RÉSEAUX DE NEURONES GÉNÉRATIFS AVEC STRUCTURE

par

Marc-Alexandre Côté

Thèse présentée au Département d'informatique
en vue de l'obtention du grade de philsophiæ doctor (Ph.D.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 28 avril 2017

Le 28 avril 2017

le jury a accepté la thèse de Monsieur Marc-Alexandre Côté dans sa version finale.

Membres du jury

Professeur Hugo Larochelle
Directeur de recherche
Département d'informatique

Professeur Maxime Descoteaux
Codirecteur de recherche
Département d'informatique

Professeur Aaron Courville
Évaluateur externe
Département d'informatique et de recherche opérationnelle
Université de Montréal

Professeur Pierre-Marc Jodoin
Évaluateur interne
Département d'informatique

Professeur Jean-Pierre Dussault
Président-rapporteur
Département d'informatique

Sommaire

Cette thèse porte sur les modèles génératifs en apprentissage automatique. Deux nouveaux modèles basés sur les réseaux de neurones y sont proposés. Le premier modèle possède une représentation interne où une certaine structure a été imposée afin d'ordonner les caractéristiques apprises. Le deuxième modèle parvient à exploiter la structure topologique des données observées, et d'en tenir compte lors de la phase générative.

Cette thèse présente également une des premières applications de l'apprentissage automatique au problème de la tractographie du cerveau. Pour ce faire, un réseau de neurones récurrent est appliqué à des données de diffusion afin d'obtenir une représentation des fibres de la matière blanche sous forme de séquences de points en trois dimensions.

Mots-clés: apprentissage automatique, réseaux de neurones, infinite restricted boltzmann machine, convolutional neural autoregressive destribution estimator, tractographie, neuroimagerie, gated recurrent unit

Remerciements

Au cours de mon doctorat, j'ai eu la chance de côtoyer de nombreuses personnes. Il est venu le temps de les remercier de m'avoir épaulé lors de cette aventure.

Je tiens d'abord à remercier mes directeurs, Hugo Larochelle et Maxime Descoteaux. Merci pour vos précieux conseils et votre jugement éclairé. Cela m'a souvent permis de voir les différents problèmes sous un oeil nouveau. Merci à Hugo de m'avoir initié à l'apprentissage automatique et d'avoir développé mon bagage mathématique. Merci à Maxime de m'avoir initié aux problématiques liées à l'imagerie cérébrale et d'avoir guidé mon parcours de chercheur depuis le début de ma maitrise.

Je suis ravi des joyeux moments passés en compagnie de mes collègues du SMART : Stan, M'mad, Adam, Phil et Thieu. Sans oublier ceux du SCIL : Gab, Mic, Elef, JC, Max C., François, Jasmeen et Gauvin. Merci aussi aux nombreux collègues du RECSUS pour ces midis en si agréable compagnie. Merci aux gars des *Balls of Steel* pour les matchs de hockey qui m'ont permis de faire sortir le méchant et de changer littéralement le mal de place. Je tiens également à remercier mes amis de mon patelin avec qui j'ai grandi : Vincent, Philippe, Bruno, Dan, JM, Nico et Lamothe. Merci pour votre amitié qui perdure malgré mon séjour à Sherbrooke d'une dizaine d'années.

Merci à mes parents Nicole et Romain d'avoir cru en moi et de m'avoir aidé financièrement afin que je puisse me concentrer pleinement sur mes études. Merci à mes deux soeurs Marième et Stéphanie d'être là pour moi, je sais que je pourrai toujours compter sur vous.

Je souhaite remercier tout spécialement Catherine qui a su me soutenir moralement tout au long de mon baccalauréat, ma maitrise et mon doctorat. Merci de ta compréhension et les encouragements lors de ces nombreuses journées, weekends et soirées (nuits !) de travail. Je me sens privilégié d'avoir dans ma vie une femme si exceptionnelle. À tes côtés, je suis enfin prêt à prendre place dans l'autobus de la vie !

Merci à tous, ce fut une expérience inoubliable.

Marc

Notations

$\boldsymbol{x}$	Lettre minuscule en gras désigne un vecteur colonne.
$\boldsymbol{M}$	Lettre majuscule en gras désigne une matrice ou un tenseur.
x_i	Le i^e élément du vecteur $\boldsymbol{x}$.
$m_{i,j}$	Élément de la matrice $\boldsymbol{M}$ associé à la i^e rangée et j^e colonne.
$f(\boldsymbol{x})$	Fonction $f : \mathbb{R}^n \to \mathbb{R}$ prenant un vecteur à n dimensions et retournant un scalaire.
$\boldsymbol{f}(\boldsymbol{x})$	Vecteur formé par l'application de la fonction $f : \mathbb{R}^n \to \mathbb{R}^n$ à chaque élément du vecteur $\boldsymbol{x}$.
$\otimes$	Produit tensoriel.
$\odot$	Produit matriciel de Hadamard.
$*$	Opération de convolution sans remplissage.
$\star$	Opération de corrélation croisée sans remplissage.
$\circledast$	Opération de convolution avec remplissage de zéros.
$\circledstar$	Opération de corrélation croisée avec remplissage de zéros.
$(\boldsymbol{s} * \boldsymbol{k})_n$	n^{e} élément du vecteur résultant de la convolution entre $\boldsymbol{s}$ et $\boldsymbol{k}$.
$\boldsymbol{e}_H$	Vecteur à H dimensons contenant que des 1.
$\mathbf{flip}(\boldsymbol{x})$	Vecteur $\boldsymbol{x}$ retourné, c'est-à-dire $\left[x_{\lvert\boldsymbol{x}\rvert}, \cdots, x_1\right]^\top$.
$\boldsymbol{h}^{(l)}$	Vecteur associé à la l^{e} couche d'un réseau de neurones.
$p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$	Modélisation, paramétrisée par $\boldsymbol{\theta}$, de la probabilité conditionelle de $\boldsymbol{y}$ étant donné $\boldsymbol{x}$.
$p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$	Modélisation, paramétrisée par $\boldsymbol{\theta}$, de la probabilité jointe entre $\boldsymbol{x}$ et $\boldsymbol{y}$.
$\mathcal{D}$	Ensemble de données.

Notations

$\boldsymbol{\theta}$	Paramètres du modèle.
$f_{\boldsymbol{\theta}}(\boldsymbol{x})$	Fonction paramétrisée par $\boldsymbol{\theta}$.
$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$	Fonction de perte à minimiser.
$(\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}))$	Gradient de la fonction $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ par rapport à $\boldsymbol{\theta}$.
$\boldsymbol{x} \leftarrow \boldsymbol{y}$	Assignation des valeurs de $\boldsymbol{x}$ aux valeurs de $\boldsymbol{y}$.
$\sigma(x)$	Fonction sigmoïde (ou fonction logistique).
$\text{soft}_+(x)$	Fonction *softplus*, une approximation lisse de la fonction rampe.

Abréviations

RBM *Restricted Boltzmann Machine*

oRBM *Ordered Restricted Boltzmann Machine*

iRBM *Infinite Restricted Boltzmann Machine*

NADE *Neural Autoregressive Density Estimator*

RNADE *Real valued Neural Autoregressive Density Estimator*

MoG *Mixture of Gaussians*

MoL *Mixture of Laplace distributions*

SAS *sinh-arcsinh distribution*

RNN *Recurrent Neural Network*

GRU *Gated Recurrent Unit*

NLL *Negative Log Likelihood*

SVM *Support Vector Machine*

PCA *Principal Component Analysis*

ICA *Independent Component Analysis*

Table des matières

Liste des figures

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction

Avec l'ère numérique déjà bien entamée, l'être humain se retrouve vite noyé dans un océan d'information. Cette information se retrouve sous plusieurs formes : documents, images, vidéos, sons, etc. Face à ces données massives (*big data*), il devient inconcevable d'y extirper manuellement toute l'information qui nous intéresse. Même en disposant d'outils et de matériels à la fine pointe de la technologie, il est difficile d'y arriver. Au cours de la dernière décennie, l'apprentissage automatique (*machine learning*) est devenu l'outil de choix pour atteindre cet objectif. En effet, cette branche de l'intelligence artificielle permet de développer des systèmes/modèles pouvant apprendre, à partir d'exemples, à extraire les caractéristiques importantes des données.

L'apprentissage automatique a permis plusieurs avancées notamment dans les domaines de la reconnaissance de la parole [Hinton et al., 2012a; Graves et al., 2013], reconnaissance d'images [Ranzato et al., 2006; Krizhevsky et al., 2012], reconnaissance d'objets [Hinton et al., 2012b] et du traitement du langage naturel [Mikolov et al., 2011]. Le secret de ces modèles réside dans l'utilisation de caractéristiques *apprises* à partir des données plutôt que d'utiliser celles extraites via des techniques conçues à la main.

Le succès qu'a connu l'apprentissage automatique au cours des dernières années portait surtout sur des problèmes de classification. Les modèles qui ont permis ces avancées sont de nature prédictive et se concentrent sur l'apprentissage de caractéristiques qui facilitent la séparation de données provenant de différentes classes. Cet apprentissage se fait généralement à l'aide d'observations étiquetées. Or, il existe une seconde famille de modèles qui tentent plutôt d'apprendre la structure inhérente aux données sans *a priori*. Une fois entrainés, ils sont en mesure de générer des données similaires à celles vues durant l'entrainement. Ces modèles dits génératifs ne nécessitent pas que les données d'entrainement soient étiquetées. Récemment, les réseaux adversariels [Goodfellow et al., 2014] et autorégressifs [van den Oord et al., 2016b] ont montré qu'ils peuvent générer des images, somme toute, réalistes compte tenu de la

difficulté de la tâche.

Le développement d'algorithmes capables de comprendre le monde dans lequel on vit, sans supervision, simplement à partir d'observations est un enjeu important dans le domaine de l'intelligence artificielle. Les modèles génératifs offrent certainement une avenue prometteuse, et il semble tout indiqué de poursuivre des travaux de recherche dans ce sens.

Dans cette optique, cette thèse se veut un apport au domaine de l'apprentissage automatique. Elle propose deux modèles génératifs innovateurs basés sur les réseaux de neurones artificiels. Elle permet également la rencontre entre la neuroimagerie et l'intelligence artificielle où, de façon originale, un réseau de neurones artificiels est utilisé comme outil permettant d'analyser la structure neuronale d'un cerveau humain.

Cette thèse est divisée en cinq chapitres. Le chapitre 1 met en contexte le domaine de l'apprentissage automatique. Le chapitre 2 explique les réseaux de neurones ainsi que trois types d'architecture les plus employés : le réseau multicouche, le réseau à convolution et le réseau récurrent. Le chapitre 3 traite d'un modèle capable d'apprendre la taille de sa couche cachée (Infinite RBM). Le chapitre 4 porte sur un modèle autorégressif utilisé pour l'estimation de densité de probabilité. Le chapitre 5 concerne l'utilisation d'un réseau de neurones récurrent pour faire la tractographie des fibres neuronales. Finalement, l'annexe A liste les contributions reliées à cette thèse.

Chapitre 1

Apprentissage automatique

“ Give computers the ability to learn without being explicitly programmed.

Arthur Lee Samuel, 1959 ”

L'**apprentissage automatique** (*Machine Learning*) est une branche de l'intelligence artificielle. Elle étudie les algorithmes qui permettent à un **agent** intelligent d'apprendre à résoudre une tâche à partir d'exemples. Cette branche regroupe plusieurs sous-domaines de l'informatique et des mathématiques appliquées tels que la théorie de l'information, l'optimisation numérique et les statistiques.

Les **algorithmes d'apprentissage** automatique ne reposent pas sur un ensemble de règles d'inférence établies *manuellement* par un humain, contrairement aux systèmes experts [Russell and Norvig, 2010, chap. 16]. En effet, lors de la phase d'apprentissage, l'agent a la capacité de concevoir ses propres règles à partir d'exemples observés dans le but d'accomplir la tâche demandée. Par règle, on entend un ensemble de caractéristiques extraites à partir des données brutes (ex. les contours dans une image).

Dans cette thèse, on s'intéressera plus particulièrement à l'**apprentissage profond** (*Deep Learning*). Il s'agit d'un sous-ensemble de méthodes d'apprentissage automatique ayant la capacité d'extraire une hiérarchie de règles dans laquelle les règles complexes ou abstraites dépendent des plus simples. À titre de comparaison, prenons la détection d'un visage dans une image. Sans hiérarchie, l'agent doit apprendre directement la relation pixels→visage. Alors qu'avec l'apprentissage profond, l'agent a la capacité d'y aller en étape, c'est-à-dire apprendre d'abord les relations pixels→nez, pixels→yeux et pixels→bouche, et ensuite la relation (nez+2·yeux+bouche)→visage.

L'apprentissage automatique, propulsé par l'apprentissage profond, connait depuis 2006 son plus grand essor jusqu'à présent. Contrairement à ce que l'on pourrait penser, l'apprentissage automatique n'est pas un domaine de recherche récent dans l'histoire

de l'informatique. Déjà dans les années 40–60, des travaux de recherche en cybernétique portaient sur l'apprentissage profond [Goodfellow et al., 2016, chap. 1.2.1].

Le domaine de l'apprentissage automatique possède de nombreux champs de recherche tout aussi actifs les uns que les autres, produisant une multitude d'algorithmes chaque année. Afin de se repérer dans ce capharnaüm, trois taxonomies sont généralement employées pour classifier les algorithmes d'apprentissage : par la tâche qu'ils tentent d'accomplir (section 1.1), par leur mode d'apprentissage (section 1.2) ou par la structure interne de l'agent (section 1.3).

Le processus d'apprentissage d'un agent consiste en fait en la résolution d'un problème d'optimisation (section 1.4). La fonction à optimiser variera en fonction de la tâche choisie, du mode d'apprentissage utilisé et de la structure de l'agent.

1.1 Tâches d'apprentissage

Dans le contexte d'apprentissage automatique, la tâche à accomplir dicte ce que l'agent doit faire lorsqu'on lui présente des données. Plus formellement, si $\boldsymbol{x} \in \mathbb{R}^n$ représente un exemple, l'objectif de l'agent est de modéliser une fonction optimale $f^*(\boldsymbol{x})$ telle que définie par la tâche d'apprentissage.

Il existe de nombreuses tâches et toutes les couvrir sort du cadre de cette thèse. Néanmoins, en voici quelques-unes qui seront utiles pour la compréhension de ce document : la classification (section 1.1.1), la régression (section 1.1.2) et l'estimation de densité (section 1.1.3).

1.1.1 Classification

Pour une tâche de classification, l'objectif est d'attribuer une classe aux données observées. Formellement, il s'agit pour l'agent de modéliser une fonction $y = f^*(\boldsymbol{x})$, où la classe prédite est un entier $y \in \{1, \ldots, C\}$ avec C un nombre fini de classes.

Alternativement, dans un contexte probabiliste, l'objectif correspond plutôt à donner, pour une observation $\boldsymbol{x}$, sa probabilité d'appartenance à chacune des classes possibles.

Ensuite, il s'agit de choisir la classe la plus probable. Formellement, l'agent apprend à modéliser $\boldsymbol{y} = f^*(\boldsymbol{x})$, où le vecteur $\boldsymbol{y} \in [0,1]^C$ contient C probabilités, c'est-à-dire une pour chaque classe, tel que $y_c = p(y = c \,|\, \boldsymbol{x})$. On sélectionne ensuite la classe ayant la plus forte probabilité

$$\underset{c \in \{1,\ldots,C\}}{\operatorname{argmax}} \, y_c. \tag{1.1}$$

La classification est utilisée, entre autres, pour la reconnaissance de caractères manuscrits [Graves and Schmidhuber, 2008], l'identification d'objets dans une image [Krizhevsky et al., 2012], la détection de courriels frauduleux [Tretyakov, 2004].

La tâche de classification peut être apprise par des algorithmes comme l'arbre de décisions [Breiman et al., 1984], les réseaux de neurones [Bishop, 2006; Goodfellow et al., 2016], la machine à vecteurs de support (*support vector machine* – SVM) [Cortes and Vapnik, 1995].

1.1.2 Régression

Pour une tâche de régression, l'objectif est de prédire une valeur continue (ex. un prix, une température, une distance, etc.) à partir d'une observation $\boldsymbol{x}$. Formellement, il s'agit pour l'agent de modéliser une fonction $y = f^*(\boldsymbol{x})$, où la prédiction $y \in \mathbb{R}$ n'est pas nécessairement un entier fini, contrairement à la tâche de classification.

Un exemple de régression pourrait être de déterminer l'évaluation municipale d'une maison en fonction des caractéristiques de l'habitation (ex. superficie, quartier, année de construction, etc.). Au chapitre 5, nous verrons comment la modélisation de fibres neuronales peut être formulée comme une tâche de régression.

La tâche de régression peut être apprise par des algorithmes comme la régression linéaire [Murphy, 2012, chap. 7], l'arbre de décisions [Breiman et al., 1984] et les réseaux de neurones [Bishop, 2006; Goodfellow et al., 2016].

1.1.3 Estimation de densité de probabilité

Pour une tâche d'estimation de densité de probabilité, l'objectif est de modéliser explicitement la distribution $p_{\mathcal{D}}(\boldsymbol{x})$ qui a généré les observations $\boldsymbol{x}$. Formellement, cela correspond à modéliser $p(\boldsymbol{x}) = f^*(\boldsymbol{x})$ qui représente soit la densité de probabilité (variables continues) ou bien la fonction de masse (variables discrètes) de la distribution.

Être en mesure de calculer la vraisemblance d'une observation offre un large éventail de possibilités en matière d'applications. À titre d'exemple, supposons que l'on présente à l'agent une image détériorée (bruit, occlusions, etc.), il sera en mesure d'apporter les correctifs nécessaires afin de restaurer l'image. Au chapitre 4, nous verrons comment un réseau de neurones parvient à résoudre ce type de tâche.

La tâche d'estimation de densité de probabilité peut être apprise par des algorithmes comme la mélange de Bernoulli [Murphy, 2012, chap. 11], le *Fully Visible Sigmoid Belief Network* [Neal, 1992] et le *Neural Autoregressive Distribution Estimator* [Larochelle and Murray, 2011].

1.2 Mode d'apprentissage

Les algorithmes peuvent être regroupés selon leur mode d'apprentissage. Cette catégorisation dépend de la nature des données comprises dans l'ensemble d'entrainement $\mathcal{D}$. Le choix d'un mode d'apprentissage influence le type de tâches pouvant être accomplies et la manière dont l'agent traite les données.

Il existe plusieurs modes d'apprentissage. Par souci de concision, seulement ceux utiles à la compréhension de ce document seront abordés : l'apprentissage supervisé (section 1.2.1) et l'apprentissage non supervisé (section 1.2.2).

1.2.1 Apprentissage supervisé

Lors d'apprentissage supervisé, l'agent reçoit l'observation accompagnée de la réponse que l'on s'attend à ce qu'il produise. Formellement, l'ensemble d'entrainement $\mathcal{D}$ est composé d'exemples $(\boldsymbol{x}, \boldsymbol{y})$ où $\boldsymbol{x} \in \mathbb{R}^n$ est l'observation et $\boldsymbol{y} \in \mathbb{R}^m$ est la cible. On dit alors que les données sont **étiquetées**.

L'avantage de l'apprentissage supervisé est d'avoir un objectif précis à optimiser : prédire la cible. Cet objectif permet d'évaluer précisément la performance de l'agent (ex. pourcentage d'erreurs). L'inconvénient est qu'il y a très peu de données étiquetées et les obtenir est souvent couteux en temps et en argent. À titre d'exemple, dans le domaine de la neuroimagerie, il faut faire appel à un médecin pour obtenir les étiquettes et le monopoliser coûte cher.

D'ordre général, l'étiquetage est fait par un humain, c'est-à-dire un expert pour le type de données étudiées. Bien que ce soit un processus long et laborieux, on peut rarement faire autrement, à moins d'avoir prévu le coup lors de l'acquisition des données. Par exemple, dans le domaine de la vision par ordinateur, on peut exploiter les moteurs de jeu (*games engines*) pour générer des données d'entrainement étiquetées. Gaidon et al. [2016] utilise cette approche pour le suivi d'objets dans un environnement 3D.

L'apprentissage supervisé est utilisé, entre autres, pour faire de la classification ou de la régression. Ce mode d'apprentissage est d'ailleurs utilisé au chapitre 5 pour la modélisation de fibres neuronales. Parmi les algorithmes considérés supervisés, on retrouve l'arbre de décisions [Rokach, 2008], les réseaux de neurones [Bishop, 2006; Goodfellow et al., 2016], le classificateur bayésien [Murphy, 2012, chap. 10] et la SVM [Cortes and Vapnik, 1995].

1.2.2 Apprentissage non supervisé

Lors d'apprentissage non supervisé, l'agent n'observe jamais la réponse attendue. Ce mode d'apprentissage ne requiert donc pas que les données soient étiquetées. Sans cible, il doit lui-même découvrir la structure latente des données à partir des observations qu'il reçoit. Formellement, l'ensemble d'entrainement $\mathcal{D}$ est seulement composé

d'exemples $\boldsymbol{x} \in \mathbb{R}^n$.

L'avantage de l'apprentissage non supervisé est la disponibilité des données non étiquetées. En effet, il en existe une immense quantité et celle-ci est sans cesse grandissante. De plus, il est peu couteux de les acquérir puisqu'elles ne requièrent aucun étiquetage. Par contre, il est plus compliqué d'évaluer directement la performance de ces algorithmes puisqu'il n'y a pas de solution unique par rapport à ce qui est attendu de l'agent lorsqu'on lui présente un ensemble d'entrainement.

L'apprentissage non supervisé est utilisé, entre autres, pour faire de l'estimation de densité de probabilité (voir chapitre 4), du partitionnement de données (*clustering*), de l'extraction de caractéristiques (*features extraction)* et de la réduction de dimensionnalité [Tan et al., 2005; Russell and Norvig, 2010; Murphy, 2012]. Parmi les algorithmes considérés non supervisés, on retrouve le codage parcimonieux (*Sparse Coding*) [Mairal et al., 2008], les autoencodeurs [Bengio, 2009] [Goodfellow et al., 2016, chap. 14], les machines restreintes de Boltzmann (*Restricted Boltzmann Machine* – RBM) [Smolensky, 1986; Hinton, 2002], l'analyse en composantes principales (*Principal Component Analysis* – PCA) [Murphy, 2012, chap. 12.2] et l'analyse en composantes indépendantes (*Independant Component Analysis* – ICA) [Murphy, 2012, chap. 12.6].

1.3 Modèles d'apprentissage

Jusqu'à présent, aucune supposition n'a été faite sur la structure interne de l'agent ni sur la manière de l'entrainer. Ce que l'on nomme agent est en fait un modèle mathématique. Chaque modèle possède un ensemble de variables qui sont optimisées en fonction de la tâche d'apprentissage. Ces variables sont communément nommées **paramètres** et représentées par le vecteur $\boldsymbol{\theta}$. Les paramètres d'un modèle dictent comment les données observées influencent les prédictions du modèle, c'est-à-dire les décisions prises par l'agent. De façon générale, plus la structure des données est complexe, plus la **complexité** d'un modèle devra être élevée afin de bien performer, c'est-à-dire un modèle possèdant plus de paramètres.

La majorité des modèles possèdent également des **hyperparamètres** qui modifient leur **capacité d'apprentissage**, par exemple le nombre de paramètres optimisables d'un modèle. Notons qu'en apprentissage automatique, tout paramètre n'étant pas optimisé lors du processus d'apprentissage de la tâche est considéré comme un hyperparamètre. Pour faire un parallèle avec la statistique bayésienne, le terme hyperparamètre désigne un paramètre de la distribution *a priori*. À titre d'exemple, le k dans k-means [Tan et al., 2005, chap. 8.2] et le nombre de composantes dans une mixture de gaussiennes [Murphy, 2012, chap. 11] sont des hyperparamètres. En dépit de l'existence de certaines heuristiques ou de techniques pour apprendre à apprendre (*meta-learning*) [Snoek et al., 2012], l'essai-erreur reste l'approche la plus couramment employée pour trouver de bonnes valeurs à ces hyperparamètres. Généralement, les contraintes d'implémentation limiteront les valeurs que peuvent prendre les hyperparamètres.

Il existe plusieurs familles de modèles. Dans l'optique d'être concis et succinct, cette thèse traite seulement de modèles basés sur les **réseaux de neurones**. Cette famille sera détaillée au chapitre 2, mais d'abord voici une description de deux catégories de modèles : discriminatif (section 1.3.1) et génératif (section 1.3.2).

1.3.1 Modèle discriminatif

Un modèle dit discriminatif sert à représenter la dépendance conditionnelle de variables non observées $\boldsymbol{y}$ par rapport à des variables observées $\boldsymbol{x}$. D'un point de vue probabiliste, cela consiste à modéliser la distribution de probabilités conditionnelles $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ paramétrée par $\boldsymbol{\theta}$.

Les modèles discriminatifs sont privilégiés pour les tâches de classification et de régression en raison de leur nature prédictive, c'est-à-dire qu'ils sont optimisés pour prédire $\boldsymbol{y}$ étant donnée $\boldsymbol{x}$. Parmi les algorithmes considérés discriminatifs, on retrouve la régression logistique [Murphy, 2012, chap 1.4], les forêts d'arbres décisionnels (*Random Forest*) [Tan et al., 2005, chap. 5.6] et les réseaux de neurones [Bishop, 2006; Goodfellow et al., 2016].

1.3.2 Modèle génératif

Un modèle dit génératif sert à représenter les relations conjointes entre des variables observées $\boldsymbol{x}$ et des variables non observées $\boldsymbol{y}$. Cela permet de générer aléatoirement des valeurs pour $\boldsymbol{x}$ et $\boldsymbol{y}$. Dans un cadre probabiliste, cela revient à modéliser la distribution de probabilités jointes $p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ paramétrée par $\boldsymbol{\theta}$, ou de façon équivalente $p(\boldsymbol{x} \mid \boldsymbol{y}; \boldsymbol{\theta})p(\boldsymbol{y}; \boldsymbol{\theta})$.

En raison de leur capacité à décrire la structure inhérente aux données, les modèles génératifs sont privilégiés lorsqu'on souhaite générer de nouvelles données similaires à celles observées lors de l'entrainement.

Ajoutons qu'en théorie, il est toujours possible de substituer un modèle génératif à un modèle discriminatif. En effet, à l'aide du théorème de Bayes, on obtient

$$p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta}) = \frac{p(\boldsymbol{x}|\boldsymbol{y}; \boldsymbol{\theta})p(\boldsymbol{y}; \boldsymbol{\theta})}{p(\boldsymbol{x}; \boldsymbol{\theta})} = \frac{p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})}{\sum_{y'} p(\boldsymbol{x}, \boldsymbol{y}'; \boldsymbol{\theta})}.$$

Cela permet de montrer qu'à partir d'une modélisation de $p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$, on parvient à calculer une probabilité conditionnelle. Par contre, la réciproque est fausse. Ayant seulement la modélisation de $p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta})$, il est impossible d'obtenir la probabilité jointe.

En pratique, les modèles génératifs sont peu utilisés pour des tâches de classification ou de régression. Les modèles discriminatifs ont l'avantage de concentrer les ressources computationnelles directement à modéliser les séparations entre les classes. Les modèles génératifs quant à eux cherchent à modéliser aussi bien les caractéristiques individuelles de chaque classe que les caractéristiques partagées par les classes. En terme d'erreurs (classification ou régression), ces derniers s'avèrent souvent moins performants que les modèles discriminatifs lorsque beaucoup de données d'entrainement sont disponibles [Ng and Jordan, 2002]. À l'inverse, les modèles génératifs possèdent un avantage lorsqu'il y a peu de données étiquées disponibles pour l'entrainement.

Parmi les algorithmes considérés génératifs, on retrouve la mixture de gaussiennes [Murphy, 2012, chap. 11], la RBM [Smolensky, 1986; Hinton, 2002], les *Generative Adversa-*

rial Networks (GAN) [Goodfellow et al., 2014] et le *Neural Autoregressive Distribution Estimator* [Larochelle and Murray, 2011] (présenté au chapitre 4).

1.4 Optimisation

Le processus d'apprentissage d'un agent consiste à mettre à jour les paramètres de son modèle pour qu'il devienne meilleur à accomplir la tâche souhaitée. Mathématiquement, cela correspond au problème d'optimisation

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) \tag{1.2}$$

où l'on cherche les paramètres $\boldsymbol{\theta}$ minimisant la **fonction de perte** $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ calculée sur l'ensemble d'entrainement $\mathcal{D}$. La fonction de perte est différentiable et choisie spécifiquement pour la tâche et le mode d'apprentissage auxquels on s'intéresse.

Pour résoudre ce problème d'optimisation, on fait généralement appel à des algorithmes d'optimisation du premier ordre. Ce sont des techniques ne nécessitant que l'information de la première dérivée de $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$, soit le gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$. Malgré leur meilleur taux de convergence, les algorithmes d'optimisation d'ordre supérieur s'avèrent impraticables en raison du calcul couteux des dérivées d'ordre supérieures. Généralement, on utilise l'algorithme de descente du gradient ou une de ses variantes (section 1.4.2).

1.4.1 Fonction de perte

L'estimation du maximum de vraisemblance est une méthode statistique couramment utilisée pour trouver les paramètres d'un modèle statistique à partir d'un échantillon de données. De façon équivalente, on utilisera le négatif de la log-vraisemblance comme fonction de perte qui est plus stable numériquement. C'est-à-dire,

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = -\log p(\mathcal{D}; \boldsymbol{\theta}) \tag{1.3}$$

$$= -\sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}} \log p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) \tag{1.4}$$

où la probabilité jointe $p(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ est obtenue à partir du modèle paramétré par $\boldsymbol{\theta}$. L'équation (1.4) est possible puisque l'on suppose que les éléments de $\mathcal{D}$ ont été générés de façon indépendante et uniformément distribuée. Notons que dans un contexte d'apprentissage non-supervisé on aura $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = -\sum_{\boldsymbol{x} \in \mathcal{D}} \log p(\boldsymbol{x}; \boldsymbol{\theta})$.

Dans le même ordre d'idées, lorsqu'on cherche à minimiser les erreurs de prédiction qu'un modèle commet sur l'ensemble d'entrainement, on utilisera l'estimation du maximum de vraisemblance conditionnelle

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = -\sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}} \log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}), \tag{1.5}$$

où la probabilité $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ est obtenue à partir du modèle paramétré par $\boldsymbol{\theta}$.

Lorsque l'agent ne modélise pas directement une probabilité telle que $p(\boldsymbol{x}; \boldsymbol{\theta})$ ou $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, on peut faire appel aux fonctions de pertes décrites ci-dessus. Minimiser ces fonctions revient à minimiser le négatif de la log-vraisemblance ou le négatif de la log-vraisemblance conditionnelle selon le cas [Goodfellow et al., 2016, chap. 5].

Fonction *softmax*

La *softmax* n'est pas une fonction de perte à proprement dit. En classification, elle offre plutôt une façon d'obtenir $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$. Spécifiquement, elle permet de convertir un vecteur de valeurs quelconques $\boldsymbol{v}$, en un vecteur de valeurs pouvant être interprétées comme des probabilités $\boldsymbol{p}$. Formellement, on la définit par

$$\boldsymbol{p} = \mathbf{softmax}(\boldsymbol{v}) = \left[\frac{e^{v_1}}{\sum_{i=1}^{|\boldsymbol{v}|} e^{v_i}}, \ldots, \frac{e^{v_{|\boldsymbol{v}|}}}{\sum_{i=1}^{|\boldsymbol{v}|} e^{v_i}} \right]^\top \tag{1.6}$$

où $\sum_{i=1}^{|\boldsymbol{p}|} p_i = 1$ et $p_i > 0$.

Supposons que le vecteur de valeurs quelconques $\hat{\boldsymbol{y}}$ est la réponse fournie par l'agent lorsqu'il observe $\boldsymbol{x}$, c'est-à-dire $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$, la fonction de perte correspond alors à

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = -\sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}} \log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}) = -\sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}} \log \mathbf{softmax}(\hat{\boldsymbol{y}})^\top \boldsymbol{y}, \tag{1.7}$$

où $\boldsymbol{y}$ est en fait un vecteur de zéros sauf à la position c, correspondant à la classe cible, où il vaut un.

