Université de Montréal

Algorithmes d'apprentissage profonds supervisés et non-supervisés: applications et résultats théoriques

par Éric Thibodeau-Laufer

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

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

Septembre, 2013

RÉSUMÉ

La liste des domaines touchés par l'apprentissage machine s'allonge rapidement. Au fur et à mesure que la quantité de données disponibles augmente, le développement d'algorithmes d'apprentissage de plus en plus puissants est crucial. Ce mémoire est constitué de trois parties: d'abord un survol des concepts de bases de l'apprentissage automatique et les détails nécessaires pour l'entraînement de réseaux de neurones, modèles qui se livrent bien à des architectures profondes. Ensuite, le premier article présente une application de l'apprentissage machine aux jeux vidéos, puis une méthode de mesure performance pour ceux-ci en tant que politique de décision. Finalement, le deuxième article présente des résultats théoriques concernant l'entraînement d'architectures profondes non-supervisées.

Les jeux vidéos sont un domaine particulièrement fertile pour l'apprentissage automatique: il est facile d'accumuler d'importantes quantités de données, et les applications ne manquent pas. La formation d'équipes selon un critère donné est une tâche commune pour les jeux en lignes. Le premier article compare différents algorithmes d'apprentissage à des réseaux de neurones profonds appliqués à la prédiction de la balance d'un match. Ensuite nous présentons une méthode par simulation pour évaluer les modèles ainsi obtenus utilisés dans le cadre d'une politique de décision en ligne.

Dans un deuxième temps nous présentons une nouvelle méthode pour entraîner des modèles génératifs. Des résultats théoriques nous indiquent qu'il est possible d'entraîner par rétropropagation des modèles non-supervisés pouvant générer des échantillons qui suivent la distribution des données. Ceci est un résultat pertinent dans le cadre de la récente littérature scientifique investiguant les propriétés des autoencodeurs comme modèles génératifs. Ces résultats sont supportés avec des expériences qualitatives préliminaires ainsi que quelques résultats quantitatifs.

Mots clés: apprentissage machine, matchmaking, réseaux de neurones, autoencodeurs, recommendation de contenu, architectures profondes

ABSTRACT

The list of areas affected by machine learning is growing rapidly. As the amount of available training data increases, the development of more powerful learning algorithms is crucial. This thesis consists of three parts: first an overview of the basic concepts of machine learning and the details necessary for training neural networks, models that lend themselves well to deep architectures. The second part presents an application of machine learning to online video games, and a performance measurement method when using these models as decision policies. Finally, the third section presents theoretical results for unsupervised training of deep architectures.

Video games are a particularly fertile area for machine learning: it is easy to accumulate large amounts of data, and many tasks are possible. Assembling teams of equal skill is a common machine learning application for online games. The first paper compares different learning algorithms against deep neural networks applied to the prediction of match balance in online games. We then present a simulation based method to evaluate the resulting models used as decision policies for online matchmaking.

Following this we present a new method to train generative models. Theoretical results indicate that it is possible to train by backpropagation unsupervised models that can generate samples following the data's true distribution. This is a relevant result in the context of the recent literature investigating the properties of autoencoders as generative models. These results are supported with preliminary quantitative results and some qualitative experiments.

Keywords: machine learning, matchmaking, neural networks, autoencoders, content recommendation, deep architectures

Table des matières

Liste des figures

Liste des tableaux

Abréviations

MSE	**M**ean **S**quared **E**rror
NLL	**N**egative **L**og **L**ikelihood
DAE	**D**enoising **A**uto **E**ncoder
CAE	**C**ontractive **A**uto **E**ncoder
MLP	**M**ulti **L**ayer **P**erceptron
GT	**G**round **T**ruth
DM	**D**irect **M**ethod
IS	**I**mportance **S**ampling
DR	**D**oubly **R**obust
RBF	**R**adial **B**asis **F**unction
GSN	**G**enerative **S**tochastic **N**etwork
RBM	**R**estricted **B**oltzmann Machine

Notation

$\mathcal{D}$	Ensemble de données
$\mathbf{x}$	Vecteur d'entrée
y	Cible scalaire
$\mathbf{y}$	Cible vectorielle
θ	Paramètres d'une fonction, dénoté θ_t pour signifier les paramètres à une itération t
f_θ	Fonction paramétrisée par θ
$\mathcal{L}$	Fonction de perte, prenant en argument $f_\theta(\mathbf{x})$ et soit y, $\mathbf{y}$ ou $\mathbf{x}$
$\mathcal{V}(\mathcal{D}, \theta)$	Vraisemblance des données $\mathcal{D}$ étant donné les paramètres θ
$\mathbf{h}$	Unités cachées d'un réseau de neurones; transformation ou représentation apprises de l'entrée
$\mathbf{W}$	Poids associés à une couche cachée
$\mathbf{b}$	Biais associés à une couche cachée
f_{enc}	Fonction d'encodage associée à un autoencodeur
f_{dec}	Fonction de décodage associée à un autoencodeur
$\mathbf{r}$	Composition de $f_{dec} \circ f_{enc}$; reconstruction d'une entrée
$\vec{\nabla}\theta$	Gradient de $\mathcal{L}$ par rapport à θ: $\frac{\partial \mathcal{L}}{\partial \theta}$
I_{cond}	Fonction indicatrice; 1 si $cond$, 0 sinon
$tanh$	tangeante hyperbolique

À Gabrielle et Olivier...

Remerciements

Jamais n'aurais-je cru pouvoir cotoyer des gens aussi brillants et créatifs que dans ces deux dernières années qui ont passées si vite. D'abord je tiens à remercier Yoshua pour m'avoir invité dans ce monde d'exploration, de mystères et de découvertes qu'est la quête de l'intelligence artificielle. Son enthousiasme, sa passion et ses connaissances forment les piliers du LISA, laboratoire de classe mondiale dont j'ai eu la chance de faire partie. Merci à Myriam Côté, dont le dynamisme fut crucial à la réalisation des projets parfois ardus d'Ubisoft. Merci à Olivier Delalleau pour ses conseils et son regard minutieux qui nous sauva plus d'une fois. Merci à Frédéric Bastien pour avoir patiemment répondu à mes nombreuses questions. À tous les membres du LISA que j'ai cotoyé, ce fut un honneur de travailler avec vous. J'espère avoir aidé les nouveaux comme les anciens l'ont fait avec moi, car la pente à monter peut être intimidante au départ. Finalement, merci à ma famille ainsi que Marianne pour leurs encouragements soutenus.

Chapitre 1

Introduction

1.1 L'apprentissage machine

L'apprentissage machine est une branche de l'intelligence artificielle dédiée à l'étude de modèles et d'algorithmes capables d'apprendre automatiquement. Ceci implique résoudre une variété de tâches en se basant sur des données existantes, avec un minimum d'à priori humain. De plus en plus, les connaissances accumulées du monde sont stockées en format électronique. Une énorme quantité de documents, d'images, de sons, de vidéos et de discussions sont disponible grâce à l'internet. Dans le but de créer un système intelligent, il faudrait tirer avantage de l'ensemble du savoir humain contenu dans ces données. Celles-ci sont cependant très complexes dans leur forme brute; par exemple, un sous-ensemble infime de toutes les configurations possibles de pixels représentent une image naturelle. On peut imaginer la structure de ce sous-ensemble comme une surface extrêmement tordue, pliée et courbée à l'intérieur d'un cube, le cube représentant l'ensemble des configurations possibles. Cette surface est mieux représentée par des concepts abstraits. La sous-surface des images représentant un arbre est tout aussi complexe, mais il suffit de comprendre le concept d'arbre pour discerner lorsqu'une image se situe sur cette sous-surface ou non.

Le coeur du problème de l'apprentissage machine et que le lien entre la structure de cette surface et ce concept n'est pas stocké explicitement. Ce savoir est encodé implicitement dans notre cerveau lorsque nous apprenons des concepts nouveaux; c'est en partie ce qui nous permet d'agir de façon intelligente. De la même façon, un système intelligent devrait être capable de découvrir lui-même la structure d'un signal ou de données qui lui sont présentés. L'apprentissage machine est au croisement des statistiques, des probabilités, du calcul, de l'algèbre linéaire et de l'optimisation

numérique; parsemé d'allusions à la biologie et la psychologie. Main en main avec la croissance de la puissance de calcul et la quantité de données disponibles, la performance des systèmes d'apprentissage automatique ne cesse d'augmenter.

Ce premier chapitre couvre les concepts de bases de l'apprentissage machine. Le deuxième chapitre concerne les réseaux de neurones, classe de modèle inspiré par le fonctionnement du cerveau. Les spécifications, les variations et les détails nécessaire à leur bon fonctionnement seront couverts. Le troisième et quatrième chapitre présentent l'article principal de cet ouvrage: une application de l'apprentissage machine aux jeux vidéos. Finalement, le cinquième et sixième chapitre couvrent un second article présentant une nouvelle méthode pour entraîner des modèles dits génératifs.

1.2 Minimisation du risque

Le cadre classique des problèmes d'apprentissage machine est le suivant: nous avons accès a un ensemble de données $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^d$, où les $\mathbf{x}^i$ sont identiquement et indépendemment tirés d'une distribution sous-jacente $p(\mathbf{x})$, et les y^i tirés d'une distribution $p(y|\mathbf{x})$. On nomme $\mathbf{x}$ les entrées et y les cibles. En pratique, $\mathbf{x}$ réfère à un ensemble de variables auxquelles nous avons accès, tandis que y est un résultat qui dépend de $\mathbf{x}$. Nous n'avons pas nécéssairement accès à ce résultat autre que dans $\mathcal{D}$.

Nous distinguons, de façon générale, deux catégories d'apprentissage: supervisé et non-supervisé. Les problèmes d'apprentissage supervisé utilisent $\mathbf{x}$ pour former une approximation d'une propriété de y, comme l'espérance $\mathbb{E}[y|\mathbf{x}]$ ou la probabilité conditionelle $p(y|\mathbf{x})$. On retrouve deux principaux problèmes dans l'apprentissage supervisé: la régression et la classification. Lorsque y prend des valeurs quantitatives continues, nous avons à faire a un problème de régression; si y prend des valeurs catégoriques non ordonnées, un problème de classification. L'apprentissage non-supervisé a trait à la modélisation des entrées, soit en estimant la densité $p(\mathbf{x})$, en cherchant des encodages de $\mathbf{x}$ qui puissent être utiles par la suite ou en catégorisant les entrées de façon implicite (sans aide des cibles).

Ces tâches sont effectuées avec le principe de minimisation du risque empirique [77] . Le risque correspond à la valeur, donnée par une fonction de perte $\mathcal{L}$, qui indique la performance d'un modèle sur la tâche donnée. Par convention, $\mathcal{L}$ est décroissant en fonction de la performance du modèle. Nous voulons donc trouver une fonction $f(\mathbf{x})$ qui minimise le risque espéré. Dans le cadre de

l'apprentissage supervisé, le risque espéré est défini par:

$$\mathbb{E}[\mathcal{L}] = \int\int \mathcal{L}(f(\mathbf{x}), y)p(\mathbf{x}, y)\mathrm{d}\mathbf{x}\mathrm{d}y. \tag{1.1}$$

Pour l'apprentissage non-supervisé la fonction de coût est fonction de $\mathbf{x}$ et $f(\mathbf{x})$. Il est généralement impossible de calculer $\mathbb{E}[\mathcal{L}]$ exactement puisque nous n'avons pas accès à $p(\mathbf{x}, y)$, et nous n'avons qu'un ensemble de données $\mathcal{D}$ de taille finie. Grâce à celui-ci, nous obtenons une estimation:

$$\mathbb{E}[\mathcal{L}] \approx \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}^{(i)}), y^{(i)}) \tag{1.2}$$

nommée risque empirique. On cherche donc $f(\mathbf{x})$ qui minimise cette quantité.

Si f ne fait que mémoriser les exemples contenus dans $\mathcal{D}$, avec $f(\mathbf{x}^{(i)}) = y^{(i)}, (\mathbf{x}^{(i)}, y^{(i)}) \in \mathcal{D}$, le risque empirique sera nul. Cette approximation ne vaut rien puisque les exemples qui ont directement servis à optimiser f sont aussi utilisés pour estimer $\mathbb{E}[\mathcal{L}]$. Dans ce cas, l'estimation du risque espéré est biaisée de manière optimiste. Ce qui importe réellement est l'erreur de généralisation, c'est-à-dire le risque sur de nouveaux exemples tirés de $p(\mathbf{x})$. On dit d'un modèle qui performe bien sur de nouveaux exemples qu'il généralise correctement. Pour obtenir une estimation non-biaisée de l'erreur de généralisation, les exemples utilisés pour obtenir f et estimer le risque doivent être distincts. Il est pratique courante de diviser les données en deux ensembles: l'un pour l'optimisation de f, nommé ensemble d'entraînement ($\mathcal{D}_{train}$), et l'autre pour l'estimation du risque espéré, nommé ensemble de test ($\mathcal{D}_{test}$). Cette méthode simule la performance du modèle sur de nouveaux exemples. Le surapprentissage réfère à une bonne performance sur l'ensemble d'entraînement mais une mauvaise performance sur l'ensemble de test. Plus de détails sur les méthodes d'évaluations sont fournis à la section 2.6.1.

1.3 Types d'algorithmes d'apprentissage

Malgré la jeunesse du domaine de l'apprentissage machine, il existe déjà beaucoup d'algorithmes d'apprentissage différents, chacun possédant ses propres avantages. Nous distinguons deux catégories, paramétriques et non-paramétriques. Un algorithme d'apprentissage paramétrique est spécifié par l'apprentissage d'un nombre de paramètres fini et une famille de distribution que ces paramètres peuvent représenter. On nomme ces paramètres "paramètres libres" du modèle, puisque c'est eux que nous optimisons pour minimiser le risque empirique. Les algorithmes d'apprentissage paramétriques ont une capacité finie, limitée par le nombre de paramètres libres. La capacité d'un

modèle réfère à la quantité d'information qu'il peut encoder. Une faible capacité implique donc une faible performance d'entraînement et de test. Une grande capacité implique une bonne performance d'entraînement et un bon potentiel de performance de test accompagné de risque de surapprentissage. Un algorithme d'apprentissage dit "non-paramétrique" n'a pas d'à priori sur la distribution à représenter, ce qu'un nombre de paramètres fini implique. Ceux dont la capacité peut croître avec la quantité et la complexité des données sont donc dit non-paramétriques.

Quelques algorithmes paramétriques classiques incluent les régressions linéaires, les lois de probabilités spécifiées par un nombre de paramètres fixe et les réseaux de neurones à taille fixe. Par exemple, une loi normale dont la moyenne est une fonction linéaire de $\mathbf{x}$ est un modèle paramétrique : les paramètres sont les poids de la fonction linéaire et l'écart-type. L'ensemble des paramètres possibles définissent une famille de distributions que ce modèle peut représenter. Nous utiliserons la notation f_θ pour spécifier que f dépend des paramètres θ, et que c'est eux que nous cherchons à optimiser.

Parmi les algorithmes non-paramétriques on retrouve entre autres les arbres de décisions, les k plus proches voisins et les machines à vecteurs de support à noyau. Les fonctions approximées par ces algorithmes peuvent devenir arbitrairement complexe en fonction de la quantité de données disponibles. Sans contrainte de modélisation sur leur capacité (la profondeur d'un arbre de décision par exemple), l'approximation obtenue tend vers la vraie distribution estimée lorsque la quantité de données tend vers l'infini.

Dans le cas des plus proches voisins, ainsi que pour plusieurs autres modèles non-paramétriques, la dimensionalité des données peut être problématique. Imaginons le cas où la dimension des entrées est de taille 100, avec des variables d'entrées binaires. Même avec des milliards d'exemples il est impossible d'obtenir toute les configurations possibles, ni même une fraction raisonnable. Plus le nombre de dimensions est élevé, plus il est difficile de couvrir uniformément l'espace des configurations possibles, qui croît exponentiellement en fonction du nombre de dimensions. De plus, certaines intuitions géométriques acquises en trois dimensions sont erronées en haute dimension. Par exemple, imaginons un hypercube unitaire A défini par $x^d \in [0,1]^d$, et un sous-volume B contenu dans cet hypercube défini par $x^d \in [0,0.9]^d$. En trois dimensions, le volume de B est approximativement égal a 70% volume de A. Lorsque $d = 100$, cette fraction tombe à $\sim 10^{-5}\%$, malgré que chaque arête de B est presque de la même taille que les côtés de A! La vaste majorité du volume d'un hypercube en haute dimension est contenue dans les coins de celui-ci. Ce phénomène est surnommé fléau de la dimensionalité. C'est pourquoi une entrée possèdera typiquement peu de voisins au sens euclidien. Les seuls voisins seront des données qui partagent quelques variables

d'entrées proche. C'est une des raisons qui motive l'étude de modèles d'apprentissage qui apprennent a décomposer l'entrée en différentes caractéristiques, potentiellement communes à beaucoup d'exemples. Ces types d'algorithmes font usage d'une représentation dite distribuée. À l'opposé, une représentation dite locale est basée sur les données faisant partie du voisinage (bien que pas nécessairement au sens euclidien) d'un exemple.

Peu importe le type d'algorithme d'apprentissage, il y a des variables de contrôle qui ne sont pas directement optimisées via l'ensemble d'entraînement. On les nomme hyperparamètres: c'est eux qui contrôlent ce que l'algorithme peut apprendre. Le nombre de voisins utilisés dans les k plus proches voisins est un exemple, la taille d'un réseau de neurones ou la profondeur maximale d'un arbre de décision sont tous des hyperparamètres.

Couvrir en détail l'ensemble des différents algorithmes d'apprentissage est un travail colossal. Cet ouvrage se concentre sur les réseaux de neurones, couverts au prochain chapitre.

1.4 Fonctions de pertes

Le choix de la fonction de perte découle directement de la tâche que nous voulons résoudre. Pour la régression, la fonction de perte habituelle est l'erreur quadratique:

$$\mathcal{L}(y, f(\mathbf{x})) = (y - f_\theta(\mathbf{x}))^2. \tag{1.3}$$

Minimiser l'erreur quadratique correspond à estimer $\mathbb{E}[y|\mathbf{x}]$ [14, p. 46], une estimation souvent désirée. Une autre fonction de perte possible pour la régression est l'erreur absolue $L(y, f(\mathbf{x})) = |y - f(\mathbf{x})|$, puisque minimiser celle-ci correspond à estimer la médiane conditionelle à $\mathbf{x}$ de la cible [74, p. 43].

La fonction peut aussi être paramétrisée pour définir une fonction de densité, soit des cibles ou des entrées elle-mêmes. Un résultat intermédiaire nécessaire est l'espérance de cette densité; les autres moments peuvent être d'autres résultats intermédiaires du modèle ou mis fixes pour toute entrée, devenant ainsi des hyperparamètres. Par exemple, un modèle peut calculer une espérance et une variance en fonction de l'entrée, utilisées ensuite pour définir la densité d'une loi normale. Pour expliciter un problème supervisé, la notation $f_\theta(\mathbf{x})_y$ signifie que f est une fonction de densité évaluée en y, avec $\int f_\theta(\mathbf{x})_y \mathrm{d}y = 1$. Donné un ensemble d'observations $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$, la

vraisemblance de $\mathcal{D}$ étant donné θ est définie par:

$$\mathcal{V}(\mathcal{D}, \theta) = p(\mathcal{D}|\theta) = p(\{y^{(i)}\}_{i=1}^{N}|\{\mathbf{x}^{(i)}\}_{i=1}^{N}, \theta) = \prod_{i=1}^{N} f_\theta(\mathbf{x}^{(i)})_{y^{(i)}} \tag{1.4}$$

Il est généralement plus commode, en partie pour des raisons numériques, de travailler avec la log-vraisemblance. L'équation précédente devient:

$$\log \mathcal{V}(\mathcal{D}, \theta) = \log \prod_{i=1}^{N} f_\theta(\mathbf{x}^{(i)})_{y^{(i)}} = \sum_{i=1}^{N} \log f_\theta(\mathbf{x}^{(i)})_{y^{(i)}} \tag{1.5}$$

Pour suivre la convention, maximiser la log-vraisemblance équivaut à minimiser la log-vraisemblance négative (NLL). Nous pouvons alors interpréter la NLL comme notre fonction de perte, avec $\mathcal{L}(f_\theta(\mathbf{x}), y) = -\log f_\theta(\mathbf{x})_y$. En ajustant les paramètres pour minimiser la NLL, $f_\theta(\mathbf{x})$ nous donne une approximation de $p(y|\mathbf{x})$. Cette méthode est nommée estimation du maximum de vraisemblance.

