should not take extreme values. Indeed, it is shown in [46] that when $\sigma \to 0$ or $\sigma \to \infty$ and the regularization parameter C is fixed, the SVM will assign the entire data space to the majority class (and when $\sigma \to \infty$ and $C \propto \sigma^2$ it converges to a linear SVM).

4.8 The Curse of Dimensionality for Local Non-Parametric Semi-Supervised Learning

In this section we focus on algorithms of the type described in recent papers [7, 30, 89, 90], which are graph-based non-parametric semi-supervised learning algorithms. Because the analysis is centered on the decision surface and eq. 4.1, it also applies to transductive SVMs and Gaussian processes when the kernel is either Gaussian or normalized Gaussian.

The graph-based algorithms we consider here can be seen as minimizing the following cost-function, as shown in [30] :

$$C(\hat{Y}) = \|\hat{Y}_l - Y_l\|^2 + \mu \hat{Y}^\top L \hat{Y} + \mu\epsilon\|\hat{Y}\|^2 \tag{4.16}$$

with $\hat{Y} = (\hat{y}_1, \ldots, \hat{y}_n)$ the estimated labels on both labeled and unlabeled data, and L the (un-normalized) graph Laplacian derived from a similarity function W between points such that $W_{ij} = W(x_i, x_j)$ corresponds to the weights of the edges in the graph. Here, $\hat{Y}_l = (\hat{y}_1, \ldots, \hat{y}_l)$ is the vector of estimated labels on the l labeled examples, whose known labels are given by $Y_l = (y_1, \ldots, y_l)$, and one may constrain $\hat{Y}_l = Y_l$ as in [90] by letting $\mu \to 0$.

Minimization of the cost criterion of eq. 4.16 can also be seen as a *label propagation* algorithm, i.e. labels are "spread" around labeled examples, with "around" being given by the structure of the graph. An intuitive view of label propagation suggests that a region of the manifold near 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 W. We define a region with constant label as a connected subset of the graph 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 W defines a

fully connected graph).

Proposition 4.8.1. *After running a label propagation algorithm minimizing the cost of eq. 4.16, the number of regions with constant estimated label is less than (or equal to) the number of labeled examples.*

Démonstration. 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 $x_{l+1}, \ldots, x_{l+q}$ the q samples in this region. The part of the cost of eq. 4.16 depending on their labels is

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

The second term is strictly positive, and because the region we consider is maximal (by definition) all samples x_j outside of the region such that $W_{ij} > 0$ verify $\hat{y}_j < 0$ (for x_i a sample in the region). Since all $\hat{y}_i$ are stricly positive for $i \in \{l+1, \ldots, l+q\}$, this means this second term can be stricly decreased by setting all $\hat{y}_i$ to 0 for $i \in \{l+1, \ldots, l+q\}$. 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 sparse 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 W is not sparse (e.g. Gaussian kernel), obtaining such a result is less obvious. However, there often exists a sparse approximation of 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 dense weight matrices, if the weighting function is local in the sense that it is close to zero when applied to a pair of examples far from each other.

4.9 General Curse of Dimensionality Argument

Can we obtain more general results that would apply to a broader class of learning algorithms, such as transductive SVMs and Gaussian processes with Gaussian kernel ? For this we must first establish the notion that if we choose the ball $\mathcal{N}(x)$ too large the predicted function is constrained to be too smooth. The following proposition supports this statement.

Proposition 4.9.1. *For the Gaussian kernel classifier, as σ increases and becomes large compared with the diameter of the data, in the smallest sphere containing the data the decision surface becomes linear if $\sum_i \alpha_i = 0$ (e.g. for SVMs), or else the normal vector of the decision surface becomes a linear combination of two sphere surface normal vectors, with each sphere centered on a weighted average of the examples of the corresponding class.*

Démonstration. The decision surface is the set of x such that

$$f(x) = \sum_i \alpha_i e^{-\frac{1}{2}||x-x_i||^2/\sigma^2} = 0.$$

The normal vector of the decision surface of the Gaussian classifier is therefore

$$\frac{\partial f(x)}{\partial x} = -\sum_i \alpha_i \frac{(x-x_i)}{\sigma^2} e^{-\frac{1}{2}||x-x_i||^2/\sigma^2}.$$

In the smallest sphere containing the data, $||x - x_i|| \leq \Delta$, Δ being the diameter of that sphere. As σ becomes large with respect to Δ, it becomes large with respect to all the distances $||x - x_i||$, and the factors $e^{-\frac{1}{2}||x-x_i||^2/\sigma^2}$ approach 1. For example for $\sigma = 4\Delta$, these factors are in the interval $[0.969, 1]$. Let us write $\beta_i(x) = \frac{\alpha_i}{\sigma^2} e^{-\frac{1}{2}||x-x_i||^2/\sigma^2}$, which approaches $\frac{\alpha_i}{\sigma^2}$. The normal vector can thus be decomposed in three terms

$$\frac{\partial f(x)}{\partial x} = -x \sum_i \beta_i(x) + \sum_{i:\alpha_i>0} \beta_i(x) x_i - \sum_{i:\alpha_i<0} (-\beta_i(x)) x_i. \tag{4.17}$$

Let us introduce two weighted averages, corresponding to each of the two classes. Define for $k = 1$ and $k = -1$

$$\begin{aligned} S_k(x) &= k \sum_{i:sign(\alpha_i)=k} \beta_i(x) \\ m_k(x) &= k \sum_{i:sign(\alpha_i)=k} \beta_i(x) x_i \\ \mu_k(x) &= \frac{m_k(x)}{S_k(x)} \end{aligned}$$

These are centers of mass for each of the two classes, with weights $|\beta_i(x)|$. The normal vector is thereby written as

$$\frac{\partial f(x)}{\partial x} = -x S_1(x) + x S_{-1}(x) + m_1(x) - m_{-1}(x)$$

If $\sum_i \alpha_i = 0$ (as in many SVMs), then $S_1(x) - S_{-1}(x)$ approaches 0, so that the first two terms above vanish and the dominant remainder is the difference vector $m_1 - m_{-1}$, which defines a linear decision surface whose normal vector is aligned with the difference of two weighted means (one per class).

Without the condition $\sum_i \alpha_i = 0$ we obtain a very specific quadratic decision surface. More precisely, the normal vector can be rewritten

$$\frac{\partial f(x)}{\partial x} = S_1(x)(\mu_1(x) - x) - S_{-1}(x)(\mu_{-1}(x) - x).$$

We can now interpret the above as follows. As σ increases, $\mu_k(x)$ approaches a fixed center and $S_k(x)$ a fixed scalar, and the decision surface normal vector is a linear combination of two vectors : the normal vector of the surface of the sphere centered at μ_1 and the normal vector of the surface of the sphere centered at μ_{-1}. □

We are now ready to put all these arguments together to argue that the curse of dimensionality plagues a large class of semi-supervised and supervised learning algorithms.

Conjecture 4.9.2. Consider a smoothness property S of discriminant functions f. Let P be a data-generating process from which training sets are sampled from a bounded region Ω. Suppose that P is such that, with N balls covering Ω with property S in each ball, one cannot build a classifier with generalization error level less than ϵ. For kernel machines with

1. the local-derivative property definining balls $\mathcal{N}(x)$ around a given test point x,
2. a smooth decision surface within any ball $\mathcal{N}(x)$ in the sense that smoothness property S is satisfied within $\mathcal{N}(x)$ (e.g. consider S shown for the Gaussian kernel in Proposition 4.9.1),

the number of training examples required for achieving error level ϵ grows linearly with N. N may grow exponentially with the dimension of a manifold near which the density concentrates, on the decision surface.

Supporting argument. The notion of locality provided by the *local-derivative* property allows us to define a ball $\mathcal{N}(x)$ around each test point x, containing neighbors that have a dominating influence on $\frac{\partial f(x)}{\partial x}$. The smoothness property S defined in Proposition 4.9.1 constrains the decision surface to be either linear (case of SVMs) or a particular quadratic form (the decision surface normal vector is a linear combination of two vectors defined by the center of mass of examples of each class). If P is as defined above, one must have at least N balls covering Ω, and let k be the smallest number such that one needs at least k examples in each ball to reach error level ϵ. The number of examples thus required is kN. To see that N can be exponential in some dimension, consider the maximum radius r of all these balls and the radius R of Ω. If the region of the decision surface in which

the density is non-negligible has intrinsic dimension d, then N could be as large as the number of radius r balls that can tile a d-dimensional manifold of radius R, which is at least $\left(\frac{R}{r}\right)^d$. This completes the argument. □

4.10 Conclusion

The central claim of this paper is that there are fundamental problems with non-parametric local approaches to learning, due to the curse of dimensionality. Even though learning algorithms such as SVMs behave better in this respect than classical algorithms such as k-nearest-neighbor classifiers (because they can do something between the k-nearest-neighbor and linear classifier), they also suffer from the curse of dimensionality, at least when the kernels are local.

We have presented a series of theoretical results, some obvious and others less, that illustrate this phenomenon. They highlight some intrinsic limitations of those local learning algorithms, that can make them fail when applied on high-dimensional problems where one has to look beyond what happens locally in order to overcome the curse of dimensionality. However, there is still work to do in order to obtain more general theoretical results that would show precisely how the generalisation error degrades when the (effective) dimension of the data grows, as can be done for classical non-parametric algorithms.

Acknowledgments

The authors would like to thank the following funding organizations for support : NSERC, MITACS, and the Canada Research Chairs. The authors are also grateful for the feedback and stimulating exchanges that helped to shape this paper, with Yann Le Cun and Léon Bottou.

CHAPITRE 5

PRÉSENTATION DU DEUXIÈME ARTICLE

5.1 Détails de l'article

Convex neural networks

Y. Bengio, N. Le Roux, P. Vincent, O. Delalleau et P. Marcotte

Publié dans *Advances in Neural Information Processing Systems 18*, MIT Press, Cambridge, MA. en 2006.

5.2 Contexte

Le chapitre précédent a permis de mettre en lumière certaines des limitations fondamentales des méthodes à noyau. Ces limitations viennent majoritairement du fait que les caractéristiques extraites des données sont prédéfinies en fonction du noyau et qu'elles ne sont pas apprises en tant que tel. L'apprentissage des méthodes à noyau consiste simplement à déterminer le poids de chaque exemple d'apprentissage pour obtenir une bonne fonction de classification.

Ces limitations encouragent naturellement à s'intéresser aux réseaux de neurones qui apprennent les caractéristiques à extraire des données. Toutefois, la non-convexité de leur optimisation a grandement affecté leur popularité dans la communauté de l'apprentissage machine. En effet, cela a souvent été perçu comme une source potentielle de mauvaises solutions, puisque l'entraînement ne parvenait qu'à trouver un minimum local de la fonction de coût et pas un minimum global[1].

Toutefois, l'amalgame entre localité du minimum et qualité de la solution n'a pas lieu d'être, comme nous allons le voir à la section suivante.

[1] Il existe certaines fonctions non convexes pour lesquelles la descente de gradient permet d'obtenir le minimum global, mais les fonctions de coût associées aux réseaux de neurones généraux n'en font pas partie.

5.2.1 Convexité

Une fonction f de $\mathbb{R}^n$ dans $\mathbb{R}^k$ est convexe si et seulement si

$$\forall x \in \mathbb{R}^n, \forall y \in \mathbb{R}^n, \forall \lambda \in [0,1], f\left(\lambda x + (1-\lambda)y\right) \leq \lambda f(x) + (1-\lambda) f(y) \qquad (5.1)$$

Une fonction convexe n'a qu'un seul minimum local qui est également un minimum global. Cette propriété assure l'unicité de la solution finale obtenue par descente de gradient, dispensant de la recherche d'un bon jeu de paramètres initial.

5.2.2 Fonction de coût et optimisation

Lors de la résolution d'un problème d'apprentissage, il est souvent impossible de minimiser l'erreur qui nous importe réellement. Cela peut être dû à une non-dérivabilité de cette erreur (comme dans le cas de l'erreur de classification) ou à un coût prohibitif de son calcul (comme par exemple la vraisemblance dans une Restricted Boltzmann Machine). On utilise alors dans ce cas différentes approximations pour résoudre le problème initial. Ces approximations peuvent être :

1. au niveau de la fonction de coût. On utilisera donc une fonction de coût simplifiée plus facilement optimisable (c'est-à-dire dérivable, la plupart du temps)
2. au niveau de l'architecture de l'algorithme. On utilisera alors des algorithmes avec une architecture particulière pour lesquelles le minimum de la fonction de coût est facile à trouver (que la fonction soit convexe ou que le minimum puisse être trouvé de façon analytique).

La qualité de la solution d'un algorithme dépend non seulement de la qualité de l'optimisation effectuée, mais également du judicieux de la fonction d'erreur et de l'architecture utilisées. Alors que les méthodes à noyau font le pari d'une optimisation réussie d'une architecture simplifiée, les réseaux de neurones gardent une architecture complexe dont le prix est une optimisation ardue.

5.3 Commentaires

Nous avons donc voulu montrer qu'il était possible de réaliser un algorithme possédant à la fois le pouvoir de généralisation des réseaux de neurones et la simplicité d'optimisation des méthodes à noyau. Il en résulte un algorithme similaire au boosting [33] qui sélectionne les unités cachées une par une, résolvant un problème de classification à chaque fois.

Ce problème de classification peut être résolu de manière exacte en temps exponentiel ou de manière approchée en temps linéaire, cette accélération se faisant au prix de la convexité. Alors que la méthode exacte réalise une meilleure performance sur l'ensemble d'entraînement (ce qui est normal puisqu'elle atteint le minimum global), la qualité de la solution obtenue (considérant l'erreur de test) est inférieure. Cela signifie que renoncer à la convexité de l'optimisation peut entraîner une meilleure erreur de généralisation. Ce résultat surprenant est analysé plus en détail par Collobert et al. [28].

CHAPITRE 6

CONVEX NEURAL NETWORKS

6.1 Abstract

Convexity has recently received a lot of attention in the machine learning community, and the lack of convexity has been seen as a major disadvantage of many learning algorithms, such as multi-layer artificial neural networks. We show that training multi-layer neural networks in which the number of hidden units is learned can be viewed as a convex optimization problem. This problem involves an infinite number of variables, but can be solved by incrementally inserting a hidden unit at a time, each time finding a linear classifier that minimizes a weighted sum of errors.

6.2 Introduction

The objective of this paper is not to present yet another learning algorithm, but rather to point to a previously unnoticed relation between multi-layer neural networks (NNs),Boosting [33] and convex optimization. Its main contributions concern the mathematical analysis of an algorithm that is similar to previously proposed incremental NNs, with L^1 regularization on the output weights. This analysis helps to understand the underlying convex optimization problem that one is trying to solve.

This paper was motivated by the unproven conjecture (based on anecdotal experience) that when the number of hidden units is “large”, the resulting average error is rather insensitive to the random initialization of the NN parameters. One way to justify this assertion is that to really stay stuck in a local minimum, one must have second derivatives positive simultaneously in all directions. When the number of hidden units is large, it seems implausible for none of them to offer a descent direction. Although this paper does not prove or disprove the above conjecture, in trying to do so we found an interesting **characterization of the optimization problem for NNs as a convex program** if the output loss function is convex in the NN output and if the output layer weights are

regularized by a convex penalty. More specifically, if the regularization is the L^1 norm of the output layer weights, then we show that a "reasonable" solution exists, involving a finite number of hidden units (no more than the number of examples, and in practice typically much less). We present a theoretical algorithm that is reminiscent of Column Generation [26], in which hidden neurons are inserted one at a time. Each insertion requires solving a weighted classification problem, very much like in Boosting [33] and in particular Gradient Boosting [34, 57].

Neural Networks, Gradient Boosting, and Column Generation

Denote $\tilde{x} \in \mathbb{R}^{d+1}$ the extension of vector $x \in \mathbb{R}^d$ with one element with value 1. What we call "Neural Network" (NN) here is a predictor for supervised learning of the form $\hat{y}(x) = \sum_{i=1}^{m} w_i h_i(x)$ where x is an input vector, $h_i(x)$ is obtained from a linear discriminant function $h_i(x) = s(v_i \cdot \tilde{x})$ with e.g. $s(a) = \text{sign}(a)$, or $s(a) = \tanh(a)$ or $s(a) = \frac{1}{1+e^{-a}}$. A learning algorithm must specify how to select m, the w_i's and the v_i's. The classical solution [71] involves (a) selecting a loss function $Q(\hat{y}, y)$ that specifies how to penalize for mismatches between $\hat{y}(x)$ and the observed y's (target output or target class), (b) optionally selecting a regularization penalty that favors "small" parameters, and (c) choosing a method to approximately minimize the sum of the losses on the training data $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ plus the regularization penalty. Note that in this formulation, an output non-linearity can still be used, by inserting it in the loss function Q. Examples of such loss functions are the quadratic loss $||\hat{y} - y||^2$, the hinge loss $\max(0, 1 - y\hat{y})$ (used in SVMs), the cross-entropy loss $-y \log \hat{y} - (1-y) \log(1-\hat{y})$ (used in logistic regression), and the exponential loss $e^{-y\hat{y}}$ (used in Boosting).

Gradient Boosting has been introduced in [34] and [57] as a non-parametric greedy-stagewise supervised learning algorithm in which one adds a function at a time to the current solution $\hat{y}(x)$, in a steepest-descent fashion, to form an additive model as above but with the functions h_i typically taken in other kinds of sets of functions, such as those obtained with decision trees. In a stagewise approach, when the $(m+1)$-th basis h_{m+1} is added, only w_{m+1} is optimized (by a line search), like in *matching pursuit* algorithms. Such a greedy-stagewise approach is also at the basis of Boosting algorithms [33], which is usually applied using decision trees as bases and Q the exponential loss. It may be

difficult to minimize exactly for w_{m+1} and h_{m+1} when the previous bases and weights are fixed, so [34] proposes to "follow the gradient" in function space, i.e., look for a base learner h_{m+1} that is best correlated with the gradient of the average loss on the $\hat{y}(x_i)$ (that would be the residue $\hat{y}(x_i) - y_i$ in the case of the square loss). The algorithm analyzed here also involves maximizing the correlation between Q' (the derivative of Q with respect to its first argument, evaluated on the training predictions) and the next basis h_{m+1}. However, we follow a "stepwise", less greedy, approach, in which all the output weights are optimized at each step, in order to obtain convergence guarantees.

Our approach adapts the Column Generation principle [26], a decomposition technique initially proposed for solving linear programs with many variables and few constraints. In this framework, active variables, or "columns", are only generated as they are required to decrease the objective. In several implementations, the column-generation subproblem is frequently a combinatorial problem for which efficient algorithms are available. In our case, the subproblem corresponds to determining an "optimal" linear classifier.

6.3 Core Ideas

Informally, consider the set $\mathcal{H}$ of all possible hidden unit functions (i.e., of all possible hidden unit weight vectors v_i). Imagine a NN that has all the elements in this set as hidden units. We might want to impose precision limitations on those weights to obtain either a countable or even a finite set. For such a NN, we only need to learn the output weights. If we end up with a finite number of non-zero output weights, we will have at the end an ordinary feedforward NN. This can be achieved by using a regularization penalty on the output weights that yields sparse solutions, such as the L^1 penalty. If in addition the loss function is convex in the output layer weights (which is the case of squared error, hinge loss, ϵ-tube regression loss, and logistic or softmax cross-entropy), then it is easy to show that the overall training criterion is convex in the parameters (which are now only the output weights). The only problem is that there are as many variables in this convex program as there are elements in the set $\mathcal{H}$, which may be very large (possibly infinite). However, we find that with L^1 regularization, a finite solution is

obtained, and that such a solution can be obtained by greedily inserting one hidden unit at a time. Furthermore, it is theoretically possible to check that the global optimum has been reached.

Definition 6.3.1. *Let $\mathcal{H}$ be a set of functions from an input space $\mathcal{X}$ to $\mathbb{R}$. Elements of $\mathcal{H}$ can be understood as "hidden units" in a NN. Let $\mathcal{W}$ be the Hilbert space of functions from $\mathcal{H}$ to $\mathbb{R}$, with an inner product denoted by $a \cdot b$ for $a, b \in \mathcal{W}$. An element of $\mathcal{W}$ can be understood as the output weights vector in a neural network. Let $h(x) : \mathcal{H} \to \mathbb{R}$ the function that maps any element h_i of $\mathcal{H}$ to $h_i(x)$. $h(x)$ can be understood as the vector of activations of hidden units when input x is observed. Let $w \in \mathcal{W}$ represent a* parameter *(the output weights). The NN prediction is denoted $\hat{y}(x) = w \cdot h(x)$. Let $Q : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a cost function convex in its first argument that takes a scalar prediction $\hat{y}(x)$ and a scalar target value y and returns a scalar cost. This is the cost to be minimized on example pair (x, y). Let $D = \{(x_i, y_i) : 1 \leq i \leq n\}$ a training set. Let $\Omega : \mathcal{W} \to \mathbb{R}$ be a convex regularization functional that penalizes for the choice of more "complex" parameters (e.g., $\Omega(w) = \lambda ||w||_1$ according to a 1-norm in $\mathcal{W}$, if $\mathcal{H}$ is countable). We define the* convex NN criterion $C(\mathcal{H}, Q, \Omega, D, w)$ *with parameter w as follows :*

$$C(\mathcal{H}, Q, \Omega, D, w) = \Omega(w) + \sum_{t=1}^{n} Q(w \cdot h(x_t), y_t). \tag{6.1}$$

The following is a trivial lemma, but it is conceptually very important as it is the basis for the rest of the analysis in this paper.

Lemma 6.3.2. *The convex NN cost $C(\mathcal{H}, Q, \Omega, D, w)$ is a convex function of w.*

Démonstration. $Q(w \cdot h(x_t), y_t)$ is convex in w and Ω is convex in w, by the above construction. C is additive in $Q(w \cdot h(x_t), y_t)$ and additive in Ω. Hence C is convex in w. □

Note that there are no constraints in this convex optimization program, so that at the global minimum all the partial derivatives of C with respect to elements of w cancel.

Let $|\mathcal{H}|$ be the cardinality of the set $\mathcal{H}$. If it is not finite, it is not obvious that an optimal solution can be achieved in finitely many iterations.

Lemma 6.3.2 says that training NNs from a very large class (with one or more hidden layer) can be seen as convex optimization problems, usually in a very high dimensional space, **as long as we allow the number of hidden units to be selected by the learning algorithm**. By choosing a regularizer that promotes **sparse** solutions, we obtain a solution that has a **finite** number of "active" hidden units (non-zero entries in the output weights vector w). This assertion is proven below, in theorem 6.4.1, for the case of the hinge loss.

However, even if the solution involves a finite number of active hidden units, the convex optimization problem could still be computationally intractable because of the large number of variables involved. One approach to this problem is to apply the principles already successfully embedded in Gradient Boosting, but more specifically in Column Generation (an optimization technique for very large scale linear programs), i.e., add one hidden unit at a time in an incremental fashion. The **important ingredient here is a way to know that we have reached the global optimum, thus not requiring to actually visit all the possible hidden units.** We show that this can be achieved as long as we can solve the sub-problem of finding a linear classifier that minimizes the weighted sum of classification errors. This can be done exactly only on low dimensional data sets but can be well approached using weighted linear SVMs, weighted logistic regression, or Perceptron-type algorithms.

Another idea (not followed up here) would be to consider first a smaller set $\mathcal{H}_1$, for which the convex problem can be solved in polynomial time, and whose solution can theoretically be selected as initialization for minimizing the criterion $C(\mathcal{H}_2, Q, \Omega, D, w)$, with $\mathcal{H}_1 \subset \mathcal{H}_2$, and where $\mathcal{H}_2$ may have infinite cardinality (countable or not). In this way we could show that we can find a solution whose cost satisfies $C(\mathcal{H}_2, Q, \Omega, D, w) \leq C(\mathcal{H}_1, Q, \Omega, D, w)$, i.e., is at least as good as the solution of a more restricted convex optimization problem. The second minimization can be performed with a local descent algorithm, without the necessity to guarantee that the global optimum will be found.

6.4 Finite Number of Hidden Neurons

In this section we consider the special case with $Q(\hat{y}, y) = max(0, 1 - y\hat{y})$ the hinge loss, and L^1 regularization, and we show that the global optimum of the convex cost involves at most $n + 1$ hidden neurons, using an approach already exploited in [67] for L^1-loss regression Boosting with L^1 regularization of output weights.

The training criterion is $C(w) = K\|w\|_1 + \sum_{t=1}^{n} \max(0, 1 - y_t w \cdot h(x_t))$. Let us re-write this cost function as the constrained optimization problem :

$$\min_{w,\xi} \; L(w, \xi) = K\|w\|_1 + \sum_{t=1}^{n} \xi_t \quad \text{s.t.} \begin{cases} y_t \left[w \cdot h(x_t)\right] \geq 1 - \xi_t & (C_1) \\ \text{and} \quad \xi_t \geq 0, t = 1, \ldots, n & (C_2) \end{cases}$$

Using a standard technique, the above program can be recast as a linear program. Defining $\lambda = (\lambda_1, \ldots, \lambda_n)$ the vector of Lagrangian multipliers for the constraints C_1, its dual problem (P) takes the form (in the case of a finite number J of base learners) :

$$(P): \qquad \max_{\lambda} \; \sum_{t=1}^{n} \lambda_t \quad \text{s.t.} \begin{cases} \lambda \cdot Z_i - K \leq 0, i \in I \\ \text{and} \quad \lambda_t \leq 1, t = 1, \ldots, n \end{cases}$$

with $(Z_i)_t = y_t h_i(x_t)$. In the case of a finite number J of base learners, $I = \{1, \ldots, J\}$. If the number of hidden units is uncountable, then I is a closed bounded interval of $\mathbb{R}$.