Le gradient de cette fonction de perte par rapport à la réponse de l'agent est donné par

$$\nabla_{\hat{\boldsymbol{y}}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \mathbf{softmax}(\hat{\boldsymbol{y}}) - \boldsymbol{y}. \tag{1.8}$$

Erreur quadratique moyenne (*Mean Squared Error*)

L'erreur quadratique moyenne représente la moyenne du carré des différences entre deux vecteurs. Bien qu'elle peut être utilisée pour toutes les tâches d'apprentissage vues à la section 1.1, elle est généralement utilisée en régression.

Supposons que $\hat{\boldsymbol{y}}$ est la réponse fournie par l'agent lorsqu'il observe $\boldsymbol{x}$, la fonction de perte correspond alors à

$$\mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \frac{1}{|\boldsymbol{y}|}(\hat{\boldsymbol{y}} - \boldsymbol{y})^\top(\hat{\boldsymbol{y}} - \boldsymbol{y}). \tag{1.9}$$

Le gradient de cette fonction de perte par rapport à la réponse de l'agent est donné par

$$\nabla_{\hat{\boldsymbol{y}}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \frac{2}{|\boldsymbol{y}|}(\hat{\boldsymbol{y}} - \boldsymbol{y}). \tag{1.10}$$

1.4.2 Descente du gradient

Bien qu'il existe des algorithmes d'optimisation de premier ordre plus avancés (ex. quasi-newton, gradient conjugué, etc. [Nocedal and Wright, 2006]), les algorithmes de type descente de gradient restent prédominants dans le domaine. Leur simplicité et leur faible cout en temps de calcul (puisqu'ils ne requièrent pas d'approximer la matrice hessienne, ni de *recherche linéaire*) en font d'eux des algorithmes propices pour l'optimisation de modèles comportant parfois des millions de paramètres tels que les réseaux de neurones.

La descente du gradient consiste à suivre localement la direction descendante ayant la plus forte pente en un point précis. Cette direction est donnée par l'opposé du gradient. Concrètement, il s'agit d'un algorithme itératif permettant de mettre à jour les variables à optimiser. Cela revient à calculer

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) \tag{1.11}$$

où $\alpha \in \mathbb{R}^+$ est un hyperparamètre de l'algorithme d'optimisation contrôlant la longueur du pas de descente.

Bien que ce soit un algorithme d'optimisation de premier ordre assez performant, la descente du gradient a de la difficulté à gérer une grande quantité de données. Son principal inconvénient est qu'il requiert le calcul du gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right)$ à chaque itération. Or, le temps de calcul du gradient augmente avec le nombre d'exemples d'entrainement. Ceci est causé par la présence d'une somme de N termes dans la fonction $\mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right)$, où N est la taille de l'ensemble d'entrainement.

Descente du gradient stochastique

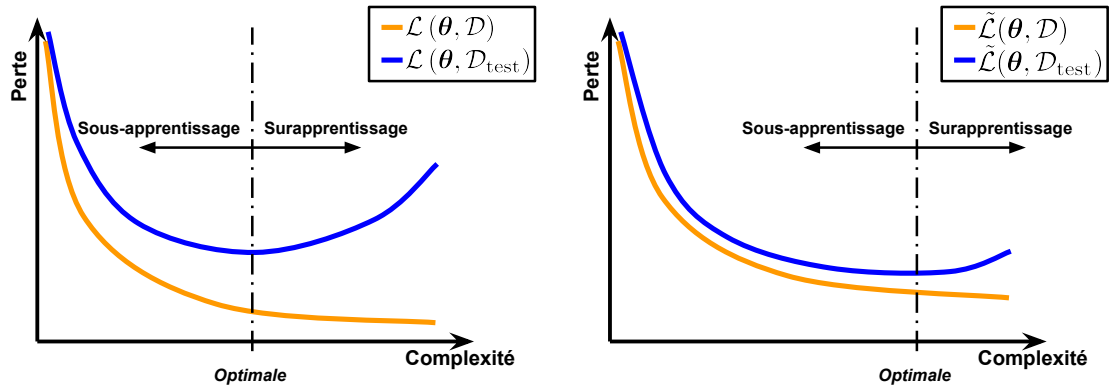
Il est de plus en plus commun d'avoir à sa disposition des dizaines de millions d'exemples pour l'entrainement d'un modèle. Il devient alors peu pratique d'utiliser l'algorithme de descente du gradient.

L'alternative est d'utiliser le gradient obtenu à partir d'un sous-ensemble d'exemples choisis aléatoirement. Ceci correspond à une approximation de la direction de descente telle qu'utilisée par l'algorithme de descente du gradient. Cette approche est connue sous le nom de descente du gradient stochastique (*Stochastic Gradient Descent – SGD*) [Goodfellow et al., 2016, chap. 8]. C'est une technique d'optimisation différente, appliquée au même problème d'optimisation.

Formellement, cela consiste à mettre à jour itérativement les variables à optimiser comme suit

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \mathcal{B}) \tag{1.12}$$

où $\mathcal{B} \subseteq \mathcal{D}$ est un sous-ensemble d'exemples choisis aléatoirement et change à chaque



(a) Sans régularisation (b) Avec régularisation

figure 1.1 – Illustrations montrant l'effet de la régularisation sur le sous-apprentissage et le surapprentissage. a) Sans régularisation, l'écart entre les erreurs d'entrainement et de généralisation a tendance à augmenter en même temps que la complexité du modèle. b) Avec régularisation, l'écart a plutôt tendance à rester petit. Bien qu'on observe que l'erreur d'entrainement soit plus élevée comparativement à celle obtenue en a), l'erreur de généralisation quant à elle est plus basse.

itération. La taille du sous-ensemble $\mathcal{B}$ est un hyperparamètre de l'algorithme d'optimisation.

1.4.3 Régularisation

En apprentissage automatique, on s'intéresse particulièrement à la capacité de généralisation d'un agent. En d'autres termes, on s'intéresse à sa capacité d'accomplir adéquatement la tâche demandée même en présence de données non observées durant l'entrainement. Pour quantifier cette propriété, on surveillera les valeurs de la fonction de perte $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_{\text{test}})$ évaluée sur un ensemble de données test $\mathcal{D}_{\text{test}}$, c'est-à-dire n'appartenant pas à l'échantillon d'apprentissage $\mathcal{D}$.

En rapportant l'**erreur d'entrainement** $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ et l'**erreur de généralisation** $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_{\text{test}})$ en fonction de la complexité d'un modèle (voir figure 1.1), on s'aperçoit qu'il existe deux régimes dans lesquels l'agent peut se retrouver : sous-apprentissage et surapprentissage.

Le **sous-apprentissage** survient lorsque l'erreur d'entrainement n'est pas assez basse. Ceci est un bon indicateur que le modèle ne parvient pas à modéliser complètement la structure des données. Pour vaincre le sous-apprentissage, il suffit généralement d'augmenter la complexité du modèle.

Le **surapprentissage** survient lorsque l'écart entre $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ et $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}_{\text{test}})$ est trop important. Ceci suggère que le modèle est en train de mémoriser naïvement les données d'entrainement plutôt que d'apprendre leur structure. Il est donc pénalisé lorsqu'il est confronté aux données de test. L'approche optimale pour vaincre le surapprentissage est d'augmenter la quantité de données observées lors de l'entrainement. Or, cette solution n'est pas toujours évidente à mettre en pratique, par exemple dans le cas du domaine médical où l'acquisition de données est couteuse. Lorsqu'il n'y a pas moyen d'obtenir plus de données, on doit recourir à des techniques de régularisation.

La **régularisation** d'un modèle vise à réduire l'erreur de généralisation sans nécessairement diminuer l'erreur d'entrainement. Pour ce faire, on ajoute à la fonction de perte un terme de régularisation qui dépend des paramètres du modèle. Formellement, on a

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}, \mathcal{D}) = \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) + \lambda \Omega(\boldsymbol{\theta}), \tag{1.13}$$

où $\lambda \in [0, \infty)$ est un hyperparamètre contrôlant l'importance du terme de régularisation $\Omega(\boldsymbol{\theta})$ par rapport à la fonction de perte originale. Il existe plusieurs fonctions $\Omega(\boldsymbol{\theta})$ pouvant être utilisées et celles-ci varient en fonction du modèle utilisé. À la section 2.1.5, nous en verrons deux qui sont typiquement employées pour les réseaux de neurones.

Chapitre 2

Réseaux de neurones artificiels

“ Tell me and I forget. Teach me and I remember. Involve me and I learn.

Benjamin Franklin ”

Inspiré par la biologie, les réseaux de neurones artificiels reprennent le concept d'axones interconnectant les neurones dans un cerveau, d'où leur nom. D'un point de vue mathématique, les neurones sont représentés par des variables observées (entrées du modèle) ou bien cachées (latentes, non observées), tandis que les axones sont représentés par les connexions entre ces variables, c'est-à-dire leurs liens de dépendance.

Le réseau de neurones est un modèle mathématique qui calcule $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ l'approximation de la fonction $f^*(\boldsymbol{x})$ définie par la tâche d'apprentissage. Le modèle est paramétré par le vecteur $\boldsymbol{\theta}$, où chaque composante représente un degré de dépendance d'une paire de variables.

Comme il a été mentionné au chapitre 1, les réseaux de neurones trouvent applications dans plusieurs contextes. Ils sont utilisés pour résoudre des tâches de classification, de régression et même d'estimation de densité de probabilité. Ils sont autant utilisés en apprentissage supervisé qu'en apprentissage non supervisé, et autant comme modèles discriminatifs que comme modèles génératifs. Bref, c'est une famille de modèles très flexible et puissante qui vaut la peine d'être exploitée.

Ce chapitre couvre trois architectures répandues de réseaux de neurones artificiels, soit le réseau multicouche 2.1, le réseau à convolution 2.2 et le réseau récurrent 2.3.

Dans ce chapitre, le terme **unité** sera préféré à celui de neurone, excepté lorsqu'on réfère au nom du modèle, afin de bien faire la distinction entre le modèle mathématique et celui biologique. Dans le même ordre d'idées, le terme **connexion** sera utilisé pour désigner un axone.

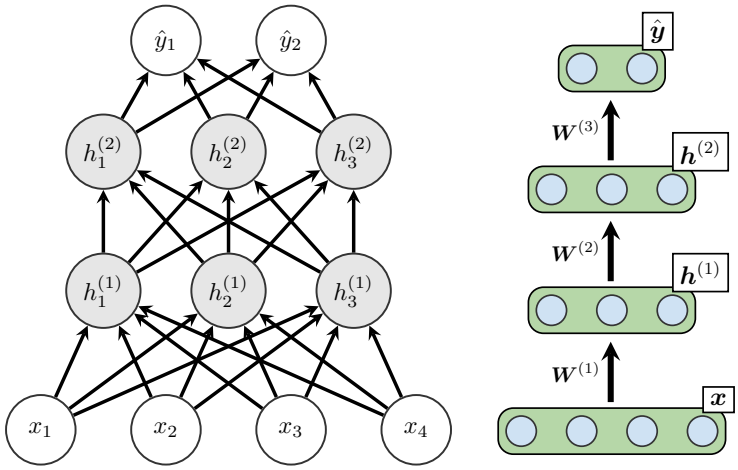


figure 2.1 – Deux représentations graphiques d'un même réseau de neurones artificiels à deux couches cachées. Spécifiquement, la couche d'entrée $\boldsymbol{x}$ comporte quatre unités, les deux couches cachées $\boldsymbol{h}^{(1)}$ et $\boldsymbol{h}^{(2)}$ en ont trois chacune, et la couche de sortie $\hat{\boldsymbol{y}}$ en possède deux. Par souci de visibilité, les biais $b_i^{(l)}$ des neurones ont été omis.

Les connexions d'un réseau de neurones peuvent être dirigées (relations de causalité) ou non dirigées. Les architectures présentées aux sections 2.1, 2.2 et 2.3 utilisent toutes des connexions dirigées. En revanche, la machine de Boltzmann restreinte, détaillée à la section 2.4, est un exemple d'un réseau possédant des connexions non dirigées.

2.1 Réseau de neurones multicouche

Cette section décrit la structure d'un réseau de neurones dans lequel les unités sont séparées en plusieurs couches. Elle servira également de base pour l'explication des variantes présentées dans les prochaines sections.

Le réseau de neurones peut être interprété comme un graphe dirigé acyclique où les noeuds et les arrêtes sont respectivement les unités et les connexions du réseau. En théorie des graphes, il est connu que, dans un graphe complet, le nombre d'interconnexions croît quadratiquement en fonction du nombre de noeuds [Wilson, 1996, chap. 2]. Plus précisément, il y aura $\mathcal{O}(H^2)$ connexions pour H noeuds.

Afin de réduire la complexité algorithmique du modèle, les réseaux de neurones artificiels sont généralement divisés en couches où les unités d'une couche peuvent seulement connecter celles d'une couche adjacente. Plus précisément, il y aura alors $\mathcal{O}\left(LH_l^2\right)$ connexions pour un réseau de neurones de L couches où la taille de la plus grande couche $H_l \ll H$. Notons que L et le nombre d'unités par couche H_l sont des hyperparamètres du modèle et peuvent être ajustés en conséquence.

Dans un réseau de neurones multicouche, chaque couche est **entièrement connectée** à la précédente (section 2.1.1). On désigne la première couche comme étant la **couche d'entrée** du modèle puisque c'est celle qui reçoit les données observées. Dans le même ordre d'idées, la dernière couche est la **couche de sortie** et est responsable de fournir le résultat pour la tâche à accomplir. Le reste des couches, c'est-à-dire celles du milieu, sont nommées **couches cachées** et contiennent des **unités cachées**. On identifiera les différentes couches comme suit : $\boldsymbol{x}$ pour la couche d'entrée, $\boldsymbol{h}^l$ pour la l^{e} couche cachée, et $\hat{\boldsymbol{y}}$ pour la couche de sortie. La figure 2.1 illustre un exemple de réseau multicouche. Par souci de lisibilité, les biais $\boldsymbol{b}$, associés aux unités d'une couche, ont volontairement été omis dans toutes les figures de cette thèse.

Dans un réseau de neurones, le rôle des couches cachées est d'extraire, à partir d'une observation, toute l'information utile pour qu'une fois rendu à la couche de sortie, le modèle soit en mesure d'accomplir correctement la tâche. Plus particulièrement, chaque couche s'occupe de transformer les valeurs provenant de la couche précédente dans un espace où les caractéristiques discriminantes sont plus facilement identifiables. On note cette transformation

$$\boldsymbol{h}^{(l)} = f_{\boldsymbol{\theta}}^{(l)}\left(\boldsymbol{h}^{(l-1)}\right) \tag{2.1}$$

où $\boldsymbol{h}^{(l)}$ désigne les valeurs de la couche courante et $\boldsymbol{h}^{(l-1)}$ celles de la couche précédente.

Le processus d'apprentissage d'un réseau de neurones comporte deux phases. On calcule d'abord, pour un exemple de l'ensemble d'entrainement $\boldsymbol{x} \in \mathcal{D}$, les valeurs de la couche de sortie $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$. Cette première phase porte le nom de **propagation avant** (section 2.1.3). Par la suite, on évalue la qualité de l'approximation $f_{\boldsymbol{\theta}}(\boldsymbol{x})$

selon des critères dictés par la tâche et le type d'apprentissage, c'est-à-dire la fonction de perte $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$. Selon les erreurs commises, des correctifs sont appliqués aux paramètres du réseau de neurones. Cette deuxième phase se nomme **propagation arrière** (section 2.1.4).

2.1.1 Couche entièrement connectée

Une couche est dite entièrement connectée si chacune de ses unités possède une connexion avec toutes celles de la couche précédente. À titre d'exemple, les couches du réseau illustré à la figure 2.1 sont toutes entièrement connectées.

Le traitement $f_{\boldsymbol{\theta}}^{(l)}$ effectué par une couche l entièrement connectée correspond à une transformation linéaire

$$\boldsymbol{a}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{2.2}$$

suivi de l'application d'une fonction non linéaire

$$\boldsymbol{h}^{(l)} = \boldsymbol{g}\left(\boldsymbol{a}^{(l)}\right). \tag{2.3}$$

La fonction $\boldsymbol{g}(\cdot)$ est une non-linéarité (voir section 2.1.2) qu'on appelle également **fonction d'activation**. Les paramètres du modèle pour une couche donnée l sont $\{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}$ où la matrice $\boldsymbol{W}^{(l)} \in \mathbb{R}^{H_l \times H_{l-1}}$ contient les poids des connexions entre la couche $l-1$ et l, et le vecteur $\boldsymbol{b}^{(l)} \in \mathbb{R}^{H_l}$ contient les biais de chaque unité de la couche l.

Pour faire un parallèle avec le domaine du traitement de signal, les poids associés aux connexions de la j^{e} unité, c'est-à-dire ceux de la rangée $\boldsymbol{W}_{j,\cdot}^{(l)}$, peuvent être interprétés comme les poids d'un filtre linéaire avec un champ récepteur couvrant l'entièreté des valeurs de la couche précédente. Dans ce contexte, une couche consiste donc en une collection de filtres capables d'identifier différentes caractéristiques dans le signal provenant de la couche précédente.

Le biais $\boldsymbol{b}_j^{(l)}$ quant à lui contrôle le seuil à partir duquel la j^{e} unité est considérée comme active, c'est-à-dire que la caractéristique recherchée a été identifiée dans le signal provenant de la couche précédente. Cela permet aux unités d'avoir leur propre

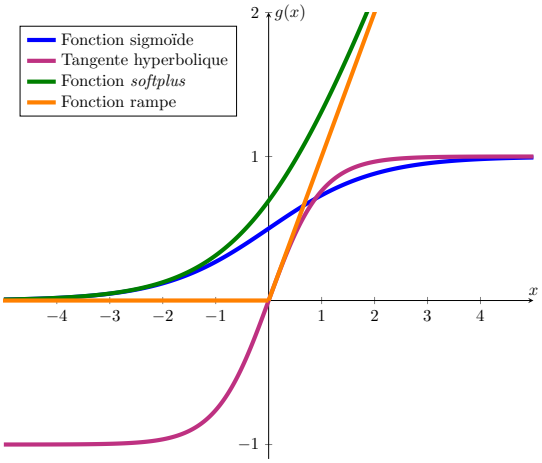


figure 2.2 – Graphiques des non-linéarités.

degré de sensibilité en favorisant différentes régions du domaine défini par la fonction d'activation. Mathématiquement, le biais correspond à une translation horizontale de la non-linéarité.

2.1.2 Non-linéarités

La présence de non-linéarités dans un réseau de neurones permet de modéliser des phénomènes pour lesquelles les variables de sorties $\hat{\boldsymbol{y}}$ ne varient pas linéairement en fonction des variables d'entrée $\boldsymbol{x}$. Sans elles, un réseau de neurones serait composé seulement de transformations linéaires. Or, peu importe le nombre de couches cachées que l'on ajoute, le réseau se comporterait toujours comme un Perceptron (réseau sans couche cachée) puisque la somme de fonctions linéaires reste une fonction linéaire.

Dans un réseau de neurones, une non-linéarité consiste en une fonction $g : \mathbb{R} \to \mathbb{E}$ permettant de contraindre les valeurs du réseau de neurones à un domaine spécifique $\mathbb{E}$ plus propice à l'apprentissage. Tel que l'on verra à la section 2.1.4, la dérivée de ces fonctions doit être définie puisqu'elle est nécessaire lors de la phase de propagation arrière.

Notons, $\boldsymbol{g}(\boldsymbol{x})$, le vecteur correspondant à l'application de la non-linéarité g élément

par élément, c'est-à-dire $\boldsymbol{g}(\boldsymbol{x}) = \left[g(x_1), \ldots, g\left(x_{|\boldsymbol{x}|}\right)\right]^\top$.

La figure 2.1.2 illustre quelques non-linéarités les plus utilisées. Voici leur fonction accompagnée de leur dérivée :

- Fonction sigmoïde (ou fonction logistique)

$$g(x) = \overbrace{1/(1+e^{-x})}^{\sigma(x)} \tag{2.4}$$
$$g'(x) = \sigma(x)(1-\sigma(x)) \tag{2.5}$$

- Tangente hyperbolique

$$g(x) = \tanh(x) \tag{2.6}$$
$$g'(x) = 1 - \tanh^2(x) \tag{2.7}$$

- Fonction rampe (*rectifier*)

$$g(x) = \max(0, x) \tag{2.8}$$
$$g'(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x < 0 \end{cases} \tag{2.9}$$

  Bien que la dérivée de la fonction rampe ne soit pas définie en $x = 0$, en pratique on supposera qu'elle vaut 0 dans les rares cas où ça arrive.

- Fonction *softplus* (approximation lisse de la fonction rampe)

$$g(x) = \ln(1+e^x) \tag{2.10}$$
$$g'(x) = 1/(1+e^{-x}) \tag{2.11}$$

  Remarquons que la dérivée de la fonction *softplus* est la fonction sigmoïde de l'équation (2.4).

2.1.3 Propagation avant

Comme expliqué plus tôt, la propagation avant désigne le processus permettant de calculer $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ à partir d'une observation $\boldsymbol{x}$. Dans un réseau de neurones, $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ correspond à une composition de L fonctions

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f_{\boldsymbol{\theta}}^{(L)}\left(f_{\boldsymbol{\theta}}^{(L-1)}\left(\dots f_{\boldsymbol{\theta}}^{(1)}(\boldsymbol{x})\right)\right) \tag{2.12}$$

avec $f_{\boldsymbol{\theta}}^{(l)}$ définie à l'équation (2.1).

Plus particulièrement, la propagation avant d'un réseau multicouche utilise les équations récurrentes suivantes (les équations 2.2 et 2.3 sont rappelées à des fins pratiques)

$$\boldsymbol{h}^{(0)} = \boldsymbol{x} \tag{2.13}$$

$$\boldsymbol{a}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{2.14}$$

$$\boldsymbol{h}^{(l)} = \boldsymbol{g}\left(\boldsymbol{a}^{(l)}\right) \tag{2.15}$$

$$\boldsymbol{h}^{(L)} = \boldsymbol{\tau}\left(\boldsymbol{a}^{(L)}\right), \tag{2.16}$$

où $\boldsymbol{h}^{(0)}$ et $\boldsymbol{h}^{(L)}$ correspondent respectivement à la couche d'entrée et à la couche de sortie. Notons que $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(L)}$. La fonction $\boldsymbol{\tau}(\cdot)$ est une non-linéarité habituellement différente de celles utilisées par les couches cachées. Elle s'assure que la sortie du modèle soit compatible avec la tâche d'apprentissage (voir exemples à la section 2.1.2). Les paramètres du modèle sont $\boldsymbol{\theta} = \left\{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)} : l \in [1, L]\right\}$.

2.1.4 Propagation arrière

La propagation arrière est un algorithme permettant de calculer $\nabla_{\boldsymbol{\theta}}\mathcal{L}$, le gradient de la fonction de perte par rapport aux paramètres du modèle pour un exemple donné. Utilisé conjointement avec un algorithme d'optimisation de premier ordre, cela permet de mettre à jour les paramètres du modèle dans l'optique de minimiser $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$.

La propagation arrière tire avantage de la dérivation en chaine pour calculer itérativement les gradients couche par couche en commençant par la sortie. Donc, pour

calculer $\nabla_{\boldsymbol{\theta}}\mathcal{L}$, il faut d'abord obtenir les gradients de $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ par rapport aux différentes composantes du réseau de neurones. Ceux-ci peuvent être obtenus en utilisant les équations récurrentes suivantes

$$\nabla_{\boldsymbol{a}^{(L)}}\mathcal{L} = (\nabla_{\boldsymbol{h}^{(L)}}\mathcal{L}) \odot \boldsymbol{\tau}'\left(\boldsymbol{a}^{(L)}\right) \tag{2.17}$$

$$\nabla_{\boldsymbol{h}^{(l)}}\mathcal{L} = \boldsymbol{W}^{(l+1)^\top}(\nabla_{\boldsymbol{a}^{(l+1)}}\mathcal{L}) \tag{2.18}$$

$$\nabla_{\boldsymbol{a}^{(l)}}\mathcal{L} = (\nabla_{\boldsymbol{h}^{(l)}}\mathcal{L}) \odot \boldsymbol{\sigma}'\left(\boldsymbol{a}^{(l)}\right) \tag{2.19}$$

$$\nabla_{\boldsymbol{W}^{(l)}}\mathcal{L} = (\nabla_{\boldsymbol{a}^{(l)}}\mathcal{L}) \otimes \boldsymbol{h}^{(l-1)} \tag{2.20}$$

$$\nabla_{\boldsymbol{b}^{(l)}}\mathcal{L} = (\nabla_{\boldsymbol{a}^{(l)}}\mathcal{L}), \tag{2.21}$$

où $\boldsymbol{\sigma}'(\cdot)$ et $\boldsymbol{\tau}'(\cdot)$ représentent les dérivées des fonctions d'activation appliquées élément par élément. Les dérivées partielles de la fonction de perte par rapport aux unités de la couche de sortie, $\nabla_{\boldsymbol{h}^{(L)}}\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$, diffèrent selon la fonction de perte utilisée (voir section 1.4.1). L'opération $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^\top$ dénote le produit dyadique (*outer product*) alors que le symbole $\odot$ représente la multiplication point à point.

Dans un réseau de neurones multicouche, les gradients des équations (2.20) et (2.21) sont utilisés pour la mise à jour des poids et des biais de la manière suivante

$$\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \alpha\nabla_{\boldsymbol{W}^{(l)}}\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) \tag{2.22}$$

$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha\nabla_{\boldsymbol{b}^{(l)}}\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}). \tag{2.23}$$

En théorie, la propagation arrière correspond uniquement au calcul du gradient via la dérivation en chaine. Par contre, on emploie souvent le terme propagation arrière dans un sens plus large qui réfère à la fois au calcul du gradient et à la mise à jour des paramètres.

2.1.5 Régularisation des paramètres

Les réseaux multicouches sont des modèles très puissants. En effet, étant donné un nombre suffisant d'unités cachées, ils peuvent approximer n'importe quelles fonctions avec une précision arbitraire [Goodfellow et al., 2016, chap. 6]. Or, il n'est pas surprenant que de tels modèles soient prédisposés au surapprentissage.

Tel que mentionné au chapitre précédent, ajouter un terme de régularisation est une technique permettant de réduire le surapprentissage. D'ordre général, pour les réseaux de neurones artificiels, on cherche à pénaliser les valeurs extrêmes des paramètres du réseau. Plus particulièrement, on opte pour une fonction $\Omega(\boldsymbol{\theta})$ qui pénalise la norme de $\boldsymbol{\omega}$, c'est-à-dire le vecteur formé en concaténant l'ensemble des poids $\boldsymbol{W}^{(l)}$ du réseau. Dans cette thèse, deux types de régularisation sont utilisés.

La **régularisation $\boldsymbol{L^2}$** consiste à utiliser la moitié de la somme du carré de tous les poids du réseau comme pénalisation. Elle force le modèle à préférer de petites valeurs pour ses poids. Formellement, on a

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\omega}\|_2^2 = \frac{1}{2}\sum_{l=1}^{L}\sum_{i=1}^{H_l}\sum_{j=1}^{H_{l-1}}\left(w_{ij}^{(l)}\right)^2 \tag{2.24}$$

et le gradient correspondant est donné par

$$\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta}) = \boldsymbol{\omega}. \tag{2.25}$$

La **régularisation $\boldsymbol{L^1}$** consiste à utiliser la somme de la valeur absolue de tous les poids du réseau comme pénalisation. En plus de forcer le modèle à préférer de petites valeurs pour ses poids, elle promeut une distribution parcimonieuse (*sparse*) de ceux-ci, c'est-à-dire que certains (les moins utiles) auront une valeur nulle. Formellement, on a

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\omega}\|_1 = \sum_{l=1}^{L}\sum_{i=1}^{H_l}\sum_{j=1}^{H_{l-1}}\left|w_{ij}^{(l)}\right|. \tag{2.26}$$

et le gradient correspondant (en fait il s'agit du sous-gradient) est donné par

$$\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta}) = \mathbf{sign}\left(\boldsymbol{\omega}\right), \tag{2.27}$$

où $\mathbf{sign}\left(\cdot\right)$ est la fonction signe appliquée élément par élément.

2.2 Réseau de neurones à convolution

Qu'il s'agisse d'une caméra faisant la mise au point sur un visage, d'un véhicule autonome évitant un piéton ou d'un logiciel segmentant des tissus cancéreux dans un cerveau, la reconnaissance d'objets dans une image est un problème inhérent au domaine de la vision par ordinateur. Récemment, les réseaux de neurones artificiels ont connu de grands succès dans ce domaine [Long et al., 2015].

De façon générale en apprentissage automatique, on désire fournir au modèle l'information la plus brute (non transformée) qu'il soit et le laisser identifier les caractéristiques importantes pour la tâche. Dans un contexte où les données sont des images, cela correspond à fournir au modèle un ensemble de pixels plutôt que des caractéristiques extraites à la main (ex. SIFT [Lowe, 1999], HOG [Dalal and Triggs, 2005], etc.).

Ceci dit, le réseau multicouche tel que présenté à la section 2.1 s'avère impraticable pour traiter des images ayant une résolution la moindrement élevée. Prenons par exemple une photo ayant une définition de 3.2 mégapixels. On désire traiter l'image avec un réseau de neurones ayant une seule couche cachée de 100 unités. Le modèle comptera au moins 320 millions paramètres ! Rappelons que les couches entièrement connectées requièrent $H_{l-1} \times H_l$ connexions, ce qui explique le nombre de paramètres démesuré pour cet exemple.

Cette section décrit la structure d'un réseau de neurones exploitant la structure topologique des données observées (ex. la corrélation entre les pixels voisins) dans lequel le nombre de paramètres ne dépend pas de la taille de la couche d'entrée. Ceci est possible grâce à un nouveau type de couche : **couche à convolution** (section 2.2.1). Bien qu'il soit possible d'utiliser uniquement des couches à convolution, en général les premières couches d'un réseau de neurones à convolution sont de type convolution et les dernières sont entièrement connectées.

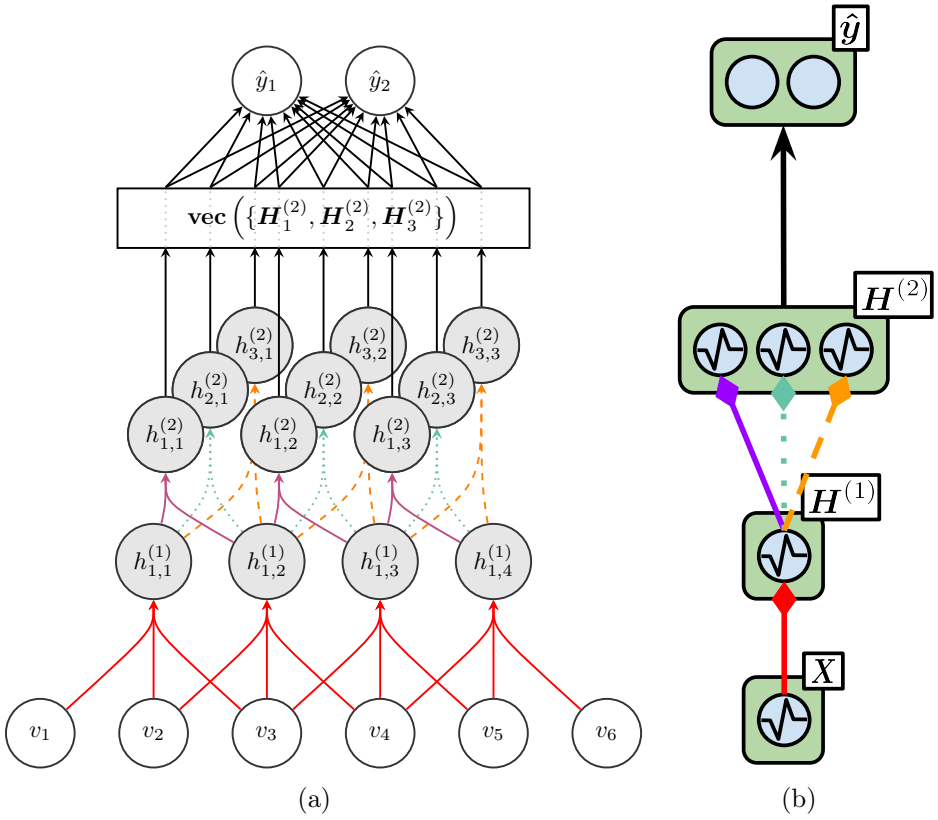


figure 2.3 – Deux représentations graphiques d'un même réseau à convolution montrant le traitement d'un signal 1D. Le modèle possède deux couches cachées suivies d'une couche de sortie entièrement connectée. Le modèle reçoit en entrée un signal $\boldsymbol{X}$ de taille 6. La première couche à convolution utilise un filtre de taille 3 produisant une carte d'activation $\boldsymbol{H}^{(1)}$ de taille 4. La deuxième utilise trois filtres de taille 2 produisant des cartes d'activation $\boldsymbol{H}^{(2)}$ de taille 3. Ces valeurs sont ensuite concaténées afin de calculer les valeurs de la couche de sortie $\hat{\boldsymbol{y}}$. À chaque couche, la couleur (ou style) des connexions sert à identifier quel filtre est utilisé pour la convolution.