Face à un problème de classification, on considère naturellement l'erreur de classification comme mesure de performance d'un algorithme d'apprentissage. Si $y, f_\theta(\mathbf{x}) \in \{1, ..., K\}$, l'erreur de classification est définie par:

$$\mathcal{L}(y, f(\mathbf{x})) = \mathrm{I}_{y \neq f(x)}, \tag{1.6}$$

moyenné sur l'ensemble de données. Cependant, il est souvent nécessaire d'utiliser une fonction de perte différentiable, à des fins d'optimisation. Puisque l'erreur de classification ne l'est pas, nous utilisons une estimation du maximum de vraisemblance et un modèle probabiliste de classification. Dans le cas binaire, nous avons $K = 2$; nous considérons cependant que $y = \{0, 1\}$, par convention. Supposons que y est tiré de $p(y|x)$, et $f_\theta(\mathbf{x}) \in [0, 1]$. Ceci nous permet d'interpréter $f_\theta(\mathbf{x})$ comme $p(y = 1|x)$ et $1 - f_\theta(\mathbf{x})$ comme $p(y = 0|x)$. La vraisemblance des données selon le modèle est alors:

$$p(\mathcal{D}|\theta) = \prod_{i=1}^{N} f_\theta(\mathbf{x})^{y^{(i)}} (1 - f_\theta(\mathbf{x}))^{1-y^{(i)}}, \tag{1.7}$$

et la log-vraisemblance négative:

$$-\log p(\mathcal{D}|\theta) = -\sum_{i=1}^{N} (y^{(i)} \log f_\theta(\mathbf{x}) + (1 - y^{(i)}) \log(1 - f_\theta(\mathbf{x}))) \tag{1.8}$$

appelée l'entropie croisée binaire [14, p. 206]. Pour une classification multiclasse, nous encodons $y \in 1, ..., K$ avec $\mathbf{y} = [\mathbf{y}_1, ..., \mathbf{y}_K]$, $\mathbf{y}_c = \mathrm{I}_{y=c}$. Le modèle est paramétrisé pour que $\sum_{c=1}^{K} f_\theta(\mathbf{x})_c =$

1, et nous interprétons $f_\theta(\mathbf{x})_c$ comme $p(y = c|x)$. Nous retrouvons l'entropie croisée comme log-vraisemblance:

$$-\log \mathcal{V}(\mathcal{D}, \theta) = -\log \prod_{i=1}^{N} \prod_{k=1}^{K} f_\theta(\mathbf{x}^{(i)})_k^{\mathbf{y}_k^{(i)}} = -\sum_{i=1}^{N} \sum_{k=1}^{K} \mathbf{y}_k^{(i)} \log f_\theta(\mathbf{x}^{(i)})_k, \tag{1.9}$$

et donc comme fonction de coût. Minimiser l'entropie croisée peut aussi être utilisée lorsque les cibles sont réelles et interprétées comme des probabilités, par exemple, la moyenne des résultats d'une expérience de classification répétée plusieurs fois pour une même entrée. Rien n'empêche l'utilisation de l'erreur de classification comme mesure finale de la qualité d'un modèle, comme c'est souvent le cas. La classe la plus probable donnée par le modèle est la classe prédite dans ce cas.

Pour un problème d'estimation de densité, $f_\theta(\mathbf{x})$ doit définir une fonction densité sur les entrées, avec $\int f_\theta(\mathbf{x})\mathrm{d}\mathbf{x} = 1$. Pour trouver les paramètres, il s'agit de maximiser la vraisemblance des données sous la distribution définie par θ, comme dans l'équation 1.5. La log-vraisemblance est alors:

$$\log \mathcal{V}(\mathcal{D}, \theta) = \log p(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}|\theta) = \log \prod_{i=1}^{N} f_\theta(\mathbf{x}^{(i)}) = \sum_{i=1}^{N} \log f_\theta(\mathbf{x}^{(i)}) \tag{1.10}$$

1.5 Optimisation par gradient

Nous supposerons dorénavant que $f_\theta(\mathbf{x})$ est continue et différentiable presque partout. La fonction de coût $\mathcal{L}(f_\theta(\mathbf{x}), y)$ en fonction de θ définit une surface d'erreur, sur laquelle nous pouvons naviguer en modifiant θ. Si cette surface est convexe, nous cherchons le minimum global de celle-ci; si la surface n'est pas convexe, il est difficile de trouver le minimum global. Dans ce cas, nous nous contentons généralement d'un minimum local suffisament bon. Beaucoup de méthodes d'optimisation numérique existent, et sont une partie importante des algorithmes d'apprentissage. Lorsque f est différentiable, la descente de gradient en est une des plus populaires, particulièrement pour les reseaux de neurones depuis [68]. Le gradient de la fonction de coût par rapport aux paramètres nous donne la direction de la pente maximale de la surface d'erreur. Une mise à jour des paramètres en effectuant un déplacement le long de cette pente nous permet d'augmenter ou diminuer le coût empirique. L'algorithme de descente de gradient (1) consiste essentiellement à itérer l'évaluation du gradient et la mise à jour des paramètres jusqu'à ce qu'une condition d'arrêt soit rencontrée. Nous utilisons comme condition d'arrêt de base le nombre d'itérations effectuées.

Algorithm 1 Descente de gradient

Require: $\mathcal{D}, \mathcal{L}, f_\theta, \theta_0, \epsilon, T$

$\tilde{\theta} \leftarrow \theta_0$

$t \leftarrow 0$

while $t < T$ **do**

$\vec{\nabla}\theta \leftarrow \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(f_\theta(\mathbf{x}^{(i)}), y^{(i)})}{\partial \theta}\Big|_{\theta=\tilde{\theta}}$

$\tilde{\theta} \leftarrow \tilde{\theta} - \epsilon\vec{\nabla}\theta$

$t \leftarrow t + 1$

end while

return $\tilde{\theta}$

Le taux d'apprentissage, ϵ, est un facteur multiplicatif qui contrôle la norme du vecteur utilisé pour la mise à jour des paramètres. On le considère généralement comme un hyperparamètre propre à l'algorithme d'apprentissage, mais une distinction possible se fait pour les hyperparamètres d'optimisation. C'est l'hyperparamètre le plus important de la descente de gradient: trop grand et les paramètre divergeront, trop petit et la convergence sera trop lente, et la performance sous-optimale. Non seulement cela, mais le taux d'apprentissage joue un rôle particulier lors de l'optimisation de fonctions non convexes. Puisque celles-ci possèdent plusieurs minima locaux, le taux d'apprentissage a un impact direct sur lequel de ces minima les paramètres convergeront. Nous pouvons imaginer une surface d'erreur ressemblant un bol, parsemée de petits creux, des minima locaux. Avec un très petit taux d'apprentissage, les paramètres peuvent se retrouver pris dans le premier creux rencontré, tandis qu'un grand taux d'apprentissage passera par dessus. D'un autre côté un grand taux peut rater ou osciller autour d'un bon minimum local. Bien que dans un contexte d'optimisation pur l'algorithme s'arrêterais lorsque $\mathcal{L}$ est inférieur à un seuil prédéfini, il n'est pas garanti que le minimum empirique soit nécessairement optimal. La surface d'erreur définie par les données est une approximation de la vrai surface d'erreur. Pour cette raison, nous lissons la surface d'erreur ou nous arrêtons l'entraînement lorsqu'un seuil de performance est atteint sur un ensemble de test alternatif (méthode nommée *early stopping*, l'ensemble alternatif correspond à l'ensemble de validation, vu en 2.6.1).

Bien que les propriétés de convergence de la descente de gradient dite *batch* sont bien définies (garanties cite), celle-ci peut être accélérée via une approximation stochastique. Au lieu de calculer la somme ou la moyenne du gradient sur tout les exemples d'entraînement, on observe que:

$$\frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(f_\theta(\mathbf{x}^{(i)}), y^{(i)})}{\partial \theta} = \mathbb{E}\left[\frac{\partial f_\theta(\mathbf{x}^{(k)}), y^{(k)})}{\partial \theta}\right], \; k \sim U(1, ..., n), \tag{1.11}$$

ce qui nous permet d'utiliser un seul exemple, aléatoirement choisi, pour calculer une estimation non-biaisée du gradient des paramètres. Il y a plusieurs avantages à cette méthode, nommée descente de gradient stochastique. Non seulement le coût de calcul est grandement réduit, mais la convergence est aussi plus rapide en terme de nombre de pas de gradients, et ce malgré le bruit ajouté par l'estimation du gradient. En fait, grâce à ce bruit, de meilleures solutions peuvent être obtenues pour les problèmes non-convexes, parce que les paramètres s'échapperons plus facilement des minima locaux [55].

Nous pouvons interpréter la descente de gradient comme l'utilisation d'une série de Taylor pour obtenir une approximation linéaire de la fonction que nous souhaitons optimiser, et supposer que cette approximation est assez bonne dans un petit rayon pour faire un pas de gradient. De la même façon, on peut utiliser une approximation quadratique avec la dérivé partielle de second ordre. La méthode d'optimisation de Newton est l'exemple classique d'optimisation de second ordre. Pour un grand nombre de paramètres, cette méthode est généralement difficile à utiliser. C'est pourquoi il existe des méthodes dites quasi-Newtoniennes, qui approximent les dérivées partielles de second ordre ou qui ne les calculent pas explicitement. Parmi ces méthodes on retrouve L-BFGS, Hessian Free, les méthodes à gradient conjugé et autres.

Chapitre 2

Réseaux de Neurones

2.1 Modèles linéaires

Les modèles linéaires sont parmis les modèles paramétriques plus simples. La régression linéaire simple suppose que la cible est une somme pondérée des variables d'entrée. Le modèle prend la forme:

$$f_\theta(x) = \sum_{i=1}^{d} x_i w_i + b \tag{2.1}$$

Étant contraint à ne représenter qu'un hyperplan, une définition plus générale de la régression linéaire consiste à admettre une transformation fixe des variables d'entrée via un ensemble de fonctions φ_i. Ceci correspond à:

$$f_\theta(x) = \sum_{i=1}^{d} \varphi(x_i) w_i + b \tag{2.2}$$

En statistique, la définition des modèles linéaires généralisés inclut une fonction appliquée au résultat de 2.1, appelée fonction de lien. Cette formulation est définie par:

$$f_\theta(x) = g(\sum_{i=1}^{d} x_i w_i + b) \tag{2.3}$$

La fonction g est généralement choisie d'après la distribution des cibles. Pour les trois modèles précédents, les paramètres sont $\theta = \{\mathbf{w}, b\}$, où $\mathbf{w} = [w_1, ..., w_d]$.

Un classifieur linéaire simple est de prendre le modèle 2.1, et de considérer l'hyperplan défini comme frontière de décision pour séparer deux classes. Ce modèle se nomme le Perceptron, et est un classifieur linéaire binaire. Cependant le gradient associé à ce modèle et la fonction de

coût 1.9 est zéro presque partout, et non différentiable à la frontière. Afin d'utiliser une optimisation à base de gradient, nous introduisons une version probabiliste de la classification binaire. La régression logistique est un modèle linéaire standard pour la classification binaire. Formellement, on considère les cibles y comme des variables de Bernoulli, prenant les valeurs 0 ou 1, avec une probabilité $p(y = 1|x)$. On modélise cette probabilité conditionelle par:

$$\begin{aligned} f_\theta(x) &= \sigma(\sum_{i=1}^{d} x_i w_i + b) \\ \sigma(x) &= \frac{1}{1 + e^{-x}} \end{aligned} \tag{2.4}$$

La régression logistique est un modèle linéaire généralisé: σ, nommée fonction sigmoïde, bornée entre 0 et 1 sert de fonction de lien. Les paramètres sont les mêmes que dans les modèles linéaires. On interprète $f(x)$ comme une approximation de $p(y = 1|x)$. Pour trouver les paramètres $\mathbf{w}$ et b on minimise l'entropie croisée, définie en 1.8.

La régression logistique multinomiale est une généralisation de la régression logistique pour un problème de classification multiclasse. La probabilité d'appartenance à une classe k est posée comme étant proportionnelle à $e^{\mathbf{x}\mathbf{W}_k+\mathbf{b}_k}$. Les rangées de la matrice $\mathbf{W} : k \times d$ correspondent aux poids des entrées pour chaque classe, et $\mathbf{b}_k$ les biais pour chacune d'elle. Il suffit ensuite d'avoir un facteur de normalisation pour conserver la sommation unitaire des probabilités d'appartenances à toute les classes. Le modèle est donc:

$$f_\theta(\mathbf{x}) = \mathrm{softmax}(\mathbf{x}\mathbf{W} + \mathbf{b}) = \frac{\mathrm{e}^{\mathbf{x}\mathbf{W}_\mathrm{k}+\mathbf{b}_\mathrm{k}}}{\sum_\mathrm{k} \mathrm{e}^{\mathbf{x}\mathbf{W}_\mathrm{k}+\mathbf{b}_\mathrm{k}}} \tag{2.5}$$

La fonction dans l'équation 2.5 se nomme "softmax", et son résultat est interprété comme les probabilités d'une loi multinomiale. Les paramètres ici sont $\theta = \{\mathbf{W}, \mathbf{b}\}$, et la fonction de coût est encore l'entropie croisée (1.9). Les modèles linéaires, bien qu'utiles pour leur simplicité, sont souvent sous-optimaux pour des données moindrement complexes, pour lesquelles une relation linéaire entre les entrées et les cibles est insuffisante. Cependant, combinés à des transformations non-linéaires qui rendent les données linéairement séparables ou qui démêlent les facteurs de variations pour obtenir des caractérisques plus indépendantes et moins corrélées entre elles, les modèles linéaires servent souvent d'étape finale pour la régression et la classification.

2.2 Réseaux de neurones artificiels

Le fonctionnement du cerveau humain est une merveille dont la compréhension ne fait que débuter. Quel meilleur modèle pour comprendre le monde qui nous entoure? Il est possible de débattre qu'émuler le cerveau, ou du moins extraire les principes qui permettent son fonctionnement, soit crucial pour la création d'une intelligence artificielle. Sans entrer dans cette discussion, certains concepts ont inspirés l'élaboration de modèles qui maintenant offrent des performances à l'état de l'art pour une multitude de tâches.

2.2.1 Réseaux supervisés

Les réseaux de neurones artificiels forment une famille de modèles inspirés par le fonctionnement du neurone, cellule à la base du cerveau. De manière (extrêmement) simplifiée, les neurones fonctionnent en reçevant le signal électrique provenant des neurones adjaçents et en émettant le leurs lorsque le signal reçu est assez fort. Les réseaux de neurones imitent ce fonctionnement à l'aide de neurones artificiels, aussi appelés unités cachées. La sortie d'une unité est essentiellement un modèle linéaire généralisé appliqué aux sorties d'un autre ensemble d'unités cachées. La fonction de lien des unités cachées est typiquement appelée "fonction d'activation". Les ensembles d'unités sont organisés en couches : les unités d'une couche reçoivent les sorties des unités appartenant à la couche précédente. Ces couches sont appelées les "couches cachées" du réseau. Voir [14, p. 225] pour un survol plus en détail.

Le réseau classique est constitué de la couche d'entrée, une ou plusieurs couches cachées puis d'une couche de sortie. La couche d'entrée correspond directement aux entrées $\mathbf{x}$, et la couche de sortie est généralement un des modèles linéaires décrit en 2.1, prenant en entrée la dernière couche cachée du réseau. Chaque couche cachée ainsi que la couche de sortie possède ses propres paramètres. Pour un réseau à une couche cachée (dénotée par $\mathbf{h}$), on a:

$$\begin{aligned} \mathbf{h} &= f_{act}(\mathbf{x}\mathbf{W} + \mathbf{b}) \\ f_\theta(x) &= g(\mathbf{h}\mathbf{V} + \mathbf{c}), \end{aligned} \tag{2.6}$$

Les hyperparamètres du réseau sont n_h et le choix de f_{act}. Les paramètres sont $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}\}$. Avec n_{out} spécifié par la tâche en question, on a $\mathbf{W} : d \times n_h$, $\mathbf{b} : 1 \times n_h$, $\mathbf{V} : n_h \times n_{out}$, $\mathbf{c} : 1 \times n_{out}$.

La définition récursive suivante généralise 2.6 pour L couches cachées:

$$\begin{aligned} \mathbf{h}^{(0)} &= \mathbf{x} \\ \mathbf{h}^{(l)} &= f_{act}(\mathbf{h}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}),\ l = 1, ..., L \\ f_\theta(\mathbf{x}) &= g(\mathbf{h}^{(L)}\mathbf{V} + \mathbf{c}) \end{aligned} \tag{2.7}$$

Ceci introduit de nouveaux hyperparamètres : chaque couche possède sa propre taille n_h^l et L contrôle la profondeur du réseau. Pour un problème de régression, g est hatibuellement l'identité, ou une fonction de lien appropriée pour une distribution à priori lorsque nous désirons estimer les paramètres par maximum de vraisemblance, comme décrit à la section 1.4. Dans un problème de classification, la sortie du réseau est habituellement la fonction $softmax$ appliquée à une transformation linéaire de dimension appropriée de la dernière couche cachée. Lors de l'optimisation, les couches cachées apprendront alors à transformer les entrées pour que les classes soit linéairement séparables.

Lorsque nous approximons plusieurs cibles simultanément, avec $n_{out} > 1$, nous avons un modèle "multi-tâche". Il est possible d'obtenir de meilleurs résultats en partageant la couche cachée pour plusieurs tâches. En forçant le réseau à apprendre une représentation commune via les couches cachées, les différentes cibles s'entraident mutuellement si celles-ci ne sont pas indépendantes. Puisque le gradient des paramètres est une fonction des différentes cibles, les couches cachées apprennent à représenter de l'information utile à chaque cible, et donc les cibles ont implicitement accès à une certaine quantité d'information qui ne serait pas accessible sans entraînement multi-tâche [11, 22].

La fonction sigmoïde fut un des premiers choix de fonction d'activation (f_{act}) pour les réseaux de neurones. Cependant, la tangeante hyperbolique ($tanh$) est généralement préférable pour des raisons d'optimisation [55]. Une autre fonction d'activation récemment populaire [40, 52] est le rectifieur, une fonction linéaire par partie. Le rectifieur est défini par:

$$rect(x) = max(0, x) \tag{2.8}$$

Cette fonction d'activation est intéressante en partie pour sa capacité à atténuer le problème de disparition du gradient. Ce phénomème a lieu dans les couches inférieures d'un réseau profond. Lors du calcul de la dérivée partielle du coût par rapport aux paramètres, la dérivée des fonctions d'activations de chaque couche supérieure fait partie de la chaîne multiplicative nécessaire pour obtenir le gradient des poids d'une couche inférieure. Puisque la dérivée des fonctions d'activations habituelles comme $sigmoid$ et $tanh$ est positive et inférieure à 1, la norme du gradient des poids

dans les couches inférieures est plus petite que celle aux couches supérieures. L'apprentissage est alors dominé par les mises à jours des couches supérieures. La sigmoïde souffre davantage de ce problème car non seulement la dérivée est bornée à 0.25, mais sa valeur moyenne est 0.5, ce qui peut saturer plus facilement les couches supérieures [39]. Quelques succès ont été reportés avec des réseaux profonds utilisant tanh [25].

Il a été démontré qu'un réseau de neurones à une couche cachée est un approximateur universel: avec suffisament d'unités cachées et des conditions faibles sur la fonction d'activation, il est possible d'approximer n'importe quelle fonction continue avec un niveau de précision arbitraire [49]. Des motivations théoriques indiquent aussi que des fonctions complexes peuvent être approximées de façon plus compacte (moins de paramètres) en utilisant des réseaux dits profonds, comportant plus de deux couches cachées [5]. De plus, un ensemble de résultats empiriques à l'état de l'art supportent cette hypothèse et motivent la continuation de l'étude des réseaux de neurones artificiels, particulièrement pour les réseaux profonds.

2.2.2 Réseaux non-supervisés

Les autoencodeurs sont des réseaux de neurones entraînés à prédire leur propre entrée : la cible y est en fait $\mathbf{x}$. Ce sont des modèles non supervisés : nous n'utilisons pas les cibles pour estimer une espérance ou probabilité conditionelle fonction de $\mathbf{x}$. L'intérêt n'est pas la prédiction de l'entrée en tant que tel, mais la transformation apprise par l'autoencodeur, qui sert de représentation alternative de l'entrée. Avec une couche cachée, on a:

$$\begin{aligned} \mathbf{h} &= f_{enc}(\mathbf{x}\mathbf{W}_{enc} + \mathbf{b}) \\ \mathbf{r} &= f_{dec}(\mathbf{h}\mathbf{W}_{dec} + \mathbf{c}), \end{aligned} \tag{2.9}$$

où $\mathbf{h}$ correspond à la transformation de l'entrée (aussi appelée encodage), et $\mathbf{r}$ la reconstruction de celle-ci. Tout comme un réseau supervisé, les hyperparamètres sont n_h, la taille de $\mathbf{h}$, et le choix de f_{enc}. Les paramètres sont $\theta = \{\mathbf{W}_{enc}, \mathbf{W}_{dec}, \mathbf{b}, \mathbf{c}\}$. Les poids du réseau, $\mathbf{W}_{enc}$ et $\mathbf{W}_{dec}$, peuvent être distincts ou les mêmes, avec $\mathbf{W}_{dec} = \mathbf{W}_{enc}^T$. Pour un $\mathbf{x}$ binaire, le coût à optimiser est généralement l'entropie croisée moyenne de la reconstruction de chaque variable d'entrée:

$$\mathcal{L}(\mathbf{x}, \mathbf{r}) = \sum_{i=1}^{d} -(\mathbf{x_i} \log(\mathbf{r_i}) + (1 - \mathbf{x_i}) \log(1 - \mathbf{r_i})), \tag{2.10}$$

l'entropie croisée entre $\mathbf{x}$ et $\mathbf{r}$. Pour un $\mathbf{x}$ continu on peut utiliser l'erreur quadratique, et l'autoencodeur estimera aussi $\mathrm{E}[\mathbf{x}|\mathbf{h}]$. Le choix de f_{dec} et de la fonction de perte, comme les réseaux supervisés,

dépend de la tâche et de la distribution des données. Dans le cadre des autoencodeurs, on nomme $\mathcal{L}(\mathbf{x}, \mathbf{r})$ coût de reconstruction. Pour une couche cachée plus petite que l'entrée, le réseau applique une compression avec perte. En entraînant un réseau avec cette propriété, les paramètres chercherons une transformation compacte de l'entrée qui conserve un maximum d'information, du moins pour les exemples d'entraînement, témoigné par un faible coût de reconstruction. Une entrée provenant d'une distribution différente aura un coût de reconstruction élevé.