Such an optimization problem satisfies all the conditions needed for using Theorem 4.2 from [38]. Indeed :

- I is compact (as a closed bounded interval of $\mathbb{R}$) ;
- $F : \lambda \mapsto \sum_{t=1}^{n} \lambda_t$ is a concave function (it is even a linear function) ;
- $g : (\lambda, i) \mapsto \lambda \cdot Z_i - K$ is convex in λ (it is actually linear in λ) ;
- $\nu(P) \leq n$ (therefore finite) ($\nu(P)$ is the largest value of F satisfying the constraints) ;
- for every set of $n + 1$ points $i_0, \ldots, i_n \in I$, there exists $\tilde{\lambda}$ such that $g(\tilde{\lambda}, i_j) < 0$ for $j = 0, \ldots, n$ (one can take $\tilde{\lambda} = 0$ since $K > 0$).

Then, from Theorem 4.2 from [38], the following theorem holds :

Theorem 6.4.1. *The solution of (P) can be attained with alternate constraints C_2' and*

only $n+1$ constraints C_1' (i.e., there exists a subset of $n+1$ constraints C_1' giving rise to the same maximum as when using the whole set of constraints). Even though these constraints are unknown, this ensures that the primal problem associated is the minimization of the cost function of a NN with $n+1$ hidden neurons.

6.5 Incremental Convex NN Algorithm

In this section we present a stepwise algorithm to optimize a NN, and show that there is a criterion that allows to verify whether the global optimum has been reached. This is a specialization of minimizing $C(\mathcal{H}, Q, \Omega, D, w)$, with $\Omega(w) = \lambda||w||_1$ and $\mathcal{H} = \{h : h(x) = s(v \cdot \tilde{x})\}$ is the set of soft or hard linear classifiers (depending on choice of $s(\cdot)$).

Algorithm ConvexNN(D,Q,λ,s)

Input : training set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, convex loss function Q, and scalar regularization penalty λ. s is either the *sign* function or the *tanh* function.

(1) Set $v_1 = (0, 0, \ldots, 1)$ and select $w_1 = \text{argmin}_w \sum_t Q(ws(1), y_t) + \lambda|w|$.

(2) Set $i = 2$.

(3) Repeat

(4) Let $q_t = Q'(\sum_{j=1}^{i-1} w_j h_j(x_t), y_t)$

(5) If $s = \text{sign}$

(5a) train linear classifier $h_i(x) = \text{sign}(v_i \cdot \tilde{x})$ with examples $\{(x_t, \text{sign}(q_t))\}$ and errors weighted by $|q_t|$, $t = 1 \ldots n$ (i.e., *maximize* $\sum_t q_t h_i(x_t)$)

(5b) else ($s = \tanh$)

(5c) train linear classifier $h_i(x) = \tanh(v_i \cdot \tilde{x})$ to *maximize* $\sum_t q_t h_i(x_t)$.

(6) If $\sum_t q_t h_i(x_t) < \lambda$, **stop**.

(7) Select $w_1, \ldots, w_i$ (and optionally $v_2, \ldots, v_i$) minimizing (exactly or approximately) $C = \sum_t Q(\sum_{j=1}^{i} w_j h_j(x_t), y_t) + \lambda \sum_{j=1} |w_j|$ such that $\frac{\partial C}{\partial w_j} = 0$ for $j = 1 \ldots i$.

(7) Increment i by 1.

(9) Return the predictor $\hat{y}(x) = \sum_{j=1}^{i} w_j h_j(x)$.

A key property of the above algorithm is that, at termination, the global optimum is reached, i.e., no hidden unit (linear classifier) can improve the objective. In the case where $s = \text{sign}$, we obtain a Boosting-like algorithm, i.e., it involves finding a classifier which minimizes the weighted cost $\sum_t q_t \text{sign}(v \cdot \tilde{x_t})$.

Theorem 6.5.1. *Algorithm* **ConvexNN** *stops when it reaches the global optimum of*

$$C(w) = \sum_t Q(w \cdot h(x_t), y_t) + \lambda||w||_1.$$

Démonstration. Let w be the output weights vector when the algorithm stops. Because the set of hidden units $\mathcal{H}$ we consider is such that when h is in $\mathcal{H}$, $-h$ is also in $\mathcal{H}$, we can assume all weights to be non-negative. By contradiction, if $w' \neq w$ is the global

optimum, with $C(w') < C(w)$, then, since C is convex in the output weights, for any $\epsilon \in (0,1)$, we have $C(\epsilon w' + (1-\epsilon)w) \leq \epsilon C(w') + (1-\epsilon)C(w) < C(w)$. Let $w_\epsilon = \epsilon w' + (1-\epsilon)w$. For ϵ small enough, we can assume all weights in w that are strictly positive to be also strictly positive in w_ϵ. Let us denote by I_p the set of strictly positive weights in w (and w_ϵ), by I_z the set of weights set to zero in w but to a non-zero value in w_ϵ, and by $\delta_{\epsilon k}$ the difference $w_{\epsilon,k} - w_k$ in the weight of hidden unit h_k between w and w_ϵ. We can assume $\delta_{\epsilon j} < 0$ for $j \in I_z$, because instead of setting a small positive weight to h_j, one can decrease the weight of $-h_j$ by the same amount, which will give either the same cost, or possibly a lower one when the weight of $-h_j$ is positive. With $o(\epsilon)$ denoting a quantity such that $\epsilon^{-1}o(\epsilon) \to 0$ when $\epsilon \to 0$, the difference $\Delta_\epsilon(w) = C(w_\epsilon) - C(w)$ can now be written :

$$
\begin{aligned}
\Delta_\epsilon(w) &= \lambda\left(\|w_\epsilon\|_1 - \|w\|_1\right) + \sum_t \left(Q(w_\epsilon \cdot h(x_t), y_t) - Q(w \cdot h(x_t), y_t)\right) \\
&= \lambda\left(\sum_{i\in I_p} \delta_{\epsilon i} + \sum_{j \in I_z} -\delta_{\epsilon j}\right) + \sum_t \sum_k \left(Q'(w \cdot h(x_t), y_t)\delta_{\epsilon k} h_k(x_t)\right) + o(\epsilon) \\
&= \sum_{i\in I_p}\left(\lambda\delta_{\epsilon i} + \sum_t q_t \delta_{\epsilon i} h_i(x_t)\right) + \sum_{j\in I_z}\left(-\lambda\delta_{\epsilon j} + \sum_t q_t \delta_{\epsilon j} h_j(x_t)\right) + o(\epsilon) \\
&= \sum_{i\in I_p} \delta_{\epsilon i}\frac{\partial C}{\partial w_i}(w) + \sum_{j\in I_z}\left(-\lambda\delta_{\epsilon j} + \sum_t q_t \delta_{\epsilon j} h_j(x_t)\right) + o(\epsilon) \\
&= 0 + \sum_{j\in I_z}\left(-\lambda\delta_{\epsilon j} + \sum_t q_t \delta_{\epsilon j} h_j(x_t)\right) + o(\epsilon)
\end{aligned}
$$

since for $i \in I_p$, thanks to step (7) of the algorithm, we have $\frac{\partial C}{\partial w_i}(w) = 0$. Thus the inequality $\epsilon^{-1}\Delta_\epsilon(w) < 0$ rewrites into

$$
\sum_{j\in I_z} \epsilon^{-1}\delta_{\epsilon j}\left(-\lambda + \sum_t q_t h_j(x_t)\right) + \epsilon^{-1}o(\epsilon) < 0
$$

which, when $\epsilon \to 0$, yields (note that $\epsilon^{-1}\delta_{\epsilon j}$ does not depend on ϵ since $\delta_{\epsilon j}$ is linear in

ϵ) :

$$\sum_{j \in I_z} \epsilon^{-1} \delta_{\epsilon j} \left(-\lambda + \sum_t q_t h_j(x_t) \right) \leq 0 \tag{6.2}$$

But, h_i being the optimal classifier chosen in step (5a) or (5c), all hidden units h_j verify $\sum_t q_t h_j(x_t) \leq \sum_t q_t h_i(x_t) < \lambda$ and $\forall j \in I_z$, $\epsilon^{-1}\delta_{\epsilon j}\left(-\lambda + \sum_t q_t h_j(x_t)\right) > 0$ (since $\delta_{\epsilon j} < 0$), contradicting eq. 6.2. □

[57] prove a related global convergence result for the AnyBoost algorithm, a non-parametric Boosting algorithm that is also similar to Gradient Boosting [34]. Again, this requires solving as a sub-problem an exact minimization to find a function $h_i \in \mathcal{H}$ that is maximally correlated with the gradient Q' on the output. We now show a simple procedure to select a hyperplane with the best weighted classification error.

Exact Minimization

In step (5a) we are required to find a linear classifier that minimizes the weighted sum of classification errors. Unfortunately, this is an NP-hard problem (w.r.t. d, see theorem 4 in [56]). However, an exact solution can be easily found in $O(n^3)$ computations for $d = 2$ inputs.

Proposition 6.5.2. *Finding a linear classifier that minimizes the weighted sum of classification error can be achieved in $O(n^3)$ steps when the input dimension is $d = 2$.*

Démonstration. We want to maximize $\sum_i c_i \text{sign}(u \cdot x_i + b)$ with respect to u and b, the c_i's being in $\mathbb{R}$. Consider u **fixed** and sort the x_i's according to their dot product with u and denote r the function which maps i to $r(i)$ such that $x_{r(i)}$ is in i-th position in the sort. Depending on the value of b, we will have $n+1$ possible sums, respectively $-\sum_{i=1}^{k} c_{r(i)} + \sum_{i=k+1}^{n} c_{r(i)}$, $k = 0, \ldots, n$. It is obvious that those sums only depend on the order of the products $u \cdot x_i$, $i = 1, \ldots, n$. When u varies smoothly on the unit circle, as the dot product is a continuous function of its arguments, the changes in the order of the dot products will occur only when there is a pair (i, j) such that $u \cdot x_i = u \cdot x_j$. Therefore, there are at most as many order changes as there are pairs of different points, i.e., $n(n-1)/2$. In the case of $d = 2$, we can enumerate all the different angles for which there is a change, namely $a_1, \ldots, a_z$ with $z \leq \frac{n(n-1)}{2}$. We then need to test at least one

$u = [\cos(\theta), \sin(\theta)]$ for each interval $a_i < \theta < a_{i+1}$, and also one u for $\theta < a_1$, which makes a total of $\frac{n(n-1)}{2}$ possibilities. □

It is possible to generalize this result in higher dimensions, and as shown in [56], one can achieve $O(\log(n)n^d)$ time.

Algorithm 1 Optimal linear classifier search

Maximizing $\sum_{i=1}^{n} c_i \delta(\text{sign}(w \cdot x_i), y_i)$ in dimension 2

(1) for $i = 1, \ldots, n$ for $j = i + 1, \ldots, n$
(3) $\Theta_{i,j} = \theta(x_i, x_j) + \frac{\pi}{2}$ where $\theta(x_i, x_j)$ is the angle between x_i and x_j
(6) sort the $\Theta_{i,j}$ in increasing order
(7) $w_0 = (1, 0)$
(8) for $k = 1, \ldots, \frac{n(n-1)}{2}$
(9) $w_k = (\cos\Theta_{i,j}, \sin\Theta_{i,j})$, $u_k = \frac{w_k + w_{k-1}}{2}$
(10) sort the x_i according to the value of $u_k \cdot x_i$
(11) compute $S(u_k) = \sum_{i=1}^{n} c_i \delta(u_k \cdot x_i), y_i)$
(12) output : $\text{argmax}_{u_k} S$

Approximate Minimization

For data in higher dimensions, the exact minimization scheme to find the optimal linear classifier is not practical. Therefore it is interesting to consider approximate schemes for obtaining a linear classifier with weighted costs. Popular schemes for doing so are the linear SVM (i.e., linear classifier with hinge loss), the logistic regression classifier, and variants of the Perceptron algorithm. In that case, step (5c) of the algorithm is not an exact minimization, and one cannot guarantee that the global optimum will be reached. However, it might be reasonable to believe that finding a linear classifier by minimizing a weighted hinge loss should yield solutions close to the exact minimization. Unfortunately, this is not generally true, as we have found out on a simple toy data set described below. On the other hand, if in step (7) one performs an optimization not only of the

output weights w_j ($j \leq i$) but also of the corresponding weight vectors v_j, then the algorithm finds a solution close to the global optimum (we could only verify this on 2D data sets, where the exact solution can be computed easily). It means that at the end of each stage, one first performs a few training iterations of the whole NN (for the hidden units $j \leq i$) with an ordinary gradient descent mechanism (we used conjugate gradients but stochastic gradient descent would work too), optimizing the w_j's and the v_j's, and then one fixes the v_j's and obtains the optimal w_j's for these v_j's (using a convex optimization procedure). In our experiments we used a quadratic Q, for which the optimization of the output weights can be done with a neural network, using the outputs of the hidden layer as inputs.

Let us consider now a bit more carefully what it means to tune the v_j's in step (7). Indeed, changing the weight vector v_j of a selected hidden neuron to decrease the cost is **equivalent to a change in the output weights w's**. More precisely, consider the step in which the value of v_j becomes v'_j. This is equivalent to the following operation on the w's, when w_j is the corresponding output weight value : the output weight associated with the value v_j of a hidden neuron is set to 0, and the output weight associated with the value v'_j of a hidden neuron is set to w_j. This corresponds to an exchange between two variables in the convex program. We are justified to take any such step as long as it allows us to decrease the cost $C(w)$. The fact that we are simultaneously making such exchanges on all the hidden units when we tune the v_j's allows us to move faster towards the global optimum.

Extension to multiple outputs

The multiple outputs case is more involved than the single-output case because it is not enough to check the condition $\sum_t h_t q_t > \lambda$. Consider a new hidden neuron whose output is h_i when the input is x_i. Let us also denote $\alpha = [\alpha_1, \ldots, \alpha_{n_o}]'$ the vector of output weights between the new hidden neuron and the n_o output neurons. The gradient with respect to α_j is $g_j = \frac{\partial C}{\partial \alpha_j} = \sum_t h_t q_{tj} - \lambda \mathrm{sign}(\alpha_j)$ with q_{tj} the value of the j-th output neuron with input x_t. This means that if, for a given j, we have $|\sum_t h_t q_{tj}| < \lambda$, moving α_j away from 0 can only increase the cost. Therefore, the right quantity to consider is $(|\sum_t h_t q_{tj}| - \lambda)_+$.

We must therefore find $\mathrm{argmax}_v \sum_j (|\sum_t h_t q_{tj}| - \lambda)_+^2$. As before, this sub-problem is not convex, but it is not as obvious how to approximate it by a convex problem. The stopping criterion becomes : if there is no j such that $|\sum_t h_t q_{tj}| > \lambda$, then all weights must remain equal to 0 and a global minimum is reached.

Experimental Results

We performed experiments on the 2D double moon toy dataset (as used in [31]), to be able to compare with the exact version of the algorithm. In these experiments, $Q(w \cdot h(x_t), y_t) = [w \cdot h(x_t) - y_t]^2$. The set-up is the following :

• Select a new linear classifier, either (a) the optimal one or (b) an approximate using logistic regression.

• Optimize the output weights using a convex optimizer.

• In case (b), tune both input and output weights by conjugate gradient descent on C and finally re-optimize the output weights using LASSO regression.

• Optionally, remove neurons whose output weight has been set to 0.

Using the approximate algorithm yielded for 100 training examples an average penalized ($\lambda = 1$) squared error of 17.11 (over 10 runs), an average test classification error of 3.68% and an average number of neurons of 5.5. The exact algorithm yielded a penalized squared error of 8.09, an average test classification error of 5.3%, and required 3 hidden neurons. A penalty of $\lambda = 1$ was nearly optimal for the exact algorithm whereas a smaller penalty further improved the test classification error of the approximate algorithm. Besides, when running the approximate algorithm for a long time, it converges to a solution whose quadratic error is extremely close to the one of the exact algorithm.

6.6 Conclusion

We have shown that training a NN can be seen as a convex optimization problem, and have analyzed an algorithm that can exactly or approximately solve this problem. We have shown that the solution with the hinge loss involved a number of non-zero weights bounded by the number of examples, and much smaller in practice. We have shown that there exists a stopping criterion to verify if the global optimum has been

reached, but it involves solving a sub-learning problem involving a linear classifier with weighted errors, which can be computationally hard if the exact solution is sought, but can be easily implemented for toy data sets (in low dimension), for comparing exact and approximate solutions.

The above experimental results are in agreement with our initial conjecture : when there are many hidden units we are much less likely to stall in the optimization procedure, because there are many more ways to descend on the convex cost $C(w)$. They also suggest, based on experiments in which we can compare with the exact sub-problem minimization, that applying Algorithm **ConvexNN** with an approximate minimization for adding each hidden unit **while continuing to tune the previous hidden units** tends to lead to fast convergence to the global minimum. What can get us stuck in a "local minimum" (in the traditional sense, i.e., of optimizing w's and v's together) is simply the **inability to find a new hidden unit weight vector that can improve the total cost (fit and regularization term) even if there exists one**.

Note that as a side-effect of the results presented here, we have a simple way to train **neural networks with hard-threshold hidden units**, since increasing $\sum_t Q'(\hat{y}(x_t), y_t)\mathrm{sign}(v_i x_t)$ can be either achieved exactly (at great price) or approximately (e.g. by using a cross-entropy or hinge loss on the corresponding linear classifier).

Acknowledgments

The authors thank the following for support : NSERC, MITACS, and the Canada Research Chairs. They are also grateful for the feedback and stimulating exchanges with Sam Roweis, Nathan Srebro, and Aaron Courville.

CHAPITRE 7

PRÉSENTATION DU TROISIÈME ARTICLE

7.1 Détails de l'article

Continuous Neural Networks
N. Le Roux et Y. Bengio
Publié dans *Proceedings of the Eleventh International Workshop on Artificial Intelligence and Statistics* en 2007.

7.2 Contexte

Les réseaux convexes présentés au chapitre précédent ouvrirent la voie des réseaux infinis en conservant une solution finale exprimable avec un nombre fini de paramètres. Toutefois, comme nous avons pu le voir, leur optimisation requiert un temps exponentiel et ils généralisaient parfois moins bien que leur version approchée.

L'article ci-après a pour but de proposer une nouvelle vision des réseaux de neurones donnant naissance à des réseaux de neurones infinis dont l'optimisation est aussi rapide que pour des réseaux normaux. Il renforce également le lien, déjà étudié par Neal [60] et Williams [83], entre les réseaux de neurones et les méthodes à noyau.

7.3 Commentaires

L'idée sous-jacente de l'article est qu'un réseau de neurones à une couche cachée modélise une primitive d'une fonction constante par morceaux, chaque unité cachée correspondant à un morceau différent. Lorsque le nombre d'unités cachées tend vers l'infini, le nombre de morceaux fait de même et le réseau de neurones modélise la primitive d'une fonction quelconque. L'apprentissage d'un réseau de neurones peut donc être vu comme l'approximation d'une fonction quelconque (la fonction dont on calcule

la primitive) par une fonction constante par morceaux.

Les réseaux continus, de leur côté, tentent d'approcher cette fonction par une fonction affine par morceaux. Cette modélisation est strictement plus puissante que les réseaux de neurones standards (en ce sens qu'il existe des réseaux continus qu'il est impossible de modéliser avec des réseaux standards alors que l'inverse est faux).

Les réseaux continus se sont effectivement avérés plus puissants que les réseaux de neurones standards sur certaines tâches, mais les résultats furent décevants sur d'autres. L'une des raisons possibles est la difficulté de l'optimisation induite par une telle paramétrisation. En effet, bien que le nombre de paramètres reste raisonnable, les unités cachées deviennent liées les unes aux autres, entraînant des dépendances complexes préjudiciables à la descente de gradient.

De son côté, le noyau induit par les réseaux infinis utilisant la fonction de Heaviside comme fonction de transfert est intéressant, tant par sa simplicité que par la qualité des résultats qu'il occasionne. Il n'est pas non plus inutile de remarquer que ce noyau n'est pas local, au contraire du noyau gaussien.

CHAPITRE 8

CONTINUOUS NEURAL NETWORKS

8.1 Abstract

This article extends neural networks to the case of an uncountable number of hidden units, in several ways. In the first approach proposed, a finite parametrization is possible, allowing gradient-based learning. For the same number of parameters than an ordinary neural network, this model yields lower error on real-world data while having a lower capacity. Better error bounds than with ordinary neural networks are shown using an affine parametrization of the input-to-hidden weights. Experimental results show that optimization is easier than with ordinary neural networks, possibly by reducing the problem of hidden unit saturation. In a second approach the input-to-hidden weights are fully non-parametric, yielding a kernel machine for which we demonstrate a simple kernel formula. Interestingly, the resulting kernel machine can be made hyperparameter-free, and still generalizes in spite of an absence of explicit regularization.

8.2 Introduction

In [60] neural networks with an infinite number of hidden units were introduced, showing that they could be interpreted as Gaussian processes, and this work served as inspiration for a large body of work on Gaussian processes. Neal's work showed a counter-example to two common beliefs in the machine learning community : (1) that a neural network with a very large number of hidden units would overfit and (2) that it would not be feasible to numerically optimize such huge networks. In spite of Neal's work, these beliefs are still commonly held. In this paper we return to Neal's idea and study a number of extensions of neural networks to the case where the number of hidden units is uncountable, and show that they yield implementable algorithms with interesting properties.

Consider a neural network with one hidden layer (and h hidden neurons), one output

neuron and a transfer function g. Let us denote V the input-to-hidden weights, $a_i (i = 1, \ldots, h)$ the hidden-to-output weights and β the output unit bias (the hidden units bias is included in V).

The output of the network with input x is then

$$f(x) = \beta + \sum_i a_i g(\tilde{x} \cdot V_i) \tag{8.1}$$

where V_i is the i^{th} column of matrix V and $\tilde{x}$ is the vector with 1 appended to x. The output of the i^{th} hidden neuron is $g(\tilde{x} \cdot V_i)$. For symmetric functions g such as $\tanh$, we can without any restriction consider the case where all the a_i's are nonnegative, since changing the sign of a_i is equivalent to changing the sign of V_i.

In ordinary neural networks we have an integer index i. To obtain an uncountable number of hidden units, we introduce a continuous-valued (possibly vector-valued) index $z \in \mathbb{R}^m$, with $m \leq d + 1$. We can replace the usual sum over hidden units by an integral that goes through the different weight vectors that can be assigned to a hidden unit :

$$f(x) = \beta + \int a(z) g[\tilde{x} \cdot V(z)] \, dz \tag{8.2}$$

where $a : \mathbb{R}^m \to \mathbb{R}$ is the hidden-to-output weight function, and $V : \mathbb{R}^m \to \mathbb{R}^{d+1}$ is the input-to-hidden weight function. Alternatively, function a can be replaced by a single scalar α when g is invertible over some range. Let $a(z) = \alpha\gamma(z)$ where α is a global constant such that $\gamma(z) g(\tilde{x} \cdot V(z))$ is in the domain of g for all z. Then one can find a function $b(z)$ such that $a(z) g(\tilde{x} \cdot V(z)) = \alpha g(b(z)\tilde{x} \cdot V(z))$, i.e. $b(z) = \frac{g^{-1}(a(z) g(\tilde{x} \cdot (V(z)))/\alpha)}{\tilde{x} \cdot V(z)}$, and $V(z)$ can be replaced by $b(z)V(z)$. An equivalent parametrization is therefore

$$f(x) = \alpha \int g[\tilde{x} \cdot V(z)] \, dz. \tag{8.3}$$

How can we prevent overfitting in these settings ? In this paper we discuss three types of solutions : (1) finite-dimensional representations of V (using formulation 8.3). (2) $\mathcal{L}^1$ regularization of the output weights with a and V completely free (3) $\mathcal{L}^2$ regularization of the output weights, with a and V completely free.

Solutions of type 1 are completely new and surprising results are obtained with them. Many possible parametrizations are possible to map the "hidden unit index" (a finite-dimensional vector) to a weight vector. This parametrization allows to show a representation theorem similar to Kolmogorov's, i.e. we show that functions from $\mathbb{R}^d$ to $\mathbb{R}$ can be represented by $d+1$ functions from $\mathbb{R}$ to $\mathbb{R}$. Here we study an affine-by-part parametrization, with interesting properties. In particular, it is shown that approximation error can be bounded in $O(h^{-2})$ where h is proportional to the number of free parameters, whereas ordinary neural networks enjoy a $O(h^{-1})$ error bound. Experiments suggest that optimizing them is easier than optimizing ordinary neural networks, probably because they are less sensitive to the saturation problem (when the weighted sum of a hidden unit is large, the derivative through it is close to 0, making it difficult to optimize the input-to-hidden weights of that hidden unit). Solutions of type 2 ($\mathcal{L}^1$ regularization) were already presented in [16] so we do not focus on them here. They also yield to a convex formulation but in an infinite number of variables, requiring approximate optimization. Solutions of type 3 ($\mathcal{L}^2$ regularization) give rise to kernel machines that are similar to Gaussian processes, albeit with a type of kernel not commonly used in the litterature. We show that an analytic formulation of the kernel is possible when hidden units compute the sign of a dot product (i.e., with formal neurons). Interestingly, experiments suggest that this kernel is more resistant to overfitting than the Gaussian kernel, allowing to train working models without a single hyper-parameter.

8.3 Affine neural networks

8.3.1 Core idea

We study here a special case of the solutions of type (1), with a finite-dimensional parametrization of the continuous neural network, based on parametrizing $V(z)$ in eq. 8.3, where z is scalar. In this case, without loss of generality we can take the domain of z to be the unit interval, thus

$$f(x) = \beta + \alpha \int_0^1 g[\tilde{x} \cdot V(z)] \, dz \tag{8.4}$$

where V is a function from $\mathbb{R}$ to $\mathbb{R}^{d+1}$ (d being the dimension of x).