2.2.1 Couche à convolution

Tout comme les couches entièrement connectées, la couche à convolution contient un nombre H_l d'unités cachées qui s'activent en fonction de la couche précédente. Par contre, le champ récepteur du filtre de chaque unité est beaucoup plus restreint, c'est-à-dire qu'il ne couvre qu'une fraction des valeurs de la couche précédente. Ainsi, le filtre est appliqué autant de fois que nécessaire pour couvrir l'entièreté de l'information provenant de la couche précédente. Cela aura pour effet d'augmenter la dimension de la valeur associée à une unité. La figure 2.3(a) illustre bien la faible densité des connexions entre la couche d'entrée et la première couche cachée, contrairement à celle d'un réseau multicouche (figure 2.1). On y remarque aussi qu'un même filtre est réutilisé plus d'une fois par unité. Ceci est réalisé à l'aide du produit de convolution discrète entre deux fonctions

$$(f * g)[n] = \sum_{i=-\infty}^{\infty} f[n-i]g[i]. \tag{2.28}$$

Dans le domaine du traitement de signal (ou de l'image), ceci correspond à l'application d'un filtre linéaire $\boldsymbol{k}$ (*kernel*) sur un signal fini $\boldsymbol{s}$ et on obtient un troisième vecteur $\boldsymbol{t} = \boldsymbol{s} * \boldsymbol{k}$.

On peut imaginer le processus de convolution comme étant le déplacement d'un filtre le long du signal un élément à la fois. À chaque position, on calcule la somme des valeurs du signal pondérée par celles du filtre. Ce décalage permet à la couche à convolution d'être robuste aux translations dans le signal d'entrée. En effet, une même caractéristique (apprise par un filtre) pourra être détectée à différents endroits le long du signal, et ce, sans nécessiter de paramètres additionnels.

Bien qu'il soit possible de déplacer le filtre de plusieurs éléments entre chacune de ses applications, dans cette thèse, on utilisera uniquement les décalages de 1.

Corrélation croisée

Notamment utilisée en traitement de signal, la corrélation croisée sert à mesurer la similitude entre deux signaux. Cette opération mathématique se distingue de la convolution par la direction dans laquelle l'application du filtre $\boldsymbol{k}$ est effectuée. Formellement, étant donnée deux fonctions discrètes f et g, on a

$$(f \star g)[n] = \sum_{i=-\infty}^{\infty} f[n+i]g[i], \tag{2.29}$$

où la fonction f est indicée par $n+i$ plutôt que $n-i$.

Dans le domaine du traitement de signal (ou de l'image), ceci correspond à l'application d'un filtre linéaire qui a été préalablement renversé. Sans perte de généralité, supposons un signal fini $\boldsymbol{s}$, on obtient l'égalité suivante

$$\boldsymbol{s} * \boldsymbol{k} = \boldsymbol{s} \star \mathbf{flip}\,(\boldsymbol{k}) \tag{2.30}$$

ou de manière équivalente

$$\boldsymbol{s} * \mathbf{flip}\,(\boldsymbol{k}) = \boldsymbol{s} \star \boldsymbol{k}, \tag{2.31}$$

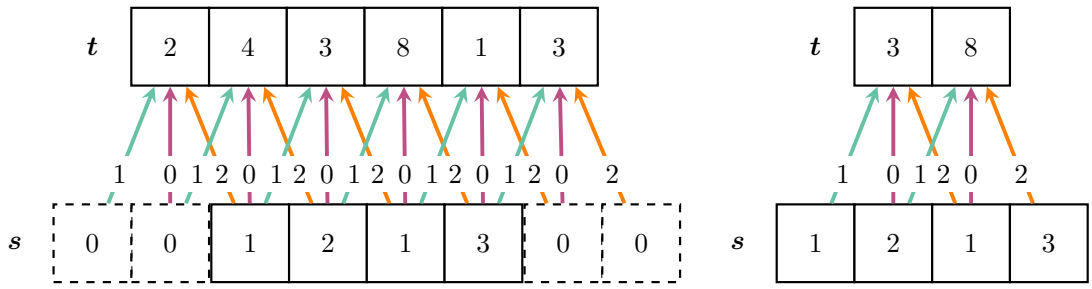
avec $\mathbf{flip}\,(\boldsymbol{k}) = [k_K, k_{K-1}, \dots k_1]$ pour un filtre $\boldsymbol{k}$ de taille K.

Gestion des bords

Puisqu'on travaille avec des données finies, quelques ajustements doivent être faits en bordure du signal lors de la convolution discrète. On peut considérer que les valeurs à l'extérieur du signal sont zéro et les intégrer à la convolution. Cette première approche est définie par

$$(\boldsymbol{s} \circledast \boldsymbol{k})_n = \sum_{i=1}^{K} s_{n-i+1} k_i, \quad \forall n \in [1, S+K-1]. \tag{2.32}$$

où $s_n = 0$ pour tout $n \notin [1, S]$, et S et K désignent respectivement la taille des vecteurs $\boldsymbol{s}$ et $\boldsymbol{k}$. De façon similaire, $\circledast$ indique une corrélation croisée où l'on ajoute des valeurs nulles en bordure du signal.



(a) $\boldsymbol{s} \circledast \boldsymbol{k}$ (b) $\boldsymbol{s} * \boldsymbol{k}$

figure 2.4 – Différence entre les deux types de gestion des bords lors de la convolution entre $\boldsymbol{s} = [1, 2, 1, 3]$ et $\boldsymbol{k} = [2, 0, 1]$. a) Convolution qui considère les valeurs à l'extérieur du signal comme étant zéro (équation 2.32). Le vecteur résultant $\boldsymbol{t}$ est plus grand que celui du signal $\boldsymbol{s}$. b) Convolution qui ignore les valeurs à l'extérieur du signal (équation 2.33). Le vecteur résultant $\boldsymbol{t}$ est plus petit que celui du signal $\boldsymbol{s}$.

Alternativement, on peut ignorer ces valeurs et effectuer la convolution strictement sur les valeurs provenant du signal. Cette seconde approche est décrite par

$$(\boldsymbol{s} * \boldsymbol{k})_n = \sum_{i=1}^{K} s_{n-i+1} k_i, \quad \forall n \in [K, S]. \tag{2.33}$$

Il existe également une approche hybride dans laquelle on ajoute un nombre suffisant de valeurs nulles avant et après le signal pour que le vecteur résultant $\boldsymbol{t}$ conserve une taille identique à celle du vecteur convolué $\boldsymbol{s}$. Cependant, cette approche n'est pas utilisée dans cette thèse.

La figure 2.4 illustre la différence entre les deux premières approches. En utilisant $\circledast$ (équation 2.32), la taille du vecteur résultant sera de $S + K - 1$ alors qu'avec $*$ (équation 2.33), elle sera de $S - K + 1$.

Convolution 2D

On utilise généralement les réseaux à convolution pour traiter signaux 2D (images). Or, le produit de convolution se généralise facilement à des problèmes de plusieurs dimensions. L'équivalent 2D des équations (2.32) et (2.33) sont données respectivement

par

$$(\boldsymbol{S} \circledast \boldsymbol{K})_{n,m} = \sum_{i=1}^{K_H} \sum_{j=1}^{K_W} s_{n-i+1,m-j+1} k_{i,j} \quad \begin{cases} \forall n \in [1, S_H + K_H - 1] \\ \forall m \in [1, S_W + K_W - 1] \end{cases} \tag{2.34}$$

$$(\boldsymbol{S} * \boldsymbol{K})_{n,m} = \sum_{i=1}^{K_H} \sum_{j=1}^{K_W} s_{n-i+1,m-j+1} k_{i,j} \quad \begin{cases} \forall n \in [K_H, S_H] \\ \forall m \in [K_W, S_W] \end{cases}, \tag{2.35}$$

où $\boldsymbol{K}$ est un filtre 2D $S_H \times S_W$ et $\boldsymbol{S}$ est une image $S_H \times S_W$ pixels avec $s_{n,m} = 0$ pour tout $n \notin [1, S_H]$ et tout $m \notin [1, S_W]$. Pour un support visuel de la convolution 2D, le lecteur peut se référer au rapport technique de Dumoulin and Visin [2016] qui illustre très bien l'impact des décalages du filtre et des techniques pour gérer la convolution en bordure des images.

2.2.2 Propagation avant

Étant donné que les réseaux de neurones à convolution seront utilisés au chapitre 4 sur des signaux 2D, les étapes de la propagation avant et arrière seront expliquées dans ce contexte.

Définissons le i^{e} exemple d'entrainement $\boldsymbol{X}_i \in \mathbb{R}^{D_C \times D_H \times D_W}$ comme étant une image couleur représentée par un tenseur 3D, où D_C est le nombre de canaux (généralement 3 pour RGB), D_H est la hauteur et D_W est la largeur, toutes deux exprimées en nombre de pixels.

Lors de la propagation avant, le filtre associé à chaque unité cachée est convolué aux valeurs de la couche précédente pour créer une **carte d'activation** (*feature map*). Cette carte contient la valeur d'activation du filtre en fonction de la position spatiale de son champ récepteur. L'état d'une couche à convolution possédant H_l unités est défini par l'ensemble des cartes d'activation $\boldsymbol{H}^{(l)} = \left\{ \boldsymbol{H}_1^{(l)}, \ldots, \boldsymbol{H}_{H_l}^{(l)} \right\}$. La figure 2.5 illustre le concept d'un réseau à convolution appliqué à des signaux 2D. On y voit également que la taille des filtres et la façon dont les bords sont gérés influencent la taille des cartes d'activation.

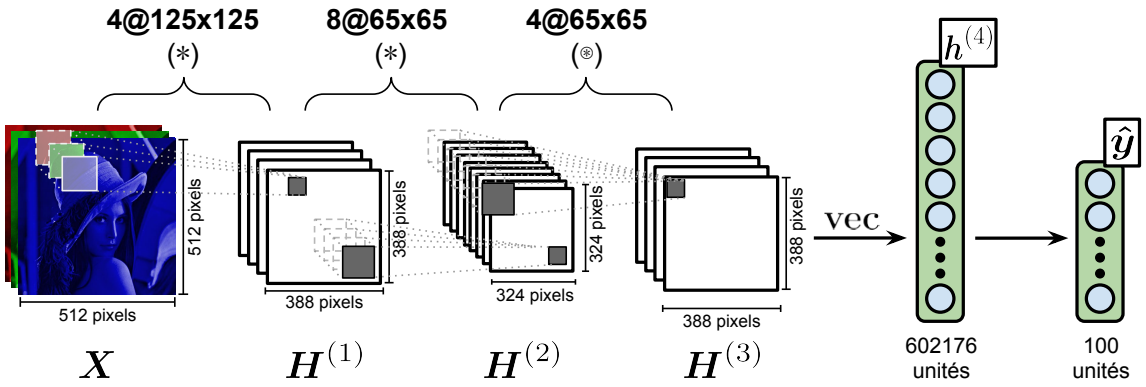


figure 2.5 – Exemple fictif servant à illustrer la différence entre $*$ et $\circledast$. Il s'agit d'un réseau de neurones à convolution utilisé pour la classification d'images couleur. Le modèle possède trois couches de convolution suivies d'une couche de sortie entièrement connectée. La première couche à convolution utilise quatre filtres de taille 125×125 produisant des cartes d'activation $\boldsymbol{H}^{(1)}$ de taille 388×388. La deuxième utilise huit filtres de taille 65×65 produisant des cartes d'activation $\boldsymbol{H}^{(2)}$ de taille 324×324. La troisième utilise quatre filtres de taille 65×65, mais ajoute également des zéros en bordure de l'image, produisant ainsi des cartes d'activation $\boldsymbol{H}^{(3)}$ de taille 388×388. Ces valeurs sont ensuite concaténées, représentées par le vecteur $\boldsymbol{h}^{(4)}$, afin de calculer les valeurs de la couche de sortie $\hat{\boldsymbol{y}}$.

Sans perte de généralité, supposons un réseau de neurones contenant *uniquement* des couches de convolution qui ignore les valeurs à l'extérieur du signal (équation (2.35)). La propagation de l'information à travers de ce réseau est définie à l'aide des équations récurrentes suivantes

$$\boldsymbol{H}^{(0)} = \boldsymbol{X} \tag{2.36}$$

$$\boldsymbol{A}_j^{(l)} = b_j^{(l)} \boldsymbol{e}_H^{(l)} \boldsymbol{e}_W^{(l)\top} + \sum_{i=1}^{H_{l-1}} \boldsymbol{H}_i^{(l-1)} * \boldsymbol{W}_{i,j}^{(l)} \tag{2.37}$$

$$\boldsymbol{H}_j^{(l)} = \boldsymbol{g}\left(\boldsymbol{A}_j^{(l)}\right) \tag{2.38}$$

$$\boldsymbol{H}_j^{(L)} = \boldsymbol{\tau}\left(\boldsymbol{A}_j^{(L)}\right), \tag{2.39}$$

où $\boldsymbol{H}^{(l)}$ représente le tenseur 3D contenant la carte des H_l caractéristiques identifiées par la couche l. Notons que $\boldsymbol{H}^{(0)}$ et $\boldsymbol{H}^{(L)}$ désignent respectivement la couche d'entrée et celle de sortie. Précisons qu'il n'y a qu'un biais par carte d'activation $\boldsymbol{H}_j^{(l)}$. Les vecteurs $\boldsymbol{e}_H^{(l)}$ et $\boldsymbol{e}_W^{(l)}$ sont remplis de 1, et leur taille correspond respectivement à la hauteur et la largeur de la carte d'activation produite par $\boldsymbol{H}_i^{(l-1)} * \boldsymbol{W}_{i,j}^{(l)}$. La matrice obtenue à l'aide de ces deux vecteurs permet au biais $b_j^{(l)}$ d'influencer l'activation obtenue à chaque position spatiale. Les fonctions $\boldsymbol{g}(\cdot)$ et $\boldsymbol{\tau}(\cdot)$ sont des non-linéarités (voir section 2.1.2) associées respectivement aux couches cachées et à celle de sortie. Les paramètres du modèle sont $\boldsymbol{\theta} = \left\{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)} : l \in [1, L]\right\}$ où les filtres de convolution $\boldsymbol{W}_{i,j}^{(l)} \in \mathbb{R}^{K_H^{(l)} \times K_W^{(l)}}$ composent le tenseur $\boldsymbol{W}^{(l)} \in \mathbb{R}^{H_{l-1} \times H_l \times K_H^{(l)} \times K_W^{(l)}}$ qui contient les poids des connexions entre la couche $l-1$ et l, et le vecteur $\boldsymbol{b}^{(l)} \in \mathbb{R}^{H_l \times 1}$ contient les biais de chaque unité de la couche l.

2.2.3 Propagation arrière

De la même façon qu'à la section 2.1.4, pour un exemple d'entrainement donné, le gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ pour un réseau de neurones à convolution est calculé à l'aide de la propagation arrière.

Sans perte de généralité, supposons un réseau de neurones contenant *uniquement* des couches de convolution qui ignore les valeurs à l'extérieur du signal (équation (2.35)).

Pour calculer $\nabla_{\boldsymbol{\theta}}\mathcal{L}$, il faut d'abord obtenir les gradients de $\mathcal{L}$ par rapport aux différentes composantes du réseau. Ils peuvent être obtenus en utilisant les équations récurrentes suivantes

$$\nabla_{\boldsymbol{A}_j^{(L)}}\mathcal{L} = (\nabla_{\boldsymbol{H}^{(L)}}\mathcal{L}) \odot \boldsymbol{\tau}'\left(\boldsymbol{A}_j^{(L)}\right) \tag{2.40}$$

$$\nabla_{\boldsymbol{H}_j^{(l)}}\mathcal{L} = \sum_{i=1}^{H_{l+1}} \left(\nabla_{\boldsymbol{A}_j^{(l+1)}}\mathcal{L}\right) \circledast \boldsymbol{W}_{i,j}^{(l+1)} \tag{2.41}$$

$$\nabla_{\boldsymbol{A}_j^{(l)}}\mathcal{L} = (\nabla_{\boldsymbol{H}^{(l)}}\mathcal{L}) \odot \boldsymbol{\sigma}'\left(\boldsymbol{A}_j^{(l)}\right) \tag{2.42}$$

$$\nabla_{\boldsymbol{W}_{i,j}^{(l)}}\mathcal{L} = \left(\nabla_{\boldsymbol{A}_j^{(l)}}\mathcal{L}\right) * \boldsymbol{H}_i^{(l-1)} \tag{2.43}$$

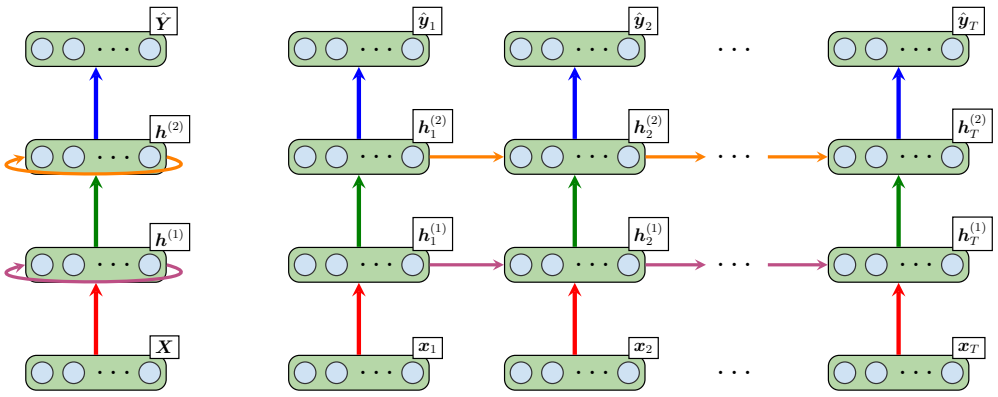
$$\nabla_{b_j^{(l)}}\mathcal{L} = {\boldsymbol{e}_H^{(l)}}^{\top}\left(\nabla_{\boldsymbol{A}_j^{(l)}}\mathcal{L}\right)\boldsymbol{e}_W^{(l)}, \tag{2.44}$$

Remarquons que l'équation (2.41) fait appel à $\circledast$, c'est-à-dire une corrélation croisée où l'on ajoute des valeurs nulles en bordure de l'image, produisant ainsi une image plus grande.

2.3 Réseau de neurones récurrent

Avec l'arrivée des téléphones intelligents, les interactions humain–machine sont inévitables. Jusqu'à tout récemment, les dialogues avec nos appareils restaient principalement unidirectionnels. Or, l'effervescence de l'apprentissage profond a permis de propulser la recherche dans le domaine de la compréhension du langage naturel.

La nature particulière des données en traitement du langage rend difficile l'utilisation de réseaux de neurones multicouches tels que présentés à la section 2.1. En effet, les données typiques dans ce domaine consistent en des collections de phrases ou de signaux vocaux. Autrement dit, il s'agit de séquences pouvant contenir un nombre différent d'éléments d'un exemple à l'autre. Puisque la taille de la couche d'entrée d'un réseau de neurones est fixée d'avance, il faudrait en choisir une assez grande de sorte à parer à toutes éventualités. Évidemment, cette solution est très peu pratique puisqu'on ne peut pas prévoir comment le réseau sera utilisé une fois entrainé.



(a) Réseau récurrent (b) Réseau récurrent déroulé

figure 2.6 – Deux représentations d'un même réseau de neurones ayant deux couches cachées récurrentes $\boldsymbol{h}^{(1)}$ et $\boldsymbol{h}^{(2)}$ et dont la couche de sortie $\hat{\boldsymbol{y}}$ est entièrement connectée. Spécifiquement, on suppose que le modèle observe une séquence $\boldsymbol{X}$ de longueur T. La couleur des flèches représente le fait que les poids des connexions entre les différents temps t sont liés. a) Représentation compacte du modèle. b) Représentation du modèle où la récurrence à été déroulée T fois. Par souci de visibilité, les biais n'apparaissent pas dans ces représentations et la taille des couches est arbitraire.

Cette section décrit la structure d'un réseau de neurones pouvant traiter des séquences de *longueur variable* tout en gardant un nombre de paramètres fixe. À la différence du réseau multicouche (tel que présenté à la section 2.1), la structure d'un réseau de neurones récurrent possède des cycles dirigés. Ces cycles sont introduits par l'utilisation d'une ou plusieurs **couches récurrentes** (section 2.3.1). Elles permettent au modèle d'exploiter la nature séquentielle (ou chronologique) des données en conservant une représentation des éléments passés.

Dans cette section, le i^{e} exemple d'entrainement sera maintenant défini comme étant une séquence $\boldsymbol{X}_i = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{T_i})$ de T_i éléments $\boldsymbol{x}_t \in \mathbb{R}^D$. On désigne, le t^{e} élément d'une séquence comme étant l'élément au **temps** t.

2.3.1 Couche récurrente

Une couche est dite récurrente si, à un temps t, elle réutilise le résultat du traitement $f_{\boldsymbol{\theta}}^{(l)}$ qu'elle a obtenu dans le passé, c'est-à-dire au temps $t-1$. Modifions l'équa-

tion (2.1) du réseau multicouche pour intégrer l'information du passé. On note cette nouvelle transformation

$$\boldsymbol{h}_t^{(l)} = f_{\boldsymbol{\theta}}^{(l)} \left(\boldsymbol{h}_t^{(l-1)}, \boldsymbol{h}_{t-1}^{(l)} \right) \tag{2.45}$$

où $\boldsymbol{h}_t^{(l)}$ et $\boldsymbol{h}_t^{(l-1)}$ désigne respectivement les valeurs de la couche l et $l-1$ au temps t, et $\boldsymbol{h}_{t-1}^{(l)}$ désigne les valeurs de la couche l obtenues au temps $t-1$.

La figure 2.6(b) montre qu'une couche récurrente propage l'information provenant du passée à l'aide de connexions transitionnelles. Remarquons que les connexions aux différents temps t sont liées, c'est-à-dire qu'elles correspondent aux mêmes paramètres. L'avantage de lier les connexions est d'être invariant aux tailles des séquences. Conceptuellement, cela signifie que la couche récurrente possède des connexions vers elle-même. La figure 2.6(a) illustre le concept de récursion.

2.3.2 Propagation avant

Dans un réseau de neurones récurrent, la propagation avant permet d'obtenir des valeurs de sortie pour chaque élément de la séquence $\boldsymbol{X}_i$. Formellement, la propagation de l'information au temps t est définie à l'aide des équations récurrentes suivantes

$$\boldsymbol{h}_t^{(0)} = \boldsymbol{x}_t \tag{2.46}$$

$$\boldsymbol{a}_t^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{h}_t^{(l-1)} + \boldsymbol{b}^{(l)} + \boldsymbol{U}^{(l)} \boldsymbol{h}_{t-1}^{(l)} \tag{2.47}$$

$$\boldsymbol{h}_t^{(l)} = \boldsymbol{g} \left(\boldsymbol{a}_t^{(l)} \right) \tag{2.48}$$

$$\boldsymbol{a}_t^{(L)} = \boldsymbol{W}^{(L)} \boldsymbol{h}_t^{(L-1)} + \boldsymbol{b}^{(L)} \tag{2.49}$$

$$\boldsymbol{h}_t^{(L)} = \boldsymbol{\tau} \left(\boldsymbol{a}^{(L)} \right), \tag{2.50}$$

où $\boldsymbol{U}^{(l)}$ est la matrice de transition contenant les poids des connexions récurrentes. Les couches cachées récurrentes sont habituellement initialisées à zéro tel que $\boldsymbol{h}_0^{(l)} = 0$. Notons que $\hat{\boldsymbol{y}}_t = \boldsymbol{h}_t^{(L)}$.

Mise à part la notion de temps, seulement le terme $\boldsymbol{U}^{(l)} \boldsymbol{h}_{t-1}^{(l)}$ de l'équation (2.47) a été ajouté à l'équation correspondante du réseau multicouche (2.2). On remarque

également que les calculs des préactivations $\boldsymbol{a}_t^{(L)}$ et des valeurs de la couche de sortie $\boldsymbol{h}_t^{(L)}$ (équations (2.49) et (2.50)) correspondent à ceux du réseau multicouche (équations (2.2) et (2.16)).

La sortie du modèle correspond à une séquence $\hat{\boldsymbol{Y}} = (\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_{T_i})$ où $\hat{\boldsymbol{y}}_t = \boldsymbol{h}_t^{(L)}$ telle que donnée par l'équation (2.50). Dépendamment de la tâche d'apprentissage (ex. classification de phrases), il arrive qu'une seule prédiction soit désirée. Dans ce cas, il est possible d'utiliser simplement la dernière prédiction de $\hat{\boldsymbol{Y}}$ tel que

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}_{T_i}. \tag{2.51}$$

Ou encore, d'utiliser une somme pondérée des prédictions

$$\hat{\boldsymbol{y}} = \sum_{t=1}^{T_i} a_t \hat{\boldsymbol{y}}_t, \tag{2.52}$$

où les poids a_t peuvent être appris.

2.3.3 Propagation arrière

Le gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ pour un réseau de neurones récurrent est également calculé à l'aide de la propagation arrière telle que présentée à la section 2.1.4. La communauté emploie généralement le terme *Backpropagation Through Time* (BPTT) pour souligner le fait que l'on doit dérouler le réseau récurrent (tel qu'illustré à la figure 2.6(b)) afin de faire la dérivation en chaine. Au final, il s'agit du même principe utilisé pour le réseau multicouche.

Formellement, pour calculer $\nabla_{\boldsymbol{\theta}} \mathcal{L}$, il faut d'abord obtenir les gradients de $\mathcal{L}$ par rapport aux différentes composantes du réseau. Ils peuvent être obtenus en utilisant les

équations récurrentes suivantes

$$\nabla_{\boldsymbol{a}_t^{(L)}} \mathcal{L} = \left(\nabla_{\boldsymbol{h}_t^{(L)}} \mathcal{L} \right) \odot \boldsymbol{\tau}' \left(\boldsymbol{a}_t^{(L)} \right) \tag{2.53}$$

$$\nabla_{\boldsymbol{h}_t^{(L-1)}} \mathcal{L} = \boldsymbol{W}^{(L)^\top} \left(\nabla_{\boldsymbol{a}_t^{(L)}} \mathcal{L} \right) \tag{2.54}$$

$$\nabla_{\boldsymbol{a}_t^{(l)}} \mathcal{L} = \left(\nabla_{\boldsymbol{h}_t^{(l)}} \mathcal{L} \right) \odot \boldsymbol{\sigma}' \left(\boldsymbol{a}_t^{(l)} \right) \tag{2.55}$$

$$\nabla_{\boldsymbol{h}_t^{(l)}} \mathcal{L} = \boldsymbol{W}^{(l+1)^\top} \left(\nabla_{\boldsymbol{a}_t^{(l+1)}} \mathcal{L} \right) + \boldsymbol{U}^{(l)^\top} \left(\nabla_{\boldsymbol{a}_{t+1}^{(l)}} \mathcal{L} \right). \tag{2.56}$$

On remarque que les gradients par rapport aux préactivations et à la dernière couche cachée (équations (2.53) à (2.55)) sont similaires à ceux que l'on retrouve pour un réseau multicouche (équations (2.17) à (2.19)).

Dans un réseau de neurones récurrent, les gradients des équations (2.55) et (2.56) sont utilisés pour obtenir les gradients de $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ par rapport aux paramètres du modèle

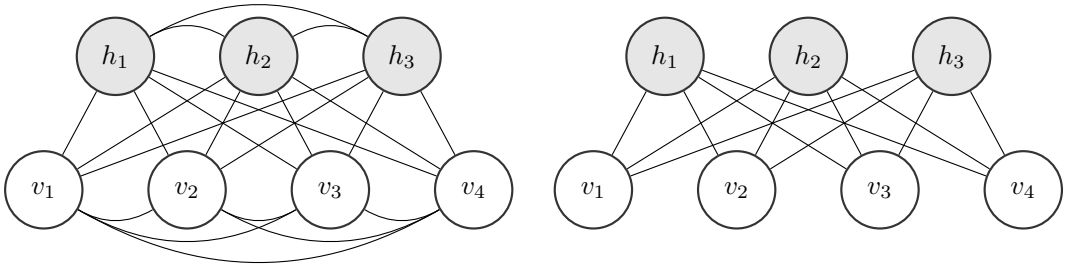
$$\nabla_{\boldsymbol{U}^{(l)}} \mathcal{L} = \sum_{t=1}^{T} \left(\left(\nabla_{\boldsymbol{a}_t^{(l)}} \mathcal{L} \right) \otimes \boldsymbol{h}_{t-1}^{(l)^\top} \right) \tag{2.57}$$

$$\nabla_{\boldsymbol{W}^{(l)}} \mathcal{L} = \sum_{t=1}^{T} \left(\left(\nabla_{\boldsymbol{a}_t^{(l)}} \mathcal{L} \right) \otimes \boldsymbol{h}_t^{(l-1)^\top} \right) \tag{2.58}$$

$$\nabla_{\boldsymbol{b}^{(l)}} \mathcal{L} = \sum_{t=1}^{T} \left(\nabla_{\boldsymbol{a}_t^{(l)}} \mathcal{L} \right). \tag{2.59}$$

2.4 Machine de Boltzmann restreinte

La machine de Boltzmann restreinte (RBM) [Hinton, 2002] est un réseau de neurones stochastiques formé de deux couches où les unités sont des variables de Bernoulli (voir figure 2.7(b)). La couche visible $\boldsymbol{v} \in \{0,1\}^D$ représente les données observées possédant D dimensions. La couche cachée $\boldsymbol{h} \in \{0,1\}^K$ représente K données latentes. Bien qu'il existe une extension de la RBM, la *Gaussian RBM* [Hinton and Salakhutdinov, 2006], dans laquelle les variables visibles sont gaussiennes, dans cette thèse nous nous limiterons à la version binaire.



(a) Machine de Boltzmann (b) Machine de Boltzmann restreinte

figure 2.7 – Comparatif entre les modèles graphiques de la machine de Boltzmann et la machine de Boltzmann restreinte.

Comme dans un réseau multicouche, les deux couches d'une RBM sont reliées par une matrice de poids $\boldsymbol{W}$. Par contre, dans la RBM les connexions entre les unitées des couches sont non dirigées. La RBM possède également un biais pour chacune de ses unitées, c'est-à-dire $\boldsymbol{b}^{\mathrm{v}} \in \mathbb{R}^D$ pour les unités visibles et $\boldsymbol{b}^{\mathrm{h}} \in \mathbb{R}^K$ pour les unités cachées. Contrairement à la machine de Boltzmann [Ackley et al., 1985], liens intracouches d'une RBM sont restreints à zéro, c'est-à-dire qu'il n'y a aucun lien entre les unités d'une même couche (voir figure 2.7). Par ailleurs, deux extensions à la RBM sont proposées au chapitre 3.

2.4.1 Structure

La RBM est un modèle génératif où la distribution de probabilités jointes $p(\boldsymbol{v}, \boldsymbol{h})$ est une distribution de Boltzmann. En physique, cette distribution décrit la dispersion des particules entre les divers états d'un système. Pour la RBM, un état $\{\boldsymbol{v}, \boldsymbol{h}\}$ représente une configuration que peuvent prendre les variables du modèle. La fonction d'énergie $E(\boldsymbol{v}, \boldsymbol{h})$ permet de mesurer l'énergie requise pour qu'un système soit dans l'état spécifié. Elle est donnée par l'équation suivante

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \boldsymbol{b}^{\mathrm{v}} - \boldsymbol{h}^\top \boldsymbol{b}^{\mathrm{h}} - \boldsymbol{h}^\top \boldsymbol{W} \boldsymbol{v}, \tag{2.60}$$

où $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}^{\mathrm{h}}, \boldsymbol{b}^{\mathrm{v}}\}$ sont les paramètres du modèle.