Si la taille de la couche cachée est supérieure à celle de l'entrée, la représentation apprise est dite surcomplète. Cependant, sans aucune autre contrainte que le coût de reconstruction, l'autoencodeur peut apprendre la fonction indentité. Cette solution comporte peu d'intérêt puisque rien ne garantit que la représentation apprise ne sera pas une permutation et/ou un changement d'échelle des entrées, avec des unités cachées redondantes. À priori, une représentation surcomplète semble superflue, mais possède plusieurs attraits. D'abord il est possible d'apprendre une représentation dite éparse, où seul quelques éléments ne sont pas nuls, ce qui permet de simuler une représentation de taille variable. Une telle représentation peut mieux démêler les facteurs de variations des variables d'entrées, contrairement à une réduction de dimensionalité, est plus susceptible d'être linéairement séparable et peut avoir des propriétés débruitantes [58]. Ces propriétés sont liées fait qu'une représentation surcomplète permet de capter la distribution des entrées: chaque unité cachée correspond à une direction locale de variation, ce qui nous permet de capter une variété non-linéaire arbitraire.

Introduit par [78], les autoencodeurs débruitants (DAE) introduisent une modification au coût de reconstruction qui permet à l'autoencodeur de capter la distribution des entrées de manière prouvable. L'intuition est simple : minimiser le coût entre l'entrée originale et la reconstruction d'une version corrumpue. Le réseau apprendra ainsi à débruiter l'entrée, et produire une reconstruction plus probable, selon les exemples d'entraînement, que la version corrompue de l'entrée. Mathématiquement, ceci correspond à:

$$\begin{aligned} \tilde{\mathbf{x}} &\sim p(\tilde{\mathbf{x}}|\mathbf{x}) \\ \mathbf{h} &= f_{enc}(\tilde{\mathbf{x}}\mathbf{W}_{enc} + \mathbf{b}) \\ \mathbf{r} &= f_{dec}(\mathbf{r}\mathbf{W}_{dec} + \mathbf{c}), \end{aligned} \tag{2.11}$$

Le processus de corruption, $\tilde{\mathbf{x}} \sim p(\tilde{\mathbf{x}}|\mathbf{x})$, est au choix et dépend de la nature des données. Les coûts de reconstruction utilisés sont les mêmes que dans un autoencodeur standard. Parmi les choix de corruption populaires il y a le bruit "poivre et sel", pour des entrées binaires, où un sous-ensemble aléatoire des variables d'entrées sont mis à 0 ou 1 avec probabilité égale. Pour des entrées continues, du bruit gaussien centré est un choix raisonnable. Le choix de la proportion des variables d'entrées

corrompues ou l'écart-type du bruit sont des hyperparamètres. Lorsque nous désirons encoder ou débruiter une entrée en pratique, la corruption n'est pas utilisé.

Les autoencodeurs contractants (CAE) [65] sont une autre variation, non-stochastique, qui possèdent des propriétés semblables à celles des autoencodeurs débruitants. Le principe est d'ajouter à la fonction de coût un terme qui pénalise les variations des unités cachées en fonction des entrées. Ce terme correspond à la somme des éléments au carré de la matrice jacobienne des unités cachées et des entrées. Ceci équivaut à $\mathcal{L}_J = \sum_{i,j} J_{i,j}^2$, avec $J_{i,j} = \frac{\partial \mathbf{h}_i}{\partial \mathbf{x}_j}$. L'optimisation doit donc balancer une reconstruction correcte et une couche cachée résistante aux variations de l'entrée. Le résultat est l'apprentissage d'une représentation sensible aux facteurs de variations des entrées, mais qui ignore les directions orthogonales aux facteurs de variations. Rappelons l'analogie de la surface courbée décrite à la section 1.1, appelée communément variété. Nous pouvons interpréter la représentation apprise (unités cachées de l'autoencodeur) comme des coordonnées qui situent les exemples sur la variété. Le CAE permet de mieux approximer la variété car nous ignorons les facteurs de variations qui ne sont pas important pour encoder un exemple. La couche cachée ne peut alors être décodée que sur un point proche de la variété. Le DAE obtient un effet similaire avec la corruption: en bruitant l'entrée, nous éloignons celle-ci de la variété mais la reconstruction doit la rapprocher. Les unités cachées doivent donc représenter la position sur la variété la plus proche de l'exemple corrompu, c'est-à-dire orthogonale à celle-ci. Il est possible de faire un lien analytique entre le débruitage et la pénalisation de la matrice jacobienne, voir [1] pour une discussion discussion détaillée du sujet, et une analyse des propriétés des autoencodeurs.

2.3 Optimisation

La méthode d'optimisation de loin la plus populaire pour trouver les paramètres d'un réseau de neurones est la descente de gradient stochastique. Celle-ci produit en général les meilleurs résultats pour les réseaux de neurones classiques, décrits dans la section précédente.

Cette section couvre les bases de l'utilisation de la descente de gradient pour les réseaux de neurones ainsi que des extensions améliorant la précision ou la vitesse de convergence de l'optimisation.

2.3.1 Rétropropagation

Pour un réseau suffisament gros, la descente de gradient implique potentiellement l'évaluation de millions de dérivées partielles, de plus en plus lourdes en calcul pour les paramètres des couches

inférieures du réseau. La rétropropagation [68] est un algorithme combinant des principes simples de calcul et de programmation dynamique afin de calculer efficacement le gradient des paramètres. On sait que:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x},$$

la règle de dérivation en chaîne. Un réseau de neurones n'étant essentiellement qu'une série d'opérations mathématique, nous pouvons appliquer la règle de dérivation en chaîne au maximum sur celle-ci. Ce faisant, nous remarquons que certains termes sont communs dans la formule de la dérivée du coût par rapport à différents paramètres.

Pour alléger la notation, on utilise $\mathcal{L}$ au lieu de $\mathcal{L}(f_\theta(x), y)$. Nous supposons que le réseau produit une sortie scalaire. Le coût pour une sortie vectorielle, comme l'entropie croisée avec la classification multiclasse, est la somme des coûts pour chaque sortie individuellement, et donc le gradient final est la somme des gradients pour chaque coût. Nous voulons calculer la dérivée de $\mathcal{L}$ par rapport aux paramètres $\theta = \{\mathbf{W}^1, \mathbf{b}^1, ..., \mathbf{W}^L, \mathbf{b}^L, \mathbf{V}, \mathbf{c}\}$. Au lieu de la calculer séparément pour tout les paramètres, on calcule d'abord l'état du réseau pour un exemple $\mathbf{x}$ donné. Ceci nous donne les valeurs pour la sortie et les couches cachées du réseau. Observons que:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{h}^L} \frac{\partial \mathbf{h}^L}{\partial \mathbf{h}^{L-1}} \cdots \frac{\partial \mathbf{h}^l}{\partial \mathbf{W}^l} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}^l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{h}^L} \frac{\partial \mathbf{h}^L}{\partial \mathbf{h}^{L-1}} \cdots \frac{\partial \mathbf{h}^l}{\partial \mathbf{b}^l} \end{aligned} \tag{2.12}$$

Puisque $\mathcal{L}$ est une fonction de $\mathbf{h}^L$ et que $\mathbf{h}^l$ est une fonction de $\mathbf{h}^{l-1}$, chacun des termes du produit peut être calculé indépendamment, et évalué grâce à l'état des couches cachées, donné par $\mathbf{x}$. Non seulement cela, mais ces termes sont communs à toutes les couches. En les évaluant une seule fois, il suffit de calculer $\frac{\partial \mathcal{L}}{\partial \mathbf{h}^L}\Big|_{\mathbf{x}}$, $\frac{\partial \mathbf{h}^l}{\partial \mathbf{h}^{l-1}}\Big|_{\mathbf{x}}$, $\frac{\partial \mathbf{h}^l}{\partial \mathbf{W}^l}\Big|_{\mathbf{x}}$, $\frac{\partial \mathbf{h}^l}{\partial \mathbf{b}^l}\Big|_{\mathbf{x}}$, $\frac{\partial f(\mathbf{x})}{\partial \mathbf{V}}\Big|_{\mathbf{x}}$ et $\frac{\partial f(\mathbf{x})}{\partial c}\Big|_{\mathbf{x}}$ pour $l = 2, ..., L$ et nous obtenons tous les éléments nécéssaires pour calculer le gradient des éléments de θ. En substituant $o = \mathbf{h}^L\mathbf{V} + c$ et $\mathbf{a}^l = \mathbf{h}^{l-1}\mathbf{W}^l + \mathbf{b}^l$ et en décomposant la chaîne de dérivées davantage on observe aussi que:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{V}} &= \frac{\partial \mathcal{L}(g(o), y)}{\partial g(o)} \frac{\partial g(o)}{\partial o} \frac{\partial o}{\partial \mathbf{V}} \\ \frac{\partial \mathcal{L}}{\partial c} &= \frac{\partial \mathcal{L}(g(o), y)}{\partial g(o)} \frac{\partial g(o)}{\partial o} \frac{\partial o}{\partial c} \end{aligned} \tag{2.13}$$

$$\begin{aligned} \frac{\partial \mathbf{h}^l}{\partial \mathbf{h}^{l-1}} &= \frac{\partial f_{act}(\mathbf{a}^l)}{\partial \mathbf{a}^l} \frac{\partial \mathbf{a}^l}{\partial \mathbf{h}^{l-1}} \\ \frac{\partial \mathbf{h}^l}{\partial \mathbf{W}^l} &= \frac{\partial f_{act}(\mathbf{a}^l)}{\partial \mathbf{a}^l} \frac{\partial \mathbf{a}^l}{\partial \mathbf{W}^l} \end{aligned}$$

avec

$$\begin{aligned} \frac{\partial o}{\partial \mathbf{V}_i} &= \frac{\partial (\mathbf{h}^L \mathbf{V} + \mathbf{c})_k}{\partial \mathbf{V}_i} = \mathbf{h}_i^L & \frac{\partial \mathbf{a}_k^l}{\partial \mathbf{W}_{i,j}^l} &= \frac{\partial (\mathbf{h}^{l-1} \mathbf{W}^l + \mathbf{b}^l)_k}{\partial \mathbf{W}_{i,j}} = \begin{cases} \mathbf{h}_i^{l-1}, j = k \\ 0, j \neq k \end{cases} \\ \frac{\partial o}{\partial c} &= \frac{\partial (\mathbf{h}^L \mathbf{V} + c)}{\partial c} = 1 & \frac{\partial \mathbf{a}_k^l}{\partial \mathbf{b}_j^l} &= \frac{\partial (\mathbf{h}^{l-1} \mathbf{W}^l + \mathbf{b}^l)_k}{\partial \mathbf{b}_j^l} = \begin{cases} 1, j = k \\ 0, j \neq k \end{cases} \end{aligned} \tag{2.14}$$

Le principe reste le même pour une architecture différente: évaluer l'état du réseau en $\mathbf{x}$, décomposer les dérivées partielles au maximum grâce la dérivation en chaîne et finalement calculer une seule fois toute les dérivées partielles nécessaires [14, p. 241].

2.3.2 Détails et ajouts à la descente de gradient

Un aspect important de la descente de gradient appliquée aux réseaux de neurones est l'initialisation des poids. Si tous les poids sont initialisés à zéro, toutes les unités calculeront la même chose et presqu'aucun apprentissage sera possible. À l'opposé, des poids trop grands combinés aux fonctions d'activations habituelles sigmoïdales (sigmoïde et tanh) saturerons les unités cachées, ce qui fera tendre le gradient vers zéro et ralentira l'entraînement. L'idéal pour l'optimisation d'un réseau de neurones sigmoïdal est que l'activation initiale des unités cachées soit située dans le régime linéaire de la fonction d'activation (autour de zéro), puis que celles-ci opèrent dans le régime non-linéaire, sans saturer au fil de l'entraînement. [39] recommande une initialisation uniforme des poids dans l'interval $\mathbf{W}_{i,j}^l \sim \mathrm{U}\left[-\sqrt{\frac{6}{n_h^l + n_h^{l-1}}}, \sqrt{\frac{6}{n_h^l + n_h^{l-1}}}\right]$ pour tanh et le même interval multiplié par 4 pour la sigmoïde. Les biais d'un réseau sont habituellement initialisés à zéro, quoique initialiser les bias des unités cachées à une valeur négative est une méthode (légèrement vulgaire) pour encourager l'éparsité .

Un autre point important lors de la descente de gradient stochastique est l'ordre des exemples présentés. Les exemples doivent être présentés au réseaux dans un ordre aléatoire sans quoi les paramètres peuvent se retrouver dans une zone loin d'une bonne solution. Prenons par exemple un problème de classification, où un grand nombre d'exemples d'une seule classe sont initialement présentés au réseau. En optimisant sur les exemples d'une seule classe, la sortie du réseau sera biaisée vers cette classe et la suite de l'entraînement devra combattre pour défaire ce biais, ralentissant ainsi la convergence et plaçant les paramètres loin d'un bon minimum local [55].

La descente de gradient stochastique "pure" évalue le gradient pour un seul point entraînement à la fois. Une astuce commune pour accélérer l'entraînement est de moyenner le gradient sur un petit échantillon d'exemples aléatoires, afin de prendre avantage des algorithmes de multiplication

matricielle tout en conservant les avantages du gradient stochastique. Le résultat est une estimation du gradient avec une variance réduite (moins bruitée) par rapport à la descente stochastique pure, et un coût de calcul par exemple réduit. On nomme cette méthode descente par "minibatch". La taille de la minibatch devient un nouvel hyperparamètre, et contrôle la variance de l'estimation du gradient ainsi que vitesse de convergence de l'entraînement.

Le taux d'apprentissage détermine, en conjonction avec le gradient lui-même, la norme des pas de gradients. Cependant, avec un taux d'apprentissage constant ϵ, les derniers exemples ayant servis à optimiser les paramètres auront une influence disproportionnée sur la solution par rapport aux exemples vus précédemment. De plus, une fois proche d'un minimum local, il est possible que les paramètres se mettent à osciller autour de celui-ci, sans jamais l'atteindre. Imaginons un long creux étroit dans la surface d'erreur; les paramètres se dirigent vers le fond via l'information donnée par le gradient. Un grand taux d'apprentissage sera utile pour couvrir plus de distance en se dirigeant vers le fond, mais une fois proche du minimum, de trop grands pas ne feront qu'osciller autour du fond. Il est difficile, dans ces conditions, de naviguer la surface d'erreur avec précision avec un taux d'apprentissage constant. Une option pour contrer ce phénomène est de réduire le taux d'apprentissage en fonction du nombre de mise à jour ayant été fait. En posant t comme le nombre de mises à jour effectuées, il existe une panoplie d'heuristiques pour réduire le taux d'apprentissage en fonction de t. Voici quelques exemples:

$$\epsilon_t = \frac{\epsilon_0 \tau}{\tau + t}$$

$$\epsilon_t = \begin{cases} \epsilon & t \leq T \\ \frac{\epsilon}{(T-t)} & t > T \end{cases}$$

$$\epsilon_t = \epsilon \alpha^t$$

Des hyperparamètres additionels contrôlant la vitesse d'amortissement sont ajoutés.

Calculer les dérivées partielles de deuxième ordre dans un réseau de neurones moindrement grand est pratiquement impossible, dû à la taille de la matrice hessienne [1], quadratique en nombre de paramètres libres. Le résultat est soit une demande de mémoire trop grande ou un temps de calcul trop élevé pour justifier les gains apportés par des méthodes d'optimisation de second ordre. Autre que les méthodes quasi-Newtonienne, il existe un moyen simple d'obtenir de l'information sur la courbure de la surface d'erreur. Au lieu d'utiliser le gradient au temps t pour effectuer la mise à jour des paramètres, on utilise une moyenne glissante exponentielle des gradients à travers le temps. La

[1] Matrice des dérivées partielles de second ordre

notion de temps correspond au nombre de mises à jour effectuées. La règle de mise à jour devient:

$$m_t = \alpha\vec{\nabla}\theta_t + (1-\alpha)m_{t-1}$$
$$\theta_t = \theta_{t-1} + \epsilon m_t$$

Pour une surface d'erreur irrégulière, certains termes du gradient auront tendance à changer de signe fréquemment. D'autres, en moyenne, seront plus stables à travers le temps: ceux-ci définissent les directions plus importantes pour l'optimisation des paramètres. La technique de "momentum" mitige l'effet d'oscillation du gradient causée par une surface d'erreur irrégulière, et allonge les pas de mise à jour dans les directions cruciales. Ce lissage des mise à jours permet en général de converger plus rapidement en ignorant les directions insignifiantes, et d'obtenir de meilleurs résultats puisque les paramètres auront moins tendance à être attirés vers des petits minima locaux [6].

2.4 Régularisation

Accélérer l'optimisation de la fonction de coût est important, mais pratiquement inutile si le réseau ne généralise pas correctement. C'est pourquoi il existe une varitété de méthodes pour prévenir le sur-entraînement. Une façon simple de régulariser un réseau de neurones est de restreindre la taille de ses couches cachées. Avec un réseau suffisament petit, dépendant du nombre d'exemples d'entraînements, il est presque impossible pour le réseau de sur-entraîner puisque sa capacité est insuffisante pour représenter une fonction qui assigne aux exemples leur cibles correspondantes. Le réseau doit donc partager sa capacité parmi tout les exemples d'entraînements. Il y a intérêt à explorer les réseaux à plus haute capacité puisque les petits réseaux sont en général moins performants. Cette section énumère quelques techniques de régularisation populaires utilisées avec succès.

2.4.1 Terme de régularisation

Une méthode commune de régularisation est d'ajouter à $\mathcal{L}$ un terme qui impose une contrainte sur les paramètres du réseau. Pour un réseau à haute capacité, il est généralement indésirable que certains poids deviennent trop grand par rapport aux autres, ce qui peut arriver lorsque le réseau mémorise un exemple ou place trop d'importance sur une entrée particulière. Des poids trop grands auront aussi tendance à saturer les unités cachées plus facilement, ce qui nuit à l'entraînement [55]. Ce phénomène peut être amorti en imposant une contrainte sur la taille des poids du réseau, en ajoutant un terme de régularisation qui croît en fonction de la taille des poids. Les choix les

plus populaires sont la sommes des poids au carré et la somme des valeurs absolues , nommés respectivement régularisation L_1 et L_2 des poids [6]. Ceci correspond mathématiquement, pour les poids $\mathbf{W}$ d'une couche du réseau, à:

$$L_1(\mathbf{W}) = \sum_{i,j} |\mathbf{W}_{i,j}|$$
$$L_2(\mathbf{W}) = \sum_{i,j} (\mathbf{W}_{i,j})^2$$

Afin de contrôler l'intensité de la régularisation, on utilise un facteur multiplicatif appliqué au terme de régularisation des poids. Celui-ci devient un nouvel hyperparamètre du modèle. On peut assigner aux poids de chaque couche un terme différent.

Comme mentionné en 2.2.2, une représentation éparse peut servir à démêler les facteurs de variations des variables d'entrées. Une méthode pour encourager l'éparsité de la représentation apprise est d'ajouter un terme de régularisation qui croît en fonction de l'activation des unités cachées. Cette méthode fonctionne particulièrement bien avec des rectifieurs et un terme $L_1(\mathbf{h}) = \sum_i |\mathbf{h}_i|$ ajouté à la fonction de coût [40].

2.4.2 Pré-entraînement

Sujet qui mérite son propre ouvrage, l'apprentissage non-supervisé s'est avéré d'une grande utilité pour l'apprentissage de réseaux profonds. Il est difficile d'entraîner des réseaux profonds sigmoidaux par descente de gradient avec des poids initialisés aléatoirement [39]. En initialisant les poids d'un réseau à l'aide d'autoencodeurs ou de variantes des machines de boltzmann, il a été possible d'entraîner avec succès des réseaux de neurones profonds, surpassant la performance de réseaux à une couche [8, 47]. Plusieurs hypothèses entourent la raison de l'efficacité du pré-entraînement. Il est tout de même possible de l'interpréter comme une forme de régularisation, puisque des réseaux profonds avec pré-entraînement obtiennent une performance d'entraînement inférieure mais une meilleure généralisation [35]. Un exemple de cette procédure suit:

1. Entraîner un autoencodeur sur les données d'entraînement.
2. Entraîner un autre autoencodeur en utilisant comme données l'encodage des entrées fournis par l'autoencodeur précédent.
3. Répéter l'étape 2 pour le nombre de couches désirées, chaque autoencodeur étant entraîné sur la transformation de l'autoencodeur précédent.