It is easy to see that this becomes formula (8.1) when V is a piecewise-constant function such that

$$V(z) = V_i \quad \text{when} \quad p_{i-1} \leq z < p_i$$

with $a_0 = 0$, $\alpha = \sum_i a_i$ and $p_i = \frac{1}{\alpha} \sum_{j=0}^{i} a_j$.

At this point, we can make an important comment. If $x \in \mathbb{R}^d$ we can rewrite $\tilde{x} \cdot V(z) = V^{d+1}(z) + \sum_{i=1}^{d} x_i V^i(z)$ where V^i is a function from $\mathbb{R}$ to $\mathbb{R}$ and x_i is the i-th coordinate of x for each i. But f is a function from $\mathbb{R}^d$ to $\mathbb{R}$. Using neural network function approximation theorems, the following theorem therefore follows :

Theorem 8.3.1. *Any function f from $\mathbb{R}^d$ to $\mathbb{R}$ can, with an arbitrary precision, be defined by $d+1$ functions V^i from $\mathbb{R}$ to $\mathbb{R}$ and two reals α and β with the relation*

$$f(x) = \beta + \alpha \int_0^1 \tanh\left(\sum_{i=1}^{d} V^i(z) \cdot x_i + V^{n+1}(z)\right) dz$$

This result is reminiscent of Kolmogorov's superposition theorem [48], but here we show that the functions V^i can be directly optimized in order to obtain a good function approximation.

8.3.2 Approximating an Integral

The formulation in eq. 8.4 shows that the only thing to optimize will be the function V, and the scalars α and β, getting rid of the optimization of the hidden-to-output weights.

In this work we consider a parametrization of V involving a finite number of parameters, and we optimize over these parameters. Since f is linked to V by an integral, it suggests to look over parametrizations yielding good approximation of an integral. Several such parametrizations already exist :

– piecewise constant functions, used in the rectangle method. This is the simplest approximation, corresponding to ordinary neural networks (eq. 8.1),

– piecewise affine functions, used in the trapezoid method. This approximation yields better results and will be the one studied here, which we coin “Affine Neural Network”.
– polynomials, used in Simpson’s method, which allow even faster convergence. However, we were not able to compute the integral of polynomials through the function $\tanh$.

8.3.3 Piecewise Affine Parametrization

Using a piecewise affine parametrization, we consider V of the form :

$$V(z) = V_{i-1} + \frac{z - p_{i-1}}{p_i - p_{i-1}}(V_i - V_{i-1}) \text{ when } p_{i-1} \leq z < p_i,$$

that is to say V is linear between p_{i-1} and p_i, $V(p_{i-1}) = V_{i-1}$ and $V(p_i) = V_i$ for each i. This will ensure the continuity of V.

Besides, we will set $V_{n+1} = V_1$ to avoid border effects and obtain an extra segment for the same number of parameters.

Rewriting $p_i = p_{i-1} = a_i$ and $V(z) = V_i(z)$ when $p_{i-1} \leq z < p_i$, the output for an input example x can be written :

$$\begin{aligned} f(x) &= \sum_i \int_{z=p_{i-1}}^{p_i} \tanh\left[V_i(z) \cdot \tilde{x}\right] \\ f(x) &= \sum_i \frac{a_i}{(V_i - V_{i-1}) \cdot \tilde{x}} \ln\left(\frac{\cosh(V_i \cdot \tilde{x})}{\cosh(V_{i-1} \cdot \tilde{x})}\right) \end{aligned} \tag{8.5}$$

To respect the continuity of the function V, we should restrict the a_i to be positive since p_i must be greater than p_{i-1}. However, in our experiments, we did not use that restriction to simplify gradient descent over the parameters. We believe that this makes little difference in the final result.

8.3.3.1 Is the continuity of V necessary ?

As said before, we want to enforce the continuity of V. The first reason is that the trapezoid method uses continuous functions and the results concerning that method can therefore be used for our approximation. Besides, we have this intuition that a "reasonable" function f will use a "reasonable" function V. This intuition will not be theoretically verified here but it seems to be validated by the experimental results.

The reader might notice at that point that there is no bijection between V and f. Indeed, since V is only defined by its integral, we can switch two different pieces of V without modifying f. Thus, it would more correct to say that if f is a "reasonable" function, then there must exist a "reasonable" V that gives rise to f.

8.3.4 Extension to multiple output neurons

The formula of the output is a linear combination of the a_i, as in the ordinary neural network. Thus, the extension to l output neurons is straightforward using the formula

$$f_j(x) = \sum_i \frac{a_{ij}}{(V_i - V_{i-1}) \cdot \tilde{x}} \ln\left(\frac{\cosh(V_i \cdot \tilde{x})}{\cosh(V_{i-1} \cdot \tilde{x})}\right) \tag{8.6}$$

for $j = 1, \ldots, l$.

8.3.5 Piecewise affine versus piecewise constant

Consider a target function f^* that we would like to approximate, and a target V^* that gives rise to it. Before going any further, we should ask ourselves two questions :

- is there a relation between the quality of the approximation of f^* and the quality of approximation of V^* ?
- are piecewise affine functions (i.e. our affine neural networks) more appropriate to approximate an arbitrary function than the piecewise constant ones (i.e. ordinary neural networks) ?

8.3.5.1 Relation between f and V

To answer the first of those questions, we will bound the error between $f(x)$ and $f^*(x)$ using the error between V and V^*. First assume that $\alpha = \alpha^*$ and $\beta = \beta^*$.

$$\begin{aligned}
|f(x) \quad - \quad & f^*(x)| = \\
= \quad a^* \quad & \left| \int_z (\tanh[V(z)\cdot\tilde{x}] - \tanh[V^*(z)\cdot\tilde{x}])\, dz \right| \\
= \quad a^* \quad & | \int_z \tanh[(V(z) - V^*(z))\cdot\tilde{x}] \\
& (1 - \tanh[V(z)\cdot\tilde{x}]\tanh[V^*(z)\cdot\tilde{x}])\, dz| \\
\leq \quad a^* \quad & \int_z |\tanh[(V(z) - V^*(z))\cdot\tilde{x}]|\, dz \\
\times \quad & \sup_z |1 - \tanh[V(z)\cdot\tilde{x}]\tanh[V^*(z)\cdot\tilde{x}]| \\
\leq \quad 2a^* \quad & \int_z |\tanh[(V(z) - V^*(z))\cdot\tilde{x}]|\, dz \\
\leq \quad 2a^* \quad & \int_z \tanh[|(V(z) - V^*(z))\cdot\tilde{x}|]\, dz
\end{aligned}$$

But for a given V, the difference between f and f^* can only be smaller if we allow (α, β) to be different from (α^*, b^*) (i.e. choosing α and β optimally). Therefore, the inequality remains when α and β are not fixed.

The difference between the approximate function f using V and the true function f^* is therefore bounded by $2\alpha^* \int_z \tanh[|(V(z) - V^*(z))\cdot\tilde{x}|]\, dz$, which is itself bounded by $2\alpha^* \times \sup_z \tanh[|(V^* - V)(z)\cdot\tilde{x}|]$.

Thus, if V is never far from V^*, we are sure that the approximated function will be close from the true function. This makes reasonable our try to better approximate V^*.

We can then make an obvious remark : if we restrict ourselves to a finite number of hidden neurons, it will never be possible to have a piecewise constant function equal to a piecewise affine function (apart from the trivial case where all affine functions are in fact constant). On the other hand, any piecewise contant function composed of h pieces can be represented arbitrarily closely by a continuous piecewise affine function composed of at most $2h$ pieces (half of the pieces being constant and the other half being used to

avoid discontinuities).

A second remark can be made : for any piecewise constant function and any continuous piecewise affine function, given that they are not equal, one can find a function that is better approximated by the first class than by the second class ; it is trivial to see that one can also find a function that is better approximated by the second class than by the first class.

After those trivial comments, let us focus on less evident remarks. As shown in section 8.3.5.1, we would like our approximation V of V^* to remain close to V^* for all z.

If V is a piecewise constant function, then it is quite straightforward to prove the following inequality :

$$|V^*(z) - V(z)| \leq \frac{p_i - p_{i-1}}{2} M_1(V^*, [p_{i-1}, p_i]) \tag{8.7}$$

where $M_1(V, I) = \max_{z \in I} |V'(z)|$ ($M_1(V, I)$ is the maximum absolute value of the first derivative of V on the interval I).

The trapezoid method tells us that the approximation of a function V^* in C^2 by a continous piecewise affine function V using the trapezoid method verifies :

$$|V^*(z) - V(z)| \leq \frac{(z - p_{i-1})(p_i - z)}{2} M_2(V^*, [p_{i-1}, p_i])$$

where $M_2(V, I) = \max_{z \in I} |V''(z)|$ ($M_2(V, I)$ is the maximum absolute value of the second derivative of V on the interval I). Using the fact that, for all z in $[p_{i-1}, p_i]$, $(z - p_{i-1})(p_i - z) \leq \frac{(p_i - p_{i-1})^2}{4}$, we have

$$|V^*(z) - V(z)| \leq \frac{(p_i - p_{i-1})^2}{8} M_2(V^*, [p_{i-1}, p_i]) \tag{8.8}$$

There is however a difference between the rectangle method, the trapezoid method and the optimization of neural networks. Indeed, both the rectangle method and the trapezoid method use uniformly spaced points, whereas in the case of affine neural networks, the spacing between two consecutive points is determined by the associated output weight.

As this is less restrictive, we are guaranteed that the error with the approximated function obtained while optimizing an affine neural network is better that the bound above.

Since the integral is on $[0, 1]$ and there are h hidden units, uniformly spaced points would give $p_i - p_{i-1} = \frac{1}{h}$. This leads to the following theorem :

Theorem 8.3.2. *In an affine neural net, $\|V^* - V\|_\infty = O(h^{-2})$ with h the number of hidden units instead of $\|V^* - V\|_\infty = O(h^{-1})$ for an ordinary neural network.*

As for all x in $\mathbb{R}$, we have $\tanh(|x|) \leq |x|$, the following theorem is a consequence of theorem 8.3.2 :

Theorem 8.3.3. *There exists two scalars C_1 and C_2 such that, for all x, $|f(x) - \hat{f}(x)| \leq C_1 h^{-2}$ with h the number of hidden units in an affine neural network (pointwise convergence) instead of $|f(x) - \hat{f}(x)| \leq C_2 h^{-1}$ for an ordinary neural network.*

One must first note that those are upper bounds. It therefore does not guarantee that the optimization of affine neural networks will always be better than the one of ordinary neural networks. Furthermore, one shall keep in mind that both methods are subject to local minima. However, we will see in the following section that the affine neural networks appear less likely to get stuck during the optimization than ordinary neural networks.

8.3.6 Implied prior distribution

Up to now we have only considered a parametrized class of functions. We know that gradient descent or ordinary $\mathcal{L}^2$ regularization are more likely to yield small values of the parameters, the latter directly corresponding to a zero-mean Gaussian prior over the parameters. Hence to visualize and better understand this function, we define a zero-mean Gaussian prior distribution on the parameters β, a_i and V_i ($1 \leq i \leq h$), and sample from the corresponding functions.

We sampled from each of the two neural networks (ordinary discrete net and continuous affine net) with different numbers of hidden units and zero-mean Gaussian priors with variance σ_u for input-to-hidden weights, variance σ_a for the input biases, variance

σ_b for the output bias and variance $\frac{w_v}{\sqrt{h}}$ for hidden-to-output weights (scaled in terms of the number of hidden units h, to keep constant the variance of the network output as h changes). Randomly obtained samples are shown in figures 8.3.6 to 8.3.6. The different

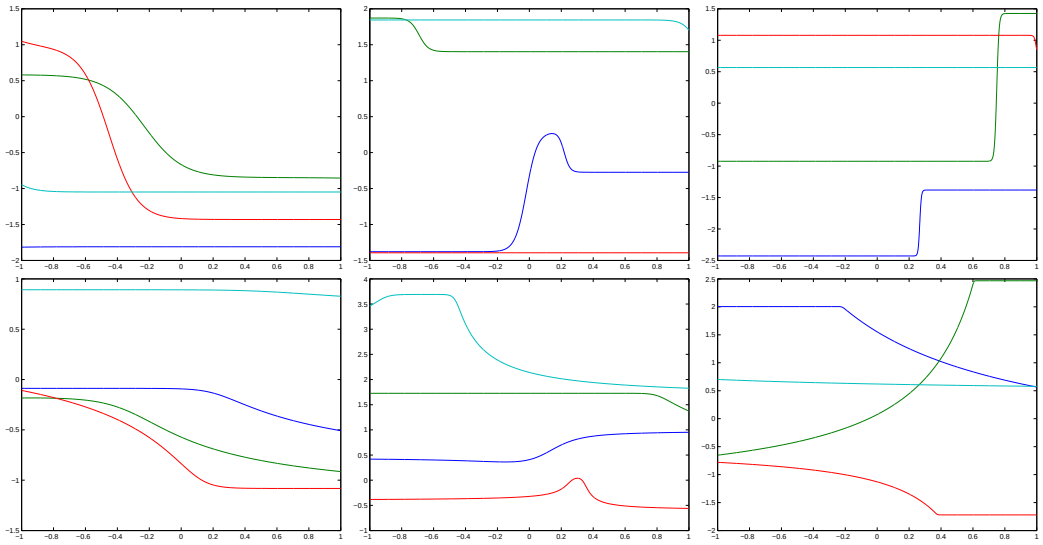


Figure 8.1 – Functions generated by an ordinary neural network (top) and an affine neural network (bottom) with 2 hidden units and $\sigma_u = 5$ (left), $\sigma_u = 20$ (mid) and $\sigma_u = 100$ (right) ($\sigma_a = \sigma_u$ and $\sigma_b = w_v = 1$).

priors over the functions show a very specific trend : when σ_u grows, the ordinary ($\tanh$) neural network tends to saturate whereas the affine neural network does not. This can easily be explained by the fact that, if $|V_i \cdot \tilde{x}|$ and $|V_{i+1} \cdot \tilde{x}|$ are both much greater than 1, hidden unit i stays in the saturation zone when V is piecewise constant (ordinary neural network). However, with a piecewise affine V, if $V_i \cdot \tilde{x}$ is positive and $V_{i+1} \cdot \tilde{x}$ is negative (or the opposite), we will go through the non-saturated zone, in which gradients on V_i and V_{i+1} flow well. This might explain the apparently easier optimization of input-to-hidden weights in the case of the affine network, even though their value is large. The result seems to be that affine neural networks usually find much better minima.

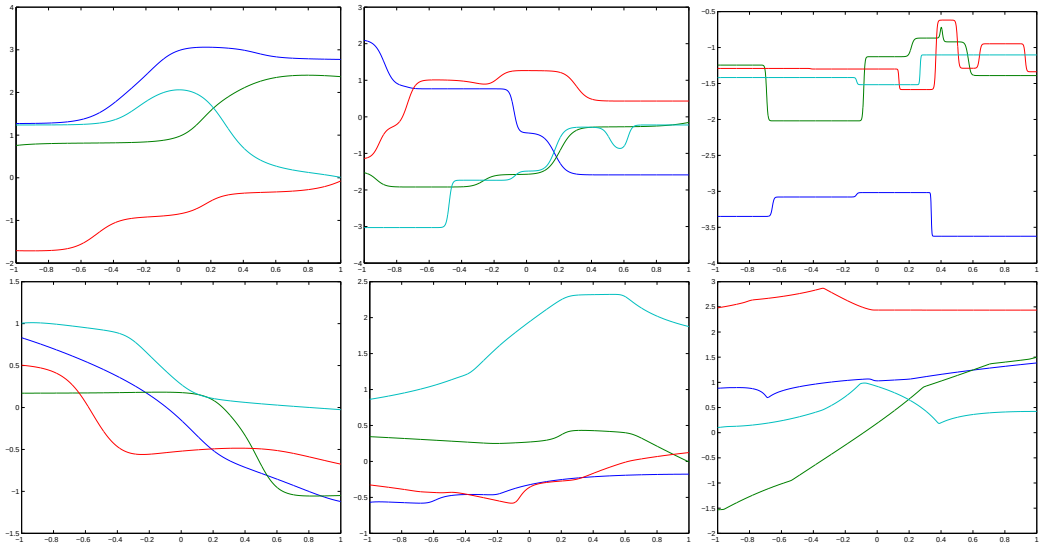


Figure 8.2 – Functions generated by an ordinary neural network (top) and an affine neural network (bottom) with 10 hidden units and $\sigma_u = 5$ (left), $\sigma_u = 20$ (mid) and $\sigma_u = 100$ (right) ($\sigma_a = \sigma_u$ and $\sigma_b = w_v = 1$).

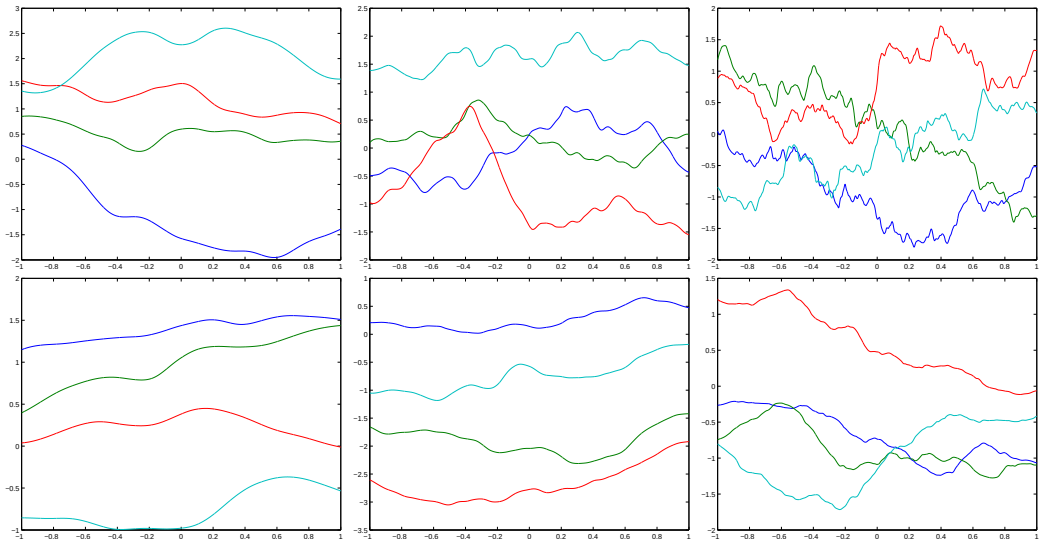


Figure 8.3 – Functions generated by an ordinary neural network (up) and an affine neural network (down) with 10000 hidden units and $\sigma_u = 5$ (left), $\sigma_u = 20$ (mid) and $\sigma_u = 100$ (right) ($\sigma_a = \sigma_u$ and $\sigma_b = w_v = 1$).

8.3.7 Experiments

In these experiments the objective was to demonstrate that, without having more capacity than ordinary neural networks (with the same number of degrees of freedom) the affine neural networks are able to obtain better fits to the data. Future experimental research should include thorough experiments to test their generalization ability.

On the LETTERS dataset, we tried some sets of hyperparameters without beating the state-of-the-art results of the ordinary neural networks in generalization error. The cause might be a lack of optimization of the hyperparameters or an intrinsic trend of the affine neural networks to overfit. However, to check this second possibility, we measured training performance on random datasets, to make sure that the good performance on real training data was not due to a larger capacity.

In all the experiments we use the conjugate gradients algorithm to perform the optimization, using Rasmussen's publically available matlab code [66].

8.3.7.1 USPS Dataset

The USPS dataset is a digit images dataset obtained from the UCI repository, consisting of 7291 training examples and 1007 test examples in dimension 256, from 10 digit classes.

We tried to minimize the average logarithm of the correct class posterior (Negative Log-Likelihood or NLL) on the training set with only 7 hidden neurons ($h = 7$). There was no explicit regularization of input-to-hidden nor hidden-to-output parameters, since the goal here was only to study the optimization with the function class. The training error curves are shown in figure 8.4. For the affine net, only the first 97000 iterations are shown (instead of 100000) because the optimization stopped at that point, having reached a local minimum.

We can clearly see that after a few iterations during which both algorithms are similar, the affine neural net optimization is much faster, and goes much lower.

So far, three explanations of that phenomenon come to mind. The first is that, as said previously, a continuous piecewise affine function is much more efficient to ap-

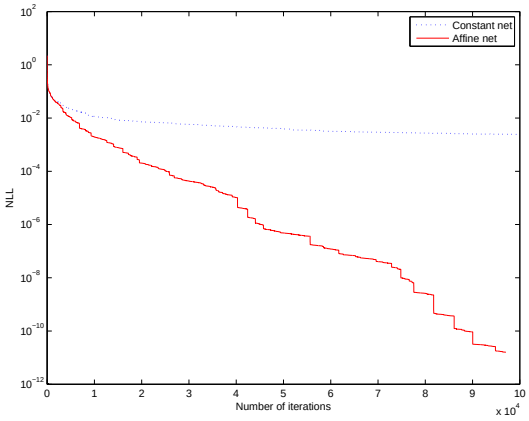


Figure 8.4 – Training curves for both algorithms on the USPS dataset with 7 hidden units.

proximate the particular function of interest. The second explanation is that the affine neural network is much less sensitive to the problem of saturation of hidden units observed in ordinary neural networks (which may correspond to a plateau rather than a local minimum in the optimization). A third explanation is that the affine network has more capacity in spite of having the same number of degrees of freedom, but the experiments below appear to disprove this hypothesis.

8.3.7.2 LETTERS Dataset

The LETTERS dataset consists of 20000 examples (16000 training examples and 4000 testing examples) in dimension 16, from 26 classes.

The difference between the two algorithms is here less obvious than with the USPS dataset. With 50 hidden units, there seems to be a clear lead of the affine net. However, with 100 hidden units, the results are much closer, with a slight lead for the standard networks. Given the very low error rate, it seems reasonable to think that both architecture learned perfectly the training set, the difference being caused by rounding errors.

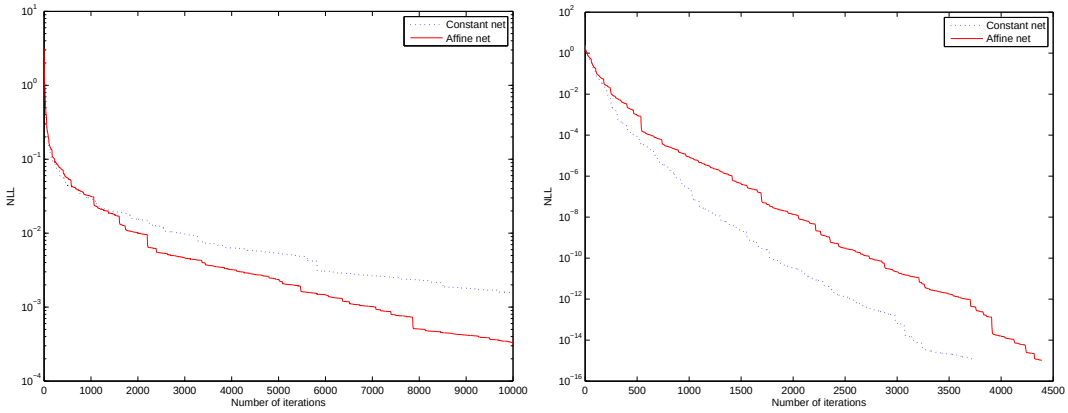


Figure 8.5 – Training curves for both algorithms on the LETTERS dataset with 50 and 100 hidden units.

8.3.7.3 Random Dataset

We wanted to verify here that the affine network is not performing better simply because it has more capacity. A way to evaluate capacity is the maximum size of a randomly labeled training set that an algorithm is able to fit.

For this purpose, we created two types of random datasets to test the affine neural network :

- a totally random dataset, where both the inputs and the targets are initialized from a uniform prior (with a mean squared error training criterion),
- a dataset where the inputs have been taken from a real dataset (USPS) and the targets have been initialized uniformly over classes (with a NLL training criterion).

We can see from the results in figure 8.6 that, on both datasets, the affine net performed slightly worse than the ordinary net. Even though only two experiments are shown, more have been performed, always with the same result (the graphs show the last two experiments we ran and have in no case been selected from a set of results). These results clearly indicate that the affine network does not have more capacity than an ordinary neural network.

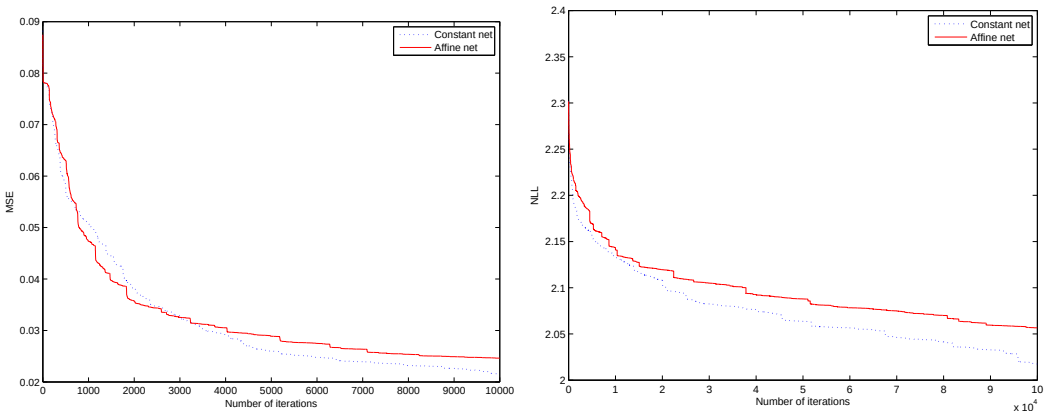


Figure 8.6 – Training curves for both algorithms on random datasets. Left : 196 random inputs and 5 random targets, using networks with 10 hidden units and the MSE. Right : Inputs of the USPS dataset and 1 random label, using networks with 7 hidden units and the NLL.