L'équation (2.60) permet d'associer une valeur à chaque état du système. Idéalement,

pour un modèle avec des paramètres optimaux, on s'attend à ce que la demande en énergie soit faible pour les configurations désirables (ex. celles associées à des données de l'ensemble d'entrainement) mais grande pour les autres configurations. D'ailleurs, la distribution de probabilités jointes $p(\boldsymbol{v}, \boldsymbol{h})$ est définie en fonction de l'énergie E associée aux différents états du modèle. On a

$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \tag{2.61}$$

où

$$Z = \sum_{\boldsymbol{v}'} \sum_{\boldsymbol{h}'} e^{-E(\boldsymbol{v}', \boldsymbol{h}')} = \sum_{\boldsymbol{v}'} e^{-F(\boldsymbol{v}')} \tag{2.62}$$

est la fonction de partition, c'est-à-dire la constante de normalisation obtenue en tenant compte de tous les états possibles du système, soit $2^D \times 2^K$ configurations.

Également inspirée de la physique, l'énergie libre (*free energy*)

$$F(\boldsymbol{v}) = -\ln \sum_{\boldsymbol{h} \in \{0,1\}^K} e^{-E(\boldsymbol{v}, \boldsymbol{h})} \tag{2.63}$$

représente la charge maximale de travail que peut accomplir un système. D'un point de vue mathématique, il s'agit d'une marginalisation d'énergies dans l'espace logarithmique. Plus formellement, F est le négatif de la log-probabilité non normalisée d'observer $\boldsymbol{v}$, ce qui donne

$$p(\boldsymbol{v}) = \frac{1}{Z} e^{-F(\boldsymbol{v})}. \tag{2.64}$$

En examinant l'équation (2.62) de plus près, on s'aperçoit que plus il y a d'unités, plus la fonction de partition Z devient **intraitable**. À titre d'exemple, une petite RBM de 100 unités cachées entrainées sur de petites images 28×28 pixels constitue environ 10^{266} configurations possibles. Il s'avère que c'est beaucoup plus que le nombre d'atomes dans l'univers observable (estimé à 10^{80} [Wikipedia, 2017]) ! Pour cette raison, il n'est pas envisageable d'évaluer $p(\boldsymbol{v})$ pour des modèles complexes.

Néanmoins, la RBM est un modèle génératif qui apprend à échantillonner des données provenant de $p(\boldsymbol{v})$, bien qu'on ne soit pas en mesure de l'évaluer. L'échantillonnage se fait à l'aide de méthodes de Monte-Carlo par chaines de Markov (MCMC). L'opérateur de transition souvent employé pour la RBM est l'échantillonnage de Gibbs

(*Gibbs Sampling* [Murphy, 2012, chap. 24.2]).

Puisque dans une RBM il n'y a pas de connexions intracouches, les unités cachées sont mutuellement indépendantes sachant la couche visible et les unités visibles sont mutuellement indépendantes sachant la couche cachée. Ceci permet de changer d'état dans la chaine de Markov en utilisant les probabilités conditionnelles suivantes

$$p(\boldsymbol{h}|\boldsymbol{v}) = \prod_{i}^{D} p(h_i|\boldsymbol{v}) \tag{2.65}$$

$$p(\boldsymbol{v}|\boldsymbol{h}) = \prod_{j}^{K} p(v_j|\boldsymbol{h}) \tag{2.66}$$

Puisque les unités du modèle sont binaires, les probabilités conditionnelles s'écrivent à l'aide de la fonction logistique (section 2.1.2) de la façon suivante

$$p(h_i = 1|\boldsymbol{v}) = \text{sigm}\left(b_i^{\text{h}} + \sum_{j=1}^{D} w_{i,j} v_j\right) \tag{2.67}$$

$$p(v_j = 1|\boldsymbol{h}) = \text{sigm}\left(b_j^{\text{v}} + \sum_{i=1}^{K} w_{i,j} h_i\right). \tag{2.68}$$

2.4.2 L'apprentissage

La RBM est propice à l'utilisation d'apprentissage non supervisé pour optimiser les paramètres du modèle. Contrairement à des modèles comme l'autoencodeur [Goodfellow et al., 2016, chap. 14] et le codage parcimonieux [Lee et al., 2007], l'apprentissage de la RBM ne se fait pas en évaluant la qualité de reconstruction des données. Elle consiste plutôt à minimiser la fonction d'énergie pour les exemples de l'ensemble d'entrainements $\mathcal{D} = \{\boldsymbol{v}_n\}_{n=1}^{N}$ relativement à d'autres configurations possibles de $\{\boldsymbol{v} \in \mathbb{R}^D | \boldsymbol{v} \notin \mathcal{D}\}$. Autrement dit, il s'agit de diminuer la surface de la fonction d'énergie pour les données désirées comparativement aux autres données possibles. De façon équivalente, ceci revient à maximiser la probabilité d'observer ces exemples

et donc à minimiser leur log-probabilité négative

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} - \ln p(\boldsymbol{v}_n). \tag{2.69}$$

Généralement, un algorithme d'optimisation de premier ordre, en l'occurrence la *descente de gradient stochastique*, est utilisé afin de trouver une solution à (2.69). Le gradient de la fonction objectif se décompose en deux phases

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) &= \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} F(\boldsymbol{v}) + \nabla_{\boldsymbol{\theta}} \ln Z \\ &= \underbrace{\frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} F(\boldsymbol{v}_n)}_{\text{Phase positive}} - \underbrace{\sum_{\boldsymbol{v}'} p(\boldsymbol{v}') \nabla_{\boldsymbol{\theta}} F(\boldsymbol{v}')}_{\text{Phase négative}}. \end{aligned} \tag{2.70}$$

La phase positive s'occupe d'augmenter la probabilité des exemples d'entrainement alors que la phase négative s'occupe de réduire la probabilité des exemples générés par le modèle.

Notons que la phase négative consiste à calculer l'espérance de l'énergie libre associée à toutes les configurations du modèle. Par conséquent, cette espérance est **intraitable** et doit être approximée. L'espérance sera donc calculée sur un échantillon $\mathcal{S} = \{\tilde{\boldsymbol{v}}_s \sim p(\boldsymbol{v})\}_{s=1}^{S}$ d'exemples générés par le modèle. Cela permet d'obtenir une approximation du gradient comme suit

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}, \mathcal{D}\right) \approx \underbrace{\frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} F(\boldsymbol{v}_n)}_{\text{Phase positive}} - \underbrace{\frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\theta}} F(\tilde{\boldsymbol{v}}_s)}_{\text{Phase négative}}. \tag{2.71}$$

2.4.3 Échantillonnage

Brièvement mentionné plus haut, l'échantillonnage de Gibbs permet d'obtenir des exemples provenant de $p(\boldsymbol{v})$. Par contre, pour avoir de « bons » échantillons, c'est-à-dire fidèles à la distribution apprise, il faut théoriquement itérer entre les états de la

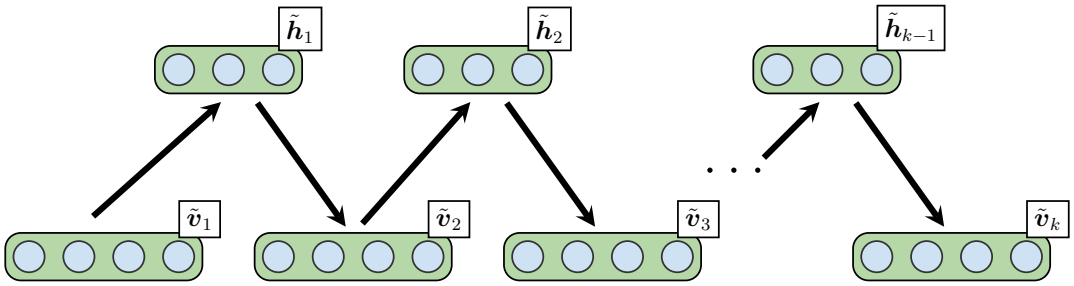


figure 2.8 – Processus itératif de l'échantillonnage de Gibbs. Une itération de Gibbs consiste en l'échantillonnage des unités cachées $\tilde{\boldsymbol{h}}_i \sim p(\boldsymbol{h}|\tilde{\boldsymbol{v}}_i)$ suivi de l'échantillonnage des unités visibles $\tilde{\boldsymbol{v}}_{i+1} \sim p(\boldsymbol{v}|\tilde{\boldsymbol{h}}_i)$.

chaine de Markov jusqu'à convergence. La figure 2.8 illustre l'alternance entre l'échantillonnage des unités cachées $\tilde{\boldsymbol{h}}_k \sim p(\boldsymbol{h}|\tilde{\boldsymbol{v}}_k)$ et l'échantillonnage des unités visibles $\tilde{\boldsymbol{v}}_{k+1} \sim p(\boldsymbol{v}|\tilde{\boldsymbol{h}}_k)$. Ceci reste un processus très long, mais est au moins calculable.

En 2002, Hinton a proposé une méthode permettant d'accélérer cette phase d'échantillonnage : la divergence contrastive (CD) [Hinton, 2002]. La première astuce est d'utiliser un exemple d'entrainement pour initialiser l'échantillonnage de Gibbs, c'est-à-dire $\tilde{\boldsymbol{v}}_1 = \boldsymbol{v}$ où $\boldsymbol{v} \in \mathcal{D}$. Cela permet d'explorer les régions se trouvant à proximité des exemples d'entrainement afin d'y modifier la surface de la fonction d'énergie. La seconde amélioration consiste simplement à arrêter avant d'avoir convergé, c'est-à-dire après k itérations de Gibbs (CDk).

Chapitre 3

Machine de Boltzmann restreinte à capacité variable

“ To infinity and beyond.

Buzz Lightyear, 1995 ”

Résumé

Dans cet article, nous présentons une nouvelle formulation de la machine de Boltzmann restreinte (RBM) qui évite d'avoir à spécifier le nombre d'unités cachées. En fait, elle permet au modèle d'adapter la taille de sa couche cachée en fonction des observations. Nous montrerons d'abord comment obtenir une version de la RBM sensible à l'ordonnance de ses unitées cachées que l'on nomme Ordered RBM (oRBM). Par la suite, nous démontrerons qu'en définissant judicieusement sa fonction d'énergie, un modèle basé sur la RBM peut croître jusqu'à l'infini tout en restant défini mathématiquement. Cette variante se nomme Infinite RBM (iRBM). Tout comme pour la RBM, le maximum de vraisemblance peut être utilisé comme fonction de perte pour entrainer ces nouvelles variantes. Durant l'entrainement, le modèle ajoute des unités à sa couche cachée ce qui lui permet d'augmenter sa capacité d'apprentissage. Nous avons étudié empiriquement le comportement de la machine de Boltzmann restreinte à capacité variable, montrant qu'elle permet d'obtenir des résultats comparables à ceux de la RBM tout en ne requièrant pas de spécifier la taille de la couche cachée.

Commentaires

L'article a été publié dans le journal (*Neural Computation*) en 2016. Mes contributions pour cet article sont les suivantes :

- Développer, en Python, la oRBM et la iRBM en utilisant la librairie

Theano [The Theano Development Team et al., 2016]. L'implémentation de la iRBM reste efficace malgré le fait que la taille du modèle s'adapte durant l'entrainement. Aussi, puisque ce sont des modèles génératifs (section 1.3.2), le code supporte la génération de nouvelles données à partir d'un modèle déjà entrainé. Le code source pour reproduire les expériences est disponible publiquement sur mon répertoire GitHub (`https://github.com/MarcCote/iRBM`).

- Explorer différentes structures de couche cachée pour la oRBM et la iRBM. Notamment, durant la phase exploratoire j'ai expérimenté les effets d'imposer une structure d'arbre binaire à la couche cachée ainsi qu'une structure étagée. Malheureusement, les modèles se sont avérés plus difficiles à entrainer et nous avons finalement décidé d'opter pour une structure linéaire.
- Évaluer la performance de la RBM, la oRBM et la iRBM sur les jeux de données *Binarized MNIST* et *CalTech101 Silhouettes*. Il s'agit de deux tâches d'estimation de densité de probabilité (section 1.1.3). L'apprentissage des modèles s'est fait de façon non supervisée (section 1.2.2).
- Contribuer au développement de la librarie Theano en ajoutant, entre autres, l'opération de la somme cumulative qui était nécessaire à l'implémentation des modèles présentés dans cet article.
- Développer un outil de visualisation permettant d'observer le poids des connexions associées à chaque unité cachée du modèle pendant l'apprentissage. Cet outil a permis de générer le vidéo mentionné dans l'article (`http://goo.gl/LGQDaI`).
- Rédiger l'article conjointement avec Hugo Larochelle.

An Infinite Restricted Boltzmann Machine

Marc-Alexandre Côté
Département d'informatique
Université de Sherbrooke
marc-alexandre.cote@usherbrooke.ca

Hugo Larochelle
Département d'informatique
Université de Sherbrooke
hugo.larochelle@usherbrooke.ca

Keywords: restricted boltzmann machine, machine learning, unsupervised learning, neural network, generative model

Abstract

We present a mathematical construction for the restricted Boltzmann machine (RBM) that doesn't require specifying the number of hidden units. In fact, the hidden layer size is adaptive and can grow during training. This is obtained by first extending the RBM to be sensitive to the ordering of its hidden units. Then, thanks to a carefully chosen definition of the energy function, we show that the limit of infinitely many hidden units is well defined. As with RBM, approximate maximum likelihood training can be performed, resulting in an algorithm that naturally and adaptively adds trained hidden units during learning. We empirically study the behaviour of this infinite RBM, showing that its performance is competitive to that of the RBM, while not requiring the tuning of a hidden layer size.

3.1 Introduction

Over the years, machine learning research has produced a large variety of latent variable probabilistic models. These include mixture models, factor analysis models, latent dynamical models, and many others. Such models usually require that the dimensionality of the latent representation be specified and fixed during learning. Adapting this quantity is then considered as a separate process, that takes the form of model selection and is normally treated as an additional hyper-parameter to tune.

For this reason, more recently, there has been a lot of work on extending these models such that the size of the representation can be treated as an adaptive quantity during training. These extensions, often referred to as "infinite" models, are non-parametric in nature where the latent space is infinite with probability 1 and can arbitrarily adapt their capacity to the training data (see Orbanz and Teh [2010] for a brief overview).

While most latent variable models have been extended to one or more infinite variants, a notable exception is the restricted Boltzmann machine (RBM). The RBM is an undirected graphical model for binary vector observations, where the latent representation is itself a binary vector (i.e. hidden layer). The RBM (and its extensions to non-binary vectors) have been successfully applied to a variety of problems and data, such as images [Ranzato et al., 2006], movie user preferences [Salakhutdinov et al., 2007], motion capture [Taylor et al., 2011], text [Dahl et al., 2012] and many others. One explanation for the lack of literature on RBMs with an adaptive hidden layer size comes from its undirected nature. Indeed, undirected models tend to be less amenable to a Bayesian treatment of learning, on which relies the majority of the literature on infinite models.

Our main contribution in this paper is thus a proposal for an infinite RBM which can adapt the effective number of hidden units during training. While our proposal is not based on a Bayesian formulation, it does correspond to the infinite limit of a finite-sized model and behaves in such a way that it effectively adapts its capacity as training progresses.

First, we propose a finite extension of the RBM that is sensitive to the position of each unit in its hidden layer. This is achieved by introducing a random variable that represents the number of hidden units intervening in the RBM's energy function. Then, thanks to the introduction of an energy cost for using each additional unit, we show that taking the infinite limit of the total number of hidden units is well defined. We describe an approximate maximum likelihood training algorithm for this infinite RBM, based on (Persistent) Contrastive Divergence, which results in a procedure where hidden units are implicitly added as training progresses. Finally, we empirically report how this model behaves in practice and show that it can achieve

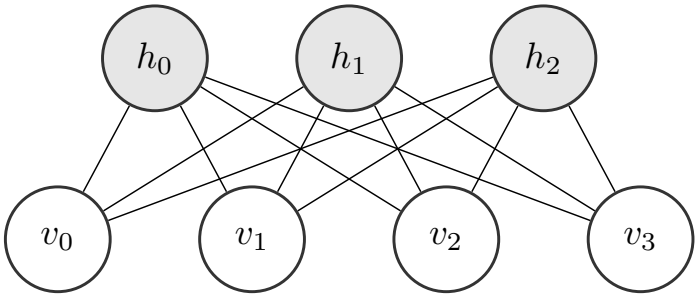


Figure 3.1: Graphical model of the restricted Boltzmann Machine. Inter-connections between visible units and hidden units using symmetric weights.

performance that is competitive to a traditional RBM on the binarized MNIST and Caltech101 Silhouettes datasets, while not requiring the tuning of a hyper-parameter for its hidden layer size.

3.2 Restricted Boltzmann Machine

We describe the basic RBM model, which we'll build on to derive its ordered and infinite versions.

An RBM is a generative stochastic neural network composed of two layers: visible $\boldsymbol{v}$ and hidden $\boldsymbol{h}$. These layers are fully connected to each other, while connections within a layer are not allowed. This means each unit v_i is connected to all h_j units via undirected weighted connections (Figure 3.1).

Given a binary RBM with D visible units and K hidden units, the set of visible vectors is $\mathbb{V} = \{0,1\}^D$, whereas the set of hidden vectors is $\mathbb{H} = \{0,1\}^K$. In an RBM model, each configuration $(\boldsymbol{v}, \boldsymbol{h}) \in \mathbb{V} \times \mathbb{H}$ has an associated energy value defined by the following function:

$$\mathrm{E}(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{h}^T \boldsymbol{W} \boldsymbol{v} - \boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \boldsymbol{h}^T \boldsymbol{b}^{\mathrm{h}} \tag{3.1}$$

The parameters $\Theta = \{\boldsymbol{W}, \boldsymbol{b}^{\mathrm{v}}, \boldsymbol{b}^{\mathrm{h}}\}$ of this model are the weights $\boldsymbol{W}$ ($K \times D$ matrix), the visible unit biases $\boldsymbol{b}^{\mathrm{v}}$ ($D \times 1$ vector) and the hidden unit biases $\boldsymbol{b}^{\mathrm{h}}$ ($K \times 1$ vector).

A probability distribution over visible and hidden vectors is defined in terms of this

energy function:

$$P_\Theta(\boldsymbol{v}, \boldsymbol{h}) = \frac{1}{Z} e^{-\mathrm{E}(\boldsymbol{v}, \boldsymbol{h})} \tag{3.2}$$

with

$$Z = \sum_{\boldsymbol{v}' \in \mathbb{V}} \sum_{\boldsymbol{h}' \in \mathbb{H}} e^{-\mathrm{E}(\boldsymbol{v}', \boldsymbol{h}')}. \tag{3.3}$$

We see from Equation (3.3) that the partition function Z (normalizing constant) is intractable, as it requires summing over all possible $2^{(D+K)}$ configurations.

The probability distribution of a visible vector is obtained by marginalizing over all configurations of hidden vectors. One property of the RBM is that the numerator of the marginal $P_\Theta(\boldsymbol{v})$ is tractable:

$$P_\Theta(\boldsymbol{v}) = \frac{1}{Z} \sum_{\boldsymbol{h}' \in \mathbb{H}} e^{-\mathrm{E}(\boldsymbol{v}, \boldsymbol{h}')} = \frac{1}{Z} e^{-F(\boldsymbol{v})} \tag{3.4}$$

with

$$F(\boldsymbol{v}) = -\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \sum_{i=1}^{K} \mathrm{soft}_+(\boldsymbol{W}_{i\cdot} \boldsymbol{v} + b_i^{\mathrm{h}}) \tag{3.5}$$

where $\mathrm{soft}_+(x) = \ln(1 + e^x)$ and the notation $\boldsymbol{W}_{i\cdot}$ designates the i^{th} row of $\boldsymbol{W}$, likewise for columns $\boldsymbol{W}_{\cdot j}$. This allows for an equivalent definition of the RBM model in terms of what is known as the free energy $F(\boldsymbol{v})$. However, the partition function still requires summing over all configurations of visible vectors, which is intractable even for moderate values of D.

RBMs can be learned as generative models, to assign high probability (i.e. low energy) to training observations and low probability otherwise. One approach is to minimize the average negative log-likelihood (NLL) for a set of examples $\mathcal{D} = \{\boldsymbol{v}_n\}_{n=1}^N$:

$$f(\Theta, \mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} - \ln P_\Theta(\boldsymbol{v}_n). \tag{3.6}$$

The gradient of this objective has a simple form, which is often referred to as the

combination of positive and negative phases:

$$\nabla_\theta f(\Theta, \mathcal{D}) = \underbrace{\frac{1}{N}\sum_{n=1}^{N} \nabla_\theta F(\boldsymbol{v}_n)}_{\text{Positive phase}} - \underbrace{\sum_{\boldsymbol{v}' \in \mathbb{V}} P_\Theta(\boldsymbol{v}') \nabla_\theta F(\boldsymbol{v}')}_{\text{Negative phase}} \tag{3.7}$$

where

$$\nabla_{\boldsymbol{W}} F(\boldsymbol{v}) = -\mathbb{E}[\boldsymbol{h}|\boldsymbol{v}]\boldsymbol{v}^T = -\widehat{\boldsymbol{h}}(\boldsymbol{v})\boldsymbol{v}^T \tag{3.8}$$

$$\nabla_{\boldsymbol{b}^{\mathrm{h}}} F(\boldsymbol{v}) = -\mathbb{E}[\boldsymbol{h}|\boldsymbol{v}] = -\widehat{\boldsymbol{h}}(\boldsymbol{v}) \tag{3.9}$$

$$\nabla_{\boldsymbol{b}}^{\mathrm{v}} F(\boldsymbol{v}) = -\boldsymbol{v} \tag{3.10}$$

and where $\widehat{\boldsymbol{h}}(\boldsymbol{v}) = \boldsymbol{\sigma}\left(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}^{\mathrm{h}}\right)$ with $\boldsymbol{\sigma}(\cdot)$ being the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ applied element-wise. Derivation for the partial derivatives can be found in Appendix 3.A.

Intuitively, the positive phase pushes up the probability of examples coming from our training set, whereas the negative phase lowers the probability of examples generated by the model. Much like the partition function, the negative phase is intractable. To overcome this we approximate the expectation under $P_\Theta(\boldsymbol{v})$ with an average of S samples $\mathcal{S} = \{\hat{\boldsymbol{v}}_s\}_{s=1}^{S}$ drawn from $P_\Theta(\boldsymbol{v})$ i.e. the model.

$$\nabla_\theta f(\Theta, \mathcal{D}) \approx \underbrace{\frac{1}{N}\sum_{n=1}^{N} \nabla_\theta F(\boldsymbol{v}_n)}_{\text{Positive phase}} - \underbrace{\frac{1}{S}\sum_{s=1}^{S} \nabla_\theta F(\hat{\boldsymbol{v}}_s)}_{\text{Negative phase}} \tag{3.11}$$

Moreover, mini-batch training is usually employed and consists in replacing the positive phase average by one over a small subset of the training set, different for every training update.

Sampling from $P_\Theta(\boldsymbol{v})$ can be achieved using block Gibbs sampling, by alternating between sampling $\boldsymbol{v} \sim P_\Theta(\boldsymbol{v}|\boldsymbol{h})$ and $\boldsymbol{h} \sim P_\Theta(\boldsymbol{h}|\boldsymbol{v})$. It can be done efficiently because RBMs have no connections within a layer, meaning that hidden units are conditionally independent given the visible units and vice versa. The conditional distributions of a

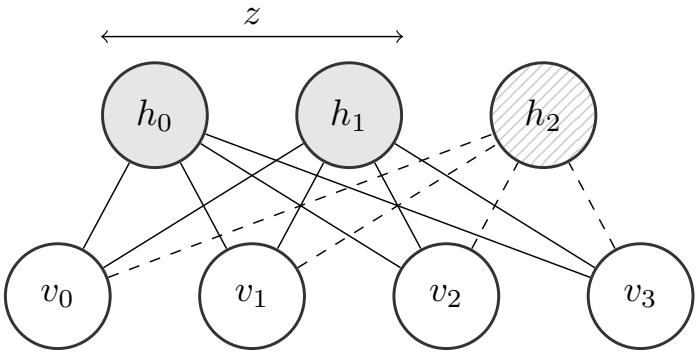


Figure 3.2: Illustration of the ordered RBM. Since $z = 2$ only the first two hidden units are selected.

binary RBM are Bernoulli distributions with parameters

$$P_\Theta(h_i = 1|\boldsymbol{v}) = \sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) \tag{3.12}$$

$$P_\Theta(v_j = 1|\boldsymbol{h}) = \sigma(\boldsymbol{h}^T\boldsymbol{W}_{\cdot j} + \boldsymbol{b}_j^{\mathrm{v}}) \tag{3.13}$$

In theory, the Markov chain should be run until equilibrium before drawing a sample for every training update, which is highly inefficient. Thus, Contrastive Divergence (CD) learning is often employed, where we initialize the update's Gibbs chains to the training examples and only perform T steps of Gibbs sampling [Hinton, 2002]. Another approach, referred to as stochastic approximation or Persistent CD (PCD) [Tieleman, 2008], is to not reinitialize the Gibbs chains between updates.

3.3 Ordered Restricted Boltzmann Machine

The model we propose is a variant of the RBM where the hidden units $\boldsymbol{h}$ are ordered from left to right, with this order being taken into account by the energy function. We refer to this model as an ordered RBM (oRBM). As shown in Figure 3.2, the oRBM takes hidden unit order into account by introducing a random variable z that can be understood as the effective number of hidden units participating to the energy. Hidden units are selected starting from the left and the selection of each hidden unit is associated with an incremental cost in energy.

Concretely, we define the energy function of the oRBM as

$$\mathrm{E}(\boldsymbol{v}, \boldsymbol{h}, z) = -\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \sum_{i=1}^{z} \left(h_i(\boldsymbol{W}_{i.}\boldsymbol{v} + b_i^{\mathrm{h}}) - \beta_i \right) \tag{3.14}$$

where z represents the number of selected hidden units that are active and β_i is a energy penalty for selecting each i^{th} hidden unit. As we will see, carefully parametrizing the per unit energy penalty will allow us to consider the case of an infinite pool of hidden units.

In our experiments, as we wanted the filters of each unit to be the dominating factor in a unit being selected, we parametrized it as $\beta_i = \beta \, \mathrm{soft}_+(b_i^{\mathrm{h}})$, where β is a global hyper-parameter (critically, as we'll discuss later, this hyper-parameter doesn't actually require tuning and a generic value for it works fine). Intuitively, the penalty term acts as a form of regularization since it forces the model to avoid using more hidden units than needed, prioritizing smaller networks.

Moreover, having the penalty depending on the hidden biases also implies that the selection of a hidden unit (i.e. influencing the outcome of the random variable z) will be mostly controlled by the values taken by the connections $\boldsymbol{W}$. Higher values of the bias of a hidden unit will not increase its probability of being selected. In other words, for the model to increase its capacity and better fit the training data, it will have to learn better filters. Note that alternative parametrizations could certainly be considered.

As with the RBM, $P_\Theta(\boldsymbol{v})$ is defined in terms of its energy function. For this, we have to specify the set of legal values for $\boldsymbol{v}$, $\boldsymbol{h}$ and z. Since, for a given z, the value of the energy is irrelevant for the dimensions of $\boldsymbol{h}$ from z to K, we will assume they are set to 0. There is thus a coupling between the value of z and the legal values of $\boldsymbol{h}$. We will note $\mathbb{H}_z = \{\boldsymbol{h} \in \mathbb{H} | h_k = 0 \ \forall k > z\}$ the legal values of $\boldsymbol{h}$ for a given z. As for z, it can vary in $\{1, \ldots, K\}$, and $\boldsymbol{v} \in \mathbb{V}$ as usual.

The joint probability over $\boldsymbol{v}$, $\boldsymbol{h}$ and z is thus:

$$P_\Theta(\boldsymbol{v}, \boldsymbol{h}, z) = \frac{1}{Z} e^{-\mathrm{E}(\boldsymbol{v}, \boldsymbol{h}, z)} \tag{3.15}$$

where

$$Z = \sum_{z'=1}^{K} \sum_{\boldsymbol{v}' \in \mathbb{V}} \sum_{\boldsymbol{h}' \in \mathbb{H}_{z'}} e^{-\mathrm{E}(\boldsymbol{v}', \boldsymbol{h}', z')}. \tag{3.16}$$

As for the marginal distribution $P_\Theta(\boldsymbol{v})$ of the oRBM model, it can also be written in terms of a free energy. Indeed, in a derivation similar to the case of the RBM, we can show:

$$P_\Theta(\boldsymbol{v}) = \frac{1}{Z} \sum_{z=1}^{K} \sum_{\boldsymbol{h} \in \mathbb{H}_z} e^{-\mathrm{E}(\boldsymbol{v}, \boldsymbol{h}, z)} = \frac{1}{Z} \sum_{z=1}^{K} e^{-F(\boldsymbol{v}, z)} \tag{3.17}$$

$$F(\boldsymbol{v}, z) = -\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \sum_{i=1}^{z} \left(\mathrm{soft}_+(\boldsymbol{W}_{i\cdot} \boldsymbol{v} + b_i^{\mathrm{h}}) - \beta_i \right) \tag{3.18}$$

This gives us a free energy where only the hidden units have been marginalized. We can also derive a formulation where the free energy depends only on $\boldsymbol{v}$:

$$P_\Theta(\boldsymbol{v}) = \frac{1}{Z} \sum_{z=1}^{K} e^{-F(\boldsymbol{v}, z)} = \frac{1}{Z} e^{-F(\boldsymbol{v})} \quad \text{with} \quad F(\boldsymbol{v}) = -\ln \left(\sum_{z=1}^{K} e^{-F(\boldsymbol{v}, z)} \right) \tag{3.19}$$

It should be noticed that, in the oRBM, z does not correspond to the number of hidden units assumed to have generated *all* observations. Instead, the model allows for different observations having been generated by a different number of hidden units. Specifically, for a given $\boldsymbol{v}$, the conditional distribution over the corresponding value of z is

$$P_\Theta(z|\boldsymbol{v}) = \frac{\exp(-F(\boldsymbol{v}, z))}{\sum_{z'=1}^{K} \exp(-F(\boldsymbol{v}, z'))} \,. \tag{3.20}$$

As for the conditional distribution over the hidden units, given a value of z it takes the same form as for the regular RBM, except for unselected hidden units which are forced to zero. Similarly, the distribution of $\boldsymbol{v}$ given a value of the hidden layer and z reflects that of the RBM:

$$P_\Theta(h_i = 1|\boldsymbol{v}, z) = \begin{cases} \sigma(\boldsymbol{W}_{i\cdot} \boldsymbol{v} + b_i^{\mathrm{h}}) & \text{if } i \leq z \\ 0 & \text{otherwise} \end{cases} \tag{3.21}$$

$$P_\Theta(v_j = 1|\boldsymbol{h}, z) = \sigma \left(\sum_{i=1}^{z} W_{ij} h_i + b_j^{\mathrm{v}} \right) \tag{3.22}$$

To train the oRBM, we can also rely on CD or PCD for estimating the gradients based on Equation 3.11 but using $F(\boldsymbol{v})$ as defined in equation 3.19. Defining $\mathbf{1}_{\leq z} = [\overbrace{1, \ldots, 1}^{z}, 0, \ldots, 0]^T$ and $\mathbf{cdf}(z|\boldsymbol{v}) = [P_\Theta(z < 1|\boldsymbol{v}), \ldots, P_\Theta(z < K|\boldsymbol{v})]^T$ with $\odot$ denoting the element-wise product, the free energy gradients are then slightly modified as follows:

$$\nabla_{\boldsymbol{W}} F(\boldsymbol{v}) = -\mathbb{E}_{\boldsymbol{h},z}[\boldsymbol{h} \odot \mathbf{1}_{\leq z}|\boldsymbol{v}]\boldsymbol{v}^T = -(\widehat{\boldsymbol{h}}(\boldsymbol{v}) \odot (1 - \mathbf{cdf}(z|\boldsymbol{v})))\boldsymbol{v}^T \tag{3.23}$$

$$\nabla_{\boldsymbol{b}^{\mathrm{h}}} F(\boldsymbol{v}) = -\mathbb{E}_{\boldsymbol{h},z}[(\boldsymbol{h} - \beta\boldsymbol{\sigma}\left(\boldsymbol{b}^{\mathrm{h}}\right)) \odot \mathbf{1}_{\leq z}|\boldsymbol{v}] = -(\widehat{\boldsymbol{h}}(\boldsymbol{v}) - \beta\boldsymbol{\sigma}\left(\boldsymbol{b}^{\mathrm{h}}\right) \odot (1 - \mathbf{cdf}(z|\boldsymbol{v})) \tag{3.24}$$

$$\nabla_{\boldsymbol{b}^{\mathrm{v}}} F(\boldsymbol{v}) = -\boldsymbol{v} \tag{3.25}$$

with $\widehat{\boldsymbol{h}}(\boldsymbol{v}) = \boldsymbol{\sigma}\left(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}^{\mathrm{h}}\right)$. Derivation for the partial derivatives can be found in Appendix 3.A.