4. Initialiser un réseau de neurones avec les poids et les biais ainsi appris. Le premier autoencodeur devient la première couche et ainsi de suite.

5. Entraîner le réseau de façon supervisée.

Optionellement, nous pouvons entraîner l'autoencodeur profond résultant de l'empilage des autoencodeurs entraînés avant l'étape 3. Les hyperparamètres des autoencodeurs pour chaque couche sont sélectionnés par validation supervisée, c'est-à-dire la performance du réseau une fois optimisé de façon supervisée. Les autoencodeurs débruitants et contractants, ainsi que les machines de boltzmann restreintes sont de bons modèles pour le pré-entraînement. Par cette initialisation, nous essayons de trouver une configuration des paramètres à partir de laquelle une bonne solution (minimum local) peut être obtenue avec la descente de gradient [35].

2.4.3 Dropout

Récemment, une technique introduite par [52] a été utilisée avec grand succès dans des réseaux profonds supervisés, améliorant significativement leurs performances. Le principe est similaire aux auto-encodeurs débruitants, i.e. ajouter du bruit lors de l'entraînement. On corrompt aussi cependant les unités cachées avec du bruit binomial: un sous-ensemble aléatoire d'unités cachées sont ignorées pour chaque exemple vu lors de l'optimisation. Ceci a pour effet d'empêcher les unités cachées à fonctionner en groupe, et devenir plus indépendantes. Ce faisant, il est plus difficile pour le réseau de mémoriser des exemples d'entraînement, et il doit décomposer l'entrée en caractéristiques utiles indépendemment des unes des autres. Voici les calculs durant l'entraînement du réseau, modifiant légèrement 2.7:

$$\begin{aligned}
\mathbf{m}^{(0)} &\sim \text{BINOMIAL}(1, p_x)_{1\times d} \\
\tilde{\mathbf{h}}^{(0)} &= \mathbf{x} \times \mathbf{m}^{(0)} \\
\mathbf{h}^{(l)} &= f_{act}(\tilde{\mathbf{h}}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}),\ l = 1, ..., L \\
\mathbf{m}^{(l)} &\sim \text{BINOMIAL}(1, p_h)_{1\times n_h^l} \\
\tilde{\mathbf{h}}^{(l)} &= \mathbf{h}^{(l)} \times \mathbf{m} \\
f_\theta(\mathbf{x}) &= g(\tilde{\mathbf{h}}^{L}\mathbf{V} + \mathbf{c})
\end{aligned} \tag{2.15}$$

L'application de ce bruit possède une interprétation intéressante. Moyenner les prédictions d'une multitude de modèles de performance similaire produit généralement des résultats supérieurs [18]. En masquant un sous-ensemble d'unités cachées, on obtient un réseau de neurones alternatif, composé des unités restantes du réseau original. Chaque exemple se voit donc assigné un sous-réseau

presque toujours différent, le nombre de sous-réseaux possibles étant énorme avec un minimum d'unité cachées. Nous entraînons ainsi implicitement un grand nombre de réseaux différents, partageant tous partiellement les mêmes poids. De plus, chaque réseau voit un ensemble différent de variables d'entrées, comme dans les forêts aléatoires. Pour moyenner ces réseaux lors d'une prédiction, le bruit est omis et nous multiplions les entrées par p_x et les unités cachées par p_h, puisque c'est la proportion d'entrées et d'unités utilisées lors de l'entraînement. Le résultat est une approximation de la sortie moyenne de tous les réseaux. Cette méthode fonctionne particulièrement bien avec les réseaux de rectifieurs profonds, typiquement avec $p_x = 0.8$ et $p_h = 0.5$. Pour plus d'information, voir [48].

2.5 Formattage des entrées

2.5.1 Encodage

Il est important, pour assurer un bon apprentissage, que les entrées soit formattées et traitées adéquatement pour une optimisation numérique. Tout d'abord, on distingue entre les variables quantitatives et qualitatives. Les variables quantitatives sont généralement normalisées pour plusieurs raisons, couvertes dans la prochaine sous-section. Une variable catégorique non ordonnée n'a presque qu'aucune signification numérique dans le cadre de l'optimisation d'un réseau de neurones; il faut la transformer de façon à ce qu'elle en aie une. Pour une variable désignant l'appartenance à une de K classes, il est préférable de l'encoder en format "one-hot", c'est-à-dire un vecteur $\mathbf{v} = [v_1, ..., v_K], v_k = 1, v_{i \neq k} = 0$. Pour une variable catégorique ordonnée, un encodage dit "thermomètre" correspondant à un vecteur $\mathbf{v} = [v_1, ..., v_K], v_{i \leq k} = 1, v_{i > k = 0}$ peut être utile. Une variable binaire est typiquement laissée intacte. Il existe beaucoup de façons d'encoder des variables, et trouver celles qui fonctionnent bien fait parti des préliminaires nécessaires à l'utilisation de réseaux de neurones, et des modèles d'apprentissage en général.

2.5.2 Normalisation

Il est préférable de normaliser les variables d'entrée pour améliorer le conditionnement du problème d'optimisation. Si certaines entrées sont systématiquement trop grandes par rapport aux autres, elles auront une participation démesurée au gradient de l'erreur, ce qui empêchera le réseau d'utiliser les autres variables. De plus, ces variables auront tendance à saturer les unités cachées au début de l'entraînement, ce qui le ralentit [55].

La standardisation est une méthode commune pour normaliser les entrées. Il suffit de soustraire la moyenne et diviser par l'écart-type chaque variable d'entrée, les deux obtenus empiriquement via l'ensemble d'entraînement. Ceci à pour d'effet de centrer les entrées et imposer un écart-type unitaire, et améliore généralement la convergence du réseau. Cette méthode fonctionne bien pour des entrées gaussiennes, mais peut être inapropriée si la variable comporte beaucoup de cas extrêmes.

Une autre méthode, non-linéaire, pour transformer les entrées est l'uniformisation. Elle consiste à appliquer la fonction de répartition empirique. La fonction est calculée d'abord en ordonnant les variables disponibles, puis en divisant la position du classement d'une variable dans cet ordre par le nombre d'éléments dans le classement. Le résultat est une estimation de la fonction de répartition de la distribution de cette variable, par conséquent borné entre 0 et 1 [6].

2.6 Sélection des hyperparamètres

Entraîner un modèle implique généralement choisir la famille de modèle puis choisir les hyperparamètres correspondant. Les hyperparamètres ont une influence critique sur le modèle final. C'est eux, en conjonction avec la quantité et la qualité des données, qui déterminent la performance d'un modèle et sa capacité à généraliser. Choisir les hyperparamètres correctement est une étape cruciale, c'est pourquoi il existe différentes techniques pour l'exploration et la validation du choix d'hyperparamètres.

2.6.1 Méthodes de test

La méthode de base pour évaluer et sélectionner un modèle est de diviser les données aléatoirement en trois ensembles: entraînement, validation et test. L'ensemble d'entraînement sert à l'optimisation du modèle et l'ensemble de test à estimer l'erreur de généralisation. Cependant, si nous sélectionnons le modèle en se basant sur l'erreur de test, l'erreur de généralisation estimée est biaisée de façon optimiste. En utilisant l'erreur de test dans le processus de sélection, nous utilisons de l'information qui ne serait pas normalement disponible si la performance du modèle était évaluée sur des exemples qui n'étaient pas disponibles lors de l'entraînement. C'est pourquoi nous utilisons un ensemble de validation, qui lui sert à la sélection et l'optimisation des hyperparamètres.

Une généralisation de la méthode précédente est la validation croisée à k plis. Après avoir choisi aléatoirement un ensemble de test, les données restantes sont divisées aléatoirement en k sous-ensembles de taille égale nommés plis. Pour les différentes configurations d'hyperparamètres,

chaque modèle est entraîné sur $k-1$ plis et validé sur le pli restant. Ceci est répété pour les k configurations possibles de $k-1$ parmi k plis d'entraînement. L'erreur de validation pour chaque modèle est obtenue en moyennant les k validations ainsi calculées, réduisant ainsi la variance de l'erreur de validation. Nous choisissons finalement le modèle avec la meilleur erreur de validation. Dans le cas extrême, k est égal au nombre d'exemples disponibles (nommé "leave-one-out validation").

Si les données ont un aspect séquentiel ou temporel, la validation séquentielle sert à estimer l'évolution de l'erreur en fonction du temps. Le principe est similaire à la validation croisée à k mais les données conservent leur ordonnement séquentiel et sont divisés en groupes chronologiques. Les premiers exemples sont gardés comme ensemble d'entraînement de base. Ensuite nous entraînons, validons et testons sur la concaténation successive des plis, en répétant cette opération pour chaque plis ajouté. On moyenne finalement l'erreur de test pour les modèles sélectionnés par validation sur chaque pli. Ceci nous donne une estimation de l'erreur de généralisation à travers le temps.

2.6.2 Méthodes de recherche

Une solution naïve pour explorer les hyperparamètres est de construire une grille de points, chaque dimension correspondant à un hyperparamètre et chaque point étant une combinaison de valeurs à tester. Nous entraînons ensuite un modèle pour chaque configuration ainsi définie. Cette méthode est inefficace à cause du fléau de la dimensionalité : le nombre de points dans la grille croît exponentiellement en fonction du nombre d'hyperparamètres. De plus, la valeur optimale d'un hyperparamètre peut résider entre les valeurs de l'espacement régulier défini par la grille. Avec 5 hyperparamètres espacés sur 10 valeurs, nous devons entraîner 10^5 modèles, ce qui est généralement infaisable. Ces principes sont analogues au concept d'intégration numérique par quadrature. Une méthode beaucoup plus efficace est l'échantillonage aléatoire, qui ressemble plus à l'intégration numérique Monte-Carlo. En tirant chaque hyperparamètre aléatoirement et indépendemment, nous pouvons mieux couvrir marginalement l'étendue des valeurs possible pour chaque hyperparamètre. Ceci nous permet d'observer plus clairement l'effet de chaque hyperparamètre. La procédure habituelle implique de sélectionner une plage raisonnable d'échantillonage pour chaque hyperparamètre, entraîner autant de modèles possible en tirant des configurations aléatoires, puis répéter en réajustant la plage d'échantillonage d'après le résultat des expériences. Plusieurs méthodes automatiques ont été développées récemment, dans le but d'automatiser la recherche d'hyperparamètre [13, 80].

Chapitre 3

Introduction au premier article

Stacked Calibration of Off-Policy Policy Evaluation for Video Game Matchmaking
Eric Laufer, Raul Chandias Ferrari, Li Yao, Olivier Delalleau et Yoshua Bengio
Accepté pour *IEEE Conference on Computational Intelligence and Games (CIG)*, 2013

3.1 Contexte

Ubisoft, une des grandes compagnies de jeux vidéos, participe présentement à une chaire de recherche avec l'Université de Montréal. C'est au sein de cette collaboration que ma maîtrise eut lieu, pendant laquelle j'ai principalement travaillé sur les projets de la chaire, plus spécifiquement ceux reliés à la recommendation automatique. Ceci inclut le filtrage collaboratif, la formation d'équipes en ligne et la recommendation de contenu. Les premières suggestions de projets lors de ma maîtrise furent axés sur le filtrage collaboratif, c'est-à-dire l'estimation de l'appréciation d'items par les utilisateurs basé sur l'appréciation des autres utilisateurs.

Ghost Recon Online est un jeu de tir à la troisième personne où des équipes de joueurs s'affrontent en ligne. La performance d'une équipe dans ce genre de jeu est généralement basée sur la coopération des joueurs au sein d'une équipe et les joueurs dont le niveau de jeu est significativement supérieur ou inférieur aux autres joueurs. L'application typique d'apprentissage machine pour ce genre de jeu est le matchmaking: assembler les joueurs pour former des équipes balancées, avec une probabilité égale de gagner. Les méthodes existantes n'étant basées que sur l'estimation du niveau du joueur, notre premier but fut de démontrer que l'utilisation des statistiques des joueurs pouvait améliorer la performance du modèle et des heuristiques en place dans le jeu. Après quelques mois, l'article

[31] fut publié, décrivant le modèle confectionné sur mesure pour cette tâche, et démontrant des performances supérieures à TrueSkill, modèle standard de matchmaking.

Vérifier si le nouveau système de matchmaking augmentait l'appréciation des joueurs fut le travail futur principal mentionné par [31]. À cette fin, les joueurs pouvaient remplir des questionnaires après les matchs pour indiquer leur appréciation. Le modèle fut modifié pour prédire l'appréciation individuelle des joueurs, afin de grouper ceux-ci pour maximiser l'amusement moyen. Un simulateur fut aussi codé pour vérifier les propriétés du système de matchmaking et comparer la performance des différentes méthodes. Cependant, après plusieurs mois d'expériences et de tests, le matchmaking basé sur l'appréciation des joueurs ne donna que des résultats marginalement meilleurs que les approches basées sur une probabilité égale de gagner.

3.2 Contributions et commentaires

Malgré cet échec apparent, nous avions en main un simulateur qui nous permettait de vérifier les propriétés des algorithmes de matchmaking. Bien que nous puissions vérifier l'erreur de classification des algorithmes prédisant l'équipe gagnante, ceci nous dit peu sur la balance des matchs qui seraient recommandés par un tel modèle. Nous avons donc développé une approche pour évaluer l'utilisation d'un algorithme de matchmaking en tant que politique de décision, basée sur l'utilisation du simulateur. Nous avons aussi profité de l'occasion pour comparer différentes sortes algorithmes d'apprentissage appliqués à cette tâche. L'article qui suit au prochain chapitre fait la comparaison de ces algorithmes en terme de performance "classique" et fait l'étude de l'utilisation du simulateur pour les évaluer en tant que politiques de décision. L'article démontre d'abord la performance supérieure d'un réseau de neurones profond avec une fonction d'activation particulière introduite par [41], puis explique une technique de régularisation pour l'utilisation du simulateur lorsque celui-ci utilise la même fonction comme politique de décision et comme estimateur de la valeur de cette politique. L'article fut accepté par *Computational Intelligence in Games*, conférence axée sur l'intelligence artificielle dans les jeux. Ma contribution personelle fut l'écriture d'une majorité de l'article, l'implémentation du cadre expérimental pour les simulations et l'obtention des résultats pour la deuxième partie.

Chapitre 4

Stacked Calibration of Off-Policy Policy Evaluation for Video Game Matchmaking

4.1 Abstract

We consider an industrial strength application of recommendation systems for video-game matchmaking in which off-policy policy evaluation is important but where standard approaches can hardly be applied. The objective of the policy is to sequentially form teams of players from those waiting to be matched, in such a way as to produce well-balanced matches. Unfortunately, the available training data comes from a policy that is not known perfectly and that is not stochastic, making it impossible to use methods based on importance weights. Furthermore, we observe that when the estimated reward function and the policy are obtained by training from the same off-policy dataset, the policy evaluation using the estimated reward function is biased. We present a simple calibration procedure that is similar to stacked regression and that removes most of the bias, in the experiments we performed. Data collected during beta tests of *Ghost Recon Online*, a first person shooter from Ubisoft, were used for the experiments.

4.2 Introduction

Video games have been a fertile area of research for machine learning and artificial intelligence in the past decade. Applications range from automated content generation [81], behavior modelling [76] to game playing [37]. Of particular interest here are ranking and matchmaking algorithms, stemming from chess rating systems, and motivated by studies reporting that an adequate challenge is an important component of player satisfaction and enjoyment [28, 75]. Starting with Harkness [44] and Elo [34], these systems were designed to rank professional chess players according to their history of tournament performance, modelling pairwise comparisons between players, inspired by Bradley and Terry [17]. Further research on the subject led to modelling uncertainty about the player skills, first introduced by Glickman [38], and spawning a variety of Bayesian rating systems, the most famous being Microsoft's own TrueSkill [45]. Going further, Delalleau *et al.* [31] and Riegelsberger *et al.* [64] presented feature-based approaches, computing player features based on their behavior (and not only the win/lose results of matches) and using them to model inter-player interactions.

Most of these systems are focused on estimating player skill and predicting game outcomes accurately. Here we will address the problem at a slightly higher level, defining matchmaking as the process of assembling and pairing available users into groups in order to maximize some measure of quality should they play together. The design of such a system is a necessary step for the development of an online multiplayer game, which has become a popular and lucrative paradigm for video games. Since changes in the matchmaking system can potentially impact user experience negatively, careful attention must be given to the evaluation and comparison of matchmaking algorithms. Faced with this policy evaluation challenge in the context of an industrial collaboration, we sought to evaluate methods using reinforcement learning [62] and contextual bandits [54] techniques. However, due to practical issues, they could not be applied on the real task: we propose here an evaluation framework for matchmaking policies, which we show to perform as well as these techniques, would it have been possible to use them in practice.

In the next section we will review and formalize the problem of online matchmaking for video games, describing popular algorithms and presenting an alternative approach to learning to predict game outcomes based on Deep Learning [12, 41]. Section 4.4 will address offline evaluation of these systems through reinforcement learning methods called off-policy policy evaluation procedures [62], and explain why it is hard to apply them to our task. Section 4.5 will present the matchmaking simulator used in our experiments in order to evaluate the different off-policy policy evaluation methods. Section 4.6 discusses a specific bias induced in direct methods to evaluate a policy and proposes a method to alleviate this bias, called *Stacked Calibration*. Section 4.7 presents

results obtained with a methodology that we propose to validate off-policy policy evaluation methods, based on a *two-stage simulation*, i.e., with a model trained on real data being used to generate data and evaluate ground truth inside the simulator.

4.3 Video Game Matchmaking by Deep Learning

4.3.1 Matchmaking in Online Video Games

As an online game grows in popularity, so will the number of players and the amount of data collected concerning their performance. With commercially successful games sometimes involving millions of players online, it has become important for game developers to implement good matchmaking to provide an adequate level of challenge, and to create leagues and leaderboards, which is an important component of player satisfaction [81]. Many of these methods are based on the Bradley-Terry model for pairwise comparison [50, 59]. Given a pool of active players, the goal of these approaches is to estimate a value for each player that conceptually represents the player's level or skill. These skill factors are estimated by modelling the probability of game outcomes between competing players as a function of their skill, and fitting this model through the historical data of wins and losses.

The chess rating system ELO does so by modelling player skills as a Gaussian or logistic distribution from which their game performance is sampled. Performance is measured by a single random scalar, and skill is its expected value. By computing the probability that one player's performance exceeds that of another, ELO provides an estimated score, or outcome of a match. When new matches are played out, the true and estimated scores are used to update the skill estimate. Several drawbacks of the model such as tracking the uncertainty about players' skills led to the creation of other, more sophisticated, systems which dealt with this uncertainty. Perhaps the most famous recent model proposed is Microsoft's TrueSkill Herbrich *et al.* [45], described as a Bayesian skill rating system, that models belief about players' skill, that explicitly models draws and that can also be used with multiple teams of players. The full description of these models is outside of the scope of this paper, and can be retrieved from Elo [34] and Herbrich *et al.* [45].

Although chess ratings have initially mostly been used to rank professional players, many video games make extensive use of rating systems for matching players together in online play. Under the assumption that a balanced match has an increased probability of being a draw, the match quality criterion defined by Herbrich *et al.* [45] is, in essence, the probability of such an outcome. The goal is thus to launch matches for which the system believes the probability of draw is high. Other

methods have been proposed by Jimenez-Rodriguez *et al.* [51], suggesting approaches like pairing players so as to minimize the difference in skills, discretizing the skill values and matching players who fall into the same skill bin. There has been some interest in matching players to optimize criteria different from draw probability or similarity of skill values, some based on reducing undesired social behavior [64] or matching players to fulfill certain roles through player modelling [51]. In most video games, we observe other outcomes besides the winning team (such as game-defined scores, which can be used in TrueSkill to rank individual players): such outcomes may provide a more objective view of balance. By using a real-valued target based on these outcomes, instead of a binary value (the identity of the winning team), we can obtain more information about the actual balance of each match.

4.3.2 Deep Learning Approach to Matchmaking

4.3.2.1 Evaluating the Balance of a Match

We have collected data from *Ghost Recon Online*, an online team-based first person shooting (FPS) game developed by Ubisoft. In this game, players form two teams to compete with each other to achieve a certain goal. The enjoyment of players is affected not only by the design of the game but also by the way the teams are formed. It is important to form teams such that the overall game is balanced — the two teams should have more or less an equal chance to win, or in other words, the combined performance of players from the two teams should be similar. TrueSkill and many other skill-based systems estimate each player's skill by only taking into account the binary result of previous games, and each team's skill by summing the skills of the team members. However, it is natural to assume that the match balance in FPS games strongly depends on the *interaction between players* — how they cooperate as a team. We take advantage of data that could be used to take these interactions and complementarities into account. Furthermore, instead of relying only on the binary win/lose outcome of each game, we use a more fine-grained measurement that is appropriate for FPS games: the relative number of kills made by each team. We apply machine learning methods to estimate this estimator of balance by taking into account the players' performance and their interactions during games played in the past. Note however that the following procedures and experiments can be easily adapted to any other measurement of balance.