8.4 Non-Parametric Continous Neural Networks

This section returns on the strong link between kernel methods and continuous neural networks, first presented by Neal [60]. It also exhibits a clear connection with Gaussian processes, with a newly motivated kernel formula. Here, we start from eq. 8.2 but use as an index z the elements of $\mathbb{R}^{d+1}$ themselves, i.e. V is completely free and fully non-parametric : we integrate over all possible weight vectors.

To make sure that the integral exists, we select a set E over which to integrate, so that the formulation becomes

$$f(x) = \int_E a(z)g(x \cdot z)\, dz \tag{8.9}$$

$$= < a, g_x > \tag{8.10}$$

with $<,>$ the usual dot product of $\mathcal{L}^2(E)$ and g_x the function such that $g_x(z) = g(x \cdot z)$.

8.4.1 $\mathcal{L}^1$-norm Output Weights Regularization

Although the optimization problem becomes convex when the $\mathcal{L}^1$-norm of a is penalized, it involves an infinite number of variables. However, we are guaranteed to obtain a finite number of hidden units with non-zero output weight, and both exact and approximate algorithms have been proposed for this case in [16]. Since this case has already been well treated in that paper, we focus here on the $\mathcal{L}^2$ regularization case.

8.4.2 $\mathcal{L}^2$-norm Output Weights Regularization

In some cases, we know that the optimal function a can be written as a linear combination of the g_{x_i} with the x_i's the training examples. For example, when the cost function is of the form $c\left((x_1, y_1, f(x_1)), ..., (x_m, y_m, f(x_m))\right) + \Omega(\|f\|_{\mathcal{H}})$ with $\|f\|_{\mathcal{H}}$ is the norm induced by the kernel k defined by $k(x_i, x_j) =< g_{x_i}, g_{x_j} >$, we can apply the representer theorem [47].

It has been known for a decade that, with Gaussian priors over the parameters, a neural network with a number of hidden units growing to infinity converges to a Gaussian process (chapter 2 of [60]). However, Neal did not compute explicitly the covariance matrices associated to specific neural network architectures. Such covariance functions have already been analytically computed [83], for the cases of sigmoid and Gaussian transfer functions. However, this has been done using a Gaussian prior on the input-to-hidden weights. The formulation presented here corresponds to a uniform prior (i.e. with no arbitrary preference for particular values of the parameters) when the transfer function is sign. The sign function has been used in [60] with a Gaussian prior on the input-to-hidden weights, but the explicit covariance function could not be computed. Instead, approximating locally the gaussian prior with a uniform prior, Neal ended up with a covariance function of the form $k(x, y) \approx A - B||x - y||$. We will see that, using a uniform prior, this is exactly the form of C one obtains.

8.4.3 Kernel when g is the sign Function

8.4.3.1 Computation of the Kernel

For the sake of shorter notation, we will denote the sign function by s and warn the reader not to get confused with the $sigmoid$ function.

We wish to compute

$$k(x,y) =< g_x, g_y >= E_{v,b}\left[s(v\cdot x+b)s(v\cdot y+b)\right].$$

Since we wish to define a uniform prior over v and b, we cannot let them span the whole space ($\mathbb{R}^n$ in the case of v and $\mathbb{R}$ in the case of b). However, the value of the function sign does not depend on the norm of its argument, so we can restrict ourselves to the case where $\|v\| = 1$. Furthermore, for values of b greater than δ, where δ is the maximum norm among the samples, the value of the sign will be constant to 1 (and -1 for opposite values of b). Therefore, we only need to integrate b on the range $[-\delta, \delta]$.

Defining a uniform prior on an interval depending on the training examples seems contradictory. We will see later that, as long as the interval is big enough, its exact value does not matter.

Let us first consider v fixed and compute the expectation over b. The product of two sign functions is equal to 1 except when the argument of one sign is positive and the other negative. In our case, this becomes :

$$\begin{cases} v\cdot x+b & < & 0 \\ v\cdot y+b & > & 0 \end{cases} \text{ or } \begin{cases} v\cdot x+b & > & 0 \\ v\cdot y+b & < & 0 \end{cases}$$

which is only true for b between $-\min(v\cdot x, v\cdot y)$ and $-\max(v\cdot x, v\cdot y)$, which is an interval of size $|v\cdot x - v\cdot y| = |v\cdot(x-y)|$.

Therefore, for each v, we have

$$\begin{aligned} E_b\left[s(v\cdot x+b)s(v\cdot y+b)\right] &= \frac{(2\delta-2\left|v\cdot(x-y)\right|)}{2\delta} \\ &= 1-\frac{|v\cdot(x-y)|}{\delta}. \end{aligned}$$

We must now compute

$$k(x,y)=1-E_v\left[\frac{|v\cdot(x-y)|}{\delta}\right] \tag{8.11}$$

It is quite obvious that the value of the second term only depends on the norm of $(x-y)$ due to the symmetry of the problem. The value of the kernel can thus be written

$$k(x,y)=1-C\|x-y\| \tag{8.12}$$

Writing $k(x,y)=1-C\|x-y\|$, we have

$$\begin{aligned} C &= \frac{1}{\delta S(d,1)}\int_{v/\|v\|=1}\left|v\cdot\frac{(x-y)}{\|x-y\|}\right|\,dv \\ &= \frac{2}{\delta S(d,1)}\int_0^{\pi/2}\cos(\theta)S\left(d-1,\sin(\theta)\right)\,d\theta \\ &= \frac{2S(d-1,1)}{\delta S(d,1)(d-1)} \end{aligned}$$

where $S(d,r)$ is the surface of the hypersphere S_d of radius r and d is the dimensionality of the data.

Therefore,

$$k(x,y)=1-\frac{2S(d-1,1)}{\delta S(d,1)(d-1)}\|x-y\|$$

Then, noticing that

$$\sqrt{\frac{d-1}{2\pi}}<\frac{S(d-1,1)}{S(d,1)}<\sqrt{\frac{d}{2\pi}}$$

, we have

$$\frac{1}{\delta}\sqrt{\frac{2}{\pi(d-1)}}<\frac{2S(d-1,1)}{\delta S(d,1)(d-1)}<\frac{\sqrt{2d}}{\delta(d-1)\pi}$$

As we could have integrated over a slightly larger integral than the minimum required, we can set our kernel function to be

$$k(x, y) = 1 - \frac{\sqrt{2}}{\delta\sqrt{(d-1)\pi}}\|x - y\| \tag{8.13}$$

with d the dimensionality of the data and δ the maximum $\mathcal{L}^2$-norm among the samples. The coefficient in front of the term $\|x - y\|$ has a slightly different form when $d = 1$ or $d = 2$.

8.4.3.2 Function sampling

As presented in [60], the functions generated by this gaussian process are Brownian (see figure 8.7). Contrary to other transfer functions and frameworks, this covariance function does not have any hyperparameter. Indeed, the solution is invariant if the covariance function is modified by adding a constant to $k(x, y)$, since the optimal solution for linear or logistic regression is a weight vector whose elements sum to 0. *Moreover, changing the covariance function by an additive factor is equivalent to accordingly multiply the weight decay on the output weights. It is also possible to set a weight decay equal to 0. The problems becomes ill-defined as many vectors are solutions. However, all solutions lead to the same function. This setting suppresses the remaining hyperparameter for the* sign *kernel.*

8.4.3.3 USPS dataset

We tried this new hyper-parameter free kernel machine on the USPS dataset, with quadratic training cost.

To prove the stability of the sign kernel, we compared it with the Gaussian kernel on the USPS dataset. We optimized the hyperparameters of the Gaussian kernel on the test set (optimization on the validation set yields 4.0% test error). As there are no hyperparameters for the sign kernel, this clearly is in favor of the Gaussian kernel. We can see that the Gaussian kernel is much more sensitive to hyperparameters, whereas the

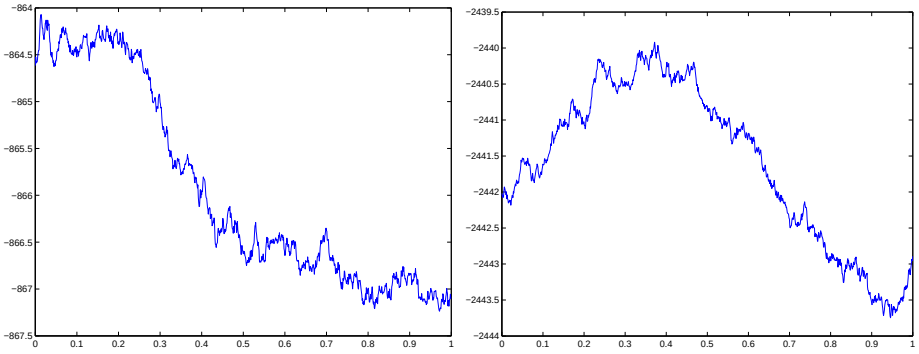


Figure 8.7 – Two functions drawn from the Gaussian process associated to the above kernel function.

Algorithm	$\lambda = 10^{-3}$	$\lambda = 10^{-6}$	$\lambda = 0$
sign	4.4687	4.1708	4.1708
gauss. $\sigma = 1$	62.86	62.7607	57.6961
gauss. $\sigma = 2$	14.2999	15.5909	6.852
gauss. $\sigma = 3$	4.0715	4.3694	4.2701
gauss. $\sigma = 4$	3.9722	4.7666	4.9652
gauss. $\sigma = 5$	3.6743	23.436	59.484

Tableau 8.1 – sign kernel *vs* Gaussian kernel on USPS dataset, with different Gaussian widths σ and weight decays λ.

performance of the sign kernel is the same for λ varying from 10^{-4} to 0.

8.4.4 Conclusions, Discussion, and Future Work

We have studied in detail two formulations of uncountable neural networks, one based on a finite parametrization of the input-to-hidden weights, and one that is fully non-parametric. The first approach delivered a number of interesting results : a new function approximation theorem, an affine parametrization in which the integrals can be computed analytically, an error bound theorem that suggests better approximation properties than ordinary neural networks, and surprising experimental results suggesting that this function class is less prone to the optimization difficulties of ordinary neural networks.

As shown in theorem 8.3.1, function V can be represented as $d+1$ functions from $\mathbb{R}$ to $\mathbb{R}$, easier to learn than one function from $\mathbb{R}^{d+1}$ to $\mathbb{R}$. We did not find parametrizations of those functions other than the continuous piecewise affine one with the same feature of analytic integration. To obtain smooth functions V with restricted complexity, one could set the functions V to be outputs of another neural network taking a discrete index in argument. However, this has not yet been exploited and will be explored in the future.

The second, non-parametric, approach delivered another set of interesting results : with sign activation functions, the integrals can be computed analytically, and correspond to a hyperparameter-free kernel machine that yields performances comparable to the Gaussian kernel. These results raise a fascinating question : why are results with the sign kernel that good with no hyper-parameter and no regularization ? To answer this, we should look at the shape of the covariance function $k(x,y) = 1 - C\|x-y\|$, which suggests the following property : it can discriminate between neighbors of a training example while being influenced by remote examples, whereas the Gaussian kernel does either one or the other, depending on the choice of σ.

CHAPITRE 9

PRÉSENTATION DU QUATRIÈME ARTICLE

9.1 Détails de l'article

Representational Power of Restricted Boltzmann Machines and Deep Belief Networks

N. Le Roux et Y. Bengio

À paraître dans *Neural Computation, 20 :6, MIT Press, Cambridge, MA* en 2008.

9.2 Contexte

Toutes les limitations des architectures peu profondes présentées dans les articles précédents encouragèrent la recherche d'architectures réellement capables d'extraire des concepts de haut niveau et insensibles au fléau de la dimensionnalité. De telles architectures existaient déjà depuis 1989 [52] dans le contexte de la reconnaissance de caractères.

En 2006, Hinton et al. [39] proposèrent une méthode gloutonne pour entraîner des réseaux profonds, et ce avec succès. Les Deep Belief Networks, ou DBN, obtinrent des résultats à l'état de l'art sur plusieurs ensembles de données, que ce soit en reconnaissance de caractères ou en classification de documents. Contrairement aux réseaux à convolution, les réseaux sigmoïdaux profonds ne sont pas limités à une tâche particulière et n'incluent pas de connaissance a priori sur le type de problème à résoudre.

9.2.1 Approches précédentes

9.2.1.1 Entraînement global

Les premières tentatives d'entraînement de réseaux profonds se firent en utilisant la descente de gradient sur les paramètres de toutes les couches à la fois. Hormis pour certaines architectures très précises [52], cette méthode tombait rapidement dans de mauvais minima locaux. Deux raisons peuvent être invoquées pour cela.

La première est l'estompement du gradient. En effet, la présence d'une fonction de transfert écrasante à chaque couche cachée diminue l'intensité du gradient au fur et à mesure que l'on s'éloigne de la couche de sortie.

La deuxième est l'extrême complexité des relations entre les paramètres dès que l'on s'éloigne de la couche de sortie. En effet, les poids de sortie agissent directement sur la sortie de la fonction et n'ont que peu d'interactions. Dès lors que l'on parcourt les couches cachées, les paramètres proches de la couche d'entrée interagissent au travers des neurones cachées, rendant leur optimisation ardue.

Dans les deux cas, ce sont les paramètres les plus éloignés de la sortie (c'est-à-dire de l'origine du signal de renforcement) qui sont les plus difficiles à optimiser. Cela motive la création de méthodes d'entraînement n'optimisant que les paramètres proches de la sortie.

9.2.1.2 Entraînement glouton

Des approches gloutonnes avaient été essayées auparavant pour entraîner des réseaux profonds, mais sans succès. Elles consistaient en l'entraînement d'un réseau de neurones à une couche cachée sur la tâche (supervisée) demandée [13]. Les poids d'entrée étaient alors fixés et les activations de la couche cachée obtenues en mettant les exemples d'entraînement en entrée devenaient les nouveaux exemples d'entraînement. Un nouveau réseau de neurones à une couche cachée était alors entraîné sur ces exemples, et ainsi de suite jusqu'à atteindre la profondeur voulue.

Malheureusement, cette technique souffre de la facilité de l'apprentissage supervisé sur l'ensemble d'apprentissage. En effet, si le nombre d'unités cachées est assez grand, il est facile d'obtenir une erreur d'entraînement faible sur l'ensemble d'apprentissage, et ce quels que soient les poids d'entrée. Autrement dit, un réseau de neurones à une couche cachée n'a pas besoin d'extraire des caractéristiques pertinentes des données pour parvenir à une faible erreur d'apprentissage. Cela a pour conséquence de jeter une grande partie de l'information contenue dans l'ensemble d'entraînement lors du passage à la première couche cachée. Ce problème est bien entendu amplifié à chaque ajout d'une nouvelle couche.

Au contraire, les DBNs utilisent comme blocs de base des algorithmes non-supervisés, les machines de Boltzmann restreintes (Restricted Boltzmann Machine, ou RBM). L'idée fondatrice est qu'un algorithme non-supervisé va tenter de conserver toute l'information contenue dans l'ensemble d'apprentissage et non seulement celle nécessaire à la résolution du problème supervisé. L'avantage immédiat de cette méthode est la conservation de l'information contenue dans les exemples d'apprentissage lors de l'ajout de couches. Il est d'ailleurs intéressant de remarquer que d'autres méthodes proposant d'autres algorithmes non-supervisés pour entraîner les couches ont obtenu des résultats comparables [49].

Cette méthode possède toutefois un inconvénient majeur : avant la phase finale de raffinement des paramètres, l'entraînement n'est aucunement guidé par la tâche supervisée à réaliser. Cela devient problèmatique dès lors que seul le problème supervisé peut être (approximativement) résolu en maintenant un modèle de taille raisonnable. Larochelle et al. [49] tentèrent d'introduire un signal supervisé au cours de l'entraînement des paramètres, améliorant quelque peu la performance finale.

9.3 Commentaires

L'article présenté ici s'intéresse exclusivement au modèle génératif et ne tient pas compte d'un éventuel problème supervisé. Il démontre de manière constructive (contrairement à [32], qui spécifient simplement que la machine de Boltzmann restreinte doit agir comme une table de conversion, chaque unité cachée étant associée à un vecteur d'entrée) l'universalité des machines de Boltzmann restreintes et propose une méthode d'entraînement moins gloutonne que celle proposée par Hinton et al. [39]. Si cette dernière permet d'obtenir de meilleurs résultats que la méthode initialement proposée par Hinton et al. [39] sur des problèmes jouets, sa complexité la rend inexploitable pour des problèmes réels.

En montrant que les machines de Boltzmann restreintes peuvent modéliser n'importe quelle distribution et que leur puissance de représentation augmente strictement à l'ajout d'une nouvelle unité cachée, cet article les relie plus fortement aux réseaux de neurones

habituels. En outre, la stratégie d'optimisation proposée montre qu'il est théoriquement possible d'effectuer un apprentissage efficace tout en étant moins glouton. Si l'algorithme proposé dans cet article n'est pas exploitable en l'état, il n'en demeure pas moins une source d'inspiration possible pour de futurs développements.

CHAPITRE 10

REPRESENTATIONAL POWER OF RESTRICTED BOLTZMANN MACHINES AND DEEP BELIEF NETWORKS

10.1 Abstract

Deep Belief Networks (DBN) are generative neural network models with many layers of hidden explanatory factors, recently introduced by Hinton et al., along with a greedy layer-wise unsupervised learning algorithm. The building block of a DBN is a probabilistic model called a Restricted Boltzmann Machine (RBM), used to represent one layer of the model. Restricted Boltzmann Machines are interesting because inference is easy in them, and because they have been successfully used as building blocks for training deeper models. We first prove that adding hidden units yields strictly improved modelling power, while a second theorem shows that RBMs are universal approximators of discrete distributions. We then study the question of whether DBNs with more layers are strictly more powerful in terms of representational power. This suggests a new and less greedy criterion for training RBMs within DBNs.

10.2 Introduction

Learning algorithms that learn to represent functions with many levels of composition are said to have a *deep architecture*. Bengio et Le Cun [15] discuss results in computational theory of circuits that strongly suggest that deep architectures are much more efficient in terms of representation (number of computational elements, number of parameters) than their shallow counterparts. 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 [44]), they may need a huge number of elements and, consequently, of training examples. For example, the parity function on d bits (which associates the value 1 with a vector $\mathbf{v}$ if $\mathbf{v}$ has an odd number of bits equal to 1 and 0 otherwise) can be implemented by a digital circuit of depth $\log(d)$ with $O(d)$

elements but requires $O(2^d)$ elements to be represented by a 2-level digital circuit [2] (e.g., in conjunctive or disjunctive normal form). We proved a similar result for Gaussian kernel machines : they require $O(2^d)$ non-zero coefficients (i.e., support vectors in a Support Vector Machine) to represent such highly varying functions [10]. On the other hand, training learning algorithms with a deep architecture (such as neural networks with many hidden layers) appears to be a challenging optimization problem [13, 80].

Hinton et al. [39] introduced a greedy layer-wise *unsupervised* learning algorithm for Deep Belief Networks (DBN). The training strategy for such networks may hold great promise as a principle to help address the problem of training deep networks. Upper layers of a DBN are supposed to represent more "abstract" concepts that explain the input data whereas lower layers extract "low-level features" from the data. In [13, 65], this greedy layer-wise principle is found to be applicable to models other than DBNs. DBNs and RBMs have already been applied successfully to a number of classification, dimensionality reduction, information retrieval, and modelling tasks [13, 39, 42, 72, 82].

In this paper we show that adding hidden units yields strictly improved modelling power, unless the RBM already perfectly models the data. Then, we prove that an RBM can model any discrete distribution, a property similar to those of neural networks with one hidden layer. Finally, we discuss the representational power of DBNs and find a puzzling result about the best that could be achieved when going from 1-layer to 2-layer DBNs. Note that the proofs of universal approximation by RBMs are constructive but these constructions are not practical as they would lead to RBMs with potentially as many hidden units as examples, and this would defy the purpose of using RBMs as building blocks of a deep network that efficiently represents the input distribution. Important theoretical questions therefore remain unanswered concerning the potential for DBNs that stack multiple RBMs to represent a distribution efficiently.

10.2.1 Background on RBMs

10.2.1.1 Definition and properties

A Restricted Boltzmann Machine (RBM) is a particular form of the Product of Experts model [40, 41] which is also a Boltzmann Machine [1] with a bipartite connectivity graph. An RBM with n hidden units is a parametric model of the joint distribution between hidden variables h_i (explanatory factors, collected in vector $\mathbf{h}$) and observed variables v_j (the examples, collected in vector $\mathbf{v}$), of the form

$$p(\mathbf{v}, \mathbf{h}) \propto \exp(-E(\mathbf{v}, \mathbf{h})) = e^{\mathbf{h}^T W \mathbf{v} + b^T \mathbf{v} + c^T \mathbf{h}}$$

with parameters $\theta = (W, b, c)$ and $v_j, h_i \in \{0, 1\}$. $E(\mathbf{v}, \mathbf{h})$ is called the **energy** of the state $(\mathbf{v}, \mathbf{h})$. We consider here the simpler case of binary units. It is straightforward to show that $P(\mathbf{v}|\mathbf{h}) = \prod_j P(v_j|\mathbf{h})$ and $P(v_j = 1|\mathbf{h}) = \text{sigmoid}(b_j + \sum_i W_{ij} h_i)$ (where sigmoid is the sigmoid function defined as $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$), and $P(\mathbf{h}|\mathbf{v})$ has a similar form : $P(\mathbf{h}|\mathbf{v}) = \prod_i P(h_i|\mathbf{v})$ and $P(h_i = 1|\mathbf{v}) = \text{sigmoid}(c_i + \sum_j W_{ij} v_j)$. Although the marginal distribution $p(\mathbf{v})$ is not tractable, it can be easily computed up to a normalizing constant. Furthermore, one can also sample from the model distribution using Gibbs sampling. Consider a Monte-Carlo Markov chain (MCMC) initialized with $\mathbf{v}$ sampled from the empirical data distribution (distribution denoted p_0). After sampling $\mathbf{h}$ from $P(\mathbf{h}|\mathbf{v})$, sample $\mathbf{v}'$ from $P(\mathbf{v}'|\mathbf{h})$, which follows a distribution denoted p_1. After k such steps we have samples from p_k, and the model's generative distribution is p_∞ (due to convergence of the Gibbs MCMC).

10.2.1.2 Training and Contrastive Divergence

We will denote

$$\begin{aligned} L(\theta) &= \frac{1}{N} \sum_{i=1}^{N} \log p\left(\mathbf{v}^{(i)}\right) \\ E(\mathbf{v}, \theta) &= -\log \left[\sum_h \exp(-E(\mathbf{v}, \mathbf{h}, \theta)) \right] \end{aligned}$$

Carreira-Perpiñan et Hinton [25] showed that the derivative of the average log-likelihood of the data under the RBM with respect to the parameters is

$$\frac{\partial \log L(\theta)}{\partial \theta} = -\left\langle \frac{\partial \log E(\mathbf{v}, \theta)}{\partial \theta} \right\rangle_0 + \left\langle \frac{\partial \log E(\mathbf{v}, \theta)}{\partial \theta} \right\rangle_\infty \tag{10.1}$$

where averaging is over both $\mathbf{v}$ and $\mathbf{h}$, $\langle \cdot \rangle_0$ denotes an average with respect to p_0 (the data distribution) multiplied by $P(\mathbf{h}|\mathbf{v})$, and $\langle \cdot \rangle_\infty$ denotes an average with respect to p_∞ (the model distribution) : $p_\infty(\mathbf{v}, \mathbf{h}) = p(\mathbf{v}, \mathbf{h})$.

Since computing the average over the true model distribution is intractable, Hinton et al. [39] use an approximation of that derivative called **contrastive divergence** [40, 41] : one replaces the average $\langle \cdot \rangle_\infty$ with $\langle \cdot \rangle_k$ for relatively small values of k. For example, in Bengio et al. [13], Hinton et al. [39], Hinton et Salakhutdinov [42], Salakhutdinov et Hinton [72], one uses $k = 1$ with great success. The average over $\mathbf{v}$'s from p_0 is replaced by a sample from the empirical distribution (this is the usual stochastic gradient sampling trick) and the average over $\mathbf{v}$'s from p_1 is replaced by a single sample from the Markov chain. The resulting gradient estimator involves only very simple computations, and for the case of binary units, the gradient estimator on weight W_{ij} is simply $P(h_i = 1|\mathbf{v})v_j - P(h_i = 1|\mathbf{v}')v'_j$, where $\mathbf{v}'$ is a sample from p_1 and $\mathbf{v}$ is the input example that starts the chain. The procedure can easily be generalized to input or hidden units that are not binary (e.g., Gaussian or exponential, for continuous-valued units [13, 82]).

10.3 RBMs are Universal Approximators

We will now prove that RBMs with a data-selected number of hidden units become non-parametric and possess universal approximation properties relating them closely to classical multilayer neural networks, but in the context of probabilistic unsupervised learning of an input distribution.

10.3.1 Better Model with Increasing Number of Units

We show below that when the number of hidden units of an RBM is increased, there are weight values for the new units that guarantee improvement in the training log-likelihood or equivalently in the KL divergence between the data distribution p_0 and the model distribution $p_\infty = p$. These are equivalent since

$$KL(p_0||p) = \sum_{\mathbf{v}} p_0(\mathbf{v}) \log \frac{p_0(\mathbf{v})}{p(\mathbf{v})} = -H(p_0) - \frac{1}{N} \sum_{i=1}^{N} \log p(\mathbf{v}^{(i)})$$

when p_0 is the empirical distribution, with $\mathbf{v}^{(i)}$ the i^{th} training vector and N the number of training vectors.