Compared to the RBM, computing these gradients requires one additional quantity: the vector of cumulative probabilities $\mathbf{cdf}(z|\boldsymbol{v})$. Fortunately, this quantity can be efficiently computed, in $\mathcal{O}(K)$, by first computing the required probabilities vector $P_\Theta(z|\boldsymbol{v})$ and performing a cumulative sum.

Sampling from $P_\Theta(\boldsymbol{v})$ slightly differs from the RBM as we need to consider z in the Markov chain. With the oRBM, Gibbs steps alternate between sampling $(\boldsymbol{h}, z) \sim P_\Theta(\boldsymbol{h}, z|\boldsymbol{v})$ and $\boldsymbol{v} \sim P_\Theta(\boldsymbol{v}|\boldsymbol{h}, z)$. Sampling from $P_\Theta(\boldsymbol{h}, z|\boldsymbol{v})$ is done in two steps: $z \sim P_\Theta(z|\boldsymbol{v})$ followed by $\boldsymbol{h} \sim P_\Theta(\boldsymbol{h}|\boldsymbol{v}, z)$.

During training, what we observe is that the hidden units are each trained gradually, in sequence, from left to right. This effect is mainly due to the multiplicative term $(1 - \mathbf{cdf}(z|\boldsymbol{v}))$ in the hidden unit parameter updates of Equations 3.23 and 3.24, which is monotonically decreasing. Effectively, the model is thus growing in capacity during training, until its maximum capacity of K hidden units.

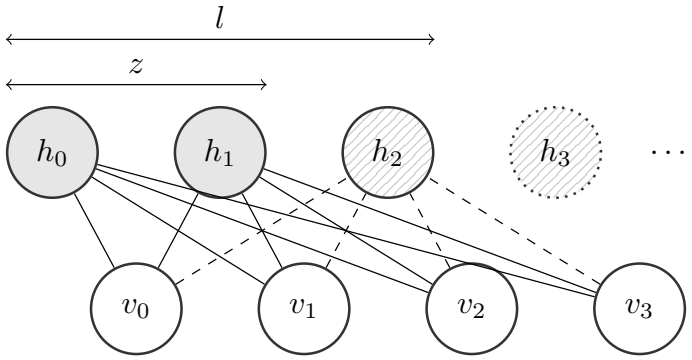


Figure 3.3: Illustration of the infinite RBM. With $z = 2$, only the first two hidden units are currently selected. The dashed lines illustrate that there are connections that are trained (non-zero) with the third hidden unit. All (infinitely many) hidden units after the third have zero-valued weights, which correspond to l being equal to 3.

3.4 Infinite Restricted Boltzmann Machine

The growing behaviour of the oRBM begs for the question: could we achieve a similar effect without having to specify a maximum capacity to the model? Indeed, while Montufar and Ay [2011] have shown that with $2^{V-1} - 1$ hidden units an RBM is a universal approximator, a variant of the RBM that could automatically increase its capacity until it is sufficiently high is likely to yield much smaller models in practice.

It turns out that this is possible by taking the limit of $K \to \infty$. For this reason, we refer to this model as the infinite RBM (iRBM).

This limit is made possible thanks to two modeling choices. The first is the assumption that a finite (but variable!) number of hidden units have non-zero weights and biases. This is trivial to ensure, for any optimization procedure, using any amount of any type of weight decay (e.g. L2 or L1 regularization) on all the weights and hidden biases. An infinite number of non-zero weights and biases could then correspond to an infinite penalty, so no proper optimization would ever diverge to this solution, no matter the initialization. This is guaranteed when using L1 regularization, thanks to its sparsity inducing property. As for L2 regularization, while it could theoretically

lead to an infinite number of hidden units (e.g. if the L2 norm of the parameters associated with each hidden unit decreases exponentially with respect to the position of the hidden unit), in practice the floating precision would clip very small parameters to zero, thus having a finite number of hidden units.

The second key choice is our parametrization of the per-unit energy penalty β_i, which will ensure that the infinite sums required in computing probabilities will be convergent. For instance, consider the conditional $P_\Theta(z|\boldsymbol{v})$:

$$P_\Theta(z|\boldsymbol{v}) = \frac{\exp(-F(\boldsymbol{v}, z))}{Z(\boldsymbol{v})} = \frac{\exp(-F(\boldsymbol{v}, z))}{\sum_{z'=1}^{\infty} \exp(-F(\boldsymbol{v}, z'))} \tag{3.26}$$

Let's note l the number of effectively trained hidden units, i.e. where all hidden units $> l$ have zero weights and biases. This is guaranteed to happen thanks to the growing behaviour that ensures hidden units are ordered from left to right. Then, we can split the normalization constant $Z(\boldsymbol{v})$ of Equation 3.26 into two parts, split at $z = l$, as follows:

$$\begin{aligned} Z(\boldsymbol{v}) &= \sum_{z=1}^{l} \exp(-F(\boldsymbol{v}, z)) + \sum_{z=l+1}^{\infty} \exp(-F(\boldsymbol{v}, z)) \\ &= \sum_{z=1}^{l} \exp(-F(\boldsymbol{v}, z)) + \sum_{z=l+1}^{\infty} \exp\left(-F(\boldsymbol{v}, l) + \sum_{i=l+1}^{z} \mathrm{soft}_+(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) - \beta_i\right) \\ &= \sum_{z=1}^{l} \exp(-F(\boldsymbol{v}, z)) + \exp(-F(\boldsymbol{v}, l)) \underbrace{\sum_{z=1}^{\infty} \exp((1-\beta)\,\mathrm{soft}_+(0))^z}_{\text{Geometric series}} \end{aligned} \tag{3.27}$$

where Equation 3.27 is obtained by exploiting the fact that all weights and biases of hidden units at position $l+1$ and higher are zero. By ensuring that $\beta > 1$, the geometric series of Equation 3.27 is finite and can be analytically computed. This in turn implies that $P_\Theta(z|\boldsymbol{v})$ is tractable and can be sampled from. Following a similar reasoning, the global partition function Z can be shown to be finite (see Appendix 3.B), thus yielding a properly defined joint distribution for any configurations with a finite number of non-zero weights and hidden biases.

One could think that, compared to a regular RBM, we have merely traded the hyper-

parameter of the hidden layer size with the hyper-parameter β. However, crucially, β's role is only to ensure that the iRBM is properly defined, and the penalty it imposes in the energy function can be compensated by the learned parameters. The extent to which the parameters can grow enough to compensate for that penalty is then controlled by the strength of weight decay, a hyper-parameter the iRBM shares with the RBM. We've thus effectively removed one hyper-parameter. Moreover, we've indeed observed that results are robust to the choice of β, that is finely tuning beta was not necessary to ultimately achieving good performance. While the choice of β can impact the number of epochs it would take for the weights to compensate for the penalty, this (the number of epochs) is a quantity that must be tuned anyways, even in regular RBMs.

The question of the identifiability of the binary RBM is a complex one, which has been studied [Cueto et al., 2009]. Unlike the RBM, the iRBM is sensitive to the ordering of its hidden units, thanks to the penalty term. This means permutations of iRBM's hidden units do not correspond to the same distribution, making its parametrization more identifiable.

As for learning, it can be done mostly by following the procedure of the oRBM, i.e. minimizing the NLL with stochastic gradient descent using (Persistent) CD to approximate the gradients. One slight modification is required however. Indeed, since the free energy gradient for the hidden weights and biases can be non-zero for all (infinite) hidden units, we cannot use the gradient of Equations 3.23 and 3.24 for all hidden units.

To avoid this issue, we consider the following observation. Instead of using the derivative of $F(\boldsymbol{v})$, we could instead use the derivative of $F(\boldsymbol{v}, z)$, where z is obtained by sampling from $P_\Theta(z|\boldsymbol{v})$:

$$\nabla_{\boldsymbol{W}} F(\boldsymbol{v}, z) = -\mathbb{E}_{\boldsymbol{h}}[\boldsymbol{h} \odot \mathbf{1}_{\leq z} | z, \boldsymbol{v}]\boldsymbol{v}^T = -(\widehat{\boldsymbol{h}}(\boldsymbol{v}) \odot \mathbf{1}_{\leq z})\boldsymbol{v}^T \tag{3.28}$$

$$\nabla_{\boldsymbol{b}^{\mathrm{h}}} F(\boldsymbol{v}, z) = -\mathbb{E}_{\boldsymbol{h}}[(\boldsymbol{h} - \beta\boldsymbol{\sigma}\left(\boldsymbol{b}^{\mathrm{h}}\right)) \odot \mathbf{1}_{\leq z} | z, \boldsymbol{v}] = -(\widehat{\boldsymbol{h}}(\boldsymbol{v}) - \beta\boldsymbol{\sigma}\left(\boldsymbol{b}^{\mathrm{h}}\right)) \odot \mathbf{1}_{\leq z}\ . \tag{3.29}$$

In this case, all weights and biases with an index greater than the sampled z have a gradient of zero, i.e. do not require any update. Moreover, the expectation of these

gradients with respect to z (conditioned on $\boldsymbol{v}$) are the gradients of $F(\boldsymbol{v})$, making them unbiased in this respect. This comes at the cost of higher variance in the updates. But thanks to this observation, we are justified to use a hybrid approach, where we use the $F(\boldsymbol{v})$ gradients only for the units with index less or equal than l, and "use" the gradient of $F(\boldsymbol{v}, z)$ for the other units, i.e. leave them set to zero.

As previously mentioned, we use weight decay to ensure that the number of non-zero parameters cannot diverge to infinity. For practical reasons, our implementation also used a capacity-limiting heuristic. If the Gibbs sampling chain ever sampled a value for z that is greater than l, then we clamped it to $l + 1$. Intuitively, this corresponds to "adding" a single hidden unit. This avoids filling all the memory in the (unlikely) event where we'd draw a large value for z. When adding a hidden unit, its associated weights and biases are initialized to zero.

We emphasize that these were not required to avoid divergence (weight decay is sufficient): it merely ensured a practical and efficient implementation of the model on the GPU. Note also that when using L1 regularisation, l can decrease in value, thanks to the sparsity promoting property of the L1 norm. Again, we highlight that while a finite number of weights and biases is maintained, that number of such weights does vary and is learned, while the implicit number of hidden units is indeed infinite (infinitely many contribute to the partition function).

3.5 Related Work

This work falls within the research literature on discovering extensions of the original RBM model to different contexts and objectives. Of note here is the implicit mixture of RBMs [Nair and Hinton, 2008]. Indeed, the oRBM can be interpreted as a special case of an implicit mixture of RBMs. Writing $P_\Theta(\boldsymbol{v})$ as $\sum_{z=1}^{K} P_\Theta(z)P_\Theta(\boldsymbol{v}|z)$ we see that the oRBM is an implicit mixture of K RBMs, where each RBM has a different number of hidden units (from 1 to K) and the weights are tied between RBMs. The prior $P_\Theta(z)$ represents the probability of using the z^{th} RBM and is also derived from the energy function. However, as in the implicit mixture of RBMs, $P_\Theta(z)$ is intractable as it would require the value of the partition function. That said, the

work of Nair and Hinton [2008] is otherwise very different and did not address the question of having an RBM with adaptive capacity.

Another related work is that of the Cardinality RBMs proposed by Swersky et al. [2012]. They used a cardinality potential to control the sparsity of the RBM, i.e. limiting the number of hidden units that can be active. In the oRBM and the iRBM, z effectively acts as an upper bound on the number of hidden units h_i that can be equal to 1, since we are limiting $\boldsymbol{h}$ to be in $\mathbb{H}_z$, a subset of $\mathbb{H}$. In their work, Swersky et al. [2012] use cardinality potentials that allow only configurations having at most k active hidden units. One difference with our work however is that their cardinality potential is order agnostic, meaning that the active hidden units can be positioned anywhere within the hidden layer while still satisfying the cardinality potential. On the other hand, in the oRBM, all units with index higher than z must be set to zero, with only the previous hidden units being allowed to be active. In addition, their parameter k is fixed during training whereas our number of active hidden units z changes depending on the input.

The oRBM also bears some similarity with autoencoders trained by a nested version of dropout [Rippel et al., 2014]. Nested dropout works by stochastically selecting the number of hidden units used to reconstruct an input example at training time, and so independently for each update and example. Rippel et al. [2014] showed that this defines a learning objective that makes the solution identifiable and no longer invariant to hidden unit permutation. In addition to being concerned with a different type of model, this work doesn't discuss the case of an unbounded and adaptive hidden layer size.

Welling et al. [2003] proposed a self supervised boosting approach, which is applicable to the RBM and in which hidden units are sequentially added and trained. However, like boosting in general and unlike the iRBM, this procedure trains each hidden unit greedily instead of jointly, which could lead to much larger networks than necessary. Moreover, it is not easily generalizable to online learning.

While the work on unsupervised neural networks with adaptive hidden layer size is otherwise relatively scarse, there's been much more work in the context of supervised

learning. There is the well known work of Fahlman and Lebiere [1990] on Cascade-Correlation networks. More recently, Zhou et al. [2012] proposed a procedure for learning discriminative features with a denoising autoencoder (a model related to the RBM). The procedure is also applicable to the online setting. It relies on invoking two heuristics that either add or merge hidden units during training. We note that the iRBM framework could easily be generalized to discriminative and hybrid training as in Zhou et al. [2012]. The corresponding mecanisms for adding and merging units would then be implicitly derived from gradient descent on the corresponding supervised training objective.

Finally, we highlight that our model is not based on a Bayesian formulation, as most of the literature on infinite models. On the other hand, it does correspond to the infinite limit of a finite-sized model and yields a model that can learn its size with training.

3.6 Experiments

We compare the performance of the oRBM and the iRBM with the classic RBM on two datasets: binarized MNIST [Salakhutdinov and Murray, 2008] and CalTech101 Silhouettes [Marlin et al., 2010b]. We aim to demonstrate that the iRBM effectively removes the need of tuning an hyper-parameter for the hidden layer size while still achieving comparable performance to the standard RBM. The code to reproduce the experiments of the paper is available on GitHub [1]. Our implementation is done using Theano [Bastien et al., 2012; Bergstra et al., 2010].

For completeness, we wish to mention that more sophisticated or deep models have reported results on one or both of these datasets (e.g. EoNADE [Uria et al., 2014], DBNs [Murray and Salakhutdinov, 2009], Deep autoregressive networks [Gregor et al., 2014b], Iterative Neural Autoregressive Distribution Estimator [Raiko and Bengio, 2014]) that improve on the standard RBM. However, since our objective with the iRBM is to effectively remove a hyper-parameter of the RBM, instead of achieving

1. `http://github.com/MarcCote/iRBM`

improved performances, we focus our comparison on this baseline.

All NLL results of this section were obtained by estimating the log-partition function $\ln \hat{Z}$ using Annealed Importance Sampling (AIS) [Salakhutdinov and Murray, 2008] with 100,000 intermediate distributions and 5000 chains. As an additional validation step, samples were generated from best models and visually inspected.

Each model was trained with mini-batch stochastic gradient descent using batch size of 64 examples and using PCD with 10 Gibbs steps between parameter updates. We used the ADAGRAD stochastic gradient update [Duchi et al., 2011], a per-dimension learning rate method, to train the oRBMs and the iRBMs. We found that having different learning rates for different hidden units was very beneficial, since units positioned earlier in the hidden layer will approach convergence faster than units to their right, and thus will benefit from a learning rate decaying more rapidly. We tried several learning rates $lr \in \{5{\times}10^{-1}, 10^{-1}, 5{\times}10^{-2}, 10^{-2}\}$ and always set ADAGRAD's epsilon parameter to 10^{-6}.

We also tested different values for both L1 and L2 regularization's factor $\lambda \in \{0, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Note that we allow the iRBM to shrink only if L1 regularization is used.

We did try varying the β found in the penalty term and as expected we've found results to be robust to its value. Since β must be greater than 1, we explored positive constants to add to 1, on a log scale (1, 0.25, 0.1, 0.01, 0.001, etc.). We settled on using $\beta = 1.01$ for all experiments as it provides a penalty high enough to have a growing behavior and requires around five hundred epochs for the weights to compensate for the penalty.

Finally, we note that improved performances could certainly have been achieved using an improved sampler (e.g. parallel tempering [Desjardins et al., 2010b]) or parametrization (e.g. enhanced gradient parametrization [Cho et al., 2013]). However, these changes would equally improve the baseline RBM, so we decided to concentrate on this more common learning setup.

Table 3.1: Average NLL on binarized MNIST test set for best RBMs, oRBM and iRBM. Partition functions were estimated using AIS with 100,000 intermediate distributions and 5000 chains. The confidence interval on the average NLL assumes $\ln \hat{Z}$ has no variance and reflects the confidence of a finite sample average. By taking the uncertainty about the partition function into account, the interval would be larger.

		Binarized MNIST		
Model	Size	$\ln \hat{Z}$	$\ln(\hat{Z} \pm 3\sigma)$	Avg. NLL
RBM	100	600.92	[600.88, 600.95]	98.17 ± 0.52
RBM	500	613.28	[613.24, 613.31]	86.50 ± 0.44
RBM	2000	1099.07	[1098.94, 1099.17]	85.03 ± 0.42
oRBM	500	40.06	[39.90, 40.19]	88.15 ± 0.46
iRBM	1208	40.32	[40.03, 40.54]	85.65 ± 0.44

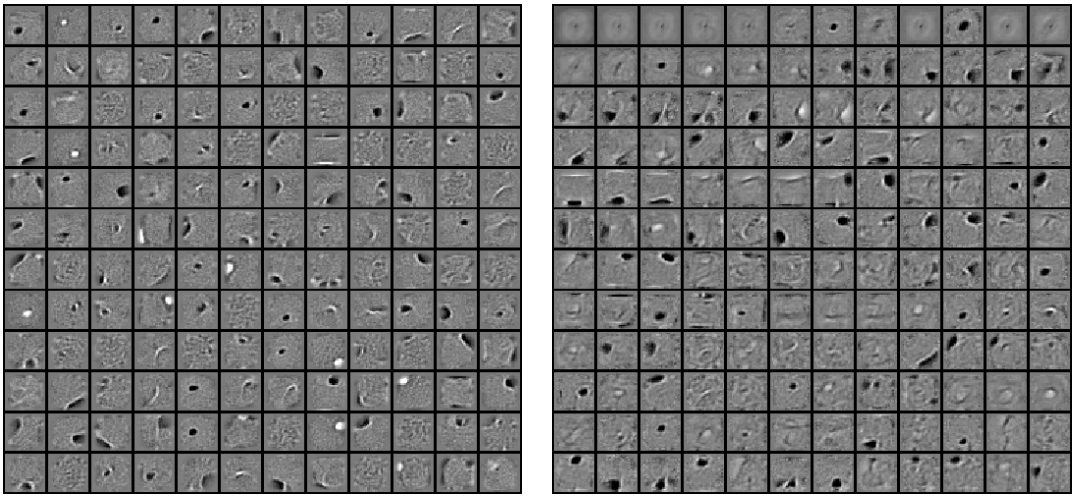
3.6.1 Binarized MNIST

The MNIST dataset [2] is composed of 70,000 images of size 28x28 pixels representing handwritten digits (0-9). Images have been stochastically binarized according to their pixel intensity as in Salakhutdinov and Murray [2008]. We use the same split as in Larochelle and Murray [2011], corresponding to 50,000 examples for training, 10,000 for validation and 10,000 for testing.

Each model was trained up to 5000 epochs but we performed AIS evaluation every 1000 epochs and kept the model having the best NLL approximation on the valid set. We report the associated NLL approximations obtained on the test set. Taking after past studies assessing RBM results on binarized MNIST, we fixed the number of hidden units to 500 for the RBM and the oRBM. Best results for the RBM, oRBM and iRBM are reported in Table 3.1. The oRBM and the iRBM models reach competitive performance compared to the RBM. Samples from all three models are illustrated in Figure 3.5.

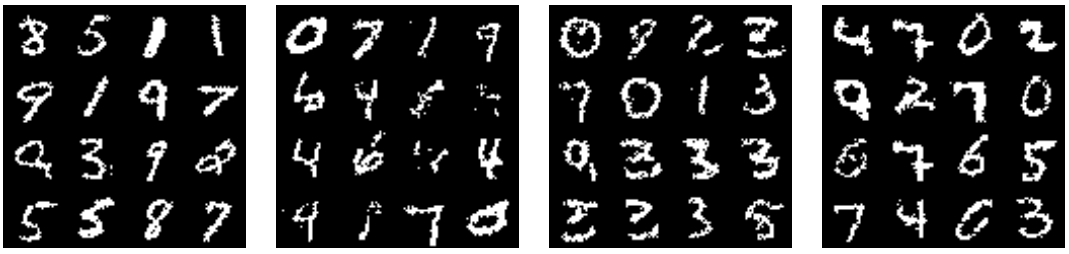
The best RBM (500 hidden units) was trained without any regularization and $lr = 10^{-2}$ for 5000 epochs. We used our own implementation to train the RBM, which is why our result slightly differs from what is reported by Salakhutdinov and Murray

2. `http://yann.lecun.com/exdb/mnist`



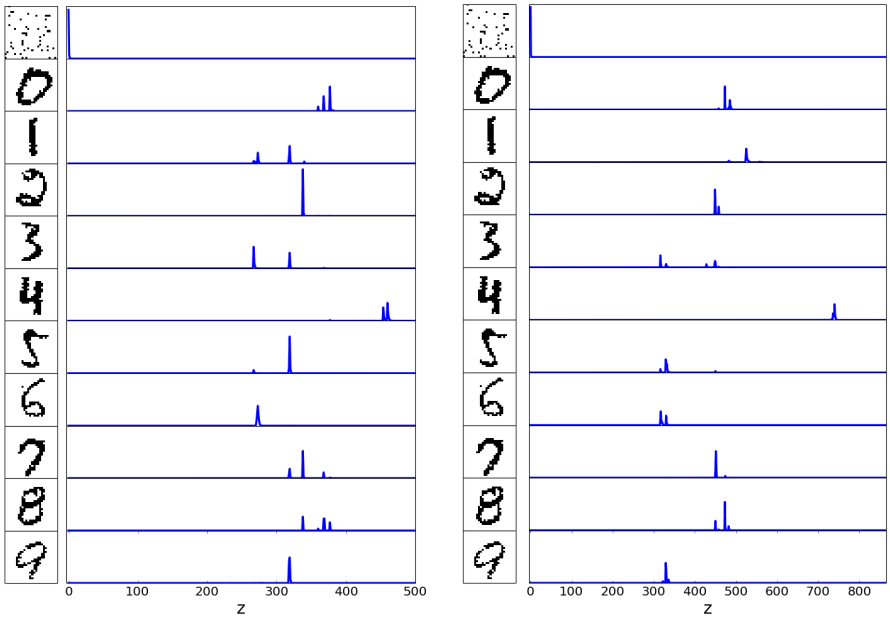
(a) RBM (b) iRBM

Figure 3.4: Comparing the filters of an RBM and an iRBM both trained on binarized MNIST. The first 96 filters are shown starting from the top-left corner and incrementing across columns first.



(a) Test set (b) RBM (c) oRBM (d) iRBM

Figure 3.5: Comparison between data from binarized MNIST and random samples generated from the three models by randomly initializing visible units and running 10,000 Gibbs steps. The RBM and oRBM both have 500 hidden units, whereas the iRBM final size is 1208 hidden units.



(a) oRBM (b) iRBM

Figure 3.6: Each row shows a plot of $P_\Theta(z|\boldsymbol{v})$ where $\boldsymbol{v}$ is a given example from MNIST test set and is displayed to the left. The first row illustrates the impact of a noisy image on sampling z. As explained in Section 3 of the paper, we see that different input images are related to different values for the number z of used units.

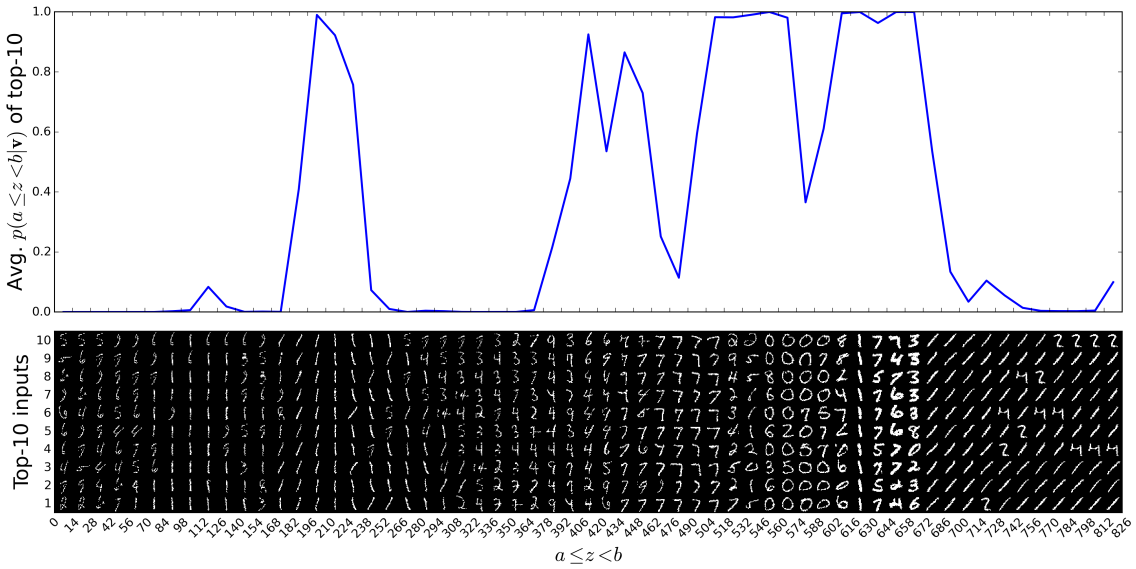


Figure 3.7: **(Bottom)** Top 10 inputs from the test set with highest value of $P_\Theta(z|\boldsymbol{v})$ within different intervals for z, i.e. $\text{argmax}_{\boldsymbol{v}}\, P_\Theta(a \leq z < b|\boldsymbol{v})$ for different intervals $[a, b)$. Interestingly, bolder inputs seem to be related to bigger values for the number z of used units. Also, simpler characters (e.g. "ones") tend to favor smaller values of z compared to more complex characters. **(Top)** Average of $P_\Theta(a \leq z < b|\boldsymbol{v})$ over the top 10 inputs. Low values highlights regions in the hidden layer where the hidden units are only useful when taken together with hidden units further right in the layer.

[2008]. The difference can be justified by the fact that they used the full 60,000 training set images, instead of a 50,000 subset. Also, they use a custom schedule to gradually increase the number of CD steps during training. That said, the oRBM and the iRBM would probably also benefit from having more training data and an improved sampling strategy.

The best oRBM (500 hidden units) was trained without any regularization and $lr = 3\times10^{-2}$ for 500 epochs. After 3000 epochs, the best iRBM had 1208 hidden units with non-zero weights. It was trained with L1 regularization using a regularization factor of $\lambda = 10^{-4}$ and $lr = 5\times10^{-2}$.

To show that our best iRBM does find an appropriate number of hidden units, we compared it with two other RBMs having respectively 100 and 2000 hidden units. Both were trained for 5000 epochs without any regularization and respectively with $lr = 10^{-1}$ and $lr = 10^{-2}$. Results are reported in Table 3.1 where we can see the oRBM and the iRBM still achieve competitive results compared to the RBM with 2000 hidden units.

Figure 3.4 shows the ordering effect on the filters obtained with an iRBM. The ordering is even more apparent when observing the hidden unit filters during training. We generated a video of this visualization, illustrating the filter values and the generated negative samples at epochs 1, 10, 50 and 100. See link: `http://goo.gl/LGQDaI`.

Interestingly, we've observed that Gibbs sampling can mix much more slowly with the oRBM. The reason is the addition of variable z increases the dependence between states and thus hurts the convergence of Gibbs sampling. In particular, we observed that when the Gibbs chain is in a state corresponding to a noisy image without any structure, it can require many steps before stepping out of this region of the input space. Yet, comparing the free energy of such random images and images that resemble digits confirmed that these random images have significantly higher free energy (and thus are unlikely samples of the model). Figure 3.6 also confirms the high dependence between z and $\boldsymbol{v}$: the distribution of the unstructured image is peaked at $z = 1$, while all digits prefer values of z greater than 250. To fix this issue,

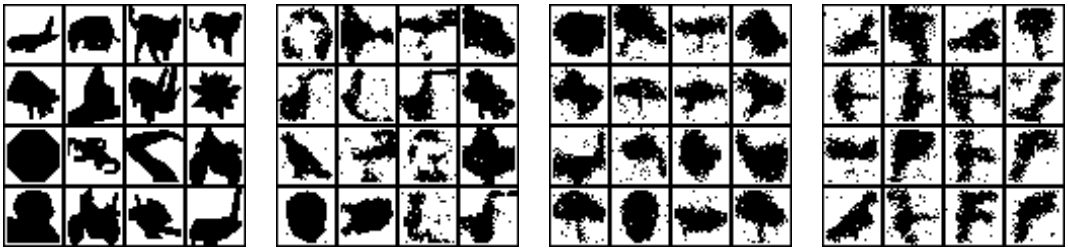
Table 3.2: Average NLL on CalTech101 Silhouettes test set estimated using AIS with 100,000 intermediate distributions and 5000 chains. The confidence interval on the average NLL assumes $\ln \hat{Z}$ has no variance and reflects the confidence of a finite sample average. By taking the uncertainty about the partition function into account, the interval would be larger.

		CalTech101 Silhouettes		
Model	Size	$\ln \hat{Z}$	$\ln(\hat{Z} \pm 3\sigma)$	Avg. NLL
RBM	100	2512.20	[2511.62, 2512.56]	177.37 ± 2.81
RBM	500	2385.91	[2385.68, 2386.10]	119.05 ± 2.27
RBM	2000	3353.47	[3349.85, 3354.15]	118.29 ± 2.25
oRBM	500	1782.96	[1782.88 1783.02]	114.99 ± 1.97
iRBM	915	2000.08	[1999.93, 2000.22]	121.47 ± 2.07

we've found that simply initializing the Gibbs chain to $z = K$ was sufficient. We used this when sampling from a trained oRBM model.

The iRBM doesn't seem to suffer as much from a low mixing rate and thus doesn't require the $z = K$ initialization heuristic for sampling. In fact, using the heuristic when sampling from an iRBM has almost no impact on the final samples when running 10,000 Gibbs steps. This could be an artefact of the model being trained progressively, i.e. we only add one hidden unit when sampling a large value for z bigger than l. Understanding how the lower mixing rate affects the proposed models and if a heuristic such as the one we mentioned earlier could be used to improve training is a topic left for future work.

We've also investigated what kind of inputs are maximizing $P_\Theta(z|\boldsymbol{v})$, for different values of z. Using our best iRBM model trained with L1 regularization, we generated Figure 3.7. It highlights the fact that $P_\Theta(z|\boldsymbol{v})$ does capture some structure about the data, as the identity of the character with highest $P_\Theta(z|\boldsymbol{v})$ vary between different values of z.



(a) Test set (b) RBM (c) oRBM (d) iRBM

Figure 3.8: Comparison between data from CalTech101 Silhouettes and random samples generated from three models by randomly initializing visible units and running 10,000 Gibbs steps. The RBM and oRBM both have 500 hidden units, whereas the iRBM final size is 915 hidden units.

3.6.2 CalTech101 Silhouettes

The CalTech101 Silhouettes dataset [3] [Marlin et al., 2010b] is composed of 8,671 images of size 28x28 binary pixels, representing object silhouettes (101 classes). The dataset is divided in three subsets: 4,100 examples for training, 2,264 for validation and 2,307 for testing.

Following a protocol similar to the one used for MNIST, each model was trained up to 5000 epochs, AIS evaluation was done every 1000 epochs. We report the NLL approximations obtained on the test set. Best results for the RBM, oRBM and iRBM are reported in Table 3.2. Again, the oRBM and the iRBM models reach competitive performance compared to the RBM. Samples from all three models are illustrated in Figure 3.8.