Following the approach described in [31], we use the following three steps to form a feature vector summarizing both players and their team performance in a match. First, various statistics of the players' performance were collected during gameplays, such as the number of kills/deaths, wins/losses, firing accuracy, time under cover and other relevant game-dependent measures. These

normalized statistics are used to form player profiles. Secondly, for each of the two teams in a match, player profiles are then aggregated to form what we call a team profile by concatenating the sum, mean, variance, minimum and maximum across each individual component of the player profile in that team. Finally, two team profiles are concatenated together to form the final feature vector of a match, denoted by $\mathbf{x}$. We define the measure of balance as

$$y = 1 - \left| log\left(\frac{kills_A}{kills_B}\right)\right|$$

where $kills_A$ (resp. $kills_B$) denotes the total number of kills from players in team A (resp. team B). The logarithm is used to reduce the effect of outliers, and the absolute value makes this quantity symmetric with respect to both teams. It is subtracted from 1 simply so that balance is perfect when equal to one, and to make finding the most balanced match a reward maximization problem. Balance evaluation models $f(\mathbf{x})$ are trained to minimize the mean square error cost between the model's prediction $f(\mathbf{x})$ and the observed target y

$$E_{(\mathbf{x},y)\sim\mathcal{D}}[(y - f(\mathbf{x}))^2] \tag{4.1}$$

where $\mathbf{x}$ and y are drawn from the unknown distribution $\mathcal{D}$. In practice, the generalization error defined in Eq. 4.1 is inaccessible due to the unknown $\mathcal{D}$. So instead, given a collection of N historical matches $(\mathbf{x}_i, y_i)$ where $i = 1 \dots N$, the following empirical cost (mean squared error) is minimized instead:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i))^2. \tag{4.2}$$

4.3.2.2 Deep Learning and Maxout

The learning algorithm we have found to work best on this task is a deep neural network. Deep Learning [5] has become a field in itself in the machine learning community, with regular workshops at the leading conferences such as NIPS and ICML, and a new conference dedicated to it, ICLR[1], sometimes under the header of *Representation Learning* or *Feature Learning*. The rapid increase in scientific activity on representation learning has been accompanied and nourished by a remarkable string of empirical successes both in academia and in industry, e.g., in speech recognition [29, 30, 46, 60, 72], music information retrieval [16, 43], object recognition [24, 52, 66] and natural language processing [4, 27]. See Bengio *et al.* [12] for a recent review.

[1] International Conference on Learning Representations

Our model is very similar to Maxout networks [41], recently introduced to train deep neural networks in the context of object recognition, but is adjusted to the regression task at hand and applied to matchmaking. It is compared with a standard Multilayer Artificial Neural Network (MLP) that has one or more hidden layers (with hyperbolic tangent non-linearity), each fully connected to the layer below.

Maxout networks use a different type of hidden layer where the hidden units are formed by taking the maximum of k filter outputs (each filter outputs the dot product of a filter weight vector with the input vector into the layer). Since this is a regression problem, the output layer is an affine transformation with a single linear output unit:

$$f(\mathbf{x}) = w'\phi(\mathbf{x}) + b \tag{4.3}$$

where $\phi(\mathbf{x})$ is the output of the top hidden layer which nonlinearly depends on the neural net input $\mathbf{x}$, w the output weight vector and b is a bias term. Our model can discover and exploit a strong nonlinear dependency between $\mathbf{x}$ and target y.

The training of Maxout networks uses the standard backpropagation equipped with a recently proposed trick called *Dropout* [48]. The basic idea of Dropout is that hidden units outputs are randomly masked (set to 0) with probability $1/2$ during training (and multiplied by $1/2$ at test time). As argued by Goodfellow *et al.* [41], it is possible to train a deep Maxout network without overfitting due to the model averaging effect brought by Dropout. Meanwhile, by making sure that exactly one filter is being used and receiving gradient for each non-masked hidden unit, it appears that optimizing Maxout networks (even deeper ones) can be done more easily than with rectifier, hyperbolic tangent or sigmoid activation functions. Although Maxout activation functions can easily lead to overfitting, combining them with Dropout provides a form of symmetry breaking and bagging regularization that generally provides substantially better performance than standard MLPs.

To compare, we also consider other standard machine learning models for the regression task. The simplest model is the Elastic Net [82]. It assumes that the interaction between the inputs and outputs are linear. This assumption is usually too strong to hold in practice. To relax this strong assumption, we consider regression trees [20], which assume that the interaction is piece-wise linear, thus overall nonlinear. This assumption works well when the input space can be separated by cells inside which the outputs are similar. However, it suffers when the true interaction is a highly variable but locally smooth function in a high dimensional space. In order to improve over an individual tree, many types of ensemble models have been tried, such as Random Forests [19] and Gradient Boosted Trees [36]. The tree-based ensemble models work well in that they use the composition of multiple

weaker learners to reduce the variance of predictions. All these methods are publicly available in scikit-learn[2].

4.3.3 Comparative Results

Our dataset consists of 436323 matches played by 159142 players. For each match, we compute a feature vector of dimension 620, the first half of which represents team A's attributes and the second half team B's attributes, as discussed in section 4.3.2. The entire dataset is chronologically split in three sets: 70% as the training set, 15% as the validation set and 15% as the test set. We train the standard MLP and Maxout network on the training set and perform early stopping by checking the cost in Eq. 4.2 on the validation set. When there is no significant drop (0.0001) for three consecutive training passes through the training set, training is stopped. For Elastic Net, training optimization is stopped when the change in cost does not exceed 0.0001. For tree-based models, training is stopped when the maximal tree depth is reached. The best model is selected based on the validation cost. We report the average cost on the test set as a measurement of models' performance in generalization.

In order to compare with different types of models, we have performed intensive hyper-parameter search for each type. We summarize the most important hyper-parameters of each model family and give the best values found. For the Elastic Net, α controls the strength of both L1 and L2 regularizations and β balances the relative importance between them. We have found that $\alpha = 1.16 \times 10^{-6}$ and $\beta = 1.42 \times 10^{-6}$ work best. For the Random Forest and Gradient Boosted Trees, the most important hyper-parameters are the number of base learners and their depth. The best model we have found for the Random Forest uses 146 base learners each of which has a depth of 16. The best Gradient Boosted Tree has 388 base learners each of which has a depth of 12. For standard MLPs and Maxout networks, there are typically many more hyper-parameters that are important such as the learning rates, the number of hidden layers and number of units per hidden layer. The best Maxout network we have found is with $k = 2$ filters per units and 2 hidden layers with 1929 hidden units for the first and 621 units for the second. The best standard MLP we found had three hidden layers with 742, 911, and 964 hidden units respectively.

Figure 4.1 shows the mean squared error (MSE) of each model as a measurement of their performance. As a baseline, the constant predictor using the average of y in the training set performs the worst, with the highest MSE. MLP performs the same as Random Forest. Maxout performs the best and has the lowest MSE. Table 4.1 shows the resulting p-values of the paired t-tests [3]. It shows the significance of the differences between all models' MSE. The standard $p = 0.05$ is adopted

[2] http://scikit-learn.org

[3] See docs.scipy.org/doc/scipy/reference/stats.html for description

to indicate the statistical significance. The bold font indicates that the difference of MSE between two models is statistically significant. In fact, all MSEs are significantly different from each other, except between Random Forest and MLP.

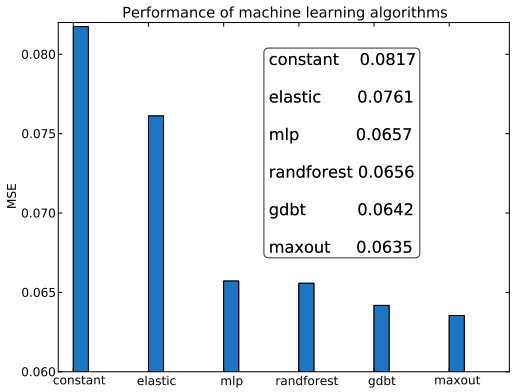

FIGURE 4.1: Performance of different types of models in predicting the balance of a match (based on the kill ratio). *constant* for the mean predictor as a baseline, *elastic* for Elastic Net, *MLP* for a standard MLP, *randforest* for Random Forest, *gdbt* for Gradient Boosted Trees, *Maxout* for Maxout networks. Maxout outperforms all the other models. See also Table 4.1 for the significance tests.

TABLE 4.1: p-values of paired t-test on MSE between models. These show if the difference in performance between any pair of models is statistically significant (in bold), i.e. p-value < 0.05, or not. All differences are significant except between RandForest and MLP, which are statistically indistinguishable. Maxout is found superior to each of the other models in a statistically significant way. See Fig. 4.1 for the labels of each of the methods used in the table below.

	MAXOUT	GDBT	RANDFOREST	MLP	ELASTIC
MAXOUT		**0.001**	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$
GDBT	**0.001**		$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$
RANDFOREST	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$		0.532	$\ll \mathbf{10^{-3}}$
MLP	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	0.532		$\ll \mathbf{10^{-3}}$
ELASTIC	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	$\ll \mathbf{10^{-3}}$	

4.4 Off-Policy Policy Evaluation

There is a class of problems in reinforcement learning named contextual bandits, also called bandits with side information, that provide a framework for learning models and evaluating algorithms on partially labeled data [2, 54]. Such data is generated in the following fashion: given a context X, a

vectorial representation of the state we are currently in, we sample an action m from a policy $\pi(X)$ that defines a multinomial distribution over K available actions. Afterwards, action m is effectively taken, and a reward y is observed, sampled from the real-world distribution of $p(y|m, X)$. The core of the problem is that $\pi(X)$ has a crucial impact on the data being collected: we will never observe the rewards associated with actions that were not taken (unless the exact same context occurs twice). This formulation lends itself well to the analysis and optimization of web ad campaigns [23], and is one of most popular frameworks for the task. Just like we will never know if a user would have clicked or not on an ad that was never shown, we will never know the outcome of a match until the players actually play it out. In our case, the decision of which players to match is taken by the matchmaking algorithm, which can thus be considered as a policy.

Here, m will denote a match, and X the set of available matches for selection, along with the associated player profiles. For a given context X (available matches), a matchmaking policy will choose one match to be launched. If $y_m \sim p(y_m|m, X)$ is the objective measure of balance given by the outcome of match m, the policy value is defined as:

$$V^\pi = \mathop{\mathbf{E}}_{X\sim D, m\sim\pi(X)}[y_m|m, X] \tag{4.4}$$

where D is the distribution of contexts, containing both the available matches and the corresponding player profiles, $m \sim \pi(X)$ is the match (selected according to policy π), and $p(y_m|m, X)$ is the underlying conditional distribution of rewards given a match and the associated player profiles. Assuming that higher values of the measure of balance are better, finding the optimal policy is formulated as estimating $\mathrm{argmax}_\pi V^\pi$.

The problem of learning on data collected by a (potentially unknown) policy is known as off-policy learning; the problem of evaluating a policy on data collected using a different policy is known as off-policy policy evaluation. Both have been well-studied for Markov decision processes [62], but are also applied in web applications such as advertising and predicting browsing behavior.

We argue that video game matchmaking suffers from these two problems, in the case where we want to replace a policy that is currently in effect and has been used to generate historical data. Suppose we fit (as in Section 4.3) a regression model $f(m, X) \approx \mathbf{E}[y_m|m, X]$ by minimizing the squared error between targets y_m and predicted values $f(m, X)$ over S_{train}, a subset of the dataset S comprised of triplets (X, m, y_m), generated by the policy b (called behavior policy). One might be tempted to use this approximation to estimate the value of a different policy π, through:

$$\hat{V}^\pi = \frac{1}{|S_{test}|} \sum_{X\in S_{test}, m\sim\pi(X)} f(m, X), \tag{4.5}$$

where S_{test} is the testing subset of S, non-overlapping with S_{train}. However, since $f(m, X)$ was trained to approximate the rewards associated with actions chosen by the behavior instead of π, this estimator (called "Direct Method") tends to suffer from strong bias, as reported by Dudik *et al.* [33]. An example of this bias would be a model trained on a dataset generated by a very good or a very poor behavior policy, in which case the model's average prediction would be, respectively, very large or very small. Naturally, the optimal way of evaluating a policy is to run it live and observe the average reward. This is problematic in practice for risk management concerns, since running a bad policy would negatively impact user experience, and harm the popularity of the game. Because of this, many methods were studied and developed to address this problem.

A popular class of algorithms for offline off-policy policy evaluation are based on importance sampling techniques. Simply put, we can estimate $\mathbf{E}_d[x]$ empirically using values sampled under a distribution different than d , i.e. $x_i \sim d'$, with

$$\hat{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \frac{d(x_i)}{d'(x_i)} \tag{4.6}$$

where $\hat{x}$ is an unbiased estimator of $\mathbf{E}_d[x]$, if d and d' are known exactly, i.e.,

$$\mathbf{E}[\hat{x}] = \mathbf{E}_d[x]. \tag{4.7}$$

Note however that importance weighted estimators, even if they are unbiased, may still have a large variance. This can be a significant issue when there are values of x for which $d'(x)$ is near 0 while $d(x) \neq 0$.

In terms of matchmaking, this means that if we know the distributions over actions given the context, $p_b(m|X)$ for the behavior policy b from which data was sampled, and $p_\pi(m|X)$ for the target policy to evaluate π, then we could use importance weights to estimate V^π:

$$V^\pi \approx \hat{V}^\pi = \frac{1}{|S|} \sum_{(X,m,y_m) \in S} y_m \frac{p_\pi(m|X)}{p_b(m|X)} \tag{4.8}$$

for data collected under policy b. Several approaches are based on importance weights to evaluate the value of a policy, and have been shown to work well in practice when importance weights are known or properly estimated.

However, to use these methods, one must know the distribution $p_b(m|X)$ of the behavior policy, or at least have a good approximation of it. This is hard in the case of matchmaking since the actual action space is enormous because of the combinatorial explosion of possible teams. The only way to

actually store available actions and their probability is to have a sampling mechanism which limits the number of possible matches to evaluate. Although the core model for the matchmaking system of a game can be known, the candidate matches at each context are not necessarily stored, as is the case in the data we were given access to. This makes the estimation of behavior action probabilites very hard, because we do not know which actions were actually available at the time of creation of each match. Not only that, but the behavior policy may actually be deterministic (as in our data), in which case we cannot direcly apply importance weighting, because the variance of the estimator does not exist and the importance sampling estimator becomes meaningless.

This motivates the exploration of techniques such as the Stacked Calibration methods proposed here, which can be applied even when the behavior policy is deterministic, i.e., where methods based on importance weights cannot be used.

In addition, we opt for a simulation-based approach to policy evaluation. Although susceptible to bias, we benefit from the fact that the core matchmaking system is already in place, and this allows us to more properly model the dynamics of matchmaking.

4.5 Matchmaking Recommendation Simulator

For reasons mentioned in the previous section, a simulator was built to evaluate different policy models. The simulator can be viewed as a type of single-server waiting queue. Each player arriving in the queue (which corresponds to a "lobby") is assigned attributes by randomly selecting them from the set of real player profiles. This is to simulate the distribution of player attributes. Since scoring each and every possible player configuration (i.e., a match) is impossible with more than a few dozen players, we randomly sample configurations of players, which will serve as match candidates. The set of randomly sampled matches corresponds to the context X from the previous section. The match candidates are scored by the policy model, and the highest-scoring match is selected. The corresponding players leave the queue, and new players can be added.

A practical concern of matchmaking systems is that every player must be matched within a certain time constraint, to avoid letting a player wait too much time before being matched. To this end, players who wait more than a certain threshold delay are forced into the next candidate match.

In our experiments, we used 100000 different attribute profiles to sample from. These are real user profiles corresponding to the profiles in the test set defined in section 4.3.3. The initial waiting queue size is 200, and players arrive at a constant rate; although this is not completely reflective of reality, it is a good approximation on a short time scale for most periods of time. A total of 50 random

matches were sampled at each iteration, from which the highest scoring is selected according to the policy model. The time constraint imposed is 20 iterations: after 20 iterations without being assigned in a match, a player is forced into the next match. In a real setting, the time constraint would be influenced by the arrival rate of players. With a simulator, we can run a policy, sample matches and generate the corresponding dataset.

4.6 Stacked Calibration

There remains an issue if we are to evaluate a model policy using the simulator. Using the same expected balance score both to select matches and to estimate matchmaking quality results in a "collusion bias". For each context, the policy must choose a match. Suppose the policy is deterministic, and chooses the match with highest expected value, given the policy model. The true most balanced match has an expected value that cannot exceed that of the match chosen by the policy. Therefore the chosen match is almost always worse than it appears based on the target policy π. Indeed, if

$$\begin{aligned} m^* &= \operatorname*{argmax}_{m \in X} f_{GT}(m) \\ \hat{m} &= \operatorname*{argmax}_{m \in X} f_{DM}(m) \end{aligned}$$

it follows that, for any given choice of matches,

$$f_{DM}(m^*) \leq f_{DM}(\hat{m}),$$

where f_{DM} is our trained estimator[4] and f_{GT} the true expected value (which we call "ground truth"). Although f_{DM} estimates the ground truth, the expected balance score of the match *selected* by the predictor during simulation (or during the live use of the predictor in the field) might still differ from it, because of the inherent limitations of various learning algorithms. This bias comes about because the same model is used *both for estimating value and for selecting an action*. This problem is analogous to the optimistic bias one observes when using training error to estimate generalization error of a learning algorithm: the learner tends to give lower errors on the training examples because these were used to select parameters. Here we have a policy that selects actions according to a predictor, and because these actions were the selected ones, they tend to receive a higher estimated value than their true value.

This motivates the proposed estimator, which we call *Stacked Calibration*. The procedure is simple. We let most of the parameters of the value predictor be estimated on a training set, but we estimate a

[4]DM stands for Direct Method [33].

calibration transformation on top of the behavior of the predictor on a validation set. We conjecture that this helps the predictor not only generalize better in general but also reduce the collusion bias. This procedure is a variant of the principle of *stacking* [79]), by which the outputs of one or more models are computed on examples not used to train them, and these outputs then enter as part of the input in examples to train a second level of models (and this can be repeated at multiple levels).

The specific procedures we used in the experiments are the following. Given the training, validation and test splits of a dataset S, we first train a value estimator $f_{DM} = f$ on the training set, and use the validation set to estimate

$$\tilde{f}(m, X) \approx \mathbf{E}[y_m | f(m, X)],$$

$\tilde{f}$ being the calibrated version of f, by minimizing

$$\sum_{(X,m,y_m) \in S_{valid}} (y_m - \tilde{f}(m, X))^2.$$

We have have tried two variants of calibration. The first is a simple linear least squares model, which we denote LDM, and the corresponding calibrated function. The second is a slight variant of radial basis function network (RBF), where

$$\tilde{f}(m, X) = \beta + f(m, X) + \sum_i w_i e^{\frac{(f(m,X)-\mu_i)^2}{\sigma^2}}. \tag{4.9}$$

Here μ_i are the radial basis function centers and w_i the associated weights, σ^2 is a scaling hyper-parameter and β is a bias term. In our experiments we used 100 centers, sampled uniformly from $f(m, X)$'s predictions on the validation set, as to easily cover the one-dimensional input space. The μ_i's were kept fixed and σ set to 1, so that the calibration parameters could be analytically estimated to minimize squared error over the validation set. In experiments we will denote the linear version as f_{LDM}, and the RBF version as f_{RBFDM}

4.7 Experimental Results

4.7.1 Experimental Setup

To assess the performance of the Stacked Calibration method we need access to the real expected value of the balance score of a match, which we will call the ground truth. The ground truth represents the underlying conditional distribution of a match's outcome given its players. Without it, we cannot conclude that calibration works without live testing, in which case we are back to square

one. To access the ground truth's expected value, we use a **two-stage simulation** methodology, in which a ground truth model f_{GT} is first trained from real data and then used to generate data and assess different learning and policy evaluation procedures. The details of this two-stage procedure are shown in Algorithm 2, allowing us to verify that the proposed stacked calibration procedure indeed improves policy value estimates.

Algorithm 2 Experimental procedure for two-stage simulation.

1: Train model f_{GT} on the real data. This model will be the ground truth.
2: Use the simulator under a specified policy b (behavior policy).
3: For the simulated matches, sample the outcome using f_{GT}. These sampled matches and outcomes constitute the artificial dataset S_b, generated by policy b.
4: Train model f_{DM} on the training subset of S_b, and use the simulator with f_{DM} as policy.
5: Using the validation subset of S_b, apply the calibration procedures described in section 4.6 to obtain f_{LDM} and f_{RBFDM}.
6: Evaluate the simulated matches using f_{DM}, the policy model, and calibrated models f_{LDM} and f_{RBFDM}.
7: Use f_{GT} to evaluate the true value of all these matches (and hence the estimated value of the policy), and compare this true policy value with the values estimated by the different off-policy policy evaluation methods compared here.