Consider the objective of approximating an arbitrary distribution p_0 with an RBM. Let p denote the distribution over visible units $\mathbf{v}$ obtained with an RBM that has n hidden units and $p_{w,c}$ denote the input distribution obtained when adding a hidden unit with weights w and bias c to that RBM. The RBM with this extra unit has the same weights and biases for all other hidden units, and the same input biases.

Lemma 10.3.1. *Let R_p be the equivalence class containing the RBMs whose associated marginal distribution over the visible units is p. The operation of adding a hidden unit to an RBM of R_p preserves the equivalence class. Thus, the set of RBMs composed of an RBM of R_p and an additional hidden unit is also an equivalence class (meaning that all the RBMs of this set have the same marginal distribution over visible units).*

Proof in section 10.6.1.

R_p will be used here to denote any RBM in this class. We also define $R_{p_{w,c}}$ as the set of RBMs obtained by adding a hidden unit with weight w and bias c to an RBM from R_p and $p_{w,c}$ the associated marginal distribution over the visible units. As demonstrated in the above lemma, this does not depend on which particular RBM from R_p we choose.

We then wish to prove, that, regardless p and p_0, if $p \neq p_0$, there exists a pair (w, c) such that $KL(p_0||p_{w,c}) < KL(p_0||p)$, i.e., that one can improve the approximation of p_0 by inserting an extra hidden unit with weight vector w and bias c.

We will first state a trivial lemma needed for the rest of the proof. It says that inserting a unit with bias $c = -\infty$ does not change the input distribution associated with the RBM.

Lemma 10.3.2. *Let p be the distribution over binary vectors $\mathbf{v}$ in $\{0,1\}^d$, obtained with an RBM R_p and let $p_{w,c}$ be the distribution obtained when adding a hidden unit with weights w and bias c to R_p. Then*

$$\forall p, \forall w \in \mathbb{R}^d, p = p_{w,-\infty}$$

Démonstration. Denoting $\widetilde{\mathbf{h}} = \begin{bmatrix} \mathbf{h} \\ h_{n+1} \end{bmatrix}$, $\widetilde{W} = \begin{bmatrix} W \\ w^T \end{bmatrix}$ and $\widetilde{C} = \begin{bmatrix} C \\ c \end{bmatrix}$ where w^T denotes the transpose of w and introducing $z(\mathbf{v}, \mathbf{h}) = \exp(\mathbf{h}^T W \mathbf{v} + B^T \mathbf{v} + C^T \mathbf{h})$, we can express $p(\mathbf{v}, \mathbf{h})$ and $p_{w,c}(\mathbf{v}, \widetilde{\mathbf{h}})$ as follows :

$$\begin{aligned} p(\mathbf{v}, \mathbf{h}) &\propto z(\mathbf{v}, \mathbf{h}) \\ p_{w,c}(\mathbf{v}, \widetilde{\mathbf{h}}) &\propto \exp\left(\widetilde{\mathbf{h}}^T \widetilde{W} \mathbf{v} + B^T \mathbf{v} + \widetilde{C}^T \widetilde{\mathbf{h}}\right) \\ &\propto z(\mathbf{v}, \mathbf{h}) \exp\left(h_{n+1} w^T \mathbf{v} + c h_{n+1}\right) \end{aligned}$$

If $c = -\infty$, $p_{w,c}(\mathbf{v}, \widetilde{\mathbf{h}}) = 0$ if $h_{n+1} = 1$. Thus, we can discard all terms where $h_{n+1} = 1$, keeping only those where $h_{n+1} = 0$. Marginalizing over the hidden units, we have :

$$\begin{aligned} p(\mathbf{v}) &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{h}^{(0)}, \mathbf{v}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)})} \\ p_{w,-\infty}(\mathbf{v}) &= \frac{\sum_{\widetilde{\mathbf{h}}} z(\mathbf{v}, \mathbf{h}) \exp\left(h_{n+1} w^T \mathbf{v} + c h_{n+1}\right)}{\sum_{\widetilde{\mathbf{h}^{(0)}}, \mathbf{v}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)}) \exp\left(h^{(0)}_{n+1} w^T \mathbf{v} + c h^{(0)}_{n+1}\right)} \\ &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h}) \exp(0)}{\sum_{\mathbf{h}^{(0)}, \mathbf{v}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)}) \exp(0)} \\ &= p(\mathbf{v}) \end{aligned}$$

□

We now state the main theorem.

Theorem 10.3.3. *Let p_0 be an arbitrary distribution over $\{0,1\}^n$ and let R_p be an RBM with marginal distribution p over the visible units such that $KL(p_0||p) > 0$. Then there exists an RBM $R_{p_{w,c}}$ composed of R_p and an additional hidden unit with parameters (w, c) whose marginal distribution $p_{w,c}$ over the visible units achieves $KL(p_0||p_{w,c}) < KL(p_0||p)$.*

Proof in section 10.6.2.

10.3.2 A Huge Model can Represent Any Distribution

The second set of results are for the limit case when the number of hidden units is very large, so that we can represent any discrete distribution exactly.

Theorem 10.3.4. *Any distribution over $\{0,1\}^n$ can be approximated arbitrarily well (in the sense of the KL divergence) with an RBM with $k+1$ hidden units where k is the number of input vectors whose probability is not 0.*

Proof sketch (Universal approximator property). We constructively build an RBM with as many hidden units as the number of input vectors whose probability is strictly positive. Each hidden unit will be assigned to one input vector. Namely, when $\mathbf{v}_i$ is the visible units vector, all hidden units have a probability 0 of being on except the one corresponding to $\mathbf{v}_i$ which has a probability $\mathrm{sigmoid}(\lambda_i)$ of being on. The value of λ_i is directly tied with $p(\mathbf{v}_i)$. On the other hand, when all hidden units are off but the i^{th} one, $p(\mathbf{v}_i|\mathbf{h}) = 1$. With probability $1 - \mathrm{sigmoid}(\lambda_i)$, all the hidden units are turned off, which yields independent draws of the visible units. The proof consists in finding the appropriate weights (and values λ_i) to yield that behaviour. □

Proof in section 10.6.3.

10.4 Representational power of Deep Belief Networks

10.4.1 Background on Deep Belief Networks

A DBN with ℓ layers models the joint distribution between observed variables v_j and ℓ hidden layers $\mathbf{h}^{(k)}, k = 1, \ldots, \ell$ made of binary units $h_i^{(k)}$ (here all binary variables), as follows :

$$p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \ldots, \mathbf{h}^{(\ell)}) = P(\mathbf{v}|\mathbf{h}^{(1)})P(\mathbf{h}^{(1)}|\mathbf{h}^{(2)}) \ldots P(\mathbf{h}^{(\ell-2)}|\mathbf{h}^{(\ell-1)})p(\mathbf{h}^{(\ell-1)}, \mathbf{h}^{(\ell)})$$

Denoting $\mathbf{v} = \mathbf{h}^{(0)}$, $b^{(k)}$ the bias vector of layer k and $W^{(k)}$ the weight matrix between layer k and layer $k+1$, we have :

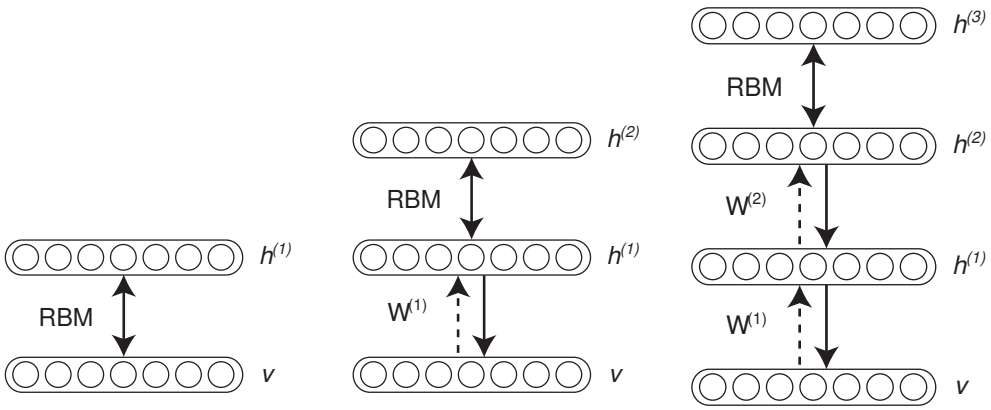
$$\begin{aligned} P(\mathbf{h}^{(k)}|\mathbf{h}^{(k+1)}) &= \prod_i P(\mathbf{h}_i^{(k)}|\mathbf{h}^{(k+1)}) \text{ (factorial conditional distribution)} \\ P(\mathbf{h}_i^{(k)} = 1|\mathbf{h}^{(k+1)}) &= \text{sigmoid}\left(b_i^{(k)} + \sum_j W_{ij}^{(k)} \mathbf{h}_j^{(k+1)}\right) \end{aligned} \tag{10.2}$$

and $p(\mathbf{h}^{(\ell-1)}, \mathbf{h}^{(\ell)})$ is an RBM.

The original motivation found in Hinton et al. [39] for having a deep network versus a single hidden layer (i.e., a DBN versus an RBM) was that the representational power of an RBM would be too limited and that more capacity could be achieved by having more hidden layers. However, we have found here that an RBM with enough hidden units can model any discrete distribution. Another motivation for deep architectures is discussed in Bengio et Le Cun [15] and Bengio et al. [13] : deep architectures can represent functions much more efficiently (in terms of number of required parameters) than shallow ones. In particular, theoretical results on circuit complexity theory prove that shallow digital circuits can be exponentially less efficient than deeper ones [2, 3, 37]. Hence the original motivation [39] was probably right when one considers the restriction to reasonably sized models.

10.4.2 Trying to Anticipate a High-Capacity Top Layer

In the greedy training procedure of Deep Belief Networks proposed in [39], one layer is added on top of the network at each stage, and only that top layer is trained (as an RBM, see figure 10.1). In that greedy phase, one does not take into account the fact that other layers will be added next. Indeed, while trying to optimize the weights, we restrict the marginal distribution over its hidden units to be the one induced by the RBM. On the contrary, when we add a new layer, that distribution (which is the marginal distribution over the visible units of the new RBM) does not have that restriction (but another one which is to be representable by an RBM of a given size). Thus, we might be able to better optimize the weights of the RBM, knowing that the marginal distribution over the hidden units will have more freedom when extra layers are added. This would lead to an alternative training criterion for DBNs.



(a) Stage 1 (b) Stage 2 (c) Stage 3

Figure 10.1 – Greedy learning of an RBM. After each RBM has been trained, the weights are frozen and a new layer is added. The new layer is trained as an RBM.

Consider a 2-layer DBN ($\ell = 2$, that is with three layers in total). To train the weights between $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$ (see figure 10.1), the greedy strategy maximizes a lower bound on the likelihood of the data (instead of the likelihood itself), called the **variational**

bound [39] :

$$\begin{aligned} \log p(\mathbf{v}) \quad \geq \quad & \sum_{\mathbf{h}^{(1)}} Q(\mathbf{h}^{(1)}|\mathbf{v}) \left[\log p(\mathbf{h}^{(1)}) + \log P(\mathbf{v}|\mathbf{h}^{(1)})\right] \\ & - \sum_{\mathbf{h}^{(1)}} Q(\mathbf{h}^{(1)}|\mathbf{v}) \log Q(\mathbf{h}^{(1)}|\mathbf{v}) \end{aligned} \tag{10.3}$$

where

- $Q(\mathbf{h}^{(1)}|\mathbf{v})$ is the posterior on hidden units $\mathbf{h}^{(1)}$ given visible vector $\mathbf{v}$, according to the first RBM model, and is determined by $W^{(1)}$. It is the assumed distribution used in the variational bound on the DBN likelihood.
- $p(\mathbf{h}^{(1)})$ is the marginal distribution over $\mathbf{h}^{(1)}$ in the DBN (thus induced by the second RBM, between $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$)
- $P(\mathbf{v}|\mathbf{h}^{(1)})$ is the posterior over $\mathbf{v}$ given $\mathbf{h}^{(1)}$ in the DBN and in the first RBM, and is determined by $W^{(1)}$.

Once the weights of the first layer ($W^{(1)}$) are frozen, the only element that can be optimized is $p(\mathbf{h}^{(1)})$. We can show that there is an analytic formulation for the distribution $p^*(\mathbf{h}^{(1)})$ that maximizes this variational bound :

$$p^*(\mathbf{h}^{(1)}) = \sum_{\mathbf{v}} p_0(\mathbf{v}) Q(\mathbf{h}^{(1)}|\mathbf{v}) \tag{10.4}$$

where p_0 is the empirical distribution of input examples. One can sample from $p^*(\mathbf{h}^{(1)})$ by first randomly sampling a $\mathbf{v}$ from the empirical distribution and then propagating it stochastically through $Q(\mathbf{h}^{(1)}|\mathbf{v})$. Using theorem 10.3.4, there exists an RBM that can approximate this optimal distribution $p^*(\mathbf{h}^{(1)})$ arbitrarily well.

Using an RBM that achieves this "optimal" $p^*(\mathbf{h}^{(1)})$ (optimal in terms of the variational bound, but not necessarily with respect to the likelihood), we can determine the distribution represented by the DBN. Let p_1 be the distribution one obtains when starting from p_0 clamped in the visible units of the lower layer ($\mathbf{v}$), sampling the hidden units $\mathbf{h}^{(1)}$ given $\mathbf{v}$ and then sampling a $\mathbf{v}$ given $\mathbf{h}^{(1)}$.

Proposition 10.4.1. *In a 2-layer DBN, using a second layer RBM achieving* $p^*(\mathbf{h}^{(1)})$,

the model distribution p is equal to p_1.

This is equivalent to making one "up-down" in the first RBM trained.

Démonstration. We can write the marginal $p^*(\mathbf{h}^{(1)})$ by summing over hidden values $\tilde{\mathbf{h}}^0$:

$$p^*(\mathbf{h}^{(1)}) = \sum_{\tilde{\mathbf{h}}^0} p_0(\tilde{\mathbf{h}}^0) Q(\mathbf{h}^{(1)}|\tilde{\mathbf{h}}^0).$$

Thus, the probability of the data under the 2-layer DBN when the top-layer RBM achieves $p^*(\mathbf{h}^{(1)})$ is

$$\begin{aligned} p(\mathbf{h}^{(0)}) &= \sum_{\mathbf{h}^{(1)}} P(\mathbf{h}^{(0)}|\mathbf{h}^{(1)}) p^*(\mathbf{h}^{(1)}) && (10.5) \\ &= \sum_{\tilde{\mathbf{h}}^0} p_0(\tilde{\mathbf{h}}^0) \sum_{\mathbf{h}^{(1)}} Q(\mathbf{h}^{(1)}|\tilde{\mathbf{h}}^0) P(\mathbf{h}^{(0)}|\mathbf{h}^{(1)}) \\ p(\mathbf{h}^{(0)}) &= p_1(\mathbf{h}^{(0)}) && (10.6) \end{aligned}$$

The last line can be seen to be true by considering the stochastic process of first picking an $\tilde{\mathbf{h}}^0$ from the empirical distribution p_0, then sampling an $\mathbf{h}^{(1)}$ from $Q(\mathbf{h}^{(1)}|\tilde{\mathbf{h}}^0)$, and finally computing the probability of $\mathbf{h}^{(0)}$ under $P(\mathbf{h}^{(0)}|\mathbf{h}^{(1)})$ for that $\mathbf{h}^{(1)}$. □

Proposition 10.4.1 tells us that, even with the best possible model for $p(\mathbf{h}^{(1)}, \mathbf{h}^{(2)})$ according to the variational bound (i.e., the model that can achieve $p^*(\mathbf{h}^{(1)})$), we obtain a KL divergence between the DBN and the data equal to $KL(p_0||p_1)$. Hence if we train the 2nd level RBM to model the stochastic output of the 1st level RBM (as suggested in Hinton et al. [39]), the best $KL(p_0||p)$ we can achieve with model p of the 2-level DBN cannot be better than $KL(p_0||p_1)$. Note that this result does not preclude that a better likelihood could be achieved with p if a better criterion is used to train the 2nd level RBM.

For $KL(p_0||p_1)$ to be 0, one should have $p_0 = p_1$. Note that a weight vector with this property would not only be a fixed point of $KL(p_0||p_1)$ but also of the likelihood and of contrastive divergence for the first-level RBM. $p_0 = p_1$ could have been obtained with a one-level DBN (i.e., a single RBM) that perfectly fits the data. This can happen when

the first RBM has infinite weights i.e., is deterministic, and just encodes $\mathbf{h}^{(0)} = \mathbf{v}$ in $\mathbf{h}^{(1)}$ perfectly. In that case the second layer $\mathbf{h}^{(2)}$ seems useless.

Does that mean that adding layers is useless ? We believe the answer is no ; first, even though having the distribution that maximizes the variational bound yields $p = p_1$, this does not mean that we cannot achieve $KL(p_0||p) < KL(p_0||p_1)$ with a 2-layer DBN (though we have no proof that it can be achieved either). Indeed, since the variational bound is not the quantity we truly want to optimize, another criterion might lead to a better model (in terms of the likelihood of the data). Besides that, even if adding layers does not allow us to perfectly fit the data (which might actually only be the case when we optimize the variational bound rather than the likelihood), the distribution of the 2-layer DBN is closer to the empirical distribution than is the first layer RBM (we do only one "up-down" Gibbs step instead of doing an infinite number of such steps). Furthermore, the extra layers allow us to regularize and hopefully obtain a representation in which even a very high capacity top layer (e.g., a memory-based non-parametric density estimator) could generalize well. This approach suggests using alternative criteria to train DBNs, that approximate $KL(p_0||p_1)$ and can be computed before $\mathbf{h}^{(2)}$ is added, but, unlike contrastive divergence, take into account the fact that more layers will be added later.

Note that computing $KL(p_0||p_1)$ exactly is intractable in an RBM because it involves summing over all possible values of the hidden vector $\mathbf{h}$. One could use a sampling or mean-field approximation (replacing the summation over values of the hidden unit vector by a sample or a mean-field value), but even then there would remain a double sum over examples :

$$\sum_{i=1}^{N} \frac{1}{N} \log \sum_{j=1}^{N} \frac{1}{N} \hat{P}(V^1 = v_i | V^0 = v_j)$$

where v_i denotes the i-th example and $\hat{P}(V^1 = v_i|V^0 = v_j)$ denotes an estimator of the probability of observing $V^1 = v_i$ at iteration 1 of the Gibbs chain (that is after a "up-down" pass) given that the chain is started from $V^0 = v_j$. We write $\hat{P}$ rather than P because computing P exactly might involve an intractable summation over all possible values of $\mathbf{h}$. In Bengio et al. [13], the reconstruction error for training an auto-encoder corresponding to one layer of a deep network is $\log \hat{P}(V^1 = v_i|V^0 = v_i)$. Hence

$\log \hat{P}(V^1 = v_i | V^0 = v_j)$ is like a reconstruction error when one tries to reconstruct or predict v_i according to $P(v_i|\mathbf{h})$ when starting from v_j, sampling $\mathbf{h}$ from $P(\mathbf{h}|v_j)$. This criterion is essentially the same as one already introduced in a different context in [14], where $\hat{P}(V^1 = v_i | V^0 = v_j)$ is computed deterministically (no hidden random variable is involved), and the inner sum (over v_j's) is approximated by using only the 5 nearest neighbours of v_i in the training set. However, the overall computation time in [14] is $O(N^2)$ because like most non-parametric learning algorithms it involves comparing all training examples with each other. In contrast, the contrastive divergence gradient estimator can be computed in $O(N)$ for a training set of size N.

To evaluate whether tractable approximations of $KL(p_0||p_1)$ would be worth investigating, we performed an experiment on a toy dataset and toy model where the computations are feasible. The data are 10-element bit vectors with patterns of 1, 2 or 3 consecutive ones (or zeros) against a background of zeros (or ones), demonstrating simple shift invariance. There are 60 possible examples (p_0), 40 of which are randomly chosen to train first an RBM with 5 binomial hidden units, and then a 2-layer DBN. The remaining 20 are a test set. The second RBM has 10 hidden units (so that we could guarantee improvement of the likelihood by the addition of the second layer). The first RBM is either trained by contrastive divergence or to minimize $KL(p_0||p_1)$, using gradient descent and a learning rate of 0.1 for 500 epochs (parameters are updated after each epoch). Other learning rates and random initialization seeds gave similar results, diverged, or converged slower. The second RBM is then trained for the same number of epochs, by contrastive divergence with the same learning rate. Figure 10.2 shows the exact $KL(p_0||p)$ of the DBN p while training the 2nd RBM. The advantage of the $KL(p_0||p_1)$ training is clear. This suggests that future research should investigate tractable approximations of $KL(p_0||p_1)$.

10.4.3 Open Questions on DBN Representational Power

The results described in the previous section were motivated by the following question : since an RBM can represent any distribution, what can be gained by adding layers to a DBN, in terms of representational power ? More formally, let R_ℓ^n be a Deep Be-

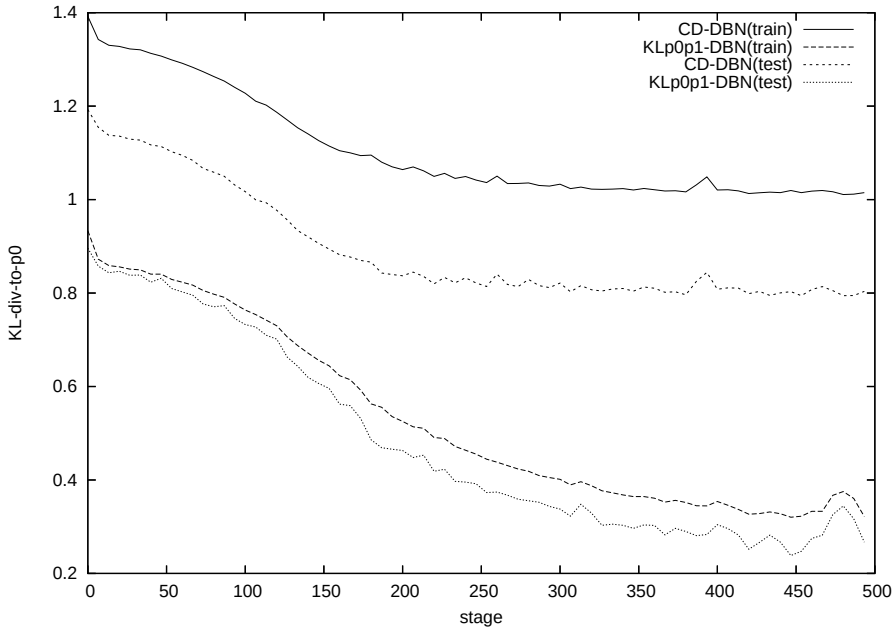


Figure 10.2 – KL divergence w.r.t. number epochs after adding the 2nd level RBM, between empirical distribution p_0 (either training or test set) and (top curves) DBN trained greedily with contrastive divergence at each layer, or (bottom curves) DBN trained greedily with $KL(p_0||p_1)$ on the 1st layer, and contrastive divergence on the 2nd.

lief Network with $\ell + 1$ hidden layers, each of them composed of n units. Can we say something about the representational power of R^n_ℓ as ℓ increases ? Denoting D^n_ℓ the set of distributions one can obtain with R^n_ℓ, it follows from the unfolding argument in Hinton et al. [39] that $D^n_\ell \subseteq D^n_{\ell+1}$. The unfolding argument shows that the last layer of an ℓ-layer DBN corresponds to an infinite directed graphical model with tied weights. By untying the weights in the $(\ell + 1)$-th RBM of this construction from those above, we obtain an $(\ell + 1)$-layer DBN. Hence every element of D^n_ℓ can be represented in $D^n_{\ell+1}$. Two questions remain :

- do we have $D^n_\ell \subset D^n_{\ell+1}$, at least for $\ell = 1$?
- what is D^n_∞ ?

10.5 Conclusions

We have shown that when the number of hidden units is allowed to vary, Restricted Boltzmann Machines are very powerful and can approximate any distribution, eventually representing them exactly when the number of hidden units is allowed to become very large (possibly 2 to the number of inputs). This only says that parameter values exist for doing so, but it does not prescribe how to obtain them efficiently. In addition, the above result is only concerned with the case of discrete inputs. It remains to be shown how to extend that type of result to the case of continuous inputs.

Restricted Boltzmann Machines are interesting chiefly because they are the building blocks of Deep Belief Networks, which can have many layers and can theoretically be much more efficient at representing complicated distributions [15]. We have introduced open questions about the expressive power of Deep Belief Networks. We have not answered these questions, but in trying to do so, we obtained an apparently puzzling result concerning Deep Belief Networks : the best that can be achieved by adding a second layer (with respect to some bound) is limited by the first layer's ability to map the data distribution to something close to itself ($KL(p_0||p_1)$), and this ability is good when the first layer is large and models well the data. So why do we need the extra layers ? We believe that the answer lies in the ability of a Deep Belief Network to generalize better by having a more compact representation. This analysis also suggests to investigate $KL(p_0||p_1)$ (or an efficient approximation of it) as a less greedy alternative to contrastive divergence for training each layer, because it would take into account that more layers will be added.

Acknowledgements

The authors would like to thank the following funding organizations for support : NSERC, MITACS, and the Canada Research Chairs. They are also grateful for the help and comments from Olivier Delalleau and Aaron Courville.