The best RBM (500 hidden units) was trained without any regularization and $lr = 10^{-2}$ for 3000 epochs. We used our own implementation to train the RBM. The best oRBM (500 hidden units) was trained with L1 regularization using a regularization factor of $\lambda = 10^{-3}$ and $lr = 10^{-2}$ for 5000 epochs. After 4000 epochs, the best iRBM had 915 hidden units with non-zero weights. It was trained with L1 regularization using a regularization factor of $\lambda = 10^{-3}$ and $lr = 5 \times 10^{-2}$.

Again, to show that our best iRBM does find an appropriate number of hidden units,

3. `http://people.cs.umass.edu/~marlin/data.shtml`

we compared it with two others RBMs having respectively 100 and 2000 hidden units. Both were trained without any regularization and respectively with $lr = 10^{-1}$ for 5000 epochs and $lr = 10^{-2}$ for 2000 epochs. Results are reported in Table 3.2 where we can see the oRBM and the iRBM still achieve competitive results compared to the RBM with 2000 hidden units.

3.7 Conclusion

We proposed a novel extension of the RBM, the infinite RBM, which obviates the need to specify the hidden layer size. The iRBM is derived from the ordered RBM by taking the infinite limit of its hidden layer size. We presented a training procedure, derived from Contrastive Divergence, such that training the iRBM yields a learning procedure where the effective hidden layer size can grow.

In future work, we are interested in generalizing the idea of a growing latent representation to structures other than a flat vector representation. We are currently exploring extensions of the RBM allowing for a tree-structured latent representation. We believe a similar construction, involving a similar z random variable, should allow us to derive a training algorithm that also learns the latent representation's size.

Acknowledgments

We thank NSERC for supporting this research, Nicolas Le Roux for discussions and comments, and Stanislas Lauly for making the iRBM's training video.

3.A Partial derivatives

Partial derivatives related to the RBM

Recall equation (3.5) representing the free energy of the RBM:

$$F(\boldsymbol{v}) = -\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \sum_{i=1}^{K} \mathrm{soft}_+(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}})$$

Taking the partial derivatives of $F(\boldsymbol{v})$ w.r.t. W_{ij}, b_i^{h} and $\boldsymbol{b}_j^{\mathrm{v}}$ respectively, we obtain the following:

$$\begin{aligned}
\frac{\partial F(\boldsymbol{v})}{\partial W_{ij}} &= -\sum_{k=1}^{K} \frac{\partial\ \mathrm{soft}_+(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}})}{\partial W_{ij}} \\
&= -\sum_{k=1}^{K} \sigma(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}}) \frac{\partial \boldsymbol{W}_{k\cdot}\boldsymbol{v}}{\partial W_{ij}} \\
&= -\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) v_j \qquad (3.30) \\
\frac{\partial F(\boldsymbol{v})}{\partial b_i^{\mathrm{h}}} &= -\sum_{k=1}^{K} \frac{\partial\ \mathrm{soft}_+(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}})}{\partial b_i^{\mathrm{h}}} \\
&= -\sum_{k=1}^{K} \sigma(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}}) \frac{\partial b_k^{\mathrm{h}}}{\partial b_i^{\mathrm{h}}} \\
&= -\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) \qquad (3.31) \\
\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{b}_j^{\mathrm{v}}} &= -\frac{\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}}}{\partial \boldsymbol{b}_j^{\mathrm{v}}} \\
&= -v_j \qquad (3.32)
\end{aligned}$$

where $\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}})$ can be expressed as a conditional expectation over h_i using equation (3.12)

$$\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) = P(h_i = 1|\boldsymbol{v}) = \sum_{h_i \in \{0,1\}} P(h_i = 1|\boldsymbol{v}) h_i = \mathbb{E}[h_i|\boldsymbol{v}].$$

Partial derivatives related to the oRBM and the iRBM

Recall equation (3.19) representing the free energy of the oRBM:

$$F(\boldsymbol{v}) = \ln\left(\sum_{z=1}^{K} e^{-F(\boldsymbol{v},z)}\right)$$

where

$$F(\boldsymbol{v}, z) = -\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}} - \sum_{i=1}^{z} \left(\mathrm{soft}_+(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) - \beta\,\mathrm{soft}_+(b_i^{\mathrm{h}})\right)$$

The partial derivatives of $F(\boldsymbol{v}, z)$ w.r.t. W_{ij}, b_i^{h} and $\boldsymbol{b}_j^{\mathrm{v}}$ are similar to equations (3.30), (3.31) and (3.32) from the RBM and are respectively given by

$$\begin{aligned}
\frac{\partial F(\boldsymbol{v}, z)}{\partial W_{ij}} &= -\sum_{k=1}^{z} \frac{\partial\ \mathrm{soft}_+(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}})}{\partial W_{ij}} \\
&= -\sum_{k=1}^{z} \sigma(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}}) \frac{\partial \boldsymbol{W}_{k\cdot}\boldsymbol{v}}{\partial W_{ij}} \\
&= -H(z-i)\,\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) v_j \qquad (3.33) \\
\frac{\partial F(\boldsymbol{v}, z)}{\partial b_i^{\mathrm{h}}} &= -\sum_{k=1}^{z} \frac{\partial\ \mathrm{soft}_+(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}}) - \beta\,\mathrm{soft}_+(b_k^{\mathrm{h}})}{\partial b_i^{\mathrm{h}}} \\
&= -\sum_{k=1}^{z} \left(\sigma(\boldsymbol{W}_{k\cdot}\boldsymbol{v} + b_k^{\mathrm{h}}) - \beta\sigma(b_k^{\mathrm{h}})\right) \frac{\partial b_k^{\mathrm{h}}}{\partial b_i^{\mathrm{h}}} \\
&= -H(z-i)\,\left(\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) - \beta\sigma(b_i^{\mathrm{h}})\right) \qquad (3.34) \\
\frac{\partial F(\boldsymbol{v}, z)}{\partial \boldsymbol{b}_j^{\mathrm{v}}} &= -\frac{\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}}}{\partial \boldsymbol{b}_j^{\mathrm{v}}} = -v_j \qquad (3.35)
\end{aligned}$$

with the Heaviside step function denoted as

$$H(n) = \begin{cases} 0, & n < 0 \\ 1, & n \geq 0 \end{cases}.$$

Then, the partial derivatives of $F(\boldsymbol{v})$ w.r.t. W_{ij}, b_i^{h} and $\boldsymbol{b}_j^{\mathrm{v}}$ are obtained respectively

as follows:

$$
\begin{aligned}
\frac{\partial F(\boldsymbol{v})}{\partial W_{ij}} &= \sum_{z=1}^{K} \frac{e^{-F(\boldsymbol{v},z)}}{\sum_{z'=1}^{K} e^{-F(\boldsymbol{v},z')}} \frac{\partial F(\boldsymbol{v},z)}{\partial W_{ij}} \\
&= -\sum_{z=1}^{K} P(z|\boldsymbol{v})\, H(z-i)\, \sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) v_j \\
&= -\sum_{z=1}^{K} H(z-i)\, P(z|\boldsymbol{v}) P(h_i = 1|\boldsymbol{v}, z) v_j \\
&= -\sum_{z=1}^{K} \sum_{h_i \in \{0,1\}} H(z-i) h_i\, P(z|\boldsymbol{v})\, P(h_i|\boldsymbol{v}, z) v_j \\
&= -\mathbb{E}_{h_i,z}[H(z-i)h_i|\boldsymbol{v}] v_j \qquad (3.36) \\
\frac{\partial F(\boldsymbol{v})}{\partial b_i^{\mathrm{h}}} &= \sum_{z=1}^{K} P(z|\boldsymbol{v}) \frac{\partial F(\boldsymbol{v},z)}{\partial b_i^{\mathrm{h}}} \\
&= -\sum_{z=1}^{K} P(z|\boldsymbol{v})\, H(z-i) \left(\sigma(\boldsymbol{W}_{i\cdot}\boldsymbol{v} + b_i^{\mathrm{h}}) - \beta\sigma(b_i^{\mathrm{h}})\right) \\
&= -\sum_{z=1}^{K} H(z-i)\, P(z|\boldsymbol{v}) \left(P(h_i = 1|\boldsymbol{v}, z) - \beta\sigma(b_i^{\mathrm{h}})\right) \\
&= -\sum_{z=1}^{K} H(z-i)\, P(z|\boldsymbol{v}) \big(-\beta\sigma(b_i^{\mathrm{h}})(1 - P(h_i = 1|\boldsymbol{v}, z)) \\
&\qquad\qquad + (1 - \beta\sigma(b_i^{\mathrm{h}}))P(h_i = 1|\boldsymbol{v}, z)\big) \\
&= -\sum_{z=1}^{K} H(z-i)\, P(z|\boldsymbol{v}) \big((0 - \beta\sigma(b_i^{\mathrm{h}}))P(h_i = 0|\boldsymbol{v}, z) \\
&\qquad\qquad + (1 - \beta\sigma(b_i^{\mathrm{h}}))P(h_i = 1|\boldsymbol{v}, z)\big) \\
&= -\sum_{z=1}^{K} \sum_{h_i \in \{0,1\}} H(z-i) \left(h_i - \beta\sigma(b_i^{\mathrm{h}})\right)\, P(z|\boldsymbol{v})\, P(h_i|\boldsymbol{v}, z) \\
&= -\mathbb{E}_{h_i,z}\left[H(z-i)\left(h_i - \beta\sigma(b_i^{\mathrm{h}})\right)|\boldsymbol{v}\right] \qquad (3.37) \\
\frac{\partial F(\boldsymbol{v})}{\partial \boldsymbol{b}_j^{\mathrm{v}}} &= -\frac{\boldsymbol{v}^T \boldsymbol{b}^{\mathrm{v}}}{\partial \boldsymbol{b}_j^{\mathrm{v}}} = -v_j
\end{aligned}
$$

Observe that in equations (3.36) and (3.37), $\sum_{z=1}^{K} P(z|\boldsymbol{v})\, H(z-i)$ corresponds to $P(z \geq i|\boldsymbol{v})$. This then translates to $(1 - \mathbf{cdf}(z|\boldsymbol{v}))$ when deriving the gradients as shown in equations (3.23) and (3.24).

3.B Convergence of the partition function for the iRBM

We will show that the partition function Z of the iRBM is finite. To do so, we take the limit of $K \to \infty$ of equation (3.16):

$$\begin{aligned} Z &= \sum_{\boldsymbol{v} \in \mathbb{V}} \sum_{z=1}^{\infty} \sum_{\boldsymbol{h} \in \mathbb{H}_z} e^{-\mathrm{E}(\boldsymbol{v},\boldsymbol{h},z)} \\ &= \sum_{\boldsymbol{v} \in \mathbb{V}} \sum_{z=1}^{\infty} e^{-F(\boldsymbol{v},z)} \\ &= \sum_{\boldsymbol{v} \in \mathbb{V}} Z(\boldsymbol{v}) \end{aligned} \tag{3.38}$$

Since the sum over all $\boldsymbol{v} \in \mathbb{V}$ is finite and we know from equation (3.27) that $Z(\boldsymbol{v})$ is finite, then Z is also finite.

Chapitre 4

Autorégression neuronale pour l'estimation de densité de probabilité

“ Standing on the shoulders of giants.

Isaac Newton, 1676 ”

Résumé

Dans cet article journal, nous présentons le *Neural Autoregressive Distribution Estimator* (NADE) et ses variantes. Il s'agit de modèles basés sur les réseaux de neurones qui sont entrainés de façon non supervisée pour la tâche d'estimation de densité de probabilité. Ils exploitent la formule des probabilités totales (*product rule*) afin d'obtenir une décomposition d'une probabilité en un produit de probabilités conditionnelles permettant ainsi d'obtenir un estimateur de densité entièrement calculable. Nous montrons que ces modèles sont compétitifs avec les méthodes existant dans la littérature en ce qui concerne la modélisation de données binaires et de données réelles. De plus, nous démontrons que les versions profondes de NADE (Deep NADE) peuvent apprendre à être invariantes à l'ordre des dimensions des données observées qui est utilisé pour obtenir le produit de probabilités conditionnelles. Finalement, nous montrons comment NADE avec une architecture profonde basée sur la convolution (Conv NADE) arrive à exploiter la structure topologique des pixels contenus dans les images.

Commentaires

L'article a été publié dans le journal (*Journal of Machine Learning Research*) en 2016. Mes contributions pour cet article sont les suivantes :

- Développer, en Python, les modèles NADE, Deep NADE et Conv NADE en utilisant la librairie Theano [The Theano Development Team et al., 2016]. Aussi, puisque ce sont des modèles génératifs (section 1.3.2), le code supporte la génération de nouvelles données à partir d'un modèle déjà entrainé. Le code source pour reproduire les expériences est disponible publiquement sur mon répertoire GitHub (`https://github.com/MarcCote/NADE`).
- Explorer différentes architectures pour Conv NADE afin de déterminer quelle configuration fonctionne le mieux. Durant cette phase exploratoire, j'ai appris beaucoup sur les réseaux à convolution, les différentes façons de gérer les bords et les *Residual Networks* [He et al., 2015, 2016].
- Évaluer la performance du modèle Conv NADE sur le jeu de données *Binarized MNIST*. Il s'agit d'une tâche d'estimation de densité de probabilité (section 1.1.3). L'apprentissage du modèle s'est fait de façon non supervisée (section 1.2.2).
- Participer à l'écriture de l'article journal. J'ai écrit la section décrivant le modèle Conv NADE ainsi que la section de discussion rapportant les résultats obtenus sur *Binarized MNIST*. J'ai également aidé à écrire la section décrivant le modèle Deep NADE.
- Créer les images illustrant l'architecture des différents modèles.

Neural Autoregressive Distribution Estimation

Benigno Uria
Google DeepMind
benigno.uria@gmail.com

Marc-Alexandre Côté
Département d'informatique
Université de Sherbrooke
marc-alexandre.cote@usherbrooke.ca

Karol Gregor
Google DeepMind
karol.gregor@gmail.com

Iain Murray
University of Edinburgh
i.murray@ed.ac.uk

Hugo Larochelle
Twitter
hlarochelle@twitter.com

Keywords: deep learning, neural networks, density modeling, unsupervised learning

Abstract

We present Neural Autoregressive Distribution Estimation (NADE) models, which are neural network architectures applied to the problem of unsupervised distribution and density estimation. They leverage the probability product rule and a weight sharing scheme inspired from restricted Boltzmann machines, to yield an estimator that is both tractable and has good generalization performance. We discuss how they achieve competitive performance in modeling both binary and real-valued observations. We also present how deep NADE models can be trained to be agnostic to the ordering of input dimensions used by the autoregressive product rule decomposition. Finally, we also show how to exploit the topological structure of pixels in images using a deep convolutional architecture for NADE.

4.1 Introduction

Distribution estimation is one of the most general problems addressed by machine learning. From a good and flexible distribution estimator, in principle it is possible to solve a variety of types of inference problem, such as classification, regression, missing value imputation, and many other predictive tasks.

4.1. Introduction

Currently, one of the most common forms of distribution estimation is based on directed graphical models. In general these models describe the data generation process as sampling a latent state $\boldsymbol{h}$ from some prior $p(\boldsymbol{h})$, followed by sampling the observed data $\boldsymbol{x}$ from some conditional $p(\boldsymbol{x} \mid \boldsymbol{h})$. Unfortunately, this approach quickly becomes intractable and requires approximations when the latent state $\boldsymbol{h}$ increases in complexity. Specifically, computing the marginal probability of the data, $p(\boldsymbol{x}) = \sum_{\boldsymbol{h}} p(\boldsymbol{x} \mid \boldsymbol{h})\, p(\boldsymbol{h})$, is only tractable under fairly constraining assumptions on $p(\boldsymbol{x} \mid \boldsymbol{h})$ and $p(\boldsymbol{h})$.

Another popular approach, based on undirected graphical models, gives probabilities of the form $p(\boldsymbol{x}) = \exp\{\phi(\boldsymbol{x})\}/Z$, where ϕ is a tractable function and Z is a normalizing constant. A popular choice for such a model is the restricted Boltzmann machine (RBM), which substantially out-performs mixture models on a variety of binary data sets [Salakhutdinov and Murray, 2008]. Unfortunately, we often cannot compute probabilities $p(\boldsymbol{x})$ exactly in undirected models either, due to the normalizing constant Z.

In this paper, we advocate a third approach to distribution estimation, based on autoregressive models and feed-forward neural networks. We refer to our particular approach as Neural Autoregressive Distribution Estimation (NADE). Its main distinguishing property is that computing $p(\boldsymbol{x})$ under a NADE model is tractable and can be computed efficiently, given an arbitrary ordering of the dimensions of $\boldsymbol{x}$.

The NADE framework was first introduced for binary variables by Larochelle and Murray [2011], and concurrent work by Gregor and LeCun [2011]. The framework was then generalized to real-valued observations [Uria et al., 2013], and to versions based on deep neural networks that can model the observations in any order [Uria et al., 2014]. This paper pulls together an extended treatement of these papers, with more experimental results, including some by Uria [2015]. We also report new work on modeling 2D images by incorporating convolutional neural networks into the NADE framework. For each type of data, we're able to reach competitive results, compared to popular directed and undirected graphical model alternatives.

4.2 NADE

We consider the problem of modeling the distribution $p(\boldsymbol{x})$ of input vector observations $\boldsymbol{x}$. For now, we will assume that the dimensions of $\boldsymbol{x}$ are binary, that is $x_d \in \{0, 1\}\ \forall d$. The model generalizes to other data types, which is explored later (Section 4.3) and in other work (Section 4.8).

NADE begins with the observation that any D-dimensional distribution $p(\boldsymbol{x})$ can be factored into a product of one-dimensional distributions, in any order o (a permutation of the integers $1, \ldots, D$):

$$p(\boldsymbol{x}) = \prod_{d=1}^{D} p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}). \tag{4.1}$$

Here $o_{<d}$ contains the first $d-1$ dimensions in ordering o and $\boldsymbol{x}_{o_{<d}}$ is the corresponding subvector for these dimensions. Thus, one can define an 'autoregressive' generative model of the data simply by specifying a parameterization of all D conditionals $p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}})$.

Frey et al. [1996] followed this approach and proposed using simple (log-)linear logistic regression models for these conditionals. This choice yields surprisingly competitive results, but are not competitive with non-linear models such as an RBM. Bengio and Bengio [2000] proposed a more flexible approach, with a single-layer feed-forward neural network for each conditional. Moreover, they allowed connections between the output of each network and the hidden layer of networks for the conditionals appearing earlier in the autoregressive ordering. Using neural networks led to some improvements in modeling performance, though at the cost of a really large model for very high-dimensional data.

In NADE, we also model each conditional using a feed-forward neural network. Specifically, each conditional $p(x_{o_d} \,|\, \boldsymbol{x}_{<d})$ is parameterized as follows:

$$p(x_{o_d} = 1 \,|\, \boldsymbol{x}_{o_{<d}}) = \text{sigm}\left(\boldsymbol{V}_{o_d,\cdot} \boldsymbol{h}_d + b_{o_d}\right) \tag{4.2}$$

$$\boldsymbol{h}_d = \text{sigm}\left(\boldsymbol{W}_{\cdot,o_{<d}} \boldsymbol{x}_{o_{<d}} + \boldsymbol{c}\right), \tag{4.3}$$

where $\text{sigm}\,(a) = 1/(1 + e^{-a})$ is the logistic sigmoid, and with H as the number of

hidden units, $\boldsymbol{V} \in \mathbb{R}^{D \times H}$, $\boldsymbol{b} \in \mathbb{R}^{D}$, $\boldsymbol{W} \in \mathbb{R}^{H \times D}$, $\boldsymbol{c} \in \mathbb{R}^{H}$ are the parameters of the NADE model.

The hidden layer matrix $\boldsymbol{W}$ and bias $\boldsymbol{c}$ are shared by each hidden layer $\boldsymbol{h}_d$ (which are all of the same size). This parameter sharing scheme (illustrated in Figure 4.1) means that NADE has $O(HD)$ parameters, rather than $O(HD^2)$ required if the neural networks were separate. Limiting the number of parameters can reduce the risk of over-fitting. Another advantage is that all D hidden layers $\boldsymbol{h}_d$ can be computed in $O(HD)$ time instead of $O(HD^2)$. Denoting the pre-activation of the d^{th} hidden layer as $\boldsymbol{a}_d = \boldsymbol{W}_{\cdot, o_{<d}} \boldsymbol{x}_{o_{<d}} + \boldsymbol{c}$, this complexity is achieved by using the recurrence

$$\boldsymbol{h}_1 = \text{sigm}\left(\boldsymbol{a}_1\right), \quad \text{where } \boldsymbol{a}_1 = \boldsymbol{c} \tag{4.4}$$

$$\boldsymbol{h}_d = \text{sigm}\left(\boldsymbol{a}_d\right), \quad \text{where } \boldsymbol{a}_d = \boldsymbol{W}_{\cdot, o_{<d}} \boldsymbol{x}_{o_{<d}} + \boldsymbol{c} = \boldsymbol{W}_{\cdot, o_{d-1}} x_{o_{d-1}} + \boldsymbol{a}_{d-1} \quad \text{for } d \in \{2, \ldots, D\}, \tag{4.5}$$

where Equation 4.5 given vector $\boldsymbol{a}_{d-1}$ can be computed in $O(H)$. Moreover, the computation of Equation 4.2 given $\boldsymbol{h}$ is also $O(H)$. Thus, computing $p(\boldsymbol{x})$ from D conditional distributions (Equation 4.1) costs $O(HD)$ for NADE. This complexity is comparable to that of regular feed-forward neural network models.

NADE can be trained by maximum likelihood, or equivalently by minimizing the average negative log-likelihood,

$$\frac{1}{N} \sum_{n=1}^{N} - \log p\left(\boldsymbol{x}^{(n)}\right) = \frac{1}{N} \sum_{n=1}^{N} \sum_{d=1}^{D} - \log p\left(x_{o_d}^{(n)} \mid \boldsymbol{x}_{o_{<d}}^{(n)}\right), \tag{4.6}$$

usually by stochastic (minibatch) gradient descent. As probabilities $p(\boldsymbol{x})$ cost $O(HD)$, gradients of the negative log-probability of training examples can also be computed in $O(HD)$. Algorithm 1 describes the computation of both $p(\boldsymbol{x})$ and the gradients of $-\log p(\boldsymbol{x})$ with respect to NADE's parameters.

4.2.1 Relationship with the RBM

The proposed weight-tying for NADE isn't simply motivated by computational reasons. It also reflects the computations of approximation inference in the RBM.

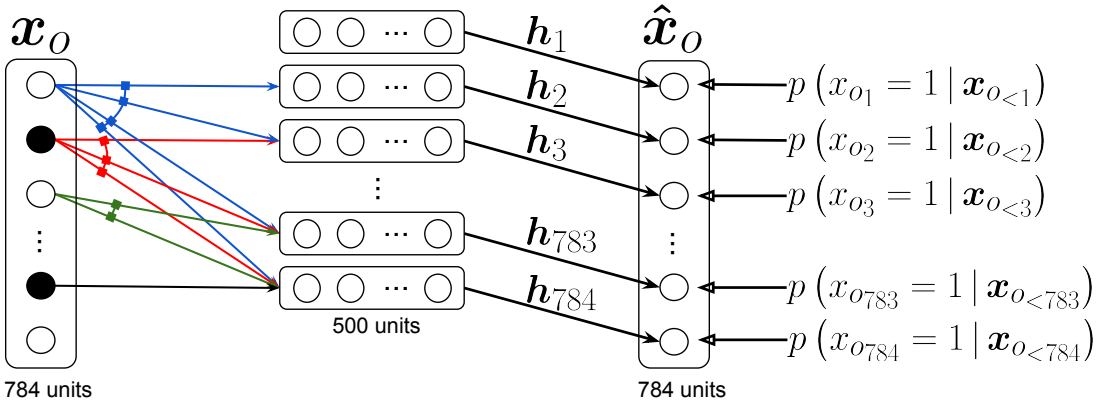


Figure 4.1: Illustration of a NADE model. In this example, in the input layer, units with value 0 are shown in black while units with value 1 are shown in white. The outputs $\hat{\boldsymbol{x}}_O$ give predictive probabilities for each dimension of a vector $\boldsymbol{x}_O$, given elements earlier in some ordering. There is no path of connections between an output and the value being predicted, or elements of $\boldsymbol{x}_O$ later in the ordering. Arrows connected together correspond to connections with shared (tied) parameters.

Denoting the energy function and distribution under an RBM as

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\boldsymbol{h}^\top \boldsymbol{W}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x} - \boldsymbol{c}^\top \boldsymbol{h} \tag{4.7}$$

$$p(\boldsymbol{x}, \boldsymbol{h}) = \exp\left\{-E(\boldsymbol{x}, \boldsymbol{h})\right\} / Z\,, \tag{4.8}$$

computing all conditionals

$$p(\boldsymbol{x}_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}) = \sum_{\boldsymbol{x}_{o_{>d}} \in \{0,1\}^{D-d}} \sum_{\boldsymbol{h} \in \{0,1\}^H} \exp\left\{-E(\boldsymbol{x}, \boldsymbol{h})\right\} / Z(\boldsymbol{x}_{o_{<d}}) \tag{4.9}$$

$$Z(\boldsymbol{x}_{\boldsymbol{o_{<d}}}) = \sum_{\boldsymbol{x}_{o_{\geq d}} \in \{0,1\}^{D-d+1}} \sum_{\boldsymbol{h} \in \{0,1\}^H} \exp\left\{-E(\boldsymbol{x}, \boldsymbol{h})\right\} \tag{4.10}$$

is intractable. However, these could be approximated using mean-field variational inference. Specifically, consider the conditional over x_{o_d}, $\boldsymbol{x}_{o_{>d}}$ and $\boldsymbol{h}$ instead:

$$p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) = \exp\left\{-E(\boldsymbol{x}, \boldsymbol{h})\right\} / Z(\boldsymbol{x}_{o_{<d}}). \tag{4.11}$$

A mean-field approach could first approximate this conditional with a factorized dis-

Algorithm 1 Computation of $p(\boldsymbol{x})$ and learning gradients for NADE.

Input: training observation vector $\boldsymbol{x}$ and ordering o of the input dimensions.
Output: $p(\boldsymbol{x})$ and gradients of $-\log p(\boldsymbol{x})$ on parameters.

Computing $p(\boldsymbol{x})$
$\boldsymbol{a}_1 \leftarrow \boldsymbol{c}$
$p(\boldsymbol{x}) \leftarrow 1$
for d from 1 to D **do**
  $\boldsymbol{h}_d \leftarrow \text{sigm}\left(\boldsymbol{a}_d\right)$
  $p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}) \leftarrow \text{sigm}\left(\boldsymbol{V}_{o_d,\cdot}\boldsymbol{h}_d + b_{o_d}\right)$
  $p(\boldsymbol{x}) \leftarrow p(\boldsymbol{x})\left(p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}})^{x_{o_d}} + (1 - p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}))^{1-x_{o_d}}\right)$
  $\boldsymbol{a}_{d+1} \leftarrow \boldsymbol{a}_d + \boldsymbol{W}_{\cdot,o_d} x_{o_d}$
end for

Computing gradients of $-\log p(\boldsymbol{x})$
$\delta\boldsymbol{a}_D \leftarrow 0$
$\delta\boldsymbol{c} \leftarrow 0$
for d from D to 1 **do**
  $\delta b_{o_d} \leftarrow \left(p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}) - x_{o_d}\right)$
  $\delta\boldsymbol{V}_{o_d,\cdot} \leftarrow \left(p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}) - x_{o_d}\right)\boldsymbol{h}_d^\top$
  $\delta\boldsymbol{h}_d \leftarrow \left(p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}) - x_{o_d}\right)\boldsymbol{V}_{o_d,\cdot}^\top$
  $\delta\boldsymbol{c} \leftarrow \delta\boldsymbol{c} + \delta\boldsymbol{h}_d \odot \boldsymbol{h}_d \odot (1 - \boldsymbol{h}_d)$
  $\delta\boldsymbol{W}_{\cdot,o_d} \leftarrow \delta\boldsymbol{a}_d x_{o_d}$
  $\delta\boldsymbol{a}_{d-1} \leftarrow \delta\boldsymbol{a}_d + \delta\boldsymbol{h}_d \odot \boldsymbol{h}_d \odot (1 - \boldsymbol{h}_d)$
end for
return $p(\boldsymbol{x}), \delta\boldsymbol{b}, \delta\boldsymbol{V}, \delta\boldsymbol{c}, \delta\boldsymbol{W}$

tribution

$$q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) = \mu_i(d)^{x_{o_d}}(1-\mu_d(d))^{1-x_{o_d}} \prod_{j>d} \mu_j(d)^{x_{o_j}}(1-\mu_j(d))^{1-x_{o_j}} \prod_k \tau_k(d)^{h_k}(1-\tau_k(d))^{1-h_k}, \tag{4.12}$$

where $\mu_j(d)$ is the marginal probability of x_{o_j} being equal to 1, given $\boldsymbol{x}_{o_{<d}}$. Similarly, $\tau_k(d)$ is the marginal for hidden variable h_k. The dependence on d comes from conditioning on $\boldsymbol{x}_{o_{<d}}$, that is on the first $d-1$ dimensions of $\boldsymbol{x}$ in ordering o.

For some d, a mean-field approximation is obtained by finding the parameters $\mu_j(d)$

for $j \in \{d, \ldots, D\}$ and $\tau_k(d)$ for $k \in \{1, \ldots, H\}$ which minimize the KL divergence between $q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}})$ and $p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}})$. This is usually done by finding message passing updates that each set the derivatives of the KL divergence to 0 for some of the parameters of $q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}})$ given others.

For some d, let us fix $\mu_j(d) = x_{o_d}$ for $j < d$, leaving only $\mu_j(d)$ for $j > d$ to be found. The KL-divergence develops as follows:

$$
\begin{aligned}
& \mathrm{KL}(q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \,||\, p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}})) \\
= & -\sum_{x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h}} q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \log p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \\
& + \sum_{x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h}} q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \log q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \\
= & \log Z(\boldsymbol{x}_{o_{<d}}) - \sum_j \sum_k \tau_k(d) W_{k,o_j} \mu_j(d) - \sum_j b_{o_j} \mu_j(d) - \sum_k c_k \tau_k(d) \\
& + \sum_{j \geq d} (\mu_j(d) \log \mu_j(d) + (1 - \mu_j(d)) \log(1 - \mu_j(d))) \\
& + \sum_k (\tau_k(d) \log \tau_k(d) + (1 - \tau_k(d)) \log(1 - \tau_k(d))) \,.
\end{aligned}
$$

Then, we can take the derivative with respect to $\tau_k(d)$ and set it to 0, to obtain:

$$
\begin{aligned}
0 &= \frac{\partial \mathrm{KL}(q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \,||\, p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}))}{\partial \tau_k(d)} \\
0 &= -c_k - \sum_j W_{k,o_j} \mu_j(d) + \log\left(\frac{\tau_k(d)}{1 - \tau_k(d)}\right) \\
\frac{\tau_k(d)}{1 - \tau_k(d)} &= \exp\left\{c_k + \sum_j W_{k,o_j} \mu_j(d)\right\} \qquad (4.13) \\
\tau_k(d) &= \frac{\exp\left\{c_k + \sum_j W_{k,o_j} \mu_j(d)\right\}}{1 + \exp\left\{c_k + \sum_j W_{k,o_j} \mu_j(d)\right\}} \\
\tau_k(d) &= \mathrm{sigm}\left(c_k + \sum_{j \geq d} W_{k,o_j} \mu_j(d) + \sum_{j < d} W_{k,o_j} x_{o_j}\right). \qquad (4.14)
\end{aligned}
$$

where in the last step we have used the fact that $\mu_j(d) = x_{o_j}$ for $j < d$. Equation 4.14

would correspond to the message passing updates of the hidden unit marginals $\tau_k(d)$ given the marginals of input $\mu_j(d)$.

Similarly, we can set the derivative with respect to $\mu_j(d)$ for $j \geq d$ to 0 and obtain:

$$\begin{aligned}
0 &= \frac{\partial \mathrm{KL}(q(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}) \,||\, p(x_{o_d}, \boldsymbol{x}_{o_{>d}}, \boldsymbol{h} \,|\, \boldsymbol{x}_{o_{<d}}))}{\partial \mu_j(d)} \\
0 &= -b_{o_d} - \sum_k \tau_k(d) W_{k,o_j} + \log\left(\frac{\mu_j(d)}{1-\mu_j(d)}\right) \\
\frac{\mu_j(d)}{1-\mu_j(d)} &= \exp\left\{b_{o_j} + \sum_k \tau_k(d) W_{k,o_j}\right\} \\
\mu_j(d) &= \frac{\exp\left\{b_{o_j} + \sum_k \tau_k(d) W_{k,o_j}\right\}}{1 + \exp\left\{b_{o_j} + \sum_k \tau_k(d) W_{k,o_j}\right\}} \\
\mu_j(d) &= \mathrm{sigm}\left(b_{o_j} + \sum_k \tau_k(d) W_{k,o_j}\right). \qquad (4.15)
\end{aligned}$$

Equation 4.15 would correspond to the message passing updates of the input marginals $\mu_j(d)$ given the hidden layer marginals $\tau_k(d)$. The complete mean-field algorithm would thus alternate between applying the updates of Equations 4.14 and 4.15, right to left.