This methodology allows us to verify which policy evaluation method works best, since f_{GT} is known. In practice, the true underlying conditional distribution $p(y_m|m, X)$ is highly complex and difficult to capture completely with statistical models (assuming the task is complex enough), and the best models tend to be overregularized. To prevent models trained on artificial data generated from an overregularized ground truth to estimate the ground truth model too well, in which case directly estimating the matches with the policy model would be satisfactory, we used as ground truth a highly overfitted gradient boosted tree. In addition, Gaussian noise with standard deviation equal to 10% of the ground truth's standard deviation on the artificial data was added: the result is a difficult to capture (because it has a rich structure that is not too smoothed out by regularization), noisy signal, as we would expect to see in real data. The behavior policy we used was a random, uniform policy. The same simulator parameters were used for policy evaluation and data generation.

In order to provide comparison with already existing methods, we saved the action probabilities of the behavior and the associated sampled matches during simulation. This gives us what we need to compare the standard importance sampling method (IS) and doubly robust estimation [33], which is a method which uses both IS and the reward estimated by the f_{DM}. The doubly robust estimator

(DR) is defined by:

$$\hat{V_{DR}^{\pi}} = \frac{1}{|S|} \sum_{(X,m,y_m)\in S} [f_{DM}(\pi(X), X) + (y_m - f_{DM}(m, X)) \frac{p_\pi(m|X)}{p_b(m|X)}], \quad (4.10)$$

where S in the above equation represents a validation set not used to estimate f_{DM}. Given the uniform policy behavior, weights $p_b(m|X)$ are known exactly, which allows us to apply this method. In the case where the evaluated policy is deterministic, $p_\pi(m, X)$ becomes the indicator function for the behavior action being the same as the policy action. For a complete review of doubly robust policy evaluation, see Dudik *et al.* [33].

4.7.2 Comparative Results

The artificial data was split in 3 sets: 50000 matches as the training set and 25000 matches each for the validation and test sets. We trained a variety of models on the artificial data: linear regressions with L2 weight decay (ridge regression), gradient boosted decision trees, random forests, k-nearest neighbours regression and linear kernel support vector machines. The point of this diversity in learning algorithms is not necessarily to actually compare these different learning algorithms, but to make sure the evaluation works across a broad range of models. We applied Algorithm 2, performing random sampling of the hyperparameters to obtain various models (as variants of the same learning algorithm type), and we report mean squared error across all the models for each different evaluation method. DM signifies that the same function is used for match selection and evaluation. LDM denotes the linear calibration and RBFDM denotes the radial basis function network version, both described in section 4.6, evaluating the matches chosen by the (uncalibrated) policy. IS stands for importance sampling, and DR for the doubly robust method (both described in section 4.4).

Figure 4.2 shows, as expected, that the uncalibrated direct method is the worst performing one. The other four are considerably better, with RBFDM leading the chart. To ensure that these differences were significant, we computed a paired t-test comparing each pair of evaluation procedures, with null hypothesis being that the expected mean squared errors are the same. The p-values for these tests are reported in table 4.3, with p-values smaller than 0.05 in bold (where the null hypothesis is rejected, i.e. indicating a statistically significant difference).

To further verify the policy value estimation, we computed Spearman's rank correlation coefficient between each model's estimated policy value and the ground truth, for each evaluation method (Table 4.2). Again, DM is the worst performing method. The important thing to note is that RBFDM

can select the best model almost as well as DR, showing that model selection is on par with this method. As reference the test errors of the models give a correlation coefficient of 0.91 with the ground truth value; we can estimate policy value accurately and select the (near) optimal one.

In analyzing these results, keep in mind that the objective was to verify if the stacked calibration methods could approach the performance of methods such as the doubly robust (DR) estimator, while not requiring a known and stochastic behavior model. The results are clearly positive in this respect.

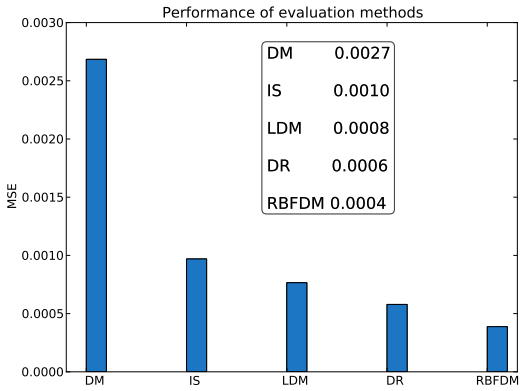

FIGURE 4.2: Precision of value approximation for different evaluation methods: *DM* for the direct uncalibrated method, *IS* for important sampling, *LDM* for linear calibration, *DR* for doubly robust, *RBFDM* for rbf calibration. RBFDM outperforms all the other methods. See also Table 4.3 for the significance tests.

TABLE 4.2: Spearman rank correlation between evaluation methods and ground truth.

	RBFDM	DR	LDM	IS	DM
correlation coefficient	0.9011	0.9366	0.8328	0.611	0.2575
p-value	1.03e-37	6.9e-47	3.6e-27	1.1e-11	0.0093

TABLE 4.3: P values of paired-t test of squared error between evaluation methods

	GT	RBFDM	DR	LDM	IS	DM
GT		$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$
RBFDM	$\ll \mathbf{10^{-4}}$		**0.0163**	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$
DR	$\ll \mathbf{10^{-4}}$	**0.0163**		**0.0473**	**0.0011**	$\ll \mathbf{10^{-4}}$
LDM	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	**0.0473**		0.1593	$\ll \mathbf{10^{-4}}$
IS	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	**0.0011**	0.1593		$\ll \mathbf{10^{-4}}$
DM	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	$\ll \mathbf{10^{-4}}$	

4.8 Conclusions

In this paper we have reviewed and compared matchmaking procedures. We have shown that modelling player-player interactions through deep architectures is helpful in predicting the outcome of a match, as these interactions are in reality complex and difficult to model with shallower models.

We then reviewed off-policy policy evaluation techniques, and pointed out some practical difficulties encountered in the context of evaluating matchmaking methods in an industrial setting where the behavior policy is unknown or not stochastic. We then proposed a technique called Stacked Calibration to improve off-policy evaluation and tested its effect through a two-stage simulator methodology.

Through these simulations, we verified that Stacked Calibration performs as well as or better than standard offline methods such as the doubly robust estimator, which are not applicable in our industrial setting. We showed that it is a reliable way of evaluating policy value. The whole approach is applicable to any setting where importance weights are not applicable and where simulation is preferred to (or in addition to) A/B testing because A/B testing is expensive, worrisome for product managers concerned with the negative effects of a poor policy, and limited in the number of policies that can be evaluated (only so many different policies can be A/B tested in a given amount of time).

Chapitre 5

Introduction au deuxième article

Deep Generative Stochastic Networks Trainable by Backprop
Yoshua Bengio et Eric Laufer
Refusé pour *Neural Information Processing Systems 2013*

5.1 Contexte

Comme mentionné en 2.2.2, un des intérêts majeurs des algorithmes d'apprentissage non-supervisés est que certains d'entre eux apprennent à décomposer les entrées selon leurs facteurs de variations importants, comme les DAE [78] et les CAE [65]. Ceci permet d'apprendre la structure des données malgré l'absence de cibles, celles-ci étant un élément essentiel de l'apprentissage supervisé, qui constitue la majorité des applications d'apprentissage machine. Une des grandes motivations pour le développement de tels algorithmes est la quantité de données non-étiquetés et donc le potentiel d'apprentissage à grande échelle, supervisé ou non.

Les machines de boltzmann restreintes (RBM) sont des réseaux de neurones avec une paramétrisation presque identique aux autoencodeurs, mais entraînés de façon très différente. Les entrées et les unités cachées sont binaires dans le cas classique. Une RBM possède une fonction d'énergie $E(\mathbf{x}, \mathbf{h}) = -\sum_i \mathbf{x}_i \mathbf{c}_i - \sum_j \mathbf{h}_j \mathbf{b}_j - \sum_{i,j} \mathbf{x}_i \mathbf{W}_{i,j} \mathbf{c}_j$, égale à une constante près à la log-vraisemblance négative de la probabilité jointe de $\mathbf{x}$ et $\mathbf{h}$. Ceci permet de définir $p(\mathbf{x}, \mathbf{h})$ en introduisant un terme de normalisation sur toutes les configurations possibles d'entrées et d'unités cachées : $p(\mathbf{x}, \mathbf{h}) = \frac{e^{-E(\mathbf{x},\mathbf{h})}}{Z}$ avec $Z = \sum_{\mathbf{h},\mathbf{x}} e^{-E(\mathbf{x},\mathbf{h})}$, nommée fonction de partition. Cependant, ce terme est incalculable à cause du nombre exponentiel de configurations possibles. Malgré cela, il est possible

d'entraîner les RBMs pour modéliser $p(\mathbf{x})$ avec une approximation stochastique du gradient de la log-vraisemblance de $\mathbf{x}$. Une RBM est un modèle génératif: nous pouvons échantilloner des exemples selon l'approximation de $p(\mathbf{x})$ avec une méthode Monte-Carlo par chaîne de Markov. Bien que les CAE ne soient pas des modèles génératifs, [67] propose une méthode pour obtenir des échantillons en ajoutant du bruit aux unités cachées, se déplaçant ainsi dans la variété des données. Les CAE et DAE sont aussi utilisables de la même façon que les RBMs pour initialiser des réseaux profonds. Ces résultats empiriques suggèrent que les autoencodeurs captent d'une certaine façon la distribution des données, comme les RBMs le font.

5.2 Contributions et commentaires

L'article suivant présente des motivations théoriques pour l'utilisation d'autoencodeurs comme modèles génératifs. Les méthodes de Monte-Carlo par chaîne de Markov pour entraîner les RBMs ne visitent que rarement les configurations improbables du réseau. Si pour passer d'un mode à l'autre de la distribution des données la chaîne de Markov doit visiter une zone improbable, il se peut que celle-ci "rebrousse chemin" et revienne au mode probable le plus proche. Nous avons donc besoin d'un très grand nombre de transitions dans la chaîne pour changer de mode, ce qui constitue un des problèmes typique associés aux RBMs (problèmes de "mixing"). Les réseaux génératifs stochastiques tentent de contourner ce problème en démontrant que les échantillons obtenus avec un processus de corruption puis de reconstruction comme celui dans les DAE peut générer des échantillons qui suivent la distributions des données sous certaines conditions faibles. Ceci nous permet d'éviter la présence d'une fonction de partition, d'entraîner par rétro-propagation et possiblement améliorer la vitesse de changement de mode lors de l'échantillonnage. De plus, le processus de corruption est généralisé au-delà des DAE. Des expériences qualitative démontrent une bonne vitesse de changement de mode lors de l'échantillonnage de chiffres écrits à la main. Au final, l'article fut rejeté (soumis à *Neural Information Processing Systems*), principalement dû au manque d'intérêt du comité de lecture pour les résultats théoriques, ainsi qu'au manque de résultats supervisés, car surpasser l'état de l'art dans une tâche supervisée est un des barêmes classiques d'acceptation. Ma contribution personelle à l'article fut l'implémentation de l'algorithme d'apprentissage proposé ainsi que l'obtention des résultats principaux.

Chapitre 6

Deep Generative Stochastic Networks Trainable by Backprop

6.1 Abstract

Recent work showed that denoising auto-encoders can be interpreted as generative models. We generalize these results to arbitrary parametrizations that learn to reconstruct their input and where noise is injected, not just in input, but also in intermediate computations. We show that under reasonable assumptions (the parametrization is rich enough to provide a consistent estimator, and it prevents the learner from just copying its input in output and producing a dirac output distribution), such models are consistent estimators of the data generating distributions, and that they define the estimated distribution through a Markov chain that consists at each step in re-injecting sampled reconstructions as a sequence of inputs into the unfolded computational graph. As a consequence, one can define deep architectures similar to deep Boltzmann machines in that units are stochastic, that the model can learn to generate a distribution similar to its training distribution, that it can easily handle missing inputs, but without the troubling problem of intractable partition function and intractable inference as stumbling blocks for both training and using these models. In particular, we argue that if the underlying latent variables of a graphical model form a highly multimodal posterior (given the input), none of the currently known training methods can appropriately deal with this multimodality (when the number modes is much greater than the number of MCMC samples one is willing to perform, and when the structure of the posterior cannot be easily approximated by some tractable variational approximation). In contrast, the proposed models can simply be trained

by back-propagating the reconstruction error (seen as log-likelihood of reconstruction) into the parameters, benefiting from the power and ease of training recently demonstrated for deep supervised networks with dropout noise.

6.2 Introduction

Research in deep learning (see Bengio [5] and Bengio *et al.* [12] for reviews) has started with breakthroughs in unsupervised learning of representations, based mostly on the Restricted Boltzmann Machine (RBM) [47], auto-encoder variants [8, 78], and sparse coding variants [57, 63]. However, the most impressive recent results have been obtained with purely supervised learning techniques for deep networks, in particular for speech recognition [29, 32, 72] and object recognition [52]. In all of these cases, the availability of large quantities of labeled data was important, and the latest breakthrough in object recognition [52] was achieved with fairly deep convolutional networks with a form of noise injection in the input and hidden layers during training, called dropout [48].

On the other hand, progress with deep unsupervised architectures has been slower. Although single-layer unsupervised learners are fairly well developed, jointly training all the layers with respect to a single unsupervised criterion remains a challenge, with the best current options being the Deep Belief Network (DBN) [47] and the Deep Boltzmann Machine (DBM) [69]. Nonetheless, joint unsupervised training of all the layers remains a difficult, much more so than for its supervised counterparts. Since the amount of unlabeled data potentially available is very large, it would be very interesting to develop unsupervised learning algorithms for generative deep architectures that can take advantage of the progress in techniques for training deep supervised ones, i.e., based on back-propagated gradients. The approach presented here is one step in this direction, allowing to train jointly train all the levels of representation of a deep unsupervised probabilistic model solely by back-propagating gradients.

Another motivation for the approach presented here is a potential problem with probabilistic models with anonymous latent variables[1], discussed in more detail in Section 6.3. The gist of the issue is the following. Graphical models with latent variables often require dealing with either or both of the following fundamentally difficult problems in the inner loop of training, or to actually use the model for taking decisions: inference (estimating the posterior distribution over latent variables h given inputs x) and sampling (from the joint model of h and x). However, if the posterior $P(h|x)$

[1] they are called anonymous because no a priori semantics is assigned to them, like in Boltzmann machines, and unlike in many knowledge-based graphical models. Whereas inference over non-anonymous latent variables is required to make sense of the model, anonymous variables are only a device to capture the structure of the distribution and need not have a clear human-readable meaning.

has a huge number of modes that matter, then all of the current approaches may be doomed for such tasks.

The main contribution of this paper is a theoretical extension of recent work (summarized in Section 6.4) on the generative view of denoising auto-encoders. It provides a statistically consistent way of estimating the underlying data distribution based on a denoising-like criterion where the noise can be injected not just in the input but anywhere in the computational graph that produces the predicted distribution for the denoised input.

We apply this idea to deep Generative Stochastic Networks (GSNs) whose computational graph resembles the one followed by Gibbs sampling in deep Boltzmann machines, but that can be trained efficiently with back-propagated gradients. The models can be trained and used in the presence of missing inputs, and they can be used to sample from the learned distribution (possibly conditioning on some of the inputs).

6.3 A Potential Problem with Anonymous Latent Variable Models

All of the graphical models studied for deep learning except the humble RBM require a non-trivial form of inference, i.e., guessing values of the latent variables h that are appropriate for the given visible input x. Several forms of inference have been investigated in the past: MAP inference is formulated like an optimization problem (looking for h that approximately maximizes $P(h \mid x)$); MCMC inference attempts to sample a sequence of h's from $P(h \mid x)$; variational inference looks for a simple (typically factorial) approximate posterior $q_x(h)$ that is close to $P(h \mid x)$, and usually involves an iterative optimization procedure. See a recent machine learning textbook for more details [61].

In addition, a challenge related to inference is sampling (not just from $P(h \mid x)$ but also from $P(h, x)$ or $P(x)$), which like inference is often needed in the inner loop of learning algorithms for probabilistic models with latent variables such as energy-based models [56] or Markov Random Fields, where $P(x)$ or $P(h, x)$ is defined in terms of a parametrized energy function whose normalized exponential gives probabilities. Deep Boltzmann machines [69] combine the difficulty of inference (for the *"positive phase"* where one tries to push the energies associated with the observed x down) and also that of sampling (for the *"negative phase"* where one tries to push up the energies associated with x's sampled from $P(x)$). Sampling for the negative phase is usually done by MCMC, although some unsupervised learning algorithms [15, 26, 42] involve "negative examples" that are sampled through simpler procedures (like perturbations of the observed input, in a spirit

reminiscent of the approach presented here). In Salakhutdinov and Hinton [69], inference for the positive phase is achieved with a mean-field variational approximation.[2]

6.3.1 Potentially Huge Number of Modes.

The challenge we propose to think about has to do with the potential existence of highly multimodal posteriors: all of the currently known approaches to inference and sampling are making very strong explicit or implicit assumptions about the form of the distribution of interest ($P(h \mid x)$ or $P(h, x)$). As we argue below, these approaches make sense if this target distribution is either approximately unimodal (MAP), (conditionally) factorizes (variational approximations, i.e., the different factors h_i are approximately independent[3] of each other given x), or has only a few modes between which it is easy to mix (MCMC). However, approximate inference can be potentially hurtful, not just at test time but for training [53], because it is often in the inner loop of the learning procedure. We want to consider here the case where neither a unimodal assumption (MAP), the assumption of a few major modes (MCMC) or of fitting a variational approximation (factorial or tree-structured distribution) are appropriate.

Imagine for example that h represents many explanatory variables of a rich audio-visual "scene" with a highly ambiguous raw input x, including the presence of several objects with ambiguous attributes or categories, such that one cannot really disambiguate one of the objects independently of the others (the so-called "structured output" scenario, but at the level of latent explanatory variables). Clearly, a factorized or unimodal representation would be inadequate (because these variables are not at all independent, given x) while the number of modes could grow exponentially with the number of ambiguous factors present in the scene. For example, consider x being the audio of speech pronounced in a foreign language that you do not master well, so that you cannot really segment and parse well each of the words. The number of plausible interpretations (given your poor knowledge of that foreign language) could be exponentially large (in the length of the utterance), and the individual factors (words) would certainly not be conditionally independent (actually having a very rich structure which corresponds to a language model). Even ignoring segmentation makes this a very difficult problem: say there are 10 word segments, each associated with 100 different plausible candidates (out of a million, counting proper nouns), but, due to the language model, only 1 out of 1000 of their combinations being plausible (i.e., the posterior does not factorize or fit a

[2] In the mean-field approximation, computation proceeds like in Gibbs sampling, but with stochastic binary values replaced by their conditional expected value (probability of being 1), given the outputs of the other units. This deterministic computation is iterated like in a recurrent network until convergence is approached, to obtain a marginal (factorized probability) approximation over all the units.

[3] this can be relaxed by considering tree-structured conditional dependencies [70] and mixtures thereof

tree structure). So one really has to consider $\frac{1}{1000} \times 100^{10} = 10^{17}$ *plausible configurations* of the latent variables. If one has to take a decision y based on x, e.g., $P(y \mid x) = \sum_h P(y \mid h)P(h \mid x)$ involves summing over a huge number of non-negligible terms of the posterior $P(h \mid x)$, which we can consider as important modes. One way or another, *summing explicitly over that many modes seems implausible*, and assuming single mode (MAP) or a factorized distribution (mean-field) would yield very poor results. Under some assumptions on the underlying data-generating process, it might well be possible to do inference that is exact or a provably good approximation, and searching for graphical models with these properties is an interesting avenue to deal with this problem. Basically, these assumptions work because we assume a specific structure in the form of the underlying distribution. Also, if we are lucky, a few Monte-Carlo samples from $P(h \mid x)$ might suffice to obtain an acceptable approximation for our y, because somehow, as far as y is concerned, many probable values of h yield the same answer y and a Monte-Carlo sample will well represent these different "types" of values of h. That is one form of regularity that could be exploited (if it exists) to approximately solve that problem. What if these assumptions are not appropriate to solve challenging AI problems? Another, more general assumption (and thus one more likely to be appropriate for these problems) is similar to what we usually do with machine learning: function approximation, i.e., although the space of functions is combinatorially large, we are able to generalize by postulating a rather large and flexible family of functions (such as a deep neural net). Thus an interesting avenue is to assume that there exists a computationally tractable function that can compute $P(y \mid x)$ in spite of the apparent complexity of going through the intermediate steps involving h, and that we may learn $P(y \mid x)$ through (x, y) examples. In fact, this is exactly what we do when we train a deep supervised neural net or any black-box supervised machine learning algorithm.

The question we want to address here is that of unsupervised learning: could we take advantage of this idea of bypassing explicit latent variables in the realm of unsupervised probabilistic models of the data? The approach proposed here has this property. It avoids the strong assumptions on the latent variable structure but still has the potential of capturing very rich distributions, by having only "function approximation" and no approximate inference. Although it avoids explicit latent variables, it still retains the property of exploiting sampling in the computations associated with the model in order to answer questions about the variables of interest.

6.4 Denoising Auto-Encoders as Generative Models

Alain and Bengio [1] showed that denoising auto-encoders with small Gaussian corruption and

squared error loss estimated the score (derivative of the log-density with respect to the input) of continuous observed random variables. More recently, Bengio *et al.* [10] generalized this to arbitrary variables (discrete, continuous or both), arbitrary corruption (not necessarily asymptotically small), and arbitrary loss function (so long as can be seen as a log-likelihood). We first summarize these results below.