10.6 Appendix

10.6.1 Proof of Lemma 10.3.1

Démonstration. Denoting $\widetilde{\mathbf{h}} = \begin{bmatrix} \mathbf{h} \\ h_{n+1} \end{bmatrix}$, $\widetilde{W} = \begin{bmatrix} W \\ w^T \end{bmatrix}$ and $\widetilde{C} = \begin{bmatrix} C \\ c \end{bmatrix}$ where w^T denotes the transpose of w and introducing $z(\mathbf{v}, \mathbf{h}) = \exp(\mathbf{h}^T W \mathbf{v} + B^T \mathbf{v} + C^T \mathbf{h})$, we can express $p(\mathbf{v}, \mathbf{h})$ and $p_{w,c}(\mathbf{v}, \widetilde{\mathbf{h}})$ as follows :

$$\begin{aligned} p(\mathbf{v}, \mathbf{h}) &\propto z(\mathbf{v}, \mathbf{h}) \\ p_{w,c}(\mathbf{v}, \widetilde{\mathbf{h}}) &\propto \exp\left(\widetilde{\mathbf{h}}^T \widetilde{W} \mathbf{v} + B^T \mathbf{v} + \widetilde{C}^T \widetilde{\mathbf{h}}\right) \\ &\propto z(\mathbf{v}, \mathbf{h}) \exp\left(h_{n+1} w^T \mathbf{v} + c h_{n+1}\right) \end{aligned}$$

Expanding the expression of $p_{w,c}(\mathbf{v})$ and regrouping the terms similar to the expression of $p(\mathbf{v})$, we get :

$$\begin{aligned} p_{w,c}(\mathbf{v}) &= \frac{\sum_{\widetilde{\mathbf{h}}} \exp\left(\mathbf{h}^T W \mathbf{v} + h_{n+1} w^T \mathbf{v} + B^T \mathbf{v} + C^T \mathbf{h} + c h_{n+1}\right)}{\sum_{\widetilde{\mathbf{h}^{(0)}}, \mathbf{v}^0} \exp\left(\mathbf{h}^{(0)T} W \mathbf{v}^0 + h_{n+1}^{(0)} w^T \mathbf{v}^0 + B^T \mathbf{v}^0 + C^T \mathbf{h}^{(0)} + c h_{n+1}^{(0)}\right)} \\ &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h}) \left(1 + \exp\left(w^T \mathbf{v} + c\right)\right)}{\sum_{\mathbf{h}^{(0)}, \mathbf{v}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)}) \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right)} \\ &= \frac{\left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) \sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0} \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right) \sum_{\mathbf{h}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)})} \end{aligned}$$

But $\sum_h z(\mathbf{v}, \mathbf{h}) = p(\mathbf{v}) K$ with $K = \sum_{\mathbf{v},\mathbf{h}} z(\mathbf{v}, \mathbf{h})$. Thus,

$$p_{w,c}(\mathbf{v}) = \frac{\left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) p(\mathbf{v})}{\sum_{\mathbf{v}^0} \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right) p(\mathbf{v}^0)}$$

which does not depend on our particular choice of R_p (since it does only depend on p).

□

10.6.2 Proof of theorem 10.3.3

Démonstration. Expanding the expression of $p_{w,c}(\mathbf{v})$ and regrouping the terms similar to the expression of $p(\mathbf{v})$, we get :

$$\begin{aligned} p_{w,c}(\mathbf{v}) &= \frac{\sum_{\widetilde{\mathbf{h}}} \exp\left(\mathbf{h}^T W \mathbf{v} + h_{n+1} w^T \mathbf{v} + B^T \mathbf{v} + C^T \mathbf{h} + c h_{n+1}\right)}{\sum_{\widetilde{\mathbf{h}^{(0)}}, \mathbf{v}^0} \exp\left(\mathbf{h}^{(0)T} W \mathbf{v}^0 + h_{n+1}^{(0)} w^T \mathbf{v}^0 + B^T \mathbf{v}^0 + C^T \mathbf{h}^{(0)} + c h_{n+1}^{(0)}\right)} \\ &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h}) \left(1 + \exp\left(w^T \mathbf{v} + c\right)\right)}{\sum_{\mathbf{h}^{(0)}, \mathbf{v}^0} z(\mathbf{v}^0, \mathbf{h}^{(0)}) \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right)} \\ &= \frac{\left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) \sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right) z(\mathbf{v}^0, \mathbf{h}^{(0)})} \end{aligned}$$

Therefore, we have :

$$\begin{aligned} KL(p_0 || p_{w,c}) &= \sum_{\mathbf{v}} p_0(\mathbf{v}) \log p_0(\mathbf{v}) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \log p_{w,c}(\mathbf{v}) \\ &= -H(p_0) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \log \left(\frac{\left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) \sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right) z(\mathbf{v}^0, \mathbf{h}^{(0)})} \right) \\ &= -H(p_0) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \log \left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \log \left(\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h}) \right) \\ &\quad + \sum_{\mathbf{v}} p_0(\mathbf{v}) \log \left(\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} \left(1 + \exp\left(w^T \mathbf{v}^0 + c\right)\right) z(\mathbf{v}^0, \mathbf{h}^{(0)}) \right) \end{aligned}$$

Assuming $w^T \mathbf{v} + c$ is a very large negative value for all $\mathbf{v}$, we can use the logarithmic series identity around 0 ($\log(1 + x) = x + o_{x \to 0}(x)$) for the second and the last term. The second term becomes[1]

$$\sum_{\mathbf{v}} p_0(\mathbf{v}) \log\left(1 + \exp\left(w^T \mathbf{v} + c\right)\right) = \sum_{\mathbf{v}} p_0(\mathbf{v}) \exp\left(w^T \mathbf{v} + c\right) + o_{c \to -\infty}\left(\exp(c)\right)$$

[1] $o_{x \to \infty}()$ notation : $f(x) = o_{x \to +\infty}\left(g(x)\right)$ if $\lim_{x \to +\infty} \frac{f(x)}{g(x)}$ exists and equals 0.

and the last term becomes

$$\begin{aligned}\left(\sum_{\mathbf{v}} p_0(\mathbf{v})\right) \quad \log \quad & \left(\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} \left(1+\exp\left(w^T\mathbf{v}^0+c\right)\right) z(\mathbf{v}^0,\mathbf{h}^{(0)})\right) \\ = \quad & \log\left(\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})\right) \\ & + \log\left(1+\frac{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} \exp\left(w^T\mathbf{v}^0+c\right) z(\mathbf{v}^0,\mathbf{h}^{(0)})}{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})}\right) \\ = \quad & \log\left(\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})\right) \\ & + \frac{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} \exp\left(w^T\mathbf{v}^0+c\right) z(\mathbf{v}^0,\mathbf{h}^{(0)})}{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})} + o_{c\rightarrow-\infty}\left(\exp(c)\right)\end{aligned}$$

But

$$\begin{aligned}\frac{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} \exp\left(w^T\mathbf{v}^0+c\right) z(\mathbf{v}^0,\mathbf{h}^{(0)})}{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})} &= \sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}+c\right) \frac{\sum_{\mathbf{h}^{(0)}} z(\mathbf{v},\mathbf{h}^{(0)})}{\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})} \\ &= \sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}+c\right) p(\mathbf{v})\end{aligned}$$

Putting all terms back together, we have

$$\begin{aligned}KL(p_0||p_{w,c}) &= -H(p_0) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \exp\left(w^T\mathbf{v}+c\right) + \sum_{\mathbf{v}} p(\mathbf{v}) \exp\left(w^T\mathbf{v}+c\right) \\ &+ o_{c\rightarrow-\infty}\left(\exp(c)\right) - \sum_{\mathbf{v}} p_0(\mathbf{v}) \log\left(\sum_{\mathbf{h}} z(\mathbf{v},\mathbf{h})\right) + \log\left(\sum_{\mathbf{v}^0,\mathbf{h}^{(0)}} z(\mathbf{v}^0,\mathbf{h}^{(0)})\right) \\ &= KL(p_0||p) + \sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}+c\right)\left(p(\mathbf{v})-p_0(\mathbf{v})\right) + o_{c\rightarrow-\infty}\left(\exp(c)\right)\end{aligned}$$

Finally, we have

$$KL(p_0||p_{w,c}) - KL(p_0||p) = \exp(c) \sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}\right)\left(p(\mathbf{v})-p_0(\mathbf{v})\right) + o_{c\rightarrow-\infty}\left(\exp(c)\right) \quad (10.7)$$

The question now becomes : can we find a w such that $\sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}\right)(p(\mathbf{v}) - p_0(\mathbf{v}))$ is negative ?

As $p_0 \neq p$, there is a $\widehat{\mathbf{v}}$ such that $p(\widehat{\mathbf{v}}) < p_0(\widehat{\mathbf{v}})$. Then there exists a positive scalar a such that $\widehat{w} = a\left(\widehat{\mathbf{v}} - \frac{1}{2}e\right)$ (with $e = [1 \ldots 1]^T$) yields $\sum_{\mathbf{v}} \exp\left(\widehat{w}^T\mathbf{v}\right)(p(\mathbf{v}) - p_0(\mathbf{v})) < 0$. Indeed, for $\mathbf{v} \neq \widehat{\mathbf{v}}$, we have

$$\begin{aligned}
\frac{\exp(\widehat{w}^T\mathbf{v})}{\exp(\widehat{w}^T\widehat{\mathbf{v}})} &= \exp\left(\widehat{w}^T(\mathbf{v} - \widehat{\mathbf{v}})\right) \\
&= \exp\left(a\left(\widehat{\mathbf{v}} - \frac{1}{2}e\right)^T(\mathbf{v} - \widehat{\mathbf{v}})\right) \\
&= \exp\left(a\sum_i \left(\widehat{\mathbf{v}}_i - \frac{1}{2}\right)(\mathbf{v}_i - \widehat{\mathbf{v}}_i)\right)
\end{aligned}$$

For i such that $\mathbf{v}_i - \widehat{\mathbf{v}}_i > 0$, we have $\mathbf{v}_i = 1$ and $\widehat{\mathbf{v}}_i = 0$. Thus, $\widehat{\mathbf{v}}_i - \frac{1}{2} = -\frac{1}{2}$ and the term inside the exponential is negative (since a is positive). For i such that $\mathbf{v}_i - \widehat{\mathbf{v}}_i < 0$, we have $\mathbf{v}_i = 0$ and $\widehat{\mathbf{v}}_i = 1$. Thus, $\widehat{\mathbf{v}}_i - \frac{1}{2} = \frac{1}{2}$ and the term inside the exponential is also negative. Furthermore, the terms come close to 0 as a goes to infinity. Since the sum can be decomposed as

$$\begin{aligned}
\sum_{\mathbf{v}} \exp\left(\widehat{w}^T\mathbf{v}\right)(p(\mathbf{v}) - p_0(\mathbf{v})) &= \exp(\widehat{w}^T\widehat{\mathbf{v}})\left(\sum_{\mathbf{v}} \frac{\exp\left(\widehat{w}^T\mathbf{v}\right)}{\exp(\widehat{w}^T\widehat{\mathbf{v}})}(p(\mathbf{v}) - p_0(\mathbf{v}))\right) \\
&= \exp(\widehat{w}^T\widehat{\mathbf{v}})\left(p(\widehat{\mathbf{v}}) - p_0(\widehat{\mathbf{v}}) + \sum_{\mathbf{v}\neq\widehat{\mathbf{v}}} \frac{\exp\left(\widehat{w}^T\mathbf{v}\right)}{\exp(\widehat{w}^T\widehat{\mathbf{v}})}(p(\mathbf{v}) - p_0(\mathbf{v}))\right)
\end{aligned}$$

we have[2]

$$\sum_{\mathbf{v}} \exp\left(\widehat{w}^T\mathbf{v}\right)(p(\mathbf{v}) - p_0(\mathbf{v})) \sim_{a\to+\infty} \exp(\widehat{w}^T\widehat{\mathbf{v}})(p(\widehat{\mathbf{v}}) - p_0(\widehat{\mathbf{v}})) < 0.$$

Therefore, there is a value $\widehat{a}$ such that, if $a > \widehat{a}$, $\sum_{\mathbf{v}} \exp\left(w^T\mathbf{v}\right)(p(\mathbf{v}) - p_0(\mathbf{v})) < 0$. This concludes the proof. □

[2] $\sim_{x\to\infty}$ notation : $f(x) \sim_{x\to+\infty} g(x)$ if $\lim_{x\to+\infty} \frac{f(x)}{g(x)}$ exists and equals 1.

10.6.3 Proof of theorem 10.3.4

Démonstration. In the former proof, we had

$$p_{w,c}(\mathbf{v}) = \frac{\left(1 + \exp\left(w^T\mathbf{v} + c\right)\right) \sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} \left(1 + \exp\left(w^T\mathbf{v}^0 + c\right)\right) z(\mathbf{v}^0, \mathbf{h}^{(0)})}$$

Let $\tilde{\mathbf{v}}$ be an arbitrary input vector and $\widehat{w}$ be defined in the same way as before, i.e. $\widehat{w} = a\left(\tilde{\mathbf{v}} - \frac{1}{2}\right)$.

Now define $\widehat{c} = -\widehat{w}^T\tilde{\mathbf{v}} + \lambda$ with $\lambda \in \mathbb{R}$. We have :

$$\begin{aligned} \lim_{a\to\infty} 1 + \exp(\widehat{w}^T\mathbf{v} + \widehat{c}) = & \quad 1 \quad \text{for } \mathbf{v} \neq \tilde{\mathbf{v}} \\ 1 + \exp(\widehat{w}^T\tilde{\mathbf{v}} + \widehat{c}) = & \quad 1 + \exp(\lambda) \end{aligned}$$

Thus, we can see that, for $\mathbf{v} \neq \tilde{\mathbf{v}}$:

$$\begin{aligned} \lim_{a\to\infty} p_{\widehat{w},\widehat{c}}(\mathbf{v}) &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0 \neq \tilde{\mathbf{v}}, \mathbf{h}^{(0)}} z(\mathbf{v}^0, \mathbf{h}^{(0)}) + \sum_{\mathbf{h}^{(0)}} \left(1 + \exp\left(\widehat{w}^T\tilde{\mathbf{v}} + \widehat{c}\right)\right) z(\tilde{\mathbf{v}}, \mathbf{h}^{(0)})} \\ &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} z(\mathbf{v}^0, \mathbf{h}^{(0)}) + \sum_{\mathbf{h}^{(0)}} \exp\left(\lambda\right) z(\tilde{\mathbf{v}}, \mathbf{h}^{(0)})} \\ &= \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} z(\mathbf{v}^0, \mathbf{h}^{(0)})} \frac{1}{1 + \exp(\lambda) \frac{\sum_{\mathbf{h}^{(0)}} z(\tilde{\mathbf{v}}, \mathbf{h}^{(0)})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} z(\mathbf{v}^0, \mathbf{h}^{(0)})}} \end{aligned}$$

Remembering $p(\mathbf{v}) = \frac{\sum_{\mathbf{h}} z(\mathbf{v}, \mathbf{h})}{\sum_{\mathbf{v}^0, \mathbf{h}^{(0)}} z(\mathbf{v}^0, \mathbf{h}^{(0)})}$, we have for $\mathbf{v} \neq \tilde{\mathbf{v}}$:

$$\lim_{a\to\infty} p_{\widehat{w},\widehat{c}}(\mathbf{v}) = \frac{p(\mathbf{v})}{1 + \exp(\lambda) p(\tilde{\mathbf{v}})} \tag{10.8}$$

Similarly, we can see that

$$\lim_{a\to\infty} p_{\widehat{w},\widehat{c}}(\tilde{\mathbf{v}}) = \frac{[1 + \exp(\lambda)] p(\tilde{\mathbf{v}})}{1 + \exp(\lambda) p(\tilde{\mathbf{v}})} \tag{10.9}$$

Depending on the value of λ, one can see that adding a hidden unit allows one to increase the probability of an arbitrary $\tilde{\mathbf{v}}$ and to uniformly decrease the probability of

every other $\mathbf{v}$ by a multiplicative factor. However, one can also see that, if $p(\tilde{\mathbf{v}}) = 0$, then $p_{\widehat{w},\widehat{c}}(\tilde{\mathbf{v}}) = 0$ for all λ.

We can therefore build the desired RBM as follows. Let us index the $\mathbf{v}$'s over the integers from 1 to 2^n and sort them such that

$$p_0(\mathbf{v}_{k+1}) = \ldots = p_0(\mathbf{v}_{2^n}) = 0 < p_0(\mathbf{v}_1) \leq p_0(\mathbf{v}_2) \leq \ldots \leq p_0(\mathbf{v}_k)$$

Let us denote p^i the distribution of an RBM with i hidden units. We start with an RBM whose weights and biases are all equal to 0. The marginal distribution over the visible units induced by that RBM is the uniform distribution. Thus,

$$p^0(\mathbf{v}_1) = \ldots = p^0(\mathbf{v}_{2^n}) = 2^{-n}$$

We define $w_1 = a_1(\mathbf{v}_1 - \frac{1}{2})$ and $c_1 = -w_1^T\mathbf{v}_1 + \lambda_1$.

As shown before, we now have :

$$\begin{aligned}
\lim_{a_1 \to +\infty} p^1(\mathbf{v}_1) &= \frac{[1+\exp(\lambda_1)]2^{-n}}{1+\exp(\lambda_1)2^{-n}} \\
\lim_{a_1 \to +\infty} p^1(\mathbf{v}_i) &= \frac{2^{-n}}{1+\exp(\lambda_1)2^{-n}} \quad \forall i \geq 2
\end{aligned}$$

As we can see, we can set $p^1(\mathbf{v}_1)$ to a value arbitrarily close to 1, with a uniform distribution over $\mathbf{v}_2, \ldots, \mathbf{v}_{2^n}$. Then, we can choose λ_2 such that $\frac{p^2(\mathbf{v}_2)}{p^2(\mathbf{v}_1)} = \frac{p(\mathbf{v}_2)}{p(\mathbf{v}_1)}$. This is possible since we can arbitrarily increase $p^2(\mathbf{v}_2)$ while multiplying the other probabilities by a constant factor and since $\frac{p(\mathbf{v}_2)}{p(\mathbf{v}_1)} \geq \frac{p^1(\mathbf{v}_2)}{p^1(\mathbf{v}_1)}$. We can continue the procedure until obtaining $p^k(\mathbf{v}_k)$. The ratio $\frac{p^i(\mathbf{v}_j)}{p^i(\mathbf{v}_{j-1})}$ does not depend on the value of i as long as $i > j$ (because at each such step i, the two probabilities are multiplied by the same factor). We will then have

$$\begin{aligned}
\frac{p^k(\mathbf{v}_k)}{p^k(\mathbf{v}_{k-1})} = \frac{p(\mathbf{v}_k)}{p(\mathbf{v}_{k-1})}, \quad \ldots \quad , \frac{p^k(\mathbf{v}_2)}{p^k(\mathbf{v}_1)} = \frac{p(\mathbf{v}_2)}{p(\mathbf{v}_1)} \\
p^k(\mathbf{v}_{k+1}) = \quad \ldots \quad = p^k(\mathbf{v}_{2^n})
\end{aligned}$$

From that, we can deduce that $p^k(\mathbf{v}_1) = \nu_k p(\mathbf{v}_1), \ldots, p^k(\mathbf{v}_k) = \nu_k p(\mathbf{v}_k)$ with $\nu_k = 1 - (2^n - k)p^k(\mathbf{v}_{2^n})$.

We also have $\frac{p^k(\mathbf{v}_1)}{p^k(\mathbf{v}_{2^n})} = \frac{p^1(\mathbf{v}_1)}{p^1(\mathbf{v}_{2^n})} = 1 + \exp(\lambda_1)$.

Thus, $p^k(\mathbf{v}_1) = p(\mathbf{v}_1)[1 - (2^n - k)p^k(\mathbf{v}_{2^n})] = (1 + \exp(\lambda_1))p^k(\mathbf{v}_{2^n})$.

Solving the above equations, we have

$$p^k(\mathbf{v}_i) = \frac{p(\mathbf{v}_1)}{1+\exp(\lambda_1)+p(\mathbf{v}_1)(2^n-k)} \quad \text{for } i > k \tag{10.10}$$

$$p^k(\mathbf{v}_i) = p(\mathbf{v}_i)\frac{1+\exp(\lambda_1)}{1+\exp(\lambda_1)+p(\mathbf{v}_1)(2^n-k)} \quad \text{for } i \leq k \tag{10.11}$$

Using the logarithmic series identity around 0 ($\log(1 + x) = x + o_{x\to 0}(x)$) for $KL(p||p^k)$ when λ_1 goes to infinity, we have

$$KL(p||p^k) = \sum_i p(\mathbf{v}_i)\frac{(2^n - k)p(\mathbf{v}_i)}{1 + \exp(\lambda_1)} + o(\exp(-\lambda_1)) \underset{\lambda_1\to\infty}{\longrightarrow} 0 \tag{10.12}$$

This concludes the proof. □

CHAPITRE 11

PRÉSENTATION DU CINQUIÈME ARTICLE

11.1 Détails de l'article

Topmoumoute Online Natural Gradient Algorithm
N. Le Roux, P.-A. Manzagol et Y. Bengio Publié dans *Advances in Neural Information Processing Systems 20*, MIT Press, Cambridge, MA en 2008.

11.2 Contexte

L'optimisation effectuée par un algorithme d'apprentissage nécessite la minimisation d'un risque $\mathcal{L}$. Cette minimisation peut se faire de différentes façons, dépendamment des structures de $\mathcal{L}$ et de notre algorithme.

Comme mentionné à la section 2.3, la descente de gradient est l'algorithme de choix pour l'optimisation des réseaux de neurones ainsi que d'autres algorithmes impossibles à optimiser analytiquement (ce qui est souvent le cas dès que l'on sort des algorithmes triviaux).

Les algorithmes comme ceux mentionnés aux chapitres précédents possèdent un grand nombre de paramètres et sont destinés à être entraînés sur un large ensemble d'entraînement. On peut même imaginer un entraînement en continu nécessitant une technique d'optimisation très rapide.

Le but de l'article présenté ici est de proposer un algorithme plus rapide que toutes les méthodes de descente de gradient actuelles pour parvenir à entraîner des algorithmes puissants sur des tâches difficiles.

11.3 Bilan

TONGA obtient des résultats extrêmement encourageants sur plusieurs jeux de données importants. Il est toutefois judicieux de se demander pourquoi ses performances,

comparées à celles du gradient stochastique, diffèrent tant d'un jeu de données à l'autre.

L'une des raisons possibles est la supposition que les gradients sont tirés de manières iid. Si cela est vrai dans le cas en ligne, ça devient une approximation dans le cas hors ligne, approximation d'autant plus erronée que l'ensemble d'apprentissage est petit. Ce problème est relativement bénin puisque, dès lors que l'ensemble d'apprentissage est petit, le besoin d'un algorithme d'optimisation rapide se fait moins ressentir.

Une autre possibilité est la sensibilité de cet algorithme aux hyperparamètres. En effet, les valeurs de λ et γ sont primordiales pour l'efficacité de l'algorithme et un mauvais réglage peut entraîner des performances très décevantes. Il serait donc désirable d'élaborer une méthode permettant le réglage automatique de ces paramètres.

CHAPITRE 12

TOPMOUMOUTE ONLINE NATURAL GRADIENT ALGORITHM

12.1 Abstract

Guided by the goal of obtaining an optimization algorithm that is both fast and yields good generalization, we study the descent direction maximizing the decrease in generalization error or the probability of not increasing generalization error. The surprising result is that from both the Bayesian and frequentist perspectives this can yield the natural gradient direction. Although that direction can be very expensive to compute we develop an efficient, general, online approximation to the natural gradient descent which is suited to large scale problems. We report experimental results showing much faster convergence in computation time and in number of iterations with TONGA (Topmoumoute Online Natural Gradient Algorithm) than with stochastic gradient descent, even on very large datasets.

Introduction

An efficient optimization algorithm is one that quickly finds a good minimum for a given cost function. An efficient learning algorithm must do the same, with the additional constraint that the function is only known through a proxy. This work aims to improve the ability to generalize through more efficient learning algorithms.

Consider the optimization of a cost on a training set with access to a validation set. As the end objective is a good solution with respect to generalization, one often uses early stopping : optimizing the training error while monitoring the validation error to fight overfitting. This approach makes the underlying assumption that overfitting happens at the later stages. A better perspective is that overfitting happens all through the learning, but starts being detrimental only at the point it overtakes the “true” learning. In terms of gradients, the gradient of the cost on the training set is never collinear with the true gradient, and the dot product between the two actually eventually becomes negative.

Early stopping is designed to determine when that happens. One can thus wonder : can one limit overfitting before that point ? Would this actually postpone that point ?

From this standpoint, we discover new justifications behind the natural gradient [4]. Depending on certain assumptions, it corresponds either to the direction minimizing the probability of increasing generalization error, or to the direction in which the generalization error is expected to decrease the fastest. Unfortunately, natural gradient algorithms suffer from poor scaling properties, both with respect to computation time and memory, when the number of parameters becomes large. To address this issue, we propose a generally applicable online approximation of natural gradient that scales linearly with the number of parameters (and requires computation time comparable to stochastic gradient descent). Experiments show that it can bring significant faster convergence and improved generalization.

12.2 Natural gradient

Let $\widetilde{\mathcal{L}}$ be a cost defined as $\widetilde{\mathcal{L}}(\theta) = \int L(x,\theta)p(x)dx$ where L is a loss function over some parameters θ and over the random variable x with distribution $p(x)$. The problem of minimizing $\widetilde{\mathcal{L}}$ over θ is often encountered and can be quite difficult. There exist various techniques to tackle it, their efficiency depending on L and p. In the case of non-convex optimization, gradient descent is a successful technique. The approach consists in progressively updating θ using the gradient $\widetilde{g} = \frac{d\widetilde{\mathcal{L}}}{d\theta}$.

Amari [4] showed that the parameter space is a Riemannian space of metric $\widetilde{C}$ (the covariance of the gradients), and introduced the natural gradient as the direction of steepest descent in this space. The natural gradient direction is therefore given by $\widetilde{C}^{-1}\widetilde{g}$. The Riemannian space is known to correspond to the space of functions represented by the parameters (instead of the space of the parameters themselves).