We now notice that Equation 4.14 corresponds to NADE's hidden layer computation (Equation 4.3) where $\mu_j(d) = 0 \;\; \forall j \geq d$. Also, Equation 4.15 corresponds to NADE's output layer computation (Equation 4.2) where $j = d$, $\tau_k(d) = h_{d,k}$ and $\boldsymbol{W}^\top = \boldsymbol{V}$. Thus, in short, NADE's forward pass is equivalent to applying a single pass of mean-field inference to approximate all the conditionals $p(\boldsymbol{x}_{o_d} \,|\, \boldsymbol{x}_{o_{<d}})$ of an RBM, where initially $\mu_j(d) = 0$ and where a separate matrix $\boldsymbol{V}$ is used for the hidden-to-input messages. A generalization of NADE based on this connection to mean field inference has been further explored by Raiko and Bengio [2014].

4.3 NADE for Non-Binary Observations

So far we have only considered the case of binary observations x_i. However, the framework of NADE naturally extends to distributions over other types of observations.

In the next section, we discuss the case of real-valued observations, which is one of the most general cases of non-binary observations and provides an illustrative example of the technical considerations one faces when extending NADE to new observations.

4.3.1 RNADE: Real-Valued NADE

A NADE model for real-valued data could be obtained by applying the derivations shown in Section 4.2.1 to the Gaussian-RBM [Welling et al., 2005]. The resulting neural network would output the mean of a Gaussian with fixed variance for each of the conditionals in Equation 4.1. Such a model is not competitive with mixture models, for example on perceptual data sets [Uria, 2015]. However, we can explore alternative models by making the neural network for each conditional distribution output the parameters of a distribution that's not a fixed-variance Gaussian.

In particular, a mixture of one-dimensional Gaussians for each autoregressive conditional provides a flexible model. Given enough components, a mixture of Gaussians can model any continuous distribution to arbitrary precision. The resulting model can be interpreted as a sequence of mixture density networks [Bishop, 1994] with shared parameters. We call this model RNADE-MoG. In RNADE-MoG, each of the conditionals is modeled by a mixture of Gaussians:

$$p(x_{o_d} \mid \boldsymbol{x}_{o_{<d}}) = \sum_{c=1}^{C} \pi_{o_d,c}\, \mathcal{N}(x_{o_d};\, \mu_{o_d,c},\, \sigma^2_{o_d,c}), \tag{4.16}$$

where the parameters are set by the outputs of a neural network:

$$\pi_{o_d,c} = \frac{\exp\left\{z^{(\pi)}_{o_d,c}\right\}}{\sum_{c=1}^{C}\exp\left\{z^{(\pi)}_{o_d,c}\right\}} \tag{4.17}$$

$$\mu_{o_d,c} = z^{(\mu)}_{o_d,c} \tag{4.18}$$

$$\sigma_{o_d,c} = \exp\left\{z^{(\sigma)}_{o_d,c}\right\} \tag{4.19}$$

$$z^{(\pi)}_{o_d,c} = b^{(\pi)}_{o_d,c} + \sum_{k=1}^{H} V^{(\pi)}_{o_d,k,c} h_{d,k} \tag{4.20}$$

$$z^{(\mu)}_{o_d,c} = b^{(\mu)}_{o_d,c} + \sum_{k=1}^{H} V^{(\mu)}_{o_d,k,c} h_{d,k} \tag{4.21}$$

$$z^{(\sigma)}_{o_d,c} = b^{(\sigma)}_{o_d,c} + \sum_{k=1}^{H} V^{(\sigma)}_{o_d,k,c} h_{d,k} \tag{4.22}$$

Parameter sharing conveys the same computational and statistical advantages as it does in the binary NADE.

Different one dimensional conditional forms may be preferred, for example due to limited data set size or domain knowledge about the form of the conditional distributions. Other choices, like single variable-variance Gaussians, sinh-arcsinh distributions, and mixtures of Laplace distributions, have been examined by Uria [2015].

Training an RNADE can still be done by stochastic gradient descent on the parameters of the model with respect to the negative log-density of the training set. It was found empirically [Uria et al., 2013] that stochastic gradient descent leads to better parameter configurations when the gradient of the mean $\left(\frac{\partial J}{\partial \mu_{o_d,c}}\right)$ was multiplied by the standard deviation ($\sigma_{o_d,c}$).

4.4 Orderless and Deep NADE

The fixed ordering of the variables in a NADE model makes the exact calculation of arbitrary conditional probabilities computationally intractable. Only a small subset of conditional distributions, those where the conditioned variables are at the be-

ginning of the ordering and marginalized variables at the end, are computationally tractable.

Another limitation of NADE is that a naive extension to a deep version, with multiple layers of hidden units, is computationally expensive. Deep neural networks [Bengio, 2009; LeCun et al., 2015] are at the core of state-of-the-art models for supervised tasks like image recognition [Krizhevsky et al., 2012] and speech recognition [Dahl et al., 2013]. The same inductive bias should also provide better unsupervised models. However, extending the NADE framework to network architectures with several hidden layers, by introducing extra non-linear calculations between Equations 4.3 and 4.2, increases its complexity to cubic in the number of units per layer. Specifically, the cost becomes $O(DH^2L)$, where L stands for the number of hidden layers and can be assumed to be a small constant, D is the number of variables modeled, and H is the number of hidden units, which we assumed to be of the same order as D. This increase in complexity is caused by no longer being able to share hidden layer computations across the conditionals in Equation 4.1, after the non-linearity in the first layer.

In this section we describe an order-agnostic training procedure, DeepNADE [Uria et al., 2014], which will address both of the issues above. This procedure trains a single deep neural network that can assign a conditional distribution to any variable given any subset of the others. This network can then provide the conditionals in Equation 4.1 for any ordering of the input observations. Therefore, the network defines a factorial number of different models with shared parameters, one for each of the $D!$ orderings of the inputs. At test time, given an inference task, the most convenient ordering of variables can be used. The models for different orderings will not be consistent with each other: they will assign different probabilities to a given test vector. However, we can use the models' differences to our advantage by creating ensembles of NADE models (Section 4.4.1), which results in better estimators than any single NADE. Moreover, the training complexity of our procedure increases linearly with the number of hidden layers $O(H^2L)$, while remaining quadratic in the size of the network's layers.

We first describe the model for an L-layer neural network modeling binary variables. A

conditional distribution is obtained directly from a hidden unit in the final layer:

$$p(x_{o_d}=1 \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d) = \boldsymbol{h}^{(L)}_{o_d}. \tag{4.23}$$

This hidden unit is computed from previous layers, all of which can only depend on the $\boldsymbol{x}_{o_{<d}}$ variables that are currently being conditioned on. We remove the other variables from the computation using a binary mask,

$$\boldsymbol{m}_{o_{<d}} = [1_{1\in o_{<d}}, 1_{2\in o_{<d}}, \ldots, 1_{D\in o_{<d}}], \tag{4.24}$$

which is element-wise multiplied with the inputs before computing the remaining layers as in a standard neural network:

$$\boldsymbol{h}^{(0)} = \boldsymbol{x} \odot \boldsymbol{m}_{o_{<d}} \tag{4.25}$$

$$\boldsymbol{a}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \tag{4.26}$$

$$\boldsymbol{h}^{(\ell)} = \boldsymbol{g}\left(\boldsymbol{a}^{(\ell)}\right) \tag{4.27}$$

$$\boldsymbol{h}^{(L)} = \mathbf{sigm}\left(\boldsymbol{a}^{(L)}\right). \tag{4.28}$$

The network is specified by a free choice of the activation function $\boldsymbol{g}(\cdot)$, and learnable parameters $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{H^{(\ell)}\times H^{(\ell-1)}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{H^{(\ell)}}$, where $H^{(l)}$ is the number of units in the ℓ-th layer. As layer zero is the masked input, $H^{(0)} = D$. The final L-th layer needs to be able to provide predictions for any element (Equation 4.23) and so also has D units.

To train a DeepNADE, the ordering of the variables is treated as a stochastic variable with a uniform distribution. Moreover, since we wish DeepNADE to provide good predictions for any ordering, we optimize the expected likelihood over the ordering of variables:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{o\in D!} -\log p(\boldsymbol{X} \,|\, \boldsymbol{\theta}, o) \propto \mathop{\mathbb{E}}_{o\in D!} \mathop{\mathbb{E}}_{\boldsymbol{x}\in\mathcal{X}} -\log p(\boldsymbol{x} \,|\, \boldsymbol{\theta}, o), \tag{4.29}$$

where we've made the dependence on the ordering o and the network's parameters $\boldsymbol{\theta}$ explicit, $D!$ stands for the set of all orderings (the permutations of D elements) and $\boldsymbol{x}$ is a uniformly sampled data point from the training set $\mathcal{X}$. Using NADE's expression

for the density of a data point in Equation 4.1 we have

$$\mathcal{J}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{o \in D!} \mathop{\mathbb{E}}_{\boldsymbol{x} \in \mathcal{X}} \sum_{d=1}^{D} -\log p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o), \tag{4.30}$$

where d indexes the elements in the ordering, o, of the variables. By moving the expectation over orderings inside the sum over the elements of the ordering, the ordering can be split in three parts: $o_{<d}$ (the indices of the $d-1$ first dimensions in the ordering), o_d (the index of the d-th variable) and $o_{>d}$ (the indices of the remaining dimensions). Therefore, the loss function can be rewritten as:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\boldsymbol{x} \in \mathcal{X}} \sum_{d=1}^{D} \mathop{\mathbb{E}}_{o_{<d}} \mathop{\mathbb{E}}_{o_d} \mathop{\mathbb{E}}_{o_{>d}} -\log p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d, o_{>d}). \tag{4.31}$$

The value of each of these terms does not depend on $o_{>d}$. Therefore, it can be simplified as:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\boldsymbol{x} \in \mathcal{X}} \sum_{d=1}^{D} \mathop{\mathbb{E}}_{o_{<d}} \mathop{\mathbb{E}}_{o_d} -\log p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d). \tag{4.32}$$

In practice, this loss function will have a very high number of terms and will have to be approximated by sampling $\boldsymbol{x}$, d and $o_{<d}$. The innermost expectation over values of o_d can be calculated cheaply, because all of the neural network computations depend only on the masked input $\boldsymbol{x}_{o_{<d}}$, and can be reused for each possible o_d. Assuming all orderings are equally probable, we will estimate $\mathcal{J}(\boldsymbol{\theta})$ by:

$$\widehat{\mathcal{J}}(\boldsymbol{\theta}) = \frac{D}{D-d+1} \sum_{o_d} -\log p(x_{o_d} \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d), \tag{4.33}$$

which is an unbiased estimator of Equation 4.29. Therefore, training can be done by descent on the gradient of $\widehat{\mathcal{J}}(\boldsymbol{\theta})$.

For binary observations, we use the cross-entropy scaled by a factor of $\frac{D}{D-d+1}$ as the training loss which corresponds to minimizing $\widehat{\mathcal{J}}$:

$$\mathcal{J}(\boldsymbol{x}) = \frac{D}{D-d+1} \, \boldsymbol{m}_{o_{\geq d}}^{\top} \left(\boldsymbol{x} \odot \log\left(\boldsymbol{h}^{(L)}\right) + (1-\boldsymbol{x}) \odot \log\left(1-\boldsymbol{h}^{(L)}\right) \right). \tag{4.34}$$

Differentiating this cost involves backpropagating the gradients of the cross-entropy

only from the outputs in $o_{\geq d}$ and rescaling them by $\frac{D}{D-d+1}$.

The resulting training procedure resembles that of a denoising autoencoder [Vincent et al., 2008]. Like the autoencoder, D outputs are used to predict D inputs corrupted by a random masking process ($\boldsymbol{m}_{o_{<d}}$ in Equation 4.25). A single forward pass can compute $\boldsymbol{h}^{(L)}_{o_{\geq d}}$, which provides a prediction $p(x_{o_d} = 1 \mid \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d)$ for every masked variable, which could be used next in an ordering starting with $o_{<d}$. Unlike the autoencoder, the outputs for variables corresponding to those provided in the input (not masked out) are ignored.

In this order-agnostic framework, missing variables and zero-valued observations are indistinguishable by the network. This shortcoming can be alleviated by concatenating the inputs to the network (masked variables $\boldsymbol{x} \odot \boldsymbol{m}_{o_{<d}}$) with the mask $\boldsymbol{m}_{o_{<d}}$. Therefore we advise substituting the input described in Equation 4.25 with

$$\boldsymbol{h}^{(0)} = \text{concat}(\boldsymbol{x} \odot \boldsymbol{m}_{o_{<d}}, \boldsymbol{m}_{o_{<d}}). \tag{4.35}$$

We found this modification to be important in order to obtain competitive statistical performance (see Table 4.3). The resulting neural network is illustrated in Figure 4.2.

4.4.1 Ensembles of NADE Models

As mentioned, the DeepNADE parameter fitting procedure effectively produces a factorial number of different NADE models, one for each ordering of the variables. These models will not, in general, assign the same probability to any particular data point. This disagreement is undesirable if we require consistent inferences for different inference problems, as it will preclude the use of the most convenient ordering of variables for each inference task.

However, it is possible to use this variability across the different orderings to our advantage by combining several models. A usual approach to improve on a particular estimator is to construct an ensemble of multiple, strong but different estimators, e.g. using bagging [Ormoneit and Tresp, 1995] or stacking [Smyth and Wolpert, 1999]. The DeepNADE training procedure suggests a way of generating ensembles of NADE

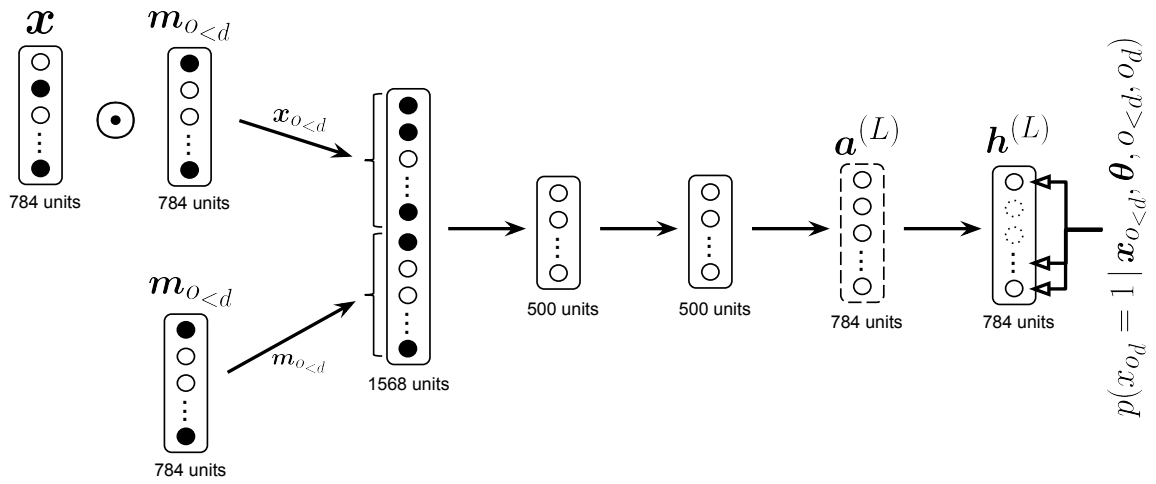


Figure 4.2: Illustration of a DeepNADE model with two hidden layers. The dashed border represents a layer pre-activation. Units with value 0 are shown in black while units with value 1 are shown in white. A mask $\boldsymbol{m}_{o_{<d}}$ specifies a subset of variables to condition on. A conditional or predictive probability of the remaining variables is given in the final layer. The output units with a corresponding input mask of value 1 (shown with dotted contour) are not involved in DeepNADE's training loss (Equation 4.34).

models: take a set of uniformly distributed orderings $\{o^{(k)}\}_{k=1}^{K}$ over the input variables and use the average probability $\frac{1}{K}\sum_{k=1}^{K} p(\boldsymbol{x} \mid \theta, o^{(k)})$ as an estimator.

The use of an ensemble increases the test-time cost of density estimation linearly with the number of orderings used. The complexity of sampling does not change however: after one of the K orderings is chosen at random, the single corresponding NADE is sampled. Importantly, the cost of training also remains the same, unlike other ensemble methods such as bagging. Furthermore, the number of components can be chosen after training and even adapted to a computational budget on the fly.

4.5 ConvNADE: Convolutional NADE

One drawback of NADE (and its variants so far) is the lack of a mechanism for truly exploiting the high-dimensional structure of the data. For example, when using NADE on binarized MNIST, we first need to flatten the 2D images before providing them to the model as a vector. As the spatial topology is not provided to the network, it can't use this information to share parameters and may learn less quickly.

Recently, convolutional neural networks (CNN) have achieved state-of-the-art performance on many supervised tasks related to images Krizhevsky et al. [2012]. Briefly, CNNs are composed of convolutional layers, each one having multiple learnable filters. The outputs of a convolutional layer are feature maps and are obtained by the convolution on the input image (or previous feature maps) of a linear filter, followed by the addition of a bias and the application of a non-linear activation function. Thanks to the convolution, spatial structure in the input is preserved and can be exploited. Moreover, as per the definition of a convolution the same filter is reused across all subregions of the entire image (or previous feature maps), yielding a parameter sharing that is natural and sensible for images.

The success of CNNs raises the question: can we exploit the spatial topology of the inputs while keeping NADE's autoregressive property? It turns out we can, simply by replacing the fully connected hidden layers of a DeepNADE model with convolutional layers. We thus refer to this variant as Convolutional NADE (ConvNADE).

First we establish some notation that we will use throughout this section. Without loss of generality, let the input $\boldsymbol{X} \in \{0,1\}^{N_X \times N_X}$ be a square binary image of size N_X and every convolution filter $\boldsymbol{W}_{ij}^{(\ell)} \in \mathbb{R}^{N_W^{(\ell)} \times N_W^{(\ell)}}$ connecting two feature maps $\boldsymbol{H}_i^{(\ell-1)}$ and $\boldsymbol{H}_j^{(\ell)}$ also be square with their size $N_W^{(\ell)}$ varying for each layer ℓ. We also define the following mask $\boldsymbol{M}_{o_{<d}} \in \{0,1\}^{N_X \times N_X}$, which is 1 for the locations of the first $d-1$ pixels in the ordering o.

Formally, Equation 4.26 is modified to use convolutions instead of dot products. Specifically for an L-layer convolutional neural network that preserves the input shape (explained below) we have

$$p(x_{o_d} = 1 \,|\, \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d) = \mathbf{vec}\left(\boldsymbol{H}_1^{(L)}\right)_{o_d}, \tag{4.36}$$

with

$$\boldsymbol{H}_1^{(0)} = \boldsymbol{X} \odot \boldsymbol{M}_{o_{<d}} \tag{4.37}$$

$$\boldsymbol{A}_j^{(\ell)} = b_j^{(\ell)} + \sum_{i=1}^{H^{(\ell-1)}} \boldsymbol{H}_i^{(\ell-1)} * \boldsymbol{W}_{ij}^{(\ell)} \tag{4.38}$$

$$\boldsymbol{H}_j^{(\ell)} = \boldsymbol{g}\left(\boldsymbol{A}_j^{(\ell)}\right) \tag{4.39}$$

$$\boldsymbol{H}_j^{(L)} = \mathbf{sigm}\left(\boldsymbol{A}_j^{(L)}\right), \tag{4.40}$$

where $H^{(\ell)}$ is the number of feature maps output by the ℓ-th layer and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{H^{(l)}}$, $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{H^{(\ell-1)} \times H^{(\ell)} \times N_W^{(\ell)} \times N_W^{(\ell)}}$, with $\odot$ denoting the element-wise multiplication, $\boldsymbol{g}(\cdot)$ being any activation function and $\mathbf{vec}(\boldsymbol{X}) \to \boldsymbol{x}$ is the concatenation of every row in $\boldsymbol{X}$. Note that $H^{(0)}$ corresponds to the number of channels the input images have.

For notational convenience, we use $*$ to denote both "valid" convolutions and "full" convolutions, instead of introducing bulky notations to differentiate these cases. The "valid" convolutions only apply a filter to complete patches of the image, resulting in a smaller image (its shape is decreased to $N_X - N_W^{(\ell)} + 1$). Alternatively, "full" convolutions zero-pad the contour of the image before applying the convolution, thus expanding the image (its shape is increased to $N_X + N_W^{(\ell)} - 1$). Which one is used should be self-explanatory depending on the context. Note that we only use convolutions with a stride of 1.

Moreover, in order for ConvNADE to output conditional probabilities as shown in Equation 4.36, the output layer must have only one feature map $\boldsymbol{H}_1^{(L)}$, whose dimension matches the dimension of the input $\boldsymbol{X}$. This can be achieved by carefully combining layers that use either "valid" or "full" convolutions.

To explore different model architectures respecting that constraint, we opted for the following strategy. Given a network, we ensured the first half of its layers was using "valid" convolutions while the other half would use "full" convolutions. In addition to that, we made sure the network was symmetric with respect to its filter shapes (i.e. the filter shape used in layer ℓ matched the one used in layer $L - \ell$).

For completeness, we wish to mention that ConvNADE can also include pooling and upsampling layers, but we did not see much improvement when using them. In fact, recent research suggests that these types of layers are not essential to obtain state-of-the-art results [Springenberg et al., 2015].

The flexibility of DeepNADE allows us to easily combine both convolutional and fully connected layers. To create such hybrid models, we used the simple strategy of having two separate networks, with their last layer fused together at the end. The 'convnet' part is only composed of convolutional layers whereas the 'fullnet' part is only composed of fully connected layers. The forward pass of both networks follows respectively Equations 4.37–4.39 and Equations 4.25–4.27. Note that in the 'fullnet' network case, $\boldsymbol{x}$ corresponds to the input image having been flattened.

In the end, the output layer $\boldsymbol{g}$ of the hybrid model corresponds to the aggregation of the last layer pre-activation of both 'convnet' and 'fullnet' networks. The conditionals are slightly modified as follows:

$$p(x_{o_d} = 1 \mid \boldsymbol{x}_{o_{<d}}, \boldsymbol{\theta}, o_{<d}, o_d) = \boldsymbol{g}_{o_d} \tag{4.41}$$

$$\boldsymbol{g} = \mathbf{sigm}\left(\mathbf{vec}\left(\boldsymbol{A}_1^{(L)}\right) + \boldsymbol{a}^{(L)}\right). \tag{4.42}$$

The same training procedure as for DeepNADE model can also be used for ConvNADE. For binary observations, the training loss is similar to Equation 4.34, with $\boldsymbol{h}^{(L)}$ being substituted for $\boldsymbol{g}$ as defined in Equation 4.42.

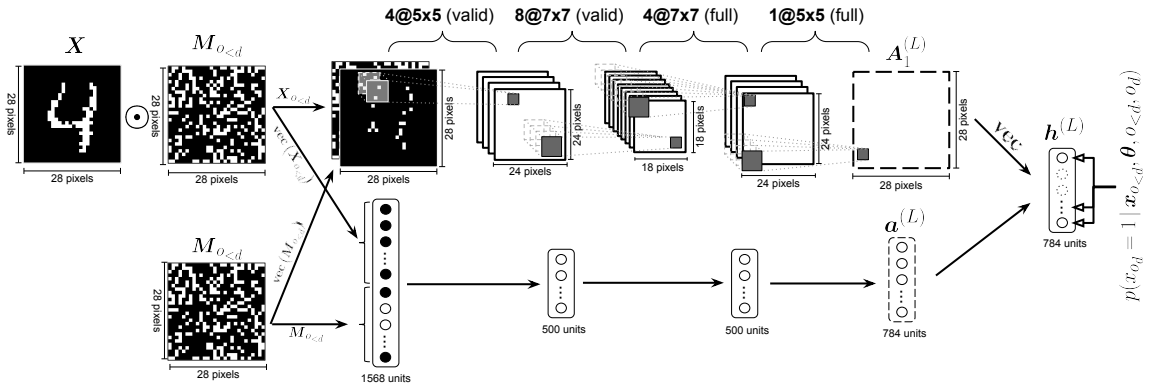


Figure 4.3: Illustration of a ConvNADE that combines a convolutional neural network with three hidden layers and a fully connected feed-forward neural network with two hidden layers. The dashed border represents a layer pre-activation. Units with a dotted contour are not valid conditionals since they depend on themselves i.e. they were given in the input.

As for the DeepNADE model, we found that providing the mask $\boldsymbol{M}_{o_{<d}}$ as an input to the model improves performance (see Table 4.4). For the 'convnet' part, the mask was provided as an additional channel to the input layer. For the 'fullnet' part, the inputs were concatenated with the mask as shown in Equation 4.35.

The final architecture is shown in Figure 4.3. In our experiments, we found that this type of hybrid model works better than only using convolutional layers (see Table 4.4). Certainly, more complex architectures could be employed but this is a topic left for future work.

4.6 Related Work

As we mentioned earlier, the development of NADE and its extensions was motivated by the question of whether a tractable distribution estimator could be designed to match a powerful but intractable model such as the restricted Boltzmann machine.

The original inspiration came from the autoregressive approach taken by fully visible sigmoid belief networks (FVSBN), which were shown by Frey et al. [1996] to

be surprisingly competitive, despite the simplicity of the distribution family for its conditionals. Bengio and Bengio [2000] later proposed using more powerful conditionals, modeled as single layer neural networks. Moreover, they proposed connecting the output of each d^{th} conditional to all of the hidden layers of the $d-1$ neural networks for the preceding conditionals. More recently, Germain et al. [2015] generalized this model by deriving a simple procedure for making it deep and orderless (akin to DeepNADE, in Section 4.4). We compare with all of these approaches in Section 4.7.1.

There exists, of course, more classical and non-autoregressive approaches to tractable distribution estimation, such as mixture models and Chow–Liu trees [Chow and Liu, 1968]. We compare with these as well in Section 4.7.1.

This work also relates directly to the recently growing literature on generative neural networks. In addition to the autoregressive approach described in this paper, there exists three other types of such models: directed generative networks, undirected generative networks and hybrid networks.

Work on directed generative networks dates back to the original work on sigmoid belief networks [Neal, 1992] and the Helmholtz machine [Hinton et al., 1995; Dayan et al., 1995]. Helmholtz machines are equivalent to a multilayer sigmoid belief network, with each using binary stochastic units. Originally they were trained using Gibbs sampling and gradient descent [Neal, 1992], or with the so-called wake sleep algorithm [Hinton et al., 1995]. More recently, many alternative directed models and training procedures have been proposed. Kingma and Welling [2013]; Rezende et al. [2014] proposed the variational autoencoder (VAE), where the model is the same as the Helmholtz machine, but with real-valued (usually Gaussian) stochastic units. Importantly, Kingma and Welling [2013] identified a reparameterization trick making it possible to train the VAE in a way that resembles the training of an autoencoder. This approach falls in the family of stochastic variational inference methods, where the encoder network corresponds to the approximate variational posterior. The VAE optimizes a bound on the likelihood which is estimated using a single sample from the variational posterior, though recent work has shown that a better bound can be obtained using an importance sampling approach [Burda et al., 2016]. Gregor

et al. [2014a] later exploited the VAE approach to develop DRAW, a directed generative model for images based on a read-write attentional mechanism. Goodfellow et al. [2014] proposed an adversarial approach to training directed generative networks, that relies on a discriminator network simultaneously trained to distinguish between data and model samples. Generative networks trained this way are referred to as Generative Adversarial Networks (GAN). While the VAE optimizes a bound of the likelihood (which is the KL divergence between the empirical and model distributions), it can be shown that the earliest versions of GANs optimize the Jensen–Shannon (JS) divergence between the empirical and model distributions. Li et al. [2015] instead propose a training objective derived from Maximum Mean Discrepancy [MMD; Gretton et al., 2007]. Recently, the directed generative model approach has been very successfully applied to model images [Denton et al., 2015; Sohl-Dickstein et al., 2011].

The undirected paradigm has also been explored extensively for developing powerful generative networks. These include the restricted Boltzmann machine [Smolensky, 1986; Freund and Haussler, 1992] and its multilayer extension, the deep Boltzmann machine [Salakhutdinov and Hinton, 2009], which dominate the literature on undirected neural networks. Salakhutdinov and Murray [2008] provided one of the first quantitative evidence of the generative modeling power of RBMs, which motivated the original parameterization for NADE [Larochelle and Murray, 2011]. Efforts to train better undirected models can vary in nature. One has been to develop alternative objectives to maximum likelihood. The proposal of Contrastive Divergence [CD; Hinton, 2002] was instrumental in the popularization of the RBM. Other proposals include pseudo-likelihood [Besag, 1975; Marlin et al., 2010a], score matching [Hyvärinen, 2005, 2007b,a], noise contrastive estimation [Gutmann and Hyvärinen, 2010] and probability flow minimization [Sohl-Dickstein et al., 2011]. Another line of development has been to optimize likelihood using Robbins–Monro stochastic approximation [Younes, 1989], also known as Persistent CD [Tieleman, 2008], and develop good MCMC samplers for deep undirected models [Salakhutdinov, 2009, 2010; Desjardins et al., 2010a; Cho et al., 2010]. Work has also been directed towards proposing improved update rules or parameterization of the model's energy function [Tieleman and Hinton, 2009; Cho et al., 2013; Montavon and Müller, 2012] as well as improved approximate inference of the hidden layers [Salakhutdinov and Larochelle, 2010]. The

work of Ngiam et al. [2011] also proposed an undirected model that distinguishes itself from deep Boltzmann machines by having deterministic hidden units, instead of stochastic.

Finally, hybrids of directed and undirected networks are also possible, though much less common. The most notable case is the Deep Belief Network [DBN; Hinton et al., 2006], which corresponds to a sigmoid belief network for which the prior over its top hidden layer is an RBM (whose hidden layer counts as an additional hidden layer). The DBN revived interest in RBMs, as they were required to successfully initialize the DBN.

NADE thus substantially differs from this literature focusing on directed and undirected models, benefiting from a few properties that these approaches lack. Mainly, NADE does not rely on latent stochastic hidden units, making it possible to tractably compute its associated data likelihood for some given ordering. This in turn makes it possible to efficiently produce exact samples from the model (unlike in undirected models) and get an unbiased gradient for maximum likelihood training (unlike in directed graphical models).

4.7 Results

In this section, we evaluate the performance of our different NADE models on a variety of data sets. The code to reproduce the experiments of the paper is available on GitHub [1]. Our implementation is done using Theano [The Theano Development Team et al., 2016].

4.7.1 Binary Vectors Data Sets

We start by evaluating the performance of NADE models on a set of benchmark data sets where the observations correspond to binary vectors. These data sets were

1. `http://github.com/MarcCote/NADE`

Name	# Inputs	Train	Valid.	Test
Adult	123	5000	1414	26147
Connect4	126	16000	4000	47557
DNA	180	1400	600	1186
Mushrooms	112	2000	500	5624
NIPS-0-12	500	400	100	1240
OCR-letters	128	32152	10000	10000
RCV1	150	40000	10000	150000
Web	300	14000	3188	32561

Table 4.1: Statistics on the binary vector data sets of Section 4.7.1.

mostly taken from the LIBSVM data sets web site [2], except for OCR-letters [3] and NIPS-0-12 [4]. Code to download these data sets is available here: `http://info.usherbrooke.ca/hlarochelle/code/nade.tar.gz`. Table 4.1 summarizes the main statistics for these data sets.