Let $\mathcal{P}(X)$ be the unknown data generating distribution. Let $\mathcal{C}(\tilde{X}|X)$ be a corruption process that stochastically transforms an X (such as sampled from $\mathcal{P}$) into a random variable $\tilde{X}$. Let $P_\theta(X|\tilde{X})$ be a denoising auto-encoder that assigns a probability to X, given $\tilde{X}$, when $\tilde{X} \sim \mathcal{C}(\tilde{X}|X)$. When n training examples are provided we obtain an estimator parametrized by θ_n. This estimator defines a Markov chain T_n obtained by sampling alternatively an $\tilde{X}$ from $\mathcal{C}(\tilde{X}|X)$ and an X from $P_\theta(X|\tilde{X})$. Let π_n be the asymptotic distribution of the chain defined by T_n, if it exists. The following theorem is proven by Bengio *et al.* [10].

Theorem 6.1. *If $P_{\theta_n}(X|\tilde{X})$ is a consistent estimator of the true conditional distribution $\mathcal{P}(X|\tilde{X})$ and T_n defines an irreducible and ergodic Markov chain, then as $n \to \infty$, the asymptotic distribution $\pi_n(X)$ of the generated samples converges to the data generating distribution $\mathcal{P}(X)$.*

It is accompanied with the following corollary, which defines some sufficient conditions for convergence of the chain, and hence applicability of Theorem 6.1.

Corollary 6.2. **If** *$P_\theta(X|\tilde{X})$ is a consistent estimator of the true conditional distribution $\mathcal{P}(X|\tilde{X})$,* **and** *both the data generating distribution and denoising model are contained in and non-zero in a finite-volume region V (i.e., $\forall \tilde{X}$, $\forall X \notin V$, $\mathcal{P}(X) = 0, P_\theta(X|\tilde{X}) = 0$),* **and** $\forall \tilde{X}$, $\forall X \in V$, $\mathcal{P}(X) > 0, P_\theta(X|\tilde{X}) > 0, \mathcal{C}(\tilde{X}|X) > 0$ **and** *these statements remain true in the limit of $n \to \infty$,* **then** *the asymptotic distribution $\pi_n(X)$ of the generated samples converges to the data generating distribution $\mathcal{P}(X)$.*

6.5 Reconstruction with Noise Injected in the Reconstruction Function: Consistent Estimation of the Underlying Data Generating Distribution

In the context where Theorem 6.1 was proven [10], $\tilde{X}$ is a noisy or corrupted version of X, i.e., it lives in the same space as X, but nothing in the proof requires that. We exploit this observation to obtain a generalization in which we consider a source of noise Z independent from X, an arbitrary

differentiable function $f_{\theta_1}(X, Z)$ from which X cannot be recovered exactly, and a reconstruction distribution $P_{\theta_2}(X|f_{\theta_1}(X, Z))$ which is trained to predict X given $f_{\theta_1}(X, Z)$.

Note that a special case that returns to the situation studied in Bengio *et al.* [10] is when $f(X, Z)$ is a fixed parameter-less corruption function that combines the noise Z with X to obtain a corrupted sample $\tilde{X}$, i.e. $\tilde{X} \sim \mathcal{C}(\tilde{X}|X)$. Equivalently, $\tilde{X}$ can be sampled by first sampling $Z \sim \mathcal{P}(Z)$ from a noise distribution and then applying the deterministic function $f(X, Z)$ to obtain $\tilde{X}$. Note that in practice, this is how random variates are generally sampled.

This view gives rise to the following corollary which is central to this paper.

Corollary 6.3. *Let training data $X \sim \mathcal{P}(X)$ and independent noise $Z \sim \mathcal{P}(Z)$. Consider a model $P_{\theta_2}(X|f_{\theta_1}(X, Z))$ trained (over both θ_1 and θ_2) by regularized conditional maximum likelihood with n examples of (X, Z) pairs. For a given θ_1, a random variable $\tilde{X} = f_{\theta_1}(X, Z)$ is defined. Assume that as n increases, P_{θ_2} is a consistent estimator of the true $\mathcal{P}(X|\tilde{X})$. Assume also that the Markov chain $X_t \sim P_{\theta_2}(X|f_{\theta_1}(X_{t-1}, Z_{t-1}))$ (where $Z_{t-1} \sim \mathcal{P}(Z)$) converges to a distribution π_n, even in the limit as $n \to \infty$. Then $\pi_n(X) \to \mathcal{P}(X)$ as $n \to \infty$.*

Proof. Consider that for a fixed n, θ_1 and θ_2 have been estimated such that, as assumed, as $n \to \infty$, θ_2 gives rise to a consistent estimator of $\mathcal{P}(X|\tilde{X})$. Note how the above Markov chain is equivalent to the Markov chain in Theorem 6.1 when we define $\tilde{X} = f_{\theta_1}(X, Z)$. We have the two conditions of Theorem 6.1 (consistent estimator of the conditional $\mathcal{P}(X|\tilde{X})$ and convergence of the Markov chain), so we can conclude that π_n converges to $\mathcal{P}(X)$. □

Since we are now considering the case where f is not a fixed function but a learned one, we have to be careful about defining it in such a way as to guarantee convergence of the Markov chain from which one would sample according to the estimated distribution. Corollary 6.2 provides some guidance for this purpose. In particular, in order to make sure that $\mathcal{C}(\tilde{X}|X) > 0$ for a given n and asymptotically, it would be necessary that f_θ be constructed such that the conditional entropy $H(f_\theta(X, Z)|X) > 0$, both for any fixed n and asymptotically. It means that the learning procedure should not have the freedom to choose parameters so as to simply eliminate the uncertainty injected with Z. Otherwise, the reconstruction distribution would simply converge to a dirac at the input X. This is the analogue of the constraint on auto-encoders that is needed to prevent them from learning the identity function. Here, we must design the family of reconstruction functions (which produces a distribution over X, given Z and X) such that when the noise Z is injected, there are always several possible values of X that could have been the correct original input.

Another extreme case to think about is when $f(X, Z)$ is overwhelmed by the noise and lost all information about X. In that case the theorems are still valid while giving uninteresting results: the learner must capture the full distribution of X in $P_{\theta_2}(X|\tilde{X})$ because the latter is now equivalent to $P_{\theta_2}(X)$, since $\tilde{X} = f(X, Z)$ does not contain any information about X. What this illustrates, though, is that when the noise is large, the reconstruction distribution (parametrized by θ_2) will need to have the expressive power to represent multiple modes. Otherwise, the reconstruction will tend to capture some kind of average output, which would visually look like a fuzzy combination of the actual modes. In the experiments performed here, we have only considered unimodal reconstruction distributions (with factorized outputs), but future work should investigate multimodal alternatives.

A related element to keep in mind is that one should pick the family of conditional distributions $P_{\theta_2}(X|\tilde{X})$ so that one can sample from them and one can easily train them when given $(X, \tilde{X})$ pairs, e.g., by maximum likelihood.

6.6 Dealing with Missing Inputs or Structured Output

In general, a simple way to deal with missing inputs is to clamp the observed inputs and then apply the Markov chain with the constraint that the observed inputs are fixed and not resampled at each time step, whereas the unobserved inputs are resampled each time. More precisely, the output (reconstruction) distribution of the model must allow us to sample a subset of variables in the vector X conditionally on the value of the rest. One can readily prove that this procedure gives rise to sampling from the appropriate conditional distribution.

Proposition 6.4. *If a subset $x^{(s)}$ of the elements of X is kept fixed (not resampled) while the remainder $X^{(-s)}$ is updated stochastically during the Markov chain of corollary 6.3, but using $P(X_{t+1}|f(X_t, Z_t), X^{(s)}_{t+1} = x^{(s)})$, then the asymptotic distribution π_n produces samples of $X^{(-s)}$ from the conditional distribution $\pi_n(X^{(-s)}|X^{(s)} = x^{(s)})$.*

Proof. Without constraint, we know that at convergence of the chain, $P(X_t|f(X_{t-1}, Z_{t-1}))$ produces a sample of π_n. A subset of these samples satisfies the condition $X = x^{(s)}$, and these constrained samples could equally have been produced by sampling from $P(X_t|f(X_{t-1}, Z_{t-1}), X^{(s)}_{t+1} = X^{(s)})$, by definition of conditional distribution. Therefore, at convergence of the chain, we have that $P(X_t|f(X_{t-1}, Z_{t-1}), X^{(s)}_{t+1} = x^{(s)})$ produces a sample from π_n under the condition $X^{(s)} = x^{(s)}$. □

Practically, it means that we must choose a reconstruction distribution from which it is not only easy to sample from, but also from which it is easy to sample conditioned on any subset of the values being known. In the experiments below, we have used a factorial distribution for the reconstruction, meaning that it is trivial to sample conditionally a subset of the input variables.

This method of dealing with missing inputs can be immediately applied to dealing with structured outputs. If $X^{(s)}$ is viewed as an "input" and $X^{(-s)}$ as an "output", then sampling from the chain $X_{t+1}^{(-s)} \sim P(X^{(-s)}|f((X^{(s)}, X_t^{(-s)}), Z_t), X^{(s)})$ will converge to estimators of $\mathcal{P}(X^{(-s)}|X^{(s)})$. This still requires good choices of the parametrization (for f as well as for the conditional probability P), but the advantages of this approach are that there is no approximate inference of latent variables and the learner is trained with respect to simpler conditional probabilities: in the limit of small noise, we conjecture that these conditional probabilities can be well approximated by unimodal distributions. One piece of theoretical evidence comes from Alain and Bengio [1]: *when the amount of corruption noise converges to 0 and the input variables have a smooth continuous density, then a unimodal Gaussian reconstruction density suffices to fully capture the joint distribution.*

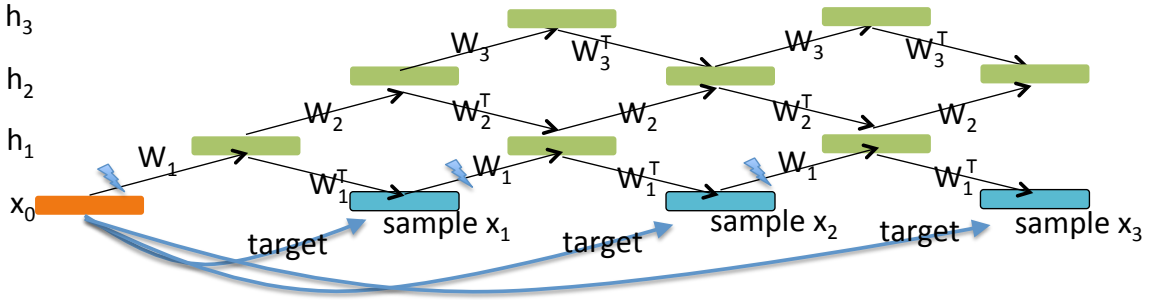

FIGURE 6.1: Unfolded computational graph inspired by the Deep Boltzmann Machine inference or sampling, but with backprop-able stochastic units at each layer. The training example $X = x_0$ starts the chain. Either odd or even layers are stochastically updated at each step. Original or sampled x_t's are corrupted by salt-and-pepper noise before entering the graph (lightning symbol). Each x_t for $t > 0$ is obtained by sampling from the reconstruction distribution for this step, and the log-likelihood of target X under that distribution is also computed and used as part of the training objective.

6.7 Deep Generative Stochastic Networks Trainable by Backprop

The theoretical results on Generative Stochastic Networks (GSNs) in this paper open a large class of possible parametrizations which will share the property that they can capture the underlying data distribution through the above Markov chain. What parametrizations will work well? Where and

how to inject noise? We present here the results of preliminary experiments with specific choices for these, but the reader should keep in mind that the space of possibilities is vast.

As a conservative starting point, we propose to explore families of parametrizations which are similar to existing deep stochastic architectures such as the Deep Boltzmann Machine (DBM) [69] and the Deep Belief Network (DBN) [47]. Basically, the idea is to construct a computational graph that is similar to the computational graph for Gibbs sampling or variational inference in Deep Boltzmann Machines. However, we have to diverge a bit from these architectures in order to accommodate the desirable property that it will be possible to back-propagate the gradient of reconstruction log-likelihood with respect to the parameters θ_1 and θ_2. Since the gradient of a binary stochastic unit is 0 almost everywhere, we have to consider related alternatives. An interesting source of inspiration regarding this question is a recent paper on estimating or propagating gradients through stochastic neurons [7]. Here we consider the following stochastic non-linearities: $h_i = \eta_{\text{out}} + \tanh(\eta_{\text{in}} + a_i)$ where a_i is the linear activation for unit i (an affine transformation applied to the input of the unit, coming from the layer below, the layer above, or both), η_{in} and η_{out} are zero-mean Gaussian noises.

To emulate a sampling procedure similar to Boltzmann machines in which the filled-in missing values can depend on the representations at the top level, the computational graph must allow information to propagate both upwards (from input to higher levels of representation) and backwards (vice-versa), giving rise to the computational graph structured illustrated in Figure 6.1, which is similar to that explored for *deterministic* recurrent auto-encoders [3, 71, 73]. Downward weight matrices have been fixed to the transpose of corresponding upward weight matrices.

The *walkback* algorithm was proposed in Bengio *et al.* [10] to make training of generalized denoising auto-encoders (a special case of the models studied here) more efficient. The basic idea is that the reconstruction is actually obtained after several steps of the sampling Markov chain. In the context presented here, it simply means that the computational graph from X to a reconstruction probability actually involves generating intermediate samples as if we were running the Markov chain starting at X. In the experiments, the graph was unfolded so that $2D$ sampled reconstructions would be produced, where D is the depth (number of hidden layers). The training loss is the sum of the reconstruction negative log-likelihoods (of target X) over all those reconstruction steps.

6.8 Experimental Validation of GSNs

Experiments evaluating the ability of the GSN models to generate good samples were performed on the MNIST and TFD datasets, following the setup in Bengio *et al.* [9]. Networks with 2 and 3

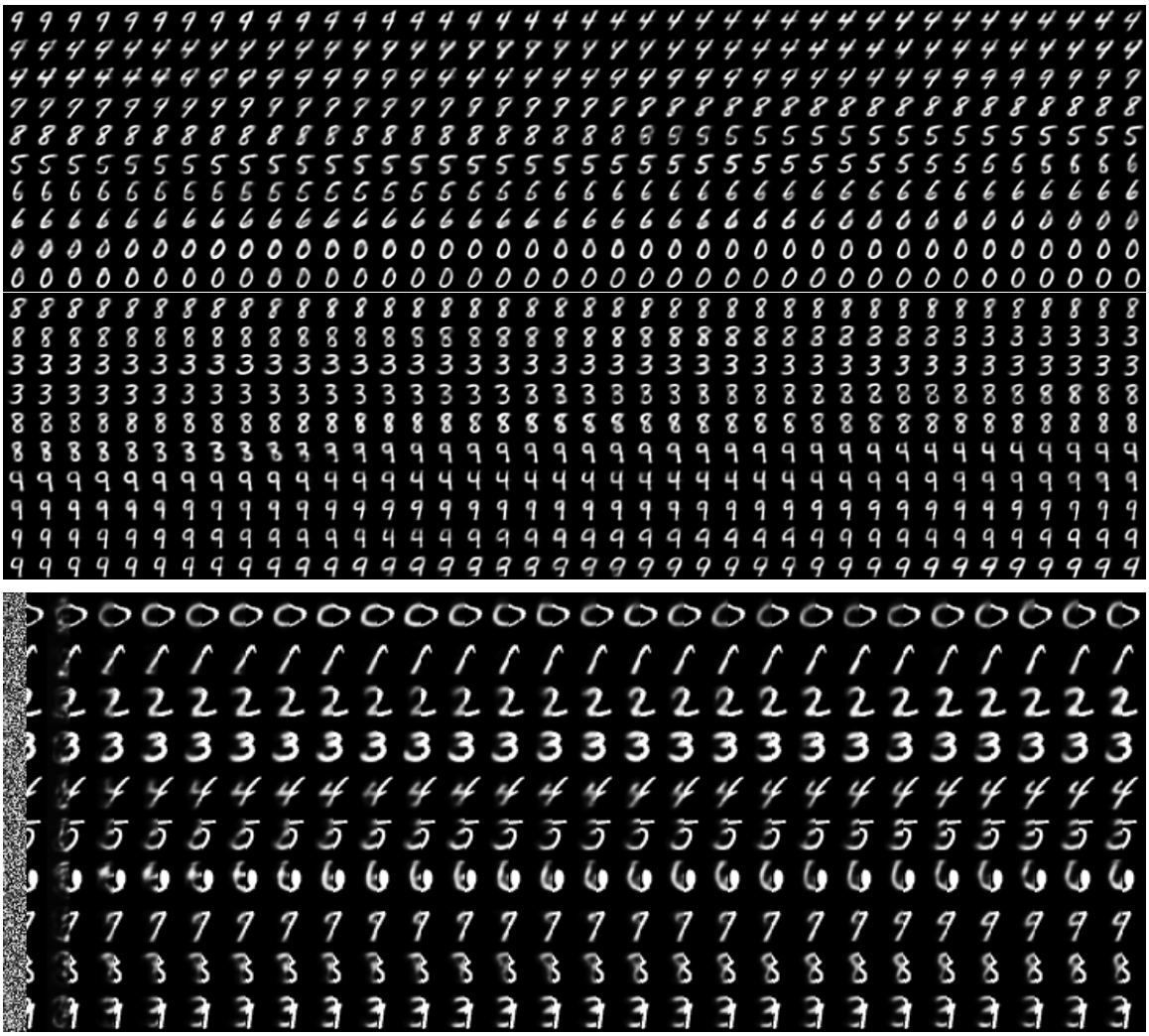

FIGURE 6.2: Top: two runs of consecutive samples (one row after the other) generated from a 2-layer GSN model, showing that it mixes well between classes and produces nice and sharp images. Bottom: conditional Markov chain, with the right half of the image clamped to one of the MNIST digit images and the left half successively resampled, illustrating the power of the trained generative model to stochastically fill-in missing inputs.

hidden layers were evaluated and compared to regular denoising auto-encoders (just 1 hidden layer, i.e., the computational graph separates into separate ones for each reconstruction step in the walk-back algorithm). They all have tanh hidden units and pre- and post-activation Gaussian noise of standard deviation 2, applied to all hidden layers except the first. In addition, at each step in the chain, the input (or the resampled X_t) is corrupted with salt-and-pepper noise of 40% (i.e., 40% of the pixels are corrupted, and replaced with a 0 or a 1 with probability 0.5). Training is over 100 to 600 epochs at most, with good results obtained after around 100 epochs. Hidden layer sizes vary between 1000 and 1500 depending on the experiments, and a learning rate of 0.25 and momentum

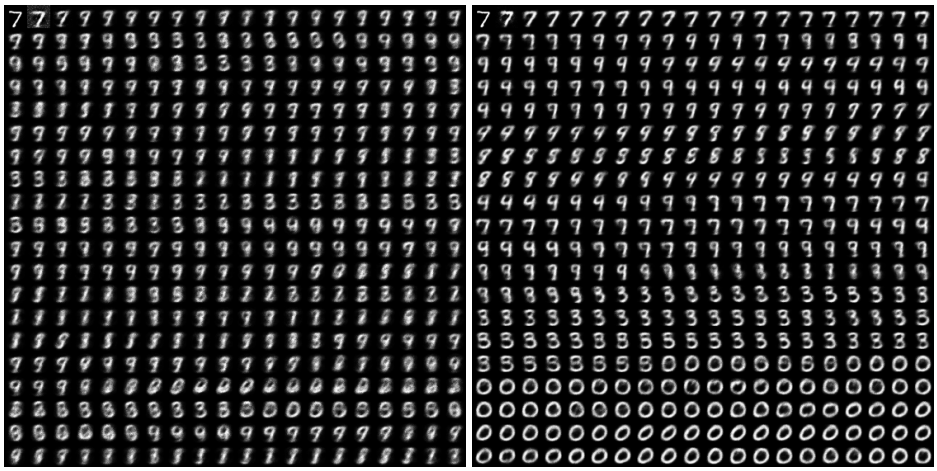

FIGURE 6.3: Left: consecutive GSN samples obtained after 10 training epochs. Right: GSN samples obtained after 25 training epochs. This shows quick convergence to a model that samples well. The samples in Figure 6.2 are obtained after 600 training epochs.