The natural gradient somewhat resembles the Newton method. LeCun et al. [53] showed that, in the case of a mean squared cost function, the Hessian is equal to the sum of the covariance matrix of the gradients and of an additional term that vanishes to 0 as the training error goes down. Indeed, when the data are generated from the model,

the Hessian and the covariance matrix are equal. There are two important differences : the covariance matrix $\widetilde{C}$ is positive-definite, which makes the technique more stable, but contains no explicit second order information. The Hessian allows to account for variations in the parameters. The covariance matrix accounts for slight variations in the set of training samples. It also means that, if the gradients highly disagree in one direction, one should not go in that direction, even if the mean suggests otherwise. In that sense, it is a conservative gradient.

12.3 A new justification for natural gradient

Until now, we supposed we had access to the true distribution p. However, this is usually not the case and, in general, the distribution p is only known through the samples of the training set. These samples define a cost $\mathcal{L}$ (resp. a gradient g) that, although close to the true cost (resp. gradient), is not equal to it. We shall refer to $\mathcal{L}$ as the training error and to $\widetilde{\mathcal{L}}$ as the generalization error. The danger is then to overfit the parameters θ to the training set, yielding parameters that are not optimal with respect to the generalization error.

A simple way to fight overfitting consists in determining the point when the continuation of the optimization on $\mathcal{L}$ will be detrimental to $\widetilde{\mathcal{L}}$. This can be done by setting aside some samples to form a validation set that will provide an independent estimate of $\widetilde{\mathcal{L}}$. Once the error starts increasing on the validation set, the optimization should be stopped. We propose a different perspective on overfitting. Instead of only monitoring the validation error, we consider using as descent direction an estimate of the direction that maximizes the probability of reducing the generalization error. The goal is to limit overfitting at every stage, with the hope that the optimal point with respect to the validation should have lower generalization error.

Consider a descent direction v. We know that if $v^T\widetilde{g}$ is negative then the generalization error drops (for a reasonably small step) when stepping in the direction of v. Likewise, if $v^T g$ is negative then the training error drops. Since the learning objective is to minimize generalization error, we would like $v^T\widetilde{g}$ as small as possible, or at least

always negative.

By definition, the gradient on the training set is $g = \frac{1}{n}\sum_{i=1}^{n} g_i$ where $g_i = \frac{\partial L(x_i, \theta)}{\partial \theta}$ and n is the number of training samples. With a rough approximation, one can consider the g_is as draws from the true gradient distribution and assume all the gradients are independent and identically distributed. The central limit theorem then gives

$$g \sim \mathbb{N}\left(\widetilde{g}, \frac{\widetilde{C}}{n}\right) \tag{12.1}$$

where $\widetilde{C}$ is the true covariance matrix of $\frac{\partial L(x,\theta)}{\partial \theta}$ wrt $p(x)$.

We will now show that, both in the Bayesian setting (with a Gaussian prior) and in the frequentist setting (with some restrictions over the type of gradient considered), the natural gradient is optimal in some sense.

12.3.1 Bayesian setting

In the Bayesian setting, $\widetilde{g}$ is a random variable. We would thus like to define a posterior over $\widetilde{g}$ given the samples g_i in order to have a posterior distribution over $v^T\widetilde{g}$ for any given direction v. The prior over $\widetilde{g}$ will be a Gaussian centered in 0 of variance $\sigma^2 I$. Thus, using eq. 12.1, the posterior over $\widetilde{g}$ given the g_is (assuming the only information over $\widetilde{g}$ given by the g_is is through g and C) is

$$\widetilde{g}|g, \widetilde{C} \sim \mathbb{N}\left(\left(I + \frac{\widetilde{C}}{n\sigma^2}\right)^{-1} g, \left(\frac{I}{\sigma^2} + n\widetilde{C}^{-1}\right)^{-1}\right) \tag{12.2}$$

Denoting $\widetilde{C}_\sigma = I + \frac{\widetilde{C}}{n\sigma^2}$, we therefore have

$$v^T\widetilde{g}|g, \widetilde{C} \sim \mathbb{N}\left(v^T\widetilde{C}_\sigma^{-1} g, \frac{v^T\widetilde{C}_\sigma^{-1}\widetilde{C}v}{n}\right) \tag{12.3}$$

Using this result, one can choose between several strategies, among which two are of particular interest :

– choosing the direction v such that the expected value of $v^T\widetilde{g}$ is the lowest possible (to maximize the immediate gain). In this setting, the direction v to choose is

$$v \propto -\widetilde{C}_\sigma^{-1} g. \tag{12.4}$$

If $\sigma < \infty$, this is the regularized natural gradient. In the case of $\sigma = \infty$, $\widetilde{C}_\sigma = I$ and this is the batch gradient descent.

– choosing the direction v to minimize the probability of $v^T\widetilde{g}$ to be positive. This is equivalent to finding

$$\text{argmin}_v \frac{v^T\widetilde{C}_\sigma^{-1} g}{\sqrt{v^T\widetilde{C}_\sigma^{-1}\widetilde{C}v}}$$

(we dropped n for the sake of clarity, since it does not change the result). If we square this quantity and take the derivative with respect to v, we find that the numerator is equal to $2\widetilde{C}_\sigma^{-1} g(v^T\widetilde{C}_\sigma^{-1} g)(v^T\widetilde{C}_\sigma^{-1}\widetilde{C}v) - 2\widetilde{C}_\sigma^{-1}\widetilde{C}v(v^T\widetilde{C}_\sigma^{-1} g)^2$. The first term is in the span of $\widetilde{C}_\sigma^{-1} g$ and the second one is in the span of $\widetilde{C}_\sigma^{-1}\widetilde{C}v$. Hence, for the derivative to be zero, we must have $g \propto \widetilde{C}v$ (since $\widetilde{C}$ and $\widetilde{C}_\sigma$ are invertible), i.e.

$$v \propto -\widetilde{C}^{-1} g. \tag{12.5}$$

This direction is the natural gradient and does not depend on the value of σ.

12.3.2 Frequentist setting

In the frequentist setting, $\widetilde{g}$ is a fixed unknown quantity. For the sake of simplicity, we will only consider (as all second-order methods do) the directions v of the form $v = M^T g$ (i.e. we are only allowed to go in a direction which is a linear function of g).

Since $g \sim \mathbb{N}\left(\widetilde{g}, \frac{\widetilde{C}}{n}\right)$, we have

$$v^T\widetilde{g} = g^T M g \sim \mathbb{N}\left(\widetilde{g}^T M\widetilde{g}, \frac{\widetilde{g}^T M^T\widetilde{C}M\widetilde{g}}{n}\right) \tag{12.6}$$

The matrix M^* which minimizes the probability of $v^T\widetilde{g}$ to be positive satisfies

$$M^* = \text{argmin}_M \frac{\widetilde{g}^T M \widetilde{g}}{\widetilde{g}^T M^T C M \widetilde{g}} \tag{12.7}$$

The numerator of the derivative of this quantity is $\widetilde{g}\widetilde{g}^T M^T \widetilde{C} M \widetilde{g}\widetilde{g}^T - 2\widetilde{C} M \widetilde{g}\widetilde{g}^T M \widetilde{g}\widetilde{g}^T$. The first term is in the span of $\widetilde{g}$ and the second one is in the span of $\widetilde{C}M\widetilde{g}$. Thus, for this derivative to be 0 for all $\widetilde{g}$, one must have $M \propto \widetilde{C}^{-1}$ and we obtain the same result as in the Bayesian case : the natural gradient represents the direction minimizing the probability of increasing the generalization error.

12.4 Online natural gradient

The previous sections provided a number of justifications for using the natural gradient. However, the technique has a prohibitive computational cost, rendering it impractical for large scale problems. Indeed, considering p as the number of parameters and n as the number of examples, a direct batch implementation of the natural gradient is $O(p^2)$ in space and $O(np^2 + p^3)$ in time, associated respectively with the gradients' covariance storage, computation and inversion. This section reviews existing low complexity implementations of the natural gradient, before proposing TONGA, a new low complexity, online and generally applicable implementation suited to large scale problems. In the previous sections we assumed the true covariance matrix $\widetilde{C}$ to be known. In a practical algorithm we of course use an empirical estimate, and here this estimate is furthermore based on a low-rank approximation denoted C (actually a sequence of estimates C_t).

12.4.1 Low complexity natural gradient implementations

Yang et Amari [87] propose a method specific to the case of multilayer perceptrons. By operating on blocks of the covariance matrix, this approach attains a lower computational complexity[1]. However, the technique is quite involved, specific to multilayer perceptrons and requires two assumptions : Gaussian distributed inputs and a number of

[1]Though the technique allows for a compact representation of the covariance matrix, the working memory requirement remains the same.

hidden units much inferior to that of input units. Amari et al. [5] offer a more general approach based on the Sherman-Morrison formula used in Kalman filters : the technique maintains an empirical estimate of the inversed covariance matrix that can be updated in $O(p^2)$. Yet the memory requirement remains $O(p^2)$. It is however not necessary to compute the inverse of the gradients' covariance, since one only needs its product with the gradient. Yang et Amari [88] offer two approaches to exploit this. The first uses conjugate gradient descent to solve $Cv = g$. The second revisits [87] thereby achieving a lower complexity. Schraudolph [78] also proposes an iterative technique based on the minimization of a different cost. This technique is used in the minibatch setting, where Cv can be computed cheaply through two matrix vector products. However, estimating the gradient covariance only from a small number of examples in one minibatch yields unstable estimation.

12.4.2 TONGA

Existing techniques fail to provide an implementation of the natural gradient adequate for the large scale setting. Their main failings are with respect to computational complexity or stability. TONGA was designed to address these issues, which it does by maintaining a low rank approximation of the covariance and by casting both problems of finding the low rank approximation and of computing the natural gradient in a lower dimensional space, thereby attaining a much lower complexity. What we exploit here is that although a covariance matrix needs many gradients to be estimated, we can take advantage of an observed property that it generally varies smoothly as training proceeds and moves in parameter space.

12.4.2.1 Computing the natural gradient direction between two eigendecompositions

Even though our motivation for the use of natural gradient implied the covariance matrix of the empirical gradients, we will use the second moment (i.e. the uncentered covariance matrix) throughout the paper (and so did Amari in his work). The main reason

is numerical stability. Indeed, in the batch setting, we have (assuming C is the centered covariance matrix and g the mean) $v = C^{-1}g$, thus $Cv = g$. But then, $(C + gg^T)v = g + gg^Tv = g(1 + g^Tv)$ and

$$(C + gg^T)^{-1}g = \frac{v}{1 + g^Tv} = \bar{v} \tag{12.8}$$

Even though the direction is the same, the scale changes and the norm of the direction is bounded by $\frac{1}{\|g\| \cos(g,v)}$.

Since TONGA operates using a low rank estimate of the gradients' non-centered covariance, we must be able to update cheaply. When presented with a new gradient, we integrate its information using the following update formula[2] :

$$C_t = \gamma \hat{C}_{t-1} + g_t g_t^T \tag{12.9}$$

where $C_0 = 0$ and $\hat{C}_{t-1}$ is the low rank approximation at time step $t-1$. C_t is now likely of greater rank, and the problem resides in computing its low rank approximation $\hat{C}_t$. Writing $\hat{C}_{t-1} = X_{t-1}X_{t-1}^T$,

$$C_t \quad = \quad X_t X_t^T \text{ with } X_t = [\sqrt{\gamma} X_{t-1} \quad g_t]$$

With such covariance matrices, computing the (regularized) natural direction v_t is equal to

$$v_t \quad = \quad (C_t + \lambda I)^{-1} g_t = (X_t X_t^T + \lambda I)^{-1} g_t \tag{12.10}$$

$$v_t \quad = \quad (X_t X_t^T + \lambda I)^{-1} X_t y_t \text{ with } y_t = [0, \ldots 0, 1]^T. \tag{12.11}$$

Using the Woodbury identity with positive definite matrices [64], we have

$$v_t = X_t (X_t^T X_t + \lambda I)^{-1} y_t \tag{12.12}$$

[2]The second term is not weighted by $1 - \gamma$ so that the influence of g_t in C_t is the same for all t, even $t = 0$.To keep the magnitude of the matrix constant, one must use a normalization constant equal to $1 + \gamma + \ldots + \gamma^t$.

If X_t is of size $p \times r$ (with $r < p$, thus yielding a covariance matrix of rank r), the cost of this computation is $O(pr^2 + r^3)$. However, since the Gram matrix $G_t = X_t^T X_t$ can be rewritten as

$$G_t = \begin{pmatrix} \gamma X_{t-1}^T X_{t-1} & \sqrt{\gamma} X_{t-1}^T g_t \\ \sqrt{\gamma} g_t^T X_{t-1} & g_t^T g_t \end{pmatrix} = \begin{pmatrix} \gamma G_{t-1} & \sqrt{\gamma} X_{t-1}^T g_t \\ \sqrt{\gamma} g_t^T X_{t-1} & g_t^T g_t \end{pmatrix}, \quad (12.13)$$

the cost of computing G_t using G_{t-1} reduces to $O(pr + r^3)$. This stresses the need to keep r small.

12.4.2.2 Updating the low-rank estimate of C_t

To keep a low-rank estimate of $C_t = X_t X_t^T$, we can compute its eigendecomposition and keep only the first k eigenvectors. This can be made at low cost using its relation to that of G_t :

$$\begin{aligned} G_t &= V D V^T \\ C_t &= (X_t V D^{-\frac{1}{2}}) D (X_t V D^{-\frac{1}{2}})^T \end{aligned} \quad (12.14)$$

The cost of such an eigendecomposition is $O(kr^2 + pkr)$ (for the computation of the eigendecomposition of the Gram matrix and the computation of the eigenvectors, respectively). Since the cost of computing the natural direction is $O(pr + r^3)$, it is computationally more efficient to let the rank of X_t grow for several steps (using formula 12.12 in between) and then compute the eigendecomposition using

$$C_{t+b} = X_{t+b} X_{t+b}^T \text{ with } X_{t+b} = \left[\gamma U_t, \quad \gamma^{\frac{b-1}{2}} g_{t+1}, \quad \dots \quad \gamma^{\frac{1}{2}} g_{t+b-1}, \quad \gamma^{\frac{t+b}{2}} g_{t+b}\right]$$

with U_t the unnormalized eigenvectors computed during the previous eigendecomposition.

12.4.2.3 Computational complexity

The computational complexity of TONGA depends on the complexity of updating the low rank approximation and on the complexity of computing the natural gradient. The cost of updating the approximation is in $O(k(k+b)^2 + p(k+b)k)$ (as above, using $r = k + b$). The cost of computing the natural gradient v_t is in $O(p(k+b) + (k+b)^3)$ (again, as above, using $r = k + b$). Assuming $k + b \ll \sqrt{(}p)$ and $k \leq b$, TONGA's total computational cost per each natural gradient computation is then $O(pb)$.

Furthermore, by operating on minibatch gradients of size b', we end up with a cost per example of $O(\frac{bp}{b'})$. Choosing $b = b'$, yields $O(p)$ per example, the same as stochastic gradient descent. Empirical comparison of cpu time also shows comparable CPU time per example, but faster convergence. In our experiments, p was in the tens of thousands, k was less than 5 and b was less than 50.

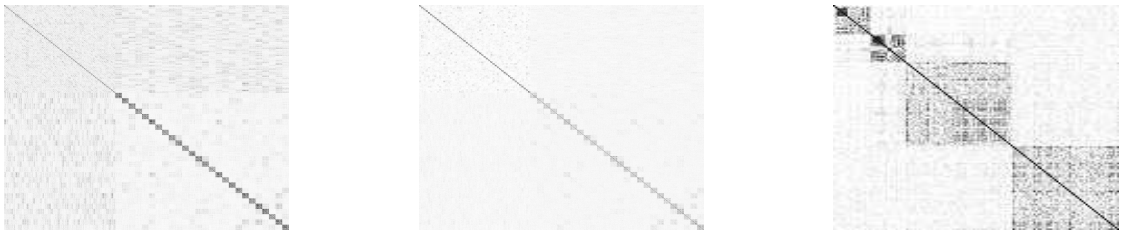
The result is an approximate natural gradient with low complexity, general applicability and flexibility over the tradoff between computations and the quality of the estimate.

12.5 Block-diagonal online natural gradient for neural networks

One might wonder if there are better approximations of the covariance matrix C than computing its first k eigenvectors. One possibility is a block-diagonal approximation from which to retain only the first k eigenvectors of every block (the value of k can be different for each block). Indeed, [27] showed that the Hessian of a neural network with one hidden layer trained with the cross-entropy cost converges to a block diagonal matrix during optimization. These blocks are composed of the weights linking all the hidden units to one output unit and all the input units to one hidden unit. Given the close relationship between the Hessian and the covariance matrices, we can assume they have a similar shape during the optimization.

Figure 12.1 shows the correlation between the standard stochastic gradients of the parameters of a $16 - 50 - 26$ neural network. The first blocks represent the weights going from the input units to each hidden unit (thus 50 blocks of size 17, bias included) and the following represent the weights going from the hidden units to each output unit

(26 blocks of size 51). One can see that the block-diagonal approximation is reasonable. Thus, instead of selecting only k eigenvectors to represent the full covariance matrix, we can select k eigenvectors for every block, yielding the same total cost. However, the rank of the approximation goes from k to $k \times$ number of blocks. In the matrices shown in figure 12.1, which are of size 2176, a value of $k = 5$ yields an approximation of rank 380.

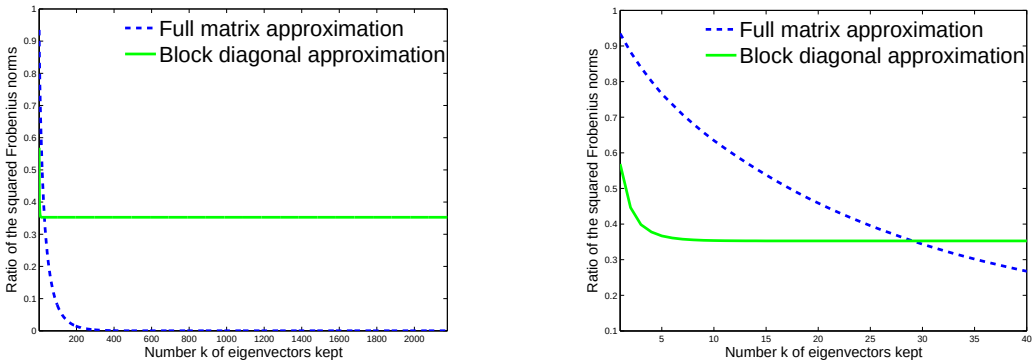


(a) Stochastic gradient (b) TONGA (c) TONGA - zoom

Figure 12.1 – Absolute correlation between the standard stochastic gradients after one epoch in a neural network with 16 input units, 50 hidden units and 26 output units when following stochastic gradient directions (left) and natural gradient directions (center and right).

Figure 12.2 shows the ratio of Frobenius norms $\frac{\|C-\bar{C}\|_F^2}{\|C\|_F^2}$ for different types of approximations $\bar{C}$ (full or block-diagonal). We can first notice that approximating only the blocks yields a ratio of $.35$ (in comparison, taking only the diagonal of C yields a ratio of $.80$), even though we considered only 82076 out of the 4734976 elements of the matrix (1.73% of the total). This ratio is almost obtained with $k = 6$. We can also notice that, for $k < 30$, the block-diagonal approximation is much better (in terms of the Frobenius norm) than the full approximation. The block diagonal approximation is therefore very cost effective.

This shows the block diagonal approximation constitutes a powerful and cheap approximation of the covariance matrix in the case of neural networks. Yet this approximation also readily applies to any mixture algorithm where we can assume independence between the components.



(a) Full view (b) Zoom

Figure 12.2 – Quality of the approximation $\bar{C}$ of the covariance C depending on the number of eigenvectors kept (k), in terms of the ratio of Frobenius norms $\frac{\|C-\bar{C}\|_F^2}{\|C\|_F^2}$, for different types of approximations $\bar{C}$ (full matrix or block diagonal)

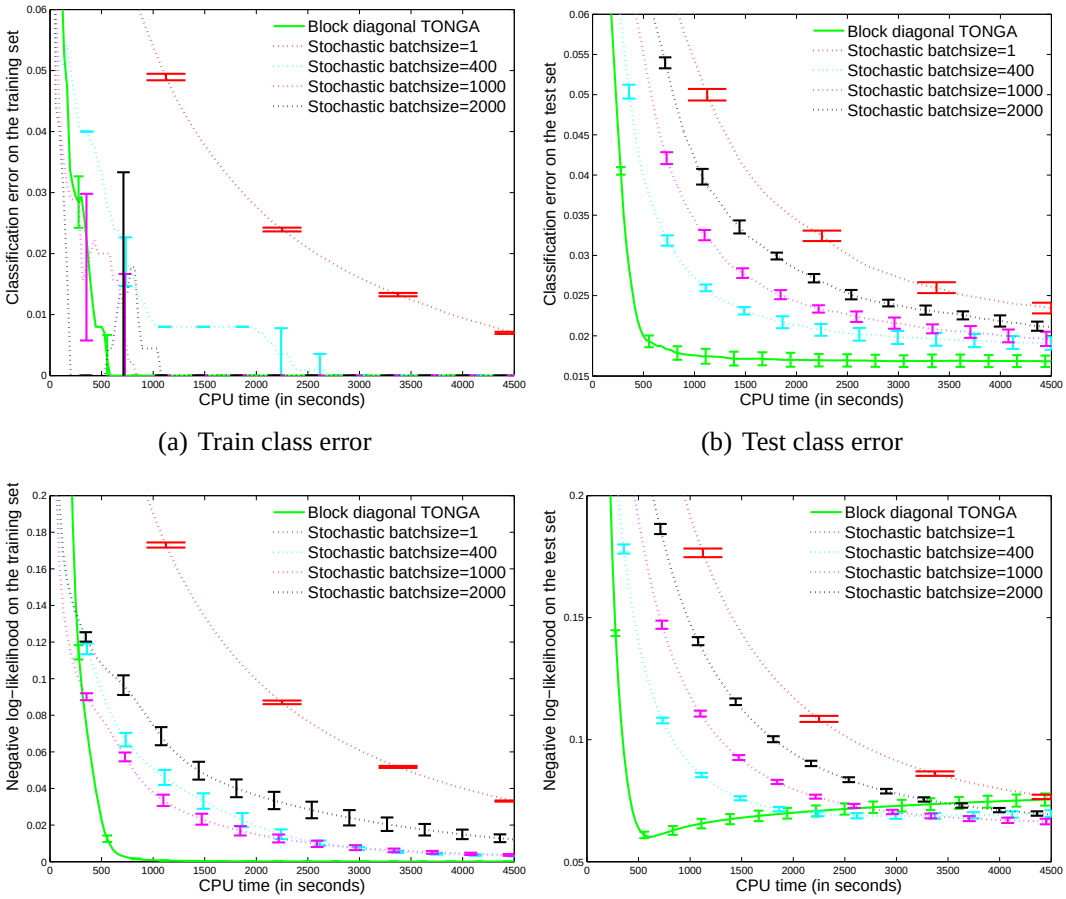
12.6 Experiments

We performed a small number of experiments with TONGA approximating the full covariance matrix, keeping the overhead of the natural gradient small (ie, limiting the rank of the approximation). Regrettably, TONGA performed only as well as stochastic gradient descent, while being rather sensitive to the hyperparameter values. The following experiments, on the other hand, use TONGA with the block diagonal approximation and yield impressive results. We believe this is a reflection of the phenomenon illustrated in figure 12.2 : the block diagonal approximation makes for a very cost effective approximation of the covariance matrix. All the experiments have been made optimizing hyperparameters on a validation set (not shown here) and selecting the best set of hyperparameters for testing, trying to keep small the overhead due to natural gradient calculations.

One could worry about the number of hyperparameters of TONGA. However, default values of $k = 5$, $b = 50$ and $\gamma = .995$ yielded good results in every experiment. When λ goes to infinity, TONGA becomes the standard stochastic gradient algorithm. Therefore, a simple heuristic for λ is to progressively tune it down. In our experiments, we only tried powers of ten.

12.6.1 MNIST dataset

The MNIST digits dataset consists of 50000 training samples, 10000 validation samples and 10000 test samples, each one composed of 784 pixels. There are 10 different classes (one for every digit).



(a) Train class error (b) Test class error

(c) Train NLL (d) Test NLL

Figure 12.3 – Comparison between stochastic gradient and TONGA on the MNIST dataset (50000 training examples), in terms of training and test classification error and Negative Log-Likelihood (NLL). The mean and standard error have been computed using 9 different initializations.

Figure 12.3 shows that in terms of training CPU time (which includes the overhead due to TONGA), TONGA allows much faster convergence in training NLL, as well as in testing classification error and testing NLL than ordinary stochastic and minibatch

gradient descent on this task. One can also note that minibatch stochastic gradient is able to profit from matrix-matrix multiplications, but this advantage is mainly seen in training classification error.

12.6.2 Rectangles problem

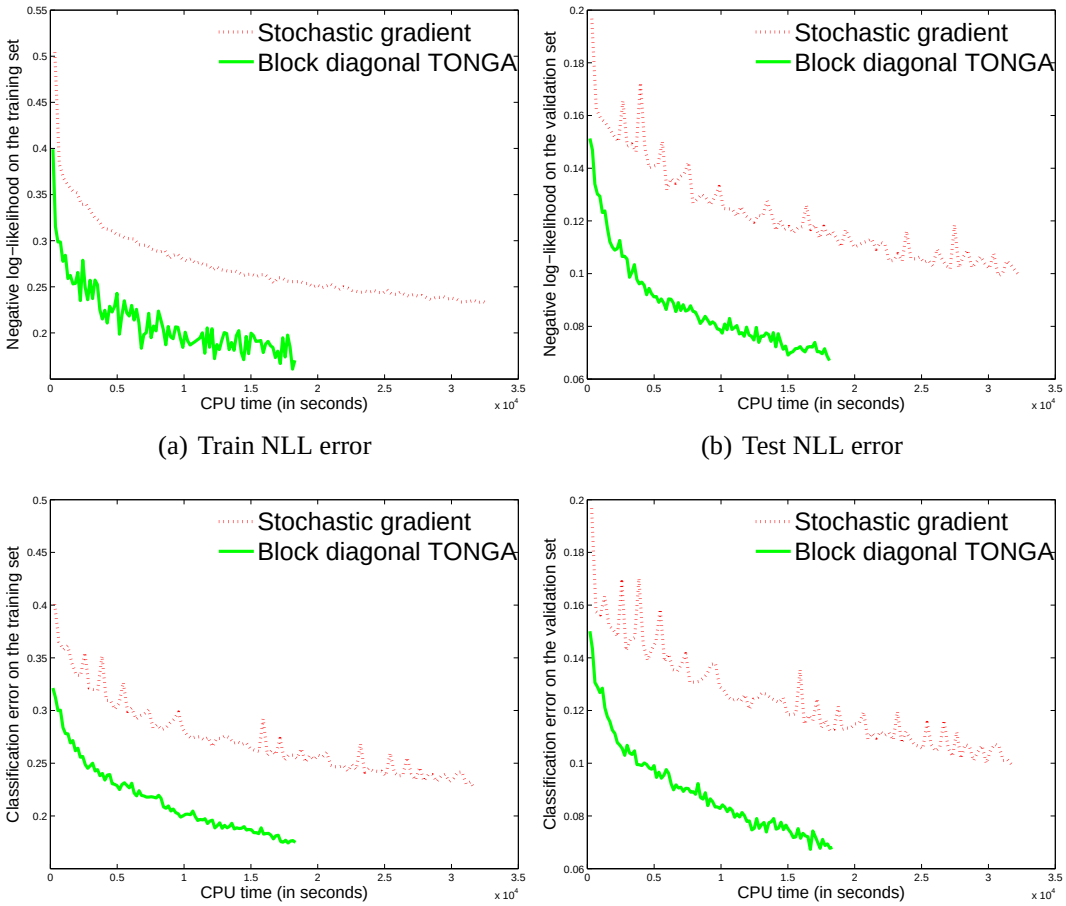
The *Rectangles-images* task has been proposed in [49] to compare deep belief networks and support vector machines. It is a two-class problem and the inputs are 28×28 grey-level images of rectangles located in varying locations and of different dimensions. The inside of the rectangle and the background are extracted from different real images. We used 900,000 training examples and 10,000 validation examples (no early stopping was performed, we show the whole training/validation curves). All the experiments are performed with a multi-layer network with a 784-200-200-100-2 architecture (previously found to work well on this dataset). Figure 12.4 shows that in terms of training CPU time, TONGA allows much faster convergence than ordinary stochastic gradient descent on this task, as well as lower classification error.

12.7 Discussion

[23] reviews the different gradient descent techniques in the online setting and discusses their respective properties. Particularly, he states that a second order online algorithm (i.e., with a search direction of is $v = Mg$ with g the gradient and M a positive semidefinite matrix) is *optimal* (in terms of convergence speed) when M converges to H^{-1}. Furthermore, the speed of convergence depends (amongst other things) on the rank of the matrix M. Given the aforementioned relationship between the covariance and the Hessian matrices, the natural gradient is close to optimal in the sense defined above, provided the model has enough capacity. On mixture models where the block-diagonal approximation is appropriate, it allows us to maintain an approximation of much higher rank than a standard low-rank approximation of the full covariance matrix.

Conclusion and future work

We bring two main contributions in this paper. First, by looking for the descent direction with either the greatest probability of not increasing generalization error or the direction with the largest expected decrease in generalization error, we obtain new justifications for the natural gradient descent direction. Second, we present an online low-rank approximation of natural gradient descent with computational complexity and CPU time similar to stochastic gradient descent. In a number of experimental comparisons we find this optimization technique to beat stochastic gradient in terms of speed and generalization (or in generalization for a given amount of training time). Even though default values for the hyperparameters yield good results, it would be interesting to have an automatic procedure to select the best set of hyperparameters.



(a) Train NLL error

(b) Test NLL error

(c) Train class error

(d) Test class error

Figure 12.4 – Comparison between stochastic gradient descent and TONGA w.r.t. NLL and classification error, on training and validation sets for the rectangles problem (900,000 training examples).

CHAPITRE 13

CONCLUSION

Les articles de cette thèse s'inscrivent dans une démarche de compréhension des limites des algorithmes majoritairement utilisés à l'heure actuelle et d'exploration de nouvelles méthodes pour pouvoir traiter les problèmes importants à venir.

Après avoir mis en évidence et étudié les limites intrinsèques de la majorité des méthodes à noyau (chapitre 4), nous avons essayé d'en transférer les propriétés attractives aux réseaux de neurones à une couche cachée : la convexité de leur optimisation (chapitre 6) et l'exploitabilité de leur universalité (chapitre 8). Malheureusement, ces avancées n'ont pas permis de vaincre la déficience fondamentable dont sont affublées ces deux familles d'algorithmes : l'impossibilité d'apprendre des concepts de haut niveau.

Nous nous sommes donc tournés vers les réseaux de neurones à plusieurs couches cachées dont la profondeur nous a semblé adaptée à l'extraction de tels concepts. Nous appuyant sur les travaux de Hinton et al. [39] sur les Deep Belief Networks, nous avons étudié certaines des propriétés de ces réseaux et proposé une méthode d'entraînement de substitution (chapitre 10).

Puisque notre but est de pouvoir traiter les problèmes importants à venir, nous nous sommes également intéressés à une méthode permettant l'analyse rapide d'un très grand nombre de données et qui est particulièrement adaptée aux réseaux de neurones possédant un nombre quelconque de couches cachées (chapitre 12).

13.1 Synthèse des articles

13.1.1 The Curse of Highly Variable Functions for Local Kernel Machines

Cet article ne propose pas d'algorithme mais est une critique argumentée des méthodes utilisant un noyau local. Deux critiques ont fréquemment été émises à l'encontre de ces arguments :

– la première invoquait le « no free lunch theorem » [86] : aucun algorithme ne

surpasse strictement un autre si l'on prend l'espérance sur tous les problèmes possibles. Cet argument insinue que nous n'avons qu'exhibé (et même construit) des problèmes pour lesquels les méthodes à noyau faillissent, ce qui ne constitue pas une preuve de leurs faiblesses. Toutefois, nous ne sommes pas intéressés par la construction d'algorithmes performants sur tous les problèmes possibles (ce qui est impossible, d'après le théorème mentionné ci-dessus), mais sur tous les problèmes que nous sommes à même de rencontrer dans notre monde. Partant de ce postulat, il est facile de voir que nous n'avons pas exhibé des problèmes choisis **au hasard**, mais bien des problèmes qui, s'ils sont d'apparence complexe, n'en demeurent pas moins très simples à résoudre pour un humain.

– la seconde arguait qu'il était impossible de généraliser loin des exemples d'apprentissage, et ce quel que soit l'algorithme utilisé. Certains algorithmes, cherchant des structures dans les données, prouvèrent qu'il n'en était rien (voir par exemple Non-Local Manifold Parzen Windows [14]).

Bien qu'apparemment illégitimes, ces critiques soulèvent toutefois un point d'importance : les humains parviennent sans souci à généraliser à des situations totalement inconnues, ce que les algorithmes actuels ont encore bien du mal à faire. Si certains y parviennent mieux que d'autres, nous sommes encore loin de savoir transférer notre connaissance d'une région de l'espace à une autre.

13.1.2 Convex Neural Networks

Cet article présente un algorithme utile à la fois pour la classification et la régression. Si l'idée originale était de prouver qu'il était possible d'entraîner un réseau de neurones de manière convexe, les implications furent toutes autres. Tout d'abord, le lien établi entre les réseaux de neurones et le boosting est d'importance puisqu'il apporte un nouveau point de vue des réseaux de neurones. Ensuite, cet article montre également qu'un réseau entraîné de manière gloutonne peut être optimal. Si cette gloutonnerie est horizontale (c'est-à-dire que ce sont les unités cachées et non les couches qui sont ajoutées de manière gloutonne), cela donne de l'espoir pour les Deep Belief Networks.

13.1.3 Continuous Neural Networks

Les réseaux continus étaient une tentative d'accorder aux réseaux de neurones l'alléchante propriété d'approximation universelle. Les limites de cette méthode théoriquement puissante ont permis de mettre en exergue deux des défis qui attendent la communauté de l'apprentissage machine dans les années à venir : l'extraction de concepts de haut niveau et l'optimisation efficace des paramètres. Ne bénéficiant d'aucune de ces deux propriétés, les réseaux continus ne purent tirer le meilleur avantage de leur capacité.

13.1.4 Representational Power of Restricted Boltzmann Machines and Deep Belief Networks

Répondant aux manques des algorithmes mentionnés dans les trois articles précédents, les Deep Belief Networks regorgent de promesses. C'est cette raison qui nous a poussé à investiguer leur puissance et les moyens optimaux pour les entraîner. Cet article permet également de mieux saisir les liens étroits reliant les machines de Boltzmann restreintes aux réseaux de neurones usuels et la validité de leur utilisation comme première étape de l'entraînement.

13.1.5 Topmoumoute Online Natural Gradient Algorithm

Comme mentionné ci-dessus, l'autre grand défi est l'optimisation d'algorithmes complexes en temps réel. En effet, de plus en plus, l'afflux de données est submergeant et il apparaît appréciable d'être capable de traiter toutes ces informations. TONGA est l'un des premiers réels algorithmes de second ordre stochastiques, qui plus est capable de battre le gradient stochastique sur les problèmes sur lesquels il a été testé.

13.2 Conclusion

Le but de cette thèse fut de mettre en lumière les limites et le potentiel des différents algorithmes utilisés dans la communauté. Si les algorithmes profonds semblent proposer des solutions aux problèmes rencontrés actuellement, ils ne sont pas pour autant la

panacée. Les algorithmes gloutons de Hinton et al. [39] et de Bengio et al. [13] sont pour l'instant sous-optimaux et ne s'accordent pas bien d'un flux incessant de nouvelles données.

On assiste également à une évolution du type de régularisation utilisée. Alors qu'il était coutume auparavant de limiter le nombre de neurones cachés (et donc le nombre de paramètres libres), les réseaux de neurones actuels utilisent un très grand nombre de paramètres tout en imposant des contraintes supplémentaires, que ce soit en liant les poids (comme dans [52] ou [62]) ou en essayant de résoudre un problème plus compliqué [39]. La qualité des résultats obtenus par ces dernières méthodes laisse à penser que l'un des grands défis futurs sera d'intégrer dans les algorithmes d'apprentissage des méthodes de régularisation dont la notion de simplicité induite imitera au mieux celle utilisée par les humains.

On peut aussi se demander l'importance du caractère non supervisé de l'entraînement préalable des Deep Belief Networks. N'est-ce qu'une méthode de régularisation, comme mentionné précédemment, ou bien un bon modèle génératif est-il nécessaire pour obtenir un bon modèle discriminant ? S'il fut longtemps accepté dans la communauté que, pour obtenir un modèle discriminant, il était inutile et même préjudiciable d'entraîner un modèle génératif, Ng et Jordan [61] montrèrent qu'il pouvait être intéressant d'essayer de modéliser toute la distribution jointe dans certains contextes. De leur côté, Lasserre et al. [50] analysèrent les suppositions implicites d'un entraînement discriminant. Les humains possèdent-ils réellement un modèle génératif du monde qui les entoure ? Est-ce nécessaire ?

Enfin, l'autre défi qui attend la communauté de l'apprentissage machine est celui de la multimodalité. En tant qu'humains, nous percevons des données de différents types (selon le sens utilisé) que nous combinons de manière efficace pour résoudre les problèmes auxquels nous devons faire face. En outre, nous sommes parfaitement capables de transférer nos connaissances abstraites acquises dans un contexte donné à des contextes totalement différents. Il n'existe pas à l'heure actuelle d'algorithmes capables d'un tel transfert.

Les algorithmes d'apprentissage futurs auront la nécessité de pouvoir extraire des

concepts de haut niveau en utilisant un nombre toujours plus grand d’exemples si on veut espérer atteindre un jour l’intelligence artificielle.

Index

BIBLIOGRAPHIE

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

[2] M. Ajtai. $\sum_1^1$-formulae on finite structures. *Annals of Pure and Applied Logic*, (24):1–48, 1983.

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

[4] S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. URL `citeseer.ist.psu.edu/article/amari98natural.html`.

[5] S. Amari, H. Park et K. Fukumizu. Adaptive method of realizing natural gradient learning for multilayer perceptrons. *Neural Computation*, 12(6):1399–1409, 2000. URL `citeseer.ist.psu.edu/amari98adaptive.html`.

[6] C.T.H. Baker. *The numerical treatment of integral equations*. Clarendon Press, Oxford, 1977.

[7] M. Belkin, I. Matveeva et P. Niyogi. Regularization and semi-supervised learning on large graphs. Dans John Shawe-Taylor et Yoram Singer, éditeurs, *COLT'2004*. Springer, 2004.

[8] M. Belkin et P. Niyogi. Using manifold structure for partially labeled classification. Dans S. Becker, S. Thrun et K. Obermayer, éditeurs, *Advances in Neural Information Processing Systems 15*, Cambridge, MA, 2003. MIT Press.

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

[10] Y. Bengio, O. Delalleau et N. Le Roux. The curse of highly variable functions for local kernel machines. Dans Y. Weiss, B. Schölkopf et J. Platt, éditeurs, *Advances in Neural Information Processing Systems 18*, pages 107–114. MIT Press, Cambridge, MA, 2006.

[11] Y. Bengio, O. Delalleau, N. Le Roux, J.-F. Paiement, P. Vincent et M. Ouimet. Learning eigenfunctions links spectral embedding and kernel PCA. *Neural Computation*, 16(10):2197–2219, 2004.

[12] Y. Bengio, R. Ducharme, P. Vincent et C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[13] Y. Bengio, P. Lamblin, D. Popovici et H. Larochelle. Greedy layer-wise training of deep networks. Dans B. Schölkopf, J. Platt et T. Hoffman, éditeurs, *Advances in Neural Information Processing Systems 19*, pages 153–160. MIT Press, 2007.

[14] Y. Bengio, H. Larochelle et P. Vincent. Non-local manifold parzen windows. Dans Y. Weiss, B. Schölkopf et J. Platt, éditeurs, *Advances in Neural Information Processing Systems 18*, pages 115–122. MIT Press, 2006.

[15] Y. Bengio et Y. Le Cun. Scaling learning algorithms towards AI. Dans L. Bottou, O. Chapelle, D. DeCoste et J. Weston, éditeurs, *Large Scale Kernel Machines*. MIT Press, 2007.

[16] Y. Bengio, N. Le Roux, P. Vincent, O. Delalleau et P. Marcotte. Convex neural networks. Dans Y. Weiss, B. Schölkopf et J. Platt, éditeurs, *Advances in Neural Information Processing Systems 18*, pages 123–130. MIT Press, Cambridge, MA, 2006.

[17] Y. Bengio et M. Monperrus. Non-local manifold tangent learning. Dans L.K. Saul, Y. Weiss et L. Bottou, éditeurs, *Advances in Neural Information Processing Systems 17*. MIT Press, 2005.

[18] Y. Bengio, J.F. Paiement, P. Vincent, O. Delalleau, N. Le Roux et M. Ouimet. Out-of-sample extensions for LLE, Isomap, MDS, Eigenmaps, and Spectral Clustering.

Dans S. Thrun, L. Saul et B. Schölkopf, éditeurs, *Advances in Neural Information Processing Systems 16*. MIT Press, 2004.

[19] K. S. Beyer, J. Goldstein, R. Ramakrishnan et U. Shaft. When is "nearest neighbor" meaningful ? Dans *Proceeding of the 7th International Conference on Database Theory*, pages 217–235. Springer-Verlag, 1999. ISBN 3-540-65452-6.

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

[21] B. Boser, I. Guyon et V. Vapnik. A training algorithm for optimal margin classifiers. Dans *Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, 1992.

[22] L. Bottou. Online algorithms and stochastic approximations. Dans David Saad, éditeur, *Online Learning in Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

[23] L. Bottou. Stochastic learning. Dans Olivier Bousquet et Ulrike von Luxburg, éditeurs, *Advanced Lectures on Machine Learning*, numbéro LNAI 3176 dans Lecture Notes in Artificial Intelligence, pages 146–168. Springer Verlag, Berlin, 2004. URL `http://leon.bottou.org/papers/bottou-mlss-2004`.

[24] M. Brand. Charting a manifold. Dans S. Becker, S. Thrun et K. Obermayer, éditeurs, *Advances in Neural Information Processing Systems 15*. MIT Press, 2003.

[25] M.A. Carreira-Perpiñan et G.E. Hinton. On contrastive divergence learning. Dans Robert G. Cowell et Zoubin Ghahramani, éditeurs, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, Jan 6-8, 2005, Savannah Hotel, Barbados*, pages 33–40. Society for Artificial Intelligence and Statistics, 2005.

[26] V. Chvátal. *Linear Programming*. W.H. Freeman, 1983.

[27] R. Collobert. *Large Scale Machine Learning*. Thèse de doctorat, Université de Paris VI, LIP6, 2004.

[28] R. Collobert, F. Sinz, J. Weston et L. Bottou. Trading convexity for scalability. Dans *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[29] C. Cortes et V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[30] O. Delalleau, Y. Bengio et N. Le Roux. Efficient non-parametric function induction in semi-supervised learning. Dans R.G. Cowell et Z. Ghahramani, éditeurs, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, Jan 6-8, 2005, Savannah Hotel, Barbados*, pages 96–103. Society for Artificial Intelligence and Statistics, 2005.

[31] O. Delalleau, Y. Bengio et N. Le Roux. Efficient non-parametric function induction in semi-supervised learning. Dans R.G. Cowell et Z. Ghahramani, éditeurs, *Proceedings of AISTATS'2005*, pages 96–103, 2005.

[32] Y. Freund et D. Haussler. Unsupervised learning of distributions of binary vectors using two layer networks. Rapport technique CRL-94-25, UCSC, 1994.

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

[34] J. Friedman. Greedy function approximation : a gradient boosting machine. *Annals of Statistics*, 29:1180, 2001.

[35] S. Geman, E. Bienenstock et R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[36] W. Härdle, M. Müller, S. Sperlich et A. Werwatz. *Nonparametric and Semiparametric Models*. Springer, http ://www.xplore-stat.de/ebooks/ebooks.html, 2004.

[37] J.T. Hastad. *Computational Limitations for Small Depth Circuits*. MIT Press, 1987.

[38] R. Hettich et K.O. Kortanek. Semi-infinite programming : theory, methods, and applications. *SIAM Review*, 35(3):380–429, 1993.

[39] G. E. Hinton, S. Osindero et Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[40] G.E. Hinton. Products of experts. Dans *Proceedings of the Ninth International Conference on Artificial Neural Networks (ICANN)*, volume 1, pages 1–6, Edinburgh, Scotland, 1999. IEE.

[41] G.E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:1771–1800, 2002.

[42] G.E. Hinton et R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

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

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

[45] E.T. Jaynes. *Probability Theory : The Logic of Science*. Cambridge University Press, 2003.

[46] S. Sathiya Keerthi et Chih-Jen Lin. Asymptotic behaviors of support vector machines with Gaussian kernel. *Neural Computation*, 15(7):1667–1689, 2003.

[47] G. Kimeldorf et G. Wahba. Some results on tchebychean spline functions. *Journal of Mathematics Analysis and Applications*, 33:82–95, 1971.

[48] A.N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Kokl. Akad. Nauk USSR*, 114:953–956, 1957.

[49] H. Larochelle, D. Erhan, A. Courville, J. Bergstra et Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. Dans *Twenty-fourth International Conference on Machine Learning (ICML'2007)*, 2007.

[50] J.A. Lasserre, C.M. Bishop et T.P. Minka. Principled hybrids of generative and discriminative models. Dans *CVPR '06 : Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 87–94, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2597-0.

[51] Y. LeCun et Y. Bengio. Convolutional networks for images, speech, and time-series. Dans M. A. Arbib, éditeur, *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

[52] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard et L.D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[53] Y. LeCun, L. Bottou, G.B. Orr et K.-R. Müller. Efficient backprop. Dans G.B. Orr et K.-R. Müller, éditeurs, *Neural Networks : Tricks of the Trade*, pages 9–50. Springer, 1998.

[54] Y. LeCun, F.-J. Huang et L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. Dans *Proceedings of CVPR'04*. IEEE Press, 2004.

[55] D.J.C. MacKay. Introduction to gaussian processes. Rapport technique, Cambridge University, 1998. URL `http://wol.ra.phy.cam.ac.uk/mackay/gpB.pdf`.

[56] P. Marcotte et G. Savard. Novel approaches to the discrimination problem. *Zeitschrift für Operations Research (Theory)*, 36:517–545, 1992.

[57] L. Mason, J. Baxter, P. L. Bartlett et M. Frean. Boosting algorithms as gradient descent. Dans *Advances in Neural Information Processing Systems 12*, pages 512–518, 2000.

[58] W.S. McCulloch et W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 1943.

[59] C. A. Micchelli. Interpolation of scattered data : distance matrices and conditionally positive definite functions. *Constructive Approximation*, 2:11–22, 1986.

[60] R. Neal. *Bayesian Learning for Neural Networks*. Thèse de doctorat, Dept. of Computer Science, University of Toronto, 1994.

[61] A.Y. Ng et M.I. Jordan. On discriminative vs. generative classifiers : A comparison of logistic regression and naive bayes. Dans *NIPS*, pages 841–848, 2001.

[62] S. J. Nowlan et G. E. Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493, 1992.

[63] E.J. Nyström. Über die praktische auflösung von linearen integralgleichungen mit anwendungen auf randwertaufgaben der potentialtheorie. *Commentationes Physico-Mathematicae*, 4(15):1–52, 1928.

[64] K. B. Petersen et M. S. Pedersen. The matrix cookbook, feb 2006. Version 20051003.

[65] Marc'Aurelio Ranzato, Christopher Poultney, Sumit Chopra et Yann LeCun. Efficient learning of sparse representations with an energy-based model. Dans B. Schölkopf, J. Platt et T. Hoffman, éditeurs, *Advances in Neural Information Processing Systems 19*. MIT Press, 2007.

[66] C.E. Rasmussen. Conjugate gradient for matlab, 2001. http ://www.kyb.tuebingen.mpg.de/bs/people/carl/code/minimize/.

[67] G. Rätsch, A. Demiriz et K.P. Bennett. Sparse regression ensembles in infinite and finite hypothesis spaces. *Machine Learning*, 2002.

[68] F. Rosenblatt. The perceptron — a perceiving and recognizing automaton. Rapport technique 85-460-1, Cornell Aeronautical Laboratory, Ithaca, N.Y., 1957.

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

[70] S. Roweis et L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, Dec. 2000.

[71] D.E. Rumelhart, G.E. Hinton et R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[72] R. Salakhutdinov et G.E. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. Dans *Proceedings of AISTATS'2007*, San Juan, Porto Rico, 2007. Omnipress.

[73] L. Saul et S. Roweis. Think globally, fit locally : unsupervised learning of low dimensional manifolds. *Journal of Machine Learning Research*, 4:119–155, 2002.

[74] M. Schmitt. Descartes' rule of signs for radial basis function neural networks. *Neural Computation*, 14(12):2997–3011, 2002.

[75] B. Schölkopf, C. J. C. Burges et A. J. Smola. *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA, 1999.

[76] B. Schölkopf, A. Smola et K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

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

[78] N.N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

[79] J. Tenenbaum, V. de Silva et J.C.L. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, Dec. 2000.

[80] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8: 257–277, 1992.

[81] Y. Weiss. Segmentation using eigenvectors : a unifying view. Dans *Proceedings IEEE International Conference on Computer Vision*, pages 975–982, 1999.

[82] M. Welling, M. Rosen-Zvi et G.E. Hinton. Exponential family harmoniums with an application to information retrieval. Dans L.K. Saul, Y. Weiss et L. Bottou, éditeurs, *Advances in Neural Information Processing Systems 17*. MIT Press, 2005.

[83] C.K.I. Williams. Computing with infinite networks. Dans M.C. Mozer, M.I. Jordan et T. Petsche, éditeurs, *Advances in Neural Information Processing Systems 9*. MIT Press, 1997.

[84] C.K.I. Williams et C.E. Rasmussen. Gaussian processes for regression. Dans D.S. Touretzky, M.C. Mozer et M.E. Hasselmo, éditeurs, *Advances in Neural Information Processing Systems 8*, pages 514–520. MIT Press, Cambridge, MA, 1996.

[85] C.K.I. Williams et M. Seeger. The effect of the input density distribution on kernel-based classifiers. Dans *Proceedings of the Seventeenth International Conference on Machine Learning*. Morgan Kaufmann, 2000.

[86] D. Wolpert et W. Macready. No free lunch theorems for search. Rapport technique SFI-TR-95-02-010, The Santa Fe Institute, 1995.

[87] H. H. Yang et S. Amari. Natural gradient descent for training multi-layer perceptrons, 1997. URL `citeseer.ist.psu.edu/hua96natural.html`.

[88] H.H. Yang et S. Amari. Complexity issues in natural gradient descent method for training multi-layer perceptrons. *Neural Computation*, 10(8):2137–2157, 1998. URL `citeseer.ist.psu.edu/91462.html`.

[89] D. Zhou, O. Bousquet, T. Navin Lal, J. Weston et B. Schölkopf. Learning with local and global consistency. Dans S. Thrun, L. Saul et B. Schölkopf, éditeurs, *Advances in Neural Information Processing Systems 16*, Cambridge, MA, 2004. MIT Press.

[90] X. Zhu, Z. Ghahramani et J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. Dans *ICML'2003*, 2003.