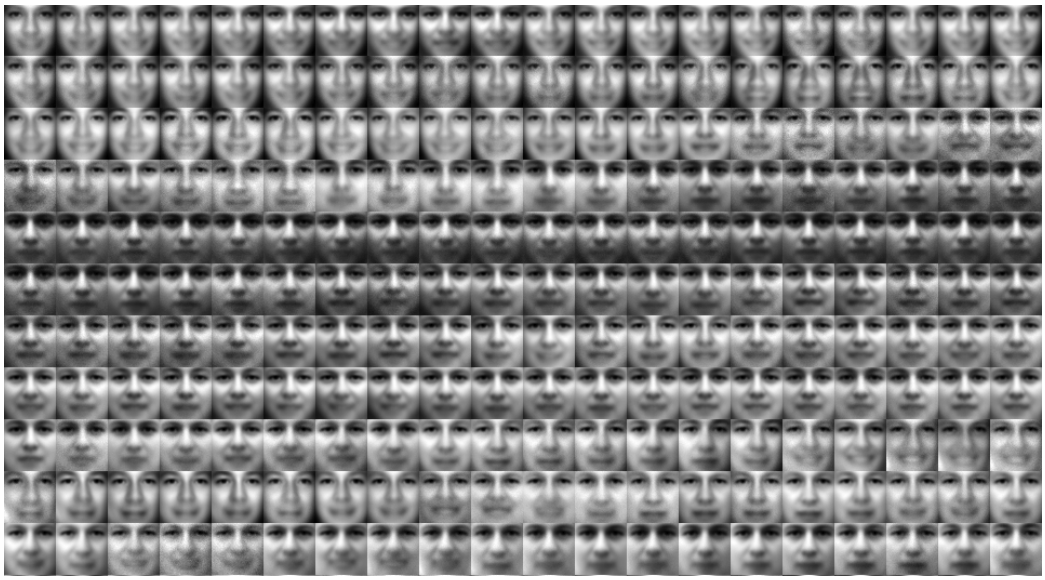

FIGURE 6.4: Consecutive GSN samples from a 3-layer model trained on the TFD dataset.

of 0.5 were selected to approximately minimize the reconstruction negative log-likelihood. The learning rate is reduced multiplicatively by 0.99 after each epoch. Following Breuleux *et al.* [21], the quality of the samples was also estimated quantitatively by measuring the log-likelihood of the test set under a Parzen density estimator constructed from 10000 consecutively generated samples (using the real-valued mean-field reconstructions as the training data for the Parzen density estimator). Results are summarized in Table 6.1. The test set Parzen log-likelihood was not used to

select among model architectures, but visual inspection of samples generated did guide the preliminary search reported here. Optimization hyper-parameters (learning rate, momentum, and learning rate reduction schedule) were selected based on the reconstruction log-likelihood training objective. The Parzen log-likelihood obtained with a two-layer model on MNIST is 214 ($\pm$ standard error of 1.1), while the log-likelihood obtained by a single-layer model (regular denoising auto-encoder, DAE in the table) is substantially worse, at -152$\pm$2.2. In comparison, Bengio *et al.* [9] report a log-likelihood of -244$\pm$54 for RBMs and 138$\pm$2 for a 2-hidden layer DBN, using the same setup. We have also evaluated a 3-hidden layer DBM [69], using the weights provided by the author, and obtained a log-likelihood of 32$\pm$2. See `http://www.mit.edu/~rsalakhu/DBM.html` for details. Interestingly, the GSN and the DBN-2 actually perform slightly better than when using samples directly coming from the MNIST training set, maybe becaue they generate more "prototypical" samples (we are using mean-field outputs).

TABLE 6.1: Test set log-likelihood obtained by a Parzen density estimator trained on 10000 generated samples, for different generative models trained on MNIST. A DBN-2 has 2 hidden layers and an DBM-3 has 3 hidden layers. The DAE is basically a GSN-1, with no injection of noise inside the network. The last column uses 10000 MNIST training examples to train the Parzen density estimator.

	GSN-2	DAE	RBM	DBM-3	DBN-2	MNIST
LOG-LIKELIHOOD	214	-152	-244	32	138	24
STANDARD ERROR	1.1	2.2	54	1.9	2.0	1.6

Figure 6.2 shows a single run of consecutive samples from this trained model, illustrating that it mixes quite well (better than RBMs) and produces rather sharp digit images. The figure shows that it can also stochastically complete missing values: the left half of the image was initialized to random pixels and the right side was clamped to an MNIST image. The Markov chain explores plausible variations of the completion according to the trained conditional distribution.

A smaller set of experiments was also run on TFD, yielding a test set Parzen log-likelihood of 1890 $\pm$29. The setup is exactly the same and was not tuned after the MNIST experiments. A DBN-2 yields a Parzen log-likelihood of 1908 $\pm$66, which is undistinguishable statistically, while an RBM yields 604 $\pm$ 15. Consecutive samples from the GSN-3 model are shown in Figure 6.4. Figure 6.3 shows consecutive samples obtained early on during training, after only 5 and 25 epochs respectively, illustrating the fast convergence of the training procedure.

6.9 Conclusion

We have introduced a new approach to training generative models that avoid the potential pitfalls of intractable or approximate inference and sampling in models with many latent variables. We argue that if the true posterior distribution of a latent variable model is highly multimodal, then the current methods for training and using such models could yield very poor results. Motivated by this possibility and the recent success of training deep but supervised neural networks, a new framework for training generative models is introduced, called Generative Stochastic Networks (GSNs). The proposed theoretical results state that if noise is injected in the networks that prevents perfect reconstruction, training them to reconstruct their observations suffices to capture the data generating distribution through a simple Markov chain. One of the theoretical assumptions that may not be guaranteed here (unless the amount of noise is very small, which may hurt mixing) is the ability to capture multimodal reconstruction distributions, and this should be the subject of future investigations. This would show up as more fuzzy reconstructions corresponding to averaging several modes. Although it did not seem to clearly hurt in the case of the reported experiments, it might become important for more complex data sets. The theoretical and empirical contributions also include a demonstration of the ability of GSNs to handle and stochastically reconstruct missing inputs, which could be applied to construct structured output models. The experiments clearly validate the theoretical results: it is possible and simple to train a GSN and sample from it, conditionally or unconditionally, while obtaining samples of comparable quality to those obtained with currently established generative models such as RBMs, DBNs and DBMs. Future work on larger scale problems should investigate whether GSNs are less sensitive to the potentially huge number of significant modes in the space of causes and whether this would actually hurt generative models based on anonymous latent variables.

Acknowledgments

The authors would like to acknowledge the stimulating discussions and help from Vincent Dumoulin, Guillaume Alain, Pascal Vincent, Yao Li, Aaron Courville, and Ian Goodfellow, as well as funding from NSERC, CIFAR (YB is a CIFAR Fellow), and the Canada Research Chairs.

Chapitre 7

Conclusion

7.1 Résumé des contributions

Un premier point important du premier article est la démonstration de performance sur le matchmaking d'un algorithme d'apprentissage (*maxout network*) originellement appliqué à la vision, une tâche de nature très différente. Ceci démontre la flexibilité de l'algorithme, renforçant les motivations déjà en place pour l'étude des réseaux de neurones. Dans un deuxième temps nous motivons l'étude du matchmaking en tant que politique de décision. Ceci est généralisable à plusieurs tâches en apprentissage machine. Souvent des modèles sont confectionnés et entraînés, et la mesure de performance finale est calculée de manière *offline*, c'est-à-dire sur des données déjà collectées. Rappelons-nous qu'un des buts des algorithmes d'apprentissage est leur utilisation dans des systèmes en temps réel. Ceci implique donc l'étude de leur performance au-delà de l'erreur de classification et la performance de régression. Dans notre cas, une approche par simulateur est justifiable par la difficulté de l'application des techniques classiques d'évaluation de politiques au problème de matchmaking en groupe. Nous rencontrons tout de même un problème de biais similaire à l'application de la méthode directe (4.4) pour l'évaluation de la valeur d'une politique. En ajoutant une régularisation additionelle à l'estimateur utilisé par la politique de décision, nos expériences démontrent que cette fonction légèrement modifiée peut servir d'estimateur raisonnable de la valeur de la politique.

Les contributions du deuxième article sont plus théoriques, et font partis du début d'une littérature qui motive l'interprétation des autoencodeurs comme modèles captant la distribution des données.

Des arguments sont présentés pour justifier l'insuffisance potentielle des modèles à variables latentes pour des tâches complexes. Ceux-ci font généralement des suppositions sur la distribution des variables latentes $\mathbf{h}$: soit la distribution est approximativement unimodale, conditionellement indépendante (les variables latentes sont indépendantes donné $\mathbf{x}$) ou possède peu de modes. De plus, la vitesse de changement de mode lors de l'échantillonage est un élément important de ces algorithmes d'apprentissage, qui utilisent l'échantillonage lors de l'entraînement. Sans changement de mode rapide, l'entraînement souffre car il faut visiter les modes probables pour que l'algorithme d'apprentissage modifie la distribution des variables latentes. Le résultat théorique principal nous indique qu'un processus de corruption et un estimateur consistant de la reconstruction des données donnent lieu à une chaîne de Markov qui converge asymptotiquement vers la distribution des données. Pour démontrer empiriquement ce résultat, nous avons implémenté un modèle avec une paramétrisation similaire à celle des machines de boltzmann profondes (DBM).

Nous démontrons d'abord qualitativement que le modèle ainsi entraîné capte la distribution des données. Celui-ci peut débruiter les données, reconstruire une portion manquante de l'entrée et générer des échantillons similaires aux données d'entraînement. Nous avons vérifié quantitativement la qualité de ces échantillons en entraînant un estimateur de densité (fenêtres de Parzen) sur les échantillons générés, pour ensuite vérifier la vraisemblance de l'ensemble de test d'après cet estimateur. Ceci nous donne une mesure de la qualité des échantillons, meilleure que celle obtenue par une DBM ou un DBN.

7.2 Directions futures

Étant plutôt appliqué, le premier article laisse des pistes pour une analyse plus poussée sur plusieurs points. Parmi celles-ci, l'étude du lien entre la performance d'un algorithme d'apprentissage et son utilisation en tant que politique de décision ressort. Dans notre contexte, nous suspectons que les gains de performance *offline* sont fortement corrélés avec la valeur de la politique de décision, c'est-à-dire la balance des matchs. Cependant, pour d'autres critère de performance, il existe peut-être une relation plus complexe entre la performance *offline* et la valeur en tant que politique de décision. Par exemple, nous avons observé cet effet dans les travaux précurseurs, où la valeur de la politique était l'amusement moyen des joueurs. Il s'est avéré qu'un modèle de régression plus performant n'était pas nécessairement meilleur en tant que politique de décision. La méthode par simulation permettrait d'investiguer ce phénomène. Dans le même ordre d'idée, des variations aux paramètres du simulateur permettraient aussi une analyse plus poussée des différentes politiques de décision. Entre autre, la méthode de sélection des matchs pourrait être altérée. L'échantillonnage

aléatoire des matchs étant un point de départ raisonnable, une approche gloutone ou diviser-pour-régner pourrait offrir de meilleurs résultats. Encore une fois, ceci dépend des paramètres de contrôle du simulateur ainsi que le critère de performance. Finalement, une analyse théorique approfondie aiderait à justifier l'utilisation de la *stacked calibration* comme méthode de réduction de variance.

À l'opposé, le deuxième article gagnerait à avoir plus de résultats empiriques. Une des applications principales des algorithmes non-supervisés comme les autoencodeurs et les RBMs est l'initialisation de réseaux profonds. Il est donc naturel de se demander si les GSN possèdent aussi la capacité de pré-entraîner des architectures profondes. Ceci est d'un intérêt particulier si un GSN profond peut être entraîné pour bien approximer la distribution des données. D'un autre côté, des méthodes de quantification de la qualité des échantillons produits par les modèles génératifs permettraient une comparaison plus révélatrice de ceux-ci. La méthode par fenêtre de Parzen est un bon début mais n'est pas sans défauts, car celle-ci semble favoriser des échantillons plus "prototypiques" qu'uniques.

Malgré l'apparence disjointe des deux travaux présentés dans ce mémoire, il y a une intersection. Les modèles génératifs pourraient s'avérer d'une grande utilité en simulation lorsque les données sont difficiles à modéliser avec des lois de probabilités simples. Par exemple, le simulateur pourrait utiliser un générateur de profil de joueurs, qui échantillonnerait des profils suivant la distribution des joueurs. Pour des données encore plus complexes, l'utilisation de modèles génératifs permettrait d'effectuer des simulations auparavant impossibles.

Bibliography

[1] Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In *International Conference on Learning Representations (ICLR'2013)*.

[2] Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. E. (2002). The nonstochastic multi-armed bandit problem. *SIAM J. Computing*, **32**(1), 48—77.

[3] Behnke, S. (2001). Learning iterative image reconstruction in the neural abstraction pyramid. *Int. J. Computational Intelligence and Applications*, **1**(4), 427–438.

[4] Bengio, Y. (2008). Neural net language models. *Scholarpedia*, **3**(1).

[5] Bengio, Y. (2009). *Learning deep architectures for AI*. Now Publishers.

[6] Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*, **abs/1206.5533**.

[7] Bengio, Y. (2013). Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal.

[8] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS'2006*.

[9] Bengio, Y., Mesnil, G., Dauphin, Y., and Rifai, S. (2013a). Better mixing via deep representations. In *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*. ACM.

[10] Bengio, Y., Li, Y., Alain, G., and Vincent, P. (2013b). Generalized denoising auto-encoders as generative models. Technical Report arXiv:1305.6663, Universite de Montreal.

[11] Bengio, Y., Courville, A., and Vincent, P. (2013c). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**(8), 1798–1828.

[12] Bengio, Y., Courville, A., and Vincent, P. (2013d). Unsupervised feature learning and deep learning: A review and new perspectives. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*.

[13] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, **13**, 281–305.

[14] Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.

[15] Bordes, A., Glorot, X., Weston, J., and Bengio, Y. (2013). A semantic matching energy function for learning with multi-relational data. *Machine Learning: Special Issue on Learning Semantics*.

[16] Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *ICML'2012*.

[17] Bradley, R. A. and Terry, M. E. (1952). The rank analysis of incomplete block designs — I. The Method of Paired Comparisons. *Biometrika*, **39**, 324–345.

[18] Breiman, L. (1996). Bagging predictors. *Mach. Learn.*, **24**(2), 123–140.

[19] Breiman, L. (2001). Random forests. *Machine Learning*, **45**(1), 5–32.

[20] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA.

[21] Breuleux, O., Bengio, Y., and Vincent, P. (2011). Quickly generating representative samples from an RBM-derived process. *Neural Computation*, **23**(8), 2053–2073.

[22] Caruana, R. (1997). Multitask learning. *Machine Learning*, **28**, 41–75.

[23] Chapelle, O. and Li, L. (2011). An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems 24 (NIPS'11)*.

[24] Ciresan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. Technical report, arXiv:1202.2745.

[25] Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, **abs/1003.0358**.

[26] Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML'2008*.

[27] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, **12**, 2493–2537.

[28] Cowley, B., Charles, D., Black, M., and Hickey, R. (2008). Toward an understanding of flow in video games. *Computers in Entertainment*, **6**(2), 20:1–20:27.

[29] Dahl, G. E., Ranzato, M., Mohamed, A., and Hinton, G. E. (2010). Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS'2010*.

[30] Dahl, G. E., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, **20**(1), 33–42.

[31] Delalleau, O., Contal, E., Thibodeau-Laufer, E., Chandias Ferrari, R., Bengio, Y., and Zhang, F. (2012). Beyond skill rating: Advanced matchmaking in ghost recon online. *IEEE Transactions on Computational Intelligence and AI in Games*, **4**(3), 167–177.

[32] Deng, L., Seltzer, M., Yu, D., Acero, A., Mohamed, A., and Hinton, G. (2010). Binary coding of speech spectrograms using a deep auto-encoder. In *Interspeech 2010*, Makuhari, Chiba, Japan.

[33] Dudik, M., Langford, J., and Li, L. (2011). Doubly robust policy evaluation and learning. In *Proceedings of the 28th International Conference on Machine learning*, ICML '11.

[34] Elo, A. E. (1978). *The rating of chessplayers, past and present*. Batsford.

[35] Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.*, **11**, 625–660.

[36] Friedman, J. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, **29**, 1180.

[37] Fürnkranz, J. (2007). Recent advances in machine learning and game playing. *ÖGAI Journal*, **26**(2), 19–28.

[38] Glickman, M. E. (1999). Parameter estimation in large dynamic paired comparison experiments. *Applied Statistics*, **48**, 377–394.

[39] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks.

[40] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*.

[41] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. Technical Report arXiv:1302.4389, Université de Montréal.

[42] Gutmann, M. and Hyvarinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS'2010*.

[43] Hamel, P., Lemieux, S., Bengio, Y., and Eck, D. (2011). Temporal pooling and multiscale learning for automatic annotation and ranking of music audio. In *ISMIR*.

[44] Harkness, K. (1967). *Official Chess Handbook*. McKay.

[45] Herbrich, R., Minka, T., and Graepel, T. (2007). Trueskill^{TM}: A bayesian skill rating system. *Advances in Neural Information Processing Systems*, **19**, 569.

[46] Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012a). Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, **29**(6), 82–97.

[47] Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, **18**, 1527–1554.

[48] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012b). Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580.

[49] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Netw.*, **2**(5), 359–366.

[50] Huang, T.-K., Lin, C.-J., and Weng, R. C. (2005). A generalized Bradley-Terry model: From group competition to individual skill. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 601–608. MIT Press, Cambridge, MA.

[51] Jimenez-Rodriguez, J., Jimenez-Diaz, G., and Diaz-Agudo, B. (2011). Matchmaking and case-based recommendations. In *Workshop on Case-Based Reasoning for Computer Games, 19th International Conference on Case Based Reasoning*.

[52] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*.

[53] Kulesza, A. and Pereira, F. (2008). Structured learning with approximate inference. In *NIPS'2007*.

[54] Langford, J. and Zhang, T. (2008). The epoch-greedy algorithm for contextual multi-armed bandits. In *NIPS'2008*, pages 1096—1103.

[55] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag.

[56] LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M.-A., and Huang, F.-J. (2006). A tutorial on energy-based learning. In G. Bakir, T. Hofman, B. Scholkopf, A. Smola, and B. Taskar, editors, *Predicting Structured Data*, pages 191–246. MIT Press.

[57] Lee, H., Battle, A., Raina, R., and Ng, A. (2007). Efficient sparse coding algorithms. In *NIPS'06*, pages 801–808. MIT Press.

[58] Lewicki, M. S., Sejnowski, T. J., and Hughes, H. (1998). Learning overcomplete representations. *Neural Computation*, **12**, 337–365.

[59] Menke, J. E. and Martinez, T. R. (2008). A Bradley-Terry artificial neural network model for individual ratings in group competitions. *Neural Computing and Applications*, **17**, 175–186.

[60] Mohamed, A., Dahl, G., and Hinton, G. (2012). Acoustic modeling using deep belief networks. *IEEE Trans. on Audio, Speech and Language Processing*, **20**(1), 14–22.

[61] Murphy, K. P. (2012). *Machine Learning: a Probabilistic Perspective*. MIT Press, Cambridge, MA, USA.

[62] Precup, D., Sutton, R. S., and Singh, S. (2000). Eligibility traces for off-policy policy evaluation. In *ICML'2000*, pages 759–766.

[63] Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007). Efficient learning of sparse representations with an energy-based model. In *NIPS'2006*.

[64] Riegelsberger, J., Counts, S., Farnham, S., and Philips, B. C. (2007). Personality matters: Incorporating detailed user attributes and preferences into the matchmaking process. In *HICSS*, page 87. IEEE Computer Society.

[65] Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011a). Contracting auto-encoders: Explicit invariance during feature extraction.

[66] Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., and Muller, X. (2011b). The manifold tangent classifier. In *NIPS'2011*.

[67] Rifai, S., Bengio, Y., Dauphin, Y., and Vincent, P. (2012). A generative process for sampling contractive auto-encoders. In *Proceedings of the Twenty-nine International Conference on Machine Learning (ICML'12)*. ACM.

[68] Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, **323**(9), 533.

[69] Salakhutdinov, R. and Hinton, G. E. (2009). Deep Boltzmann machines. In *AISTATS'2009*, pages 448–455.

[70] Saul, L. K. and Jordan, M. I. (1996). Exploiting tractable substructures in intractable networks. In *NIPS'95*. MIT Press, Cambridge, MA.

[71] Savard, F. (2011). *Réseaux de neurones à relaxation entraînés par critère d'autoencodeur débruitant*. Master's thesis, U. Montréal.

[72] Seide, F., Li, G., and Yu, D. (2011). Conversational speech transcription using context-dependent deep neural networks. In *Interspeech 2011*, pages 437–440.

[73] Seung, S. H. (1998). Learning continuous attractors in recurrent networks. In *NIPS'97*, pages 654–660. MIT Press.

[74] Stroock, D. (2011). *Probability Theory: An Analytic View*. Cambridge University Press.

[75] Sweetser, P. and Wyeth, P. (2005). Gameflow: A model for evaluating player enjoyment in games. *Comput. Entertain.*, **3**(3), 3–3.

[76] van den Herik, H. J., Donkers, H., and Spronck, P. H. (2005). Opponent modelling and commercial games. *Proceedings of IEEE*, pages 15–25.

[77] Vapnik, V. (1999). An overview of statistical learning theory. *IEEE Transactions on Neural Networks*, **10**(5), 988–999.

[78] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *ICML 2008*.

[79] Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, **5**, 241–249.

[80] Yamins, D., Tax, D., and Bergstra, J. S. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. **28**(1), 115–123.

[81] Yannakakis, G. and Togelius, J. (2011). Experience-driven procedural content generation. *Affective Computing, IEEE Transactions on*, **2**(3), 147–161.

[82] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **67**(2), 301–320.