For these experiments, we only consider tractable distribution estimators, where we can evaluate $p(\boldsymbol{x})$ on test items exactly. We consider the following baselines:

- **MoB**: A mixture of multivariate Bernoullis, trained using the EM algorithm. The number of mixture components was chosen from $\{32, 64, 128, 256, 512, 1024\}$ based on validation set performance, and early stopping was used to determine the number of EM iterations.
- **RBM**: A restricted Boltzmann machine made tractable by using only 23 hidden units, trained by contrastive divergence with up to 25 steps of Gibbs sampling. The validation set performance was used to select the learning rate from $\{0.005, 0.0005, 0.00005\}$, and the number of iterations over the training set from $\{100, 500, 1000\}$.
- **FVSBN**: Fully visible sigmoid belief network, that models each conditional $p(x_{o_d} \mid \boldsymbol{x}_{o_{<d}})$ with logistic regression. The ordering of inputs was selected randomly. Training was by stochastic gradient descent. The validation set was

2. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`
3. `http://ai.stanford.edu/~btaskar/ocr/`
4. `http://www.cs.nyu.edu/~roweis/data.html`

used for early stopping, as well as for choosing the base learning rate $\eta \in \{0.05, 0.005, 0.0005\}$, and a decreasing schedule constant γ from $\{0, 0.001, 0.000001\}$ for the learning rate schedule $\eta/(1+\gamma t)$ for the t^{th} update.

- **Chow–Liu**: A Chow–Liu tree is a graph over the observed variables, where the distribution of each variable, except the root, depends on a single parent node. There is an $O(D^2)$ fitting algorithm to find the maximum likelihood tree and conditional distributions [Chow and Liu, 1968]. We adapted an implementation provided by Harmeling and Williams [2011], who found Chow–Liu to be a strong baseline.

  The maximum likelihood parameters are not defined when conditioning on events that haven't occurred in the training set. Moreover, conditional probabilities of zero are possible, which could give infinitely bad test set performance. We re-estimated the conditional probabilities on the Chow–Liu tree using Lidstone or "add-α" smoothing:

$$p(x_d = 1 \mid x_{\text{parent}} = z) = \frac{\text{count}(x_d = 1 \mid x_{\text{parent}} = z) + \alpha}{\text{count}(x_{\text{parent}} = z) + 2\alpha}, \tag{4.43}$$

  selecting α for each data set from $\{10^{-20}, 0.001, 0.01, 0.1\}$ based on performance on the validation set.

- **MADE** [Germain et al., 2015]: Generalization of the neural network approach of Bengio and Bengio [2000], to multiple layers. We consider a version using a single (fixed) input ordering and another trained on multiple orderings from which an ensemble was constructed (which was inspired from the order-agnostic approach of Section 4.4) that we refer to as MADE-E. See Germain et al. [2015] for more details.

We compare these baselines with the two following NADE variants:

- **NADE (fixed order)**: Single layer NADE model, trained on a single (fixed) randomly generated order, as described in Section 4.2. The sigmoid activation function was used for the hidden layer, of size 500. Much like for FVSBN, training relied on stochastic gradient descent and the validation set was used

Model	Adult	Connect4	DNA	Mushrooms	NIPS-0-12	OCR-letters	RCV1	Web
MoB	-20.44	-23.41	-98.19	-14.46	-290.02	-40.56	-47.59	-30.16
RBM	-16.26	-22.66	-96.74	-15.15	-277.37	-43.05	-48.88	-29.38
FVSBN	**-13.17**	-12.39	-83.64	-10.27	-276.88	-39.30	-49.84	-29.35
Chow–Liu	-18.51	-20.57	-87.72	-20.99	-281.01	-48.87	-55.60	-33.92
MADE	**-13.12**	**-11.90**	-83.63	-9.68	-280.25	-28.34	-47.10	-28.53
MADE-E	**-13.13**	**-11.90**	**-79.66**	-9.69	-277.28	-30.04	-46.74	**-28.25**
NADE	**-13.19**	-11.99	-84.81	-9.81	**-273.08**	**-27.22**	-46.66	-28.39
NADE-E	**-13.19**	-12.58	-82.31	-9.69	**-272.39**	**-27.32**	**-46.12**	**-27.87**

Table 4.2: Average log-likelihood performance of tractable distribution baselines and NADE models, on binary vector data sets. The best result is shown in bold, along with any other result with an overlapping confidence interval.

for early stopping, as well as for choosing the learning rate from {0.05, 0.005, 0.0005}, and the decreasing schedule constant γ from {0,0.001,0.000001}.

- **NADE-E**: Single layer NADE trained according to the order-agnostic procedure described in Section 4.4. The rectified linear activation function was used for the hidden layer, also of size 500. Minibatch gradient descent was used for training, with minibatches of size 100. The initial learning rate, chosen among {0.016, 0.004, 0.001, 0.00025, 0.0000675}, was linearly decayed to zero over the course of $100,000$ parameter updates. Early stopping was used, using Equation 4.34 to get a stochastic estimate of the validation set average log-likelihood. An ensemble using 16 orderings was used to compute the test-time log-likelihood.

Table 4.2 presents the results. We observe that NADE restricted to a fixed ordering of the inputs achieves very competitive performance compared to the baselines. However, the order-agnostic version of NADE is overall the best method, being among the top performing model for 5 data sets out of 8.

The performance of fixed-order NADE is surprisingly robust to variations of the chosen input ordering. The standard deviation on the average log-likelihood when varying the ordering was small: on Mushrooms, DNA and NIPS-0-12, we observed standard deviations of 0.045, 0.05 and 0.15, respectively. However, models with different orders can do well on different test examples, which explains why ensembling

can still help.

4.7.2 Binary Image Data Set

We now consider the case of an image data set, constructed by binarizing the MNIST digit data set. Each image has been stochastically binarized according to their pixel intensity as generated by Salakhutdinov and Murray [2008]. This benchmark has been a popular choice for the evaluation of generative neural network models. Here, we investigate two questions:

1. How does NADE compare to intractable generative models?
2. Does the use of a convolutional architecture improve the performance of NADE?

For these experiments, in addition to the baselines already described in Section 4.7.1, we consider the following:

- **DARN** [Gregor et al., 2014b]: This deep generative autoencoder has two hidden layers, one deterministic and one with binary stochastic units. Both layers have 500 units (denoted as $n_h = 500$). Adaptive weight noise (adaNoise) was either used or not to avoid the need for early stopping [Graves, 2011]. Evaluation of exact test probabilities is intractable for large latent representations. Hence, Monte Carlo was used to approximate the expected description length, which corresponds to an upper bound on the negative log-likelihood.
- **DRAW** [Gregor et al., 2014a]: Similar to a variational autoencoder where both the encoder and the decoder are LSTMs, guided (or not) by an attention mechanism. In this model, both LSTMs (encoder and decoder) are composed of 256 recurrent hidden units and always perform 64 timesteps. When the attention mechanism is enabled, patches (2×2 pixels) are provided as inputs to the encoder instead of the whole image and the decoder also produces patches (5×5 pixels) instead of a whole image.
- **Pixel RNN** [van den Oord et al., 2016b]: NADE-like model for natural images that is based on convolutional and LSTM hidden units. This model has 7 hidden layers, each composed of 16 units. van den Oord et al. [2016b] proposed a novel

Model	$-\log p$	$\approx$
MoBernoullis K=10	168.95	
MoBernoullis K=500	137.64	
Chow–Liu tree	134.99	
MADE 2hl (32 masks)	86.64	
RBM (500 h, 25 CD steps)		86.34
DBN 2hl		84.55
DARN $n_h = 500$		84.71
DARN $n_h = 500$ (adaNoise)		**84.13**
NADE (fixed order)	88.33	
DeepNADE 1hl (no input masks)	99.37	
DeepNADE 2hl (no input masks)	95.33	
DeepNADE 1hl	92.17	
DeepNADE 2hl	89.17	
DeepNADE 3hl	89.38	
DeepNADE 4hl	89.60	
EoNADE 1hl (2 orderings)	90.69	
EoNADE 1hl (128 orderings)	87.71	
EoNADE 2hl (2 orderings)	87.96	
EoNADE 2hl (128 orderings)	**85.10**	

Table 4.3: Negative log-likelihood test results of models ignorant of the 2D topology on the binarized MNIST data set.

two-dimensional LSTM, named Diagonal BiLSTM, which is used in this model. Unlike our ConvNADE, the ordering is fixed before training and at test time, and corresponds to a scan of the image in a diagonal fashion starting from a corner at the top and reaching the opposite corner at the bottom.

We compare these baselines with some NADE variants. The performance of a basic (fixed-order, single hidden layer) NADE model is provided in Table 4.3 and samples are illustrated in Figure 4.4. More importantly, we will focus on whether the following variants achieve better test set performance:

- **DeepNADE**: Multiple layers (1hl, 2hl, 3hl or 4hl) trained according to the order-agnostic procedure described in Section 4.4. Information about which inputs are masked was either provided or not (no input masks) to the model. The rectified linear activation function was used for all hidden layers. Minibatch

gradient descent was used for training, with minibatches of size 1000. Training consisted of 200 iterations of 1000 parameter updates. Each hidden layer was pre-trained according to Algorithm 2. We report an average of the average test log-likelihoods over ten different random orderings.

- **EoNADE**: This variant is similar to DeepNADE except for the log-likelihood on the test set, which is instead computed from an ensemble that averages predictive probabilities over 2 or 128 orderings. To clarify, the DeepNADE results report the typical performance of one ordering, by averaging results after taking the log, and so do not combine the predictions of the models like EoNADE does.

- **ConvNADE**: Multiple convolutional layers trained according to the order-agnostic procedure described in Section 4.4. The exact architecture is shown in Figure 4.5(a). Information about which inputs are masked was either provided or not (no input masks). The rectified linear activation function was used for all hidden layers. The Adam optimizer [Kingma and Ba, 2015] was used with a learning rate of 10^{-4}. Early stopping was used with a look ahead of 10 epochs, using Equation 4.34 to get a stochastic estimate of the validation set average log-likelihood. An ensemble using 128 orderings was used to compute the log-likelihood on the test set.

- **ConvNADE + DeepNADE**: This variant is similar to ConvNADE except for the aggregation of a separate DeepNADE model at the end of the network. The exact architecture is shown in Figure 4.5(b). The training procedure is the same as with ConvNADE.

Table 4.3 presents the results obtained by models ignorant of the 2D topology, such as the basic NADE model. Addressing the first question, we observe that the order-agnostic version of NADE with two hidden layers is competitive with intractable generative models. Moreover, examples of the ability of DeepNADE to solve inference tasks by marginalization and conditional sampling are shown in Figure 4.6.

Now, addressing the second question, we can see from Table 4.4 that convolutions do improve the performance of NADE. Moreover, we observe that providing informa-

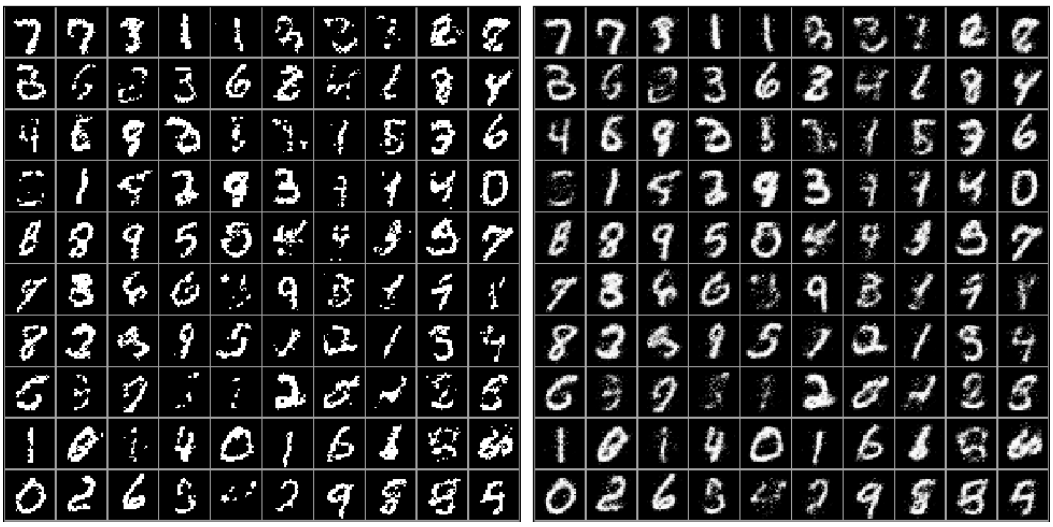
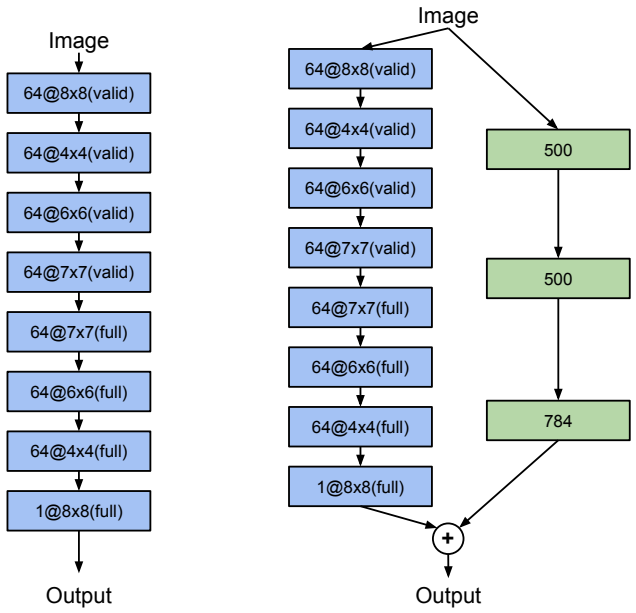


Figure 4.4: **Left**: samples from NADE trained on binarized MNIST. **Right**: probabilities from which each pixel was sampled. Ancestral sampling was used with the same fixed ordering used during training.

Algorithm 2 Pre-training of a NADE with n hidden layers on data set X.

```
procedure PRETRAIN(n, X)
    if n = 1 then
        return RANDOM-ONE-HIDDEN-LAYER-NADE
    else
        nade ← PRETRAIN(n − 1, X)
        nade ← REMOVE-OUTPUT-LAYER(nade)
        nade ← ADD-A-NEW-HIDDEN-LAYER(nade)
        nade ← ADD-A-NEW-OUTPUT-LAYER(nade)
        nade ← TRAIN-ALL(nade,X,iters=20)          ▷ Train for 20 iterations.
        return nade
    end if
end procedure
```



(a) ConvNADE (b) ConvNADE + DeepNADE

Figure 4.5: Network architectures for binarized MNIST. (a) ConvNADE with 8 convolutional layers (depicted in blue). The number of feature maps for a given layer is given by the number before the "@" symbol followed by the filter size and the type of convolution is specified in parentheses. (b) The same ConvNADE combined with a DeepNADE consisting of three fully-connected layers of respectively 500, 500 and 784 units.

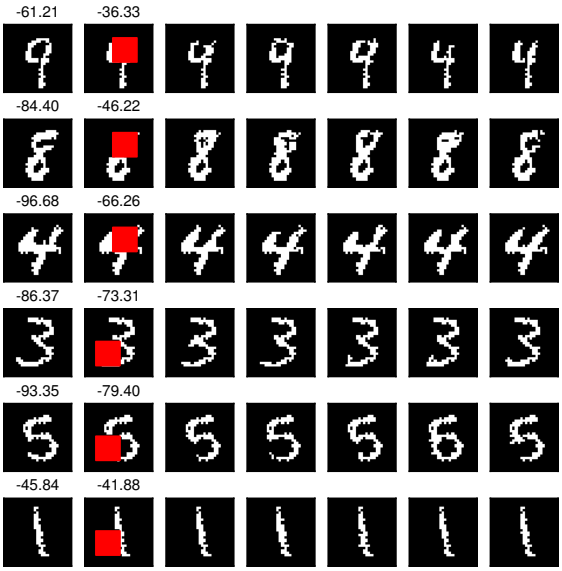


Figure 4.6: Example of marginalization and sampling. The first column shows five examples from the test set of the MNIST data set. The second column shows the density of these examples when a random 10×10 pixel region is marginalized. The right-most five columns show samples for the hollowed region. Both tasks can be done easily with a NADE where the pixels to marginalize are at the end of the ordering.

Model	$-\log p$	$\leq$
DRAW (without attention)		87.40
DRAW		80.97
Pixel RNN	**79.20**	
ConvNADE+DeepNADE (no input masks)	85.25	
ConvNADE	81.30	
ConvNADE+DeepNADE	80.82	

Table 4.4: Negative log-likelihood test results of models exploiting 2D topology on the binarized MNIST data set.

tion about which inputs are masked is essential to obtaining good results. We can also see that combining convolutional and fully-connected layers helps. Even though ConvNADE+DeepNADE performs slightly worse than Pixel RNN, we note that our proposed approach is order-agnostic, whereas Pixel RNN requires a fixed ordering. Figure 4.7 shows samples obtained from the ConvNADE+DeepNADE model using ancestral sampling on a random ordering.

4.7.3 Real-Valued Observations Data Sets

In this section, we compare the statistical performance of RNADE to mixtures of Gaussians (MoG) and factor analyzers (MFA), which are surprisingly strong baselines in some tasks [Tang et al., 2012; Zoran and Weiss, 2012].

Low-dimensional data

We start by considering three UCI data sets [Bache and Lichman, 2013], previously used to study the performance of other density estimators [Silva et al., 2011; Tang et al., 2012], namely: *red wine*, *white wine* and *parkinsons*. These are low dimensional data sets (see Table 4.5) with hard thresholds and non-linear dependencies that make it difficult to fit mixtures of Gaussians or factor analyzers.

Following Tang et al. [2012], we eliminated discrete-valued attributes and an attribute from every pair with a Pearson correlation coefficient greater than 0.98. We

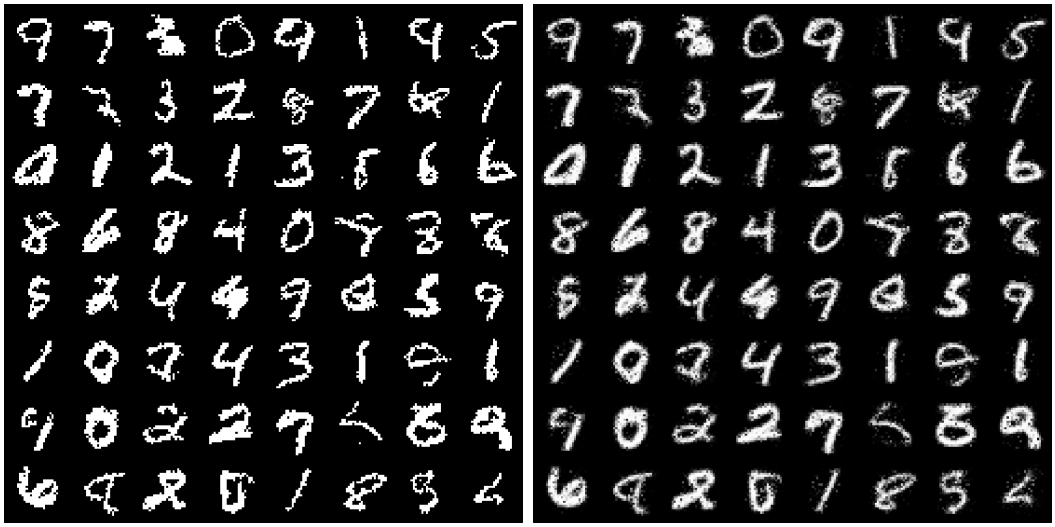


Figure 4.7: **Left**: samples from ConvNADE+DeepNADE trained on binarized MNIST. **Right**: probabilities from which each pixel was sampled. Ancestral sampling was used with a different random ordering for each sample.

normalized each dimension of the data by subtracting its training-subset sample mean and dividing by its standard deviation. All results are reported on the normalized data.

We use full-covariance Gaussians and mixtures of factor analysers as baselines. Models were compared on their log-likelihood on held-out test data. Due to the small size of the data sets (see Table 4.5), we used 10-folds, using 90% of the data for training, and 10% for testing.

We chose the hyperparameter values for each model by doing per-fold cross-validation, using a ninth of the training data as validation data. Once the hyperparameter values have been chosen, we train each model using all the training data (including the validation data) and measure its performance on the 10% of held-out testing data. In order to avoid overfitting, we stopped the training after reaching a training likelihood higher than the one obtained on the best validation-wise iteration of the best validation run. Early stopping was important to avoid overfitting the RNADE models. It also improved the results of the MFAs, but to a lesser degree.

The MFA models were trained using the EM algorithm [Ghahramani and Hinton,

	Red wine	White wine	Parkinsons
Dimensionality	11	11	15
Total number of data points	1599	4898	5875

Table 4.5: Dimensionality and size of the UCI data sets used in Section 4.7.3

1996; Verbeek, 2005]. We cross-validated the number of components and factors. We also selected the number of factors from $2, 4, \ldots D$, where choosing D results in a mixture of Gaussians, and the number of components was chosen among $2, 4, \ldots 50$. Cross-validation selected fewer than 50 components in every case.

We report the performance of several RNADE models using different parametric forms for the one-dimensional conditionals: Gaussian with fixed variance (RNADE-FV), Gaussian with variable variance (RNADE-Gaussian), *sinh-arcsinh* distribution (RNADE-SAS), mixture of Gaussians (RNADE-MoG), and mixture of Laplace distributions (RNADE-MoL). All RNADE models were trained by stochastic gradient descent, using minibatches of size 100, for 500 epochs, each epoch comprising 10 minibatches. We fixed the number of hidden units to 50, and the non-linear activation function of the hidden units to ReLU. Three hyperparameters were cross-validated using grid-search: the number of components on each one-dimensional conditional (only applicable to the RNADE-MoG and RNADE-MoL models) was chosen from $\{2, 5, 10, 20\}$, the weight-decay (used only to regularize the input to hidden weights) from $\{2.0, 1.0, 0.1, 0.01, 0.001, 0\}$, and the learning rate from $\{0.1, 0.05, 0.025, 0.0125\}$. Learning rates were decreased linearly to reach 0 after the last epoch.

The results are shown in Table 4.6. RNADE with mixture of Gaussian conditionals was among the statistically significant group of best models on all data sets. As shown in Figure 4.8, RNADE-SAS and RNADE-MoG models are able to capture hard thresholds and heteroscedasticity.

Natural image patches

We also measured the ability of RNADE to model small patches of natural images. Following the work of Zoran and Weiss [2011], we use 8-by-8-pixel patches

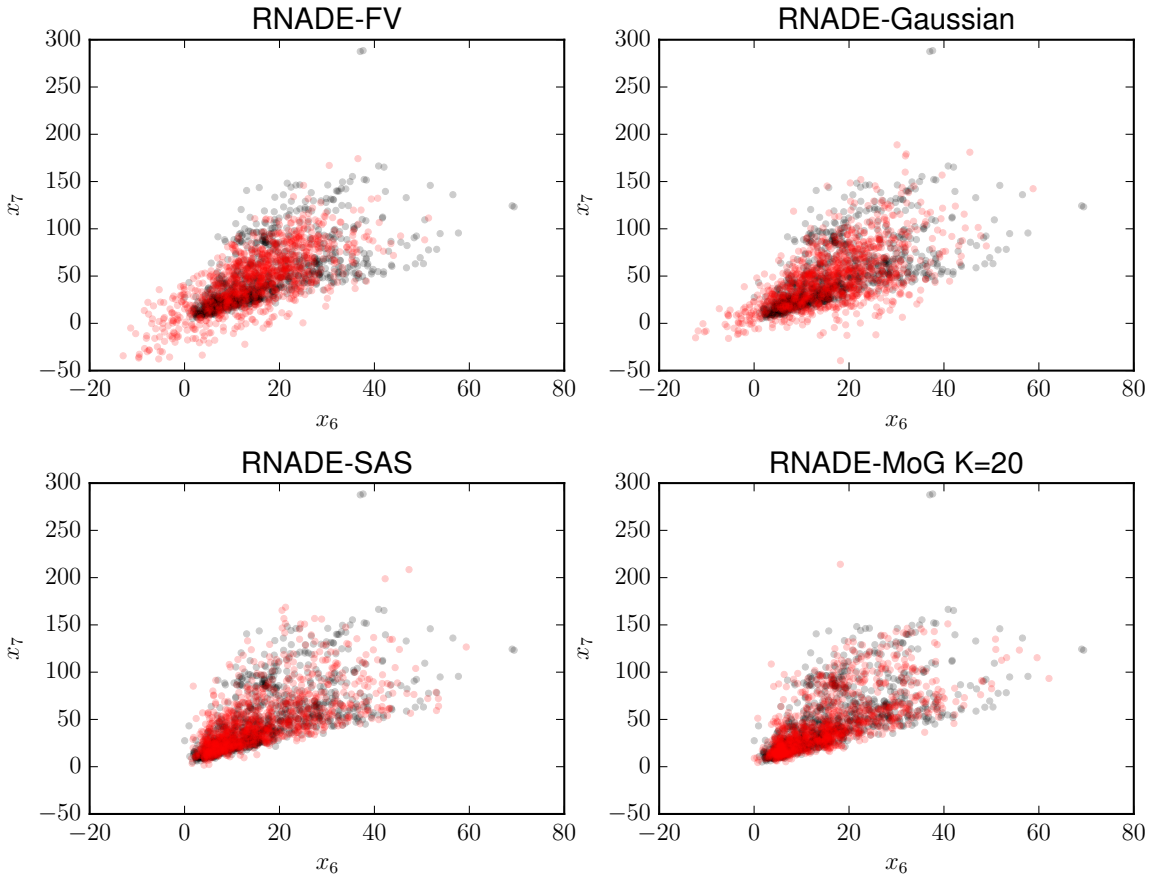


Figure 4.8: Scatter plot of dimensions x_7 vs x_6 of the *red wine* data set. A thousand data points from the data set are shown in black in all subfigures. As can be observed, this conditional distribution $p(x_7 \mid x_6)$ is heteroscedastic, skewed and has hard thresholds. In red, a thousand samples from four RNADE models with different one-dimensional conditional forms are shown. **Top-left**: In red, one thousand samples from a RNADE-FV model. **Top-right**: In red, one thousand samples from a RNADE-Gaussian model. **Bottom-left**: In red, one thousand samples from a RNADE-SAS (sinh-arcsinh distribution) model. **Bottom-right**: In red, one thousand samples from a RNADE-MoG model with 20 components per one-dimensional conditional. The RNADE-SAS and RNADE-MoG models successfully capture all the characteristics of the data.

Model	Red wine	White wine	Parkinsons
Gaussian	−13.18	−13.20	−10.85
MFA	−10.19	−10.73	−1.99
RNADE-FV	−12.29	−12.50	−8.87
RNADE-Gaussian	−11.99	−12.20	−3.47
RNADE-SAS	−9.86	−11.22	−3.07
RNADE-MoG	**−9.36**	**−10.23**	**−0.90**
RNADE-MoL	**−9.46**	−10.38	−2.63

Table 4.6: Average test set log-likelihoods per data point for seven models on three UCI data sets. Performances not in bold can be shown to be significantly worse than at least one of the results in bold as per a paired t-test on the ten mean-likelihoods (obtained from each data fold), with significance level 0.05.

of monochrome natural images, obtained from the BSDS300 data set [Martin et al., 2001; Figure 4.9 gives examples].

Pixels in this data set can take a finite number of brightness values ranging from 0 to 255. We added uniformly distributed noise between 0 and 1 to the brightness of each pixel. We then divided by 256, making the pixels take continuous values in the range $[0, 1]$. Adding noise prevents deceivingly high-likelihood solutions that assign narrow high-density spikes around some of the possible discrete values.

We subtracted the mean pixel value from each patch. Effectively reducing the dimensionality of the data. Therefore we discarded the 64th (bottom-right) pixel, which would be perfectly predictable and models could fit arbitrarily high densities to it. All of the results in this section were obtained by fitting the pixels in a raster-scan order.

Experimental details follow. We trained our models by using patches randomly drawn from 180 images in the training subset of BSDS300. We used the remaining 20 images in the training subset as validation data. We used 1000 random patches from the validation subset to early-stop training of RNADE. We measured the performance of each model by their log-likelihood on one million patches drawn randomly from the test subset of 100 images not present in the training data. Given the larger scale of this data set, hyperparameters of the RNADE and MoG models were chosen manually

using the performance of preliminary runs on the validation data, rather than by grid search.

All RNADE models reported use ReLU activations for the hidden units. The RNADE models were trained by stochastic gradient descent, using 25 data points per minibatch, for a total of 1,000 epochs, each comprising 1,000 minibatches. The learning rate was initialized to 0.001, and linearly decreased to reach 0 after the last epoch. Gradient momentum with factor 0.9 was used, but initiated after the first epoch. A weight decay rate of 0.001 was applied to the input-to-hidden weight matrix only. We found that multiplying the gradient of the mean output parameters by the standard deviation improves results of the models with mixture outputs [5]. RNADE training was early stopped but didn't show signs of overfitting. Even larger models might perform better.

The MoG models were trained using 1,000 iterations of minibatch EM. At each iteration 20,000 randomly sampled data points were used in an EM update. A step was taken from the previous parameters' value towards the parameters resulting from the M-step: $\boldsymbol{\theta_t} = (1-\eta)\boldsymbol{\theta_{t-1}} + \eta\boldsymbol{\theta_{EM}}$. The step size, η, was scheduled to start at 0.1 and linearly decreased to reach 0 after the last update. The training of the MoG was early-stopped and also showed no signs of overfitting.

The results are shown in Table 4.7. We report the average log-likelihood of each model for a million image patches from the test set. The ranking of RNADE models is maintained when ordered by validation likelihood: the model with best test-likelihood would have been chosen using crossvalidation across all the RNADE models shown in the table. We also compared RNADE with a MoG trained by Zoran and Weiss (downloaded from Daniel Zoran's website) from which we removed the 64th row and column of each covariance matrix. There are two differences in the set-up of our experiments and those of Zoran and Weiss. First, we learned the means of the MoG components, while Zoran and Weiss [2011] fixed them to zero. Second, we held-out 20 images from the training set to do early-stopping and hyperparameter optimisation, while they used the 200 images for training.

5. Empirically, we found this to work better than regular gradients and also better than multiplying by the variances, which would provide a step with the right units.

Model	Test log-likelihood
MoG $K=200$ [Zoran and Weiss, 2012] [a]	152.8
MoG $K=100$	144.7
MoG $K=200$	150.4
MoG $K=300$	150.4
RNADE-FV $h=512$	100.3
RNADE-Gaussian $h=512$	143.9
RNADE-Laplace $h=512$	145.9
RNADE-SAS [b] $h=512$	148.5
RNADE-MoG $K=2$ $h=512$	149.5
RNADE-MoG $K=2$ $h=1024$	150.3
RNADE-MoG $K=5$ $h=512$	152.4
RNADE-MoG $K=5$ $h=1024$	152.7
RNADE-MoG $K=10$ $h=512$	153.5
RNADE-MoG $K=10$ $h=1024$	**153.7**
RNADE-MoL $K=2$ $h=512$	149.3
RNADE-MoL $K=2$ $h=1024$	150.1
RNADE-MoL $K=5$ $h=512$	151.5
RNADE-MoL $K=5$ $h=1024$	151.4
RNADE-MoL $K=10$ $h=512$	152.3
RNADE-MoL $K=10$ $h=1024$	152.5

Table 4.7: Average per-example log-likelihood of several mixture of Gaussian and RNADE models on 8×8 pixel patches of natural images. These results are reported in nats and were calculated using one million patches. Standard errors due to the finite test sample size are lower than 0.1 nats in every case. h indicates the number of hidden units in the RNADE models, and K the number of one-dimensional components for each conditional in RNADE or the number of full-covariance components for MoG.

a. This model was trained using the full 200 images in the BSDS training data set, the rest of the models were trained using 180, reserving 20 for hyperparameter crossvalidation and early-stopping.

b. Training an RNADE with sinh-arcsinh conditionals required the use of a starting learning rate 20 times smaller to avoid divergence during training. For this reason, this model was trained for 2000 epochs.

The RNADE-FV model with fixed conditional variances obtained very low statistical performance. Adding an output parameter per dimension to have variable standard deviations made our models competitive with MoG with 100 full-covariance components. However, in order to obtain results superior to the mixture of Gaussians model trained by Zoran and Weiss, we had to use richer conditional distributions: one-dimensional mixtures of Gaussians (RNADE-MoG). On average, the best RNADE model obtained 3.3 nats per patch higher log-density than a MoG fitted with the same training data.

In Figure 4.9, we show one hundred examples from the test set, one hundred examples from Zoran and Weiss' mixture of Gaussians, and a hundred samples from our best RNADE-MoG model. Similar patterns can be observed in the three cases: uniform patches, edges, and locally smooth noisy patches.

Speech acoustics

We also measured the ability of RNADE to model small patches of speech spectrograms, extracted from the TIMIT data set [Garofolo et al., 1993]. The patches contained 11 frames of 20 filter-banks plus energy; totalling 231 dimensions per data point. A good generative model of speech acoustics could be used, for example, in denoising, or speech detection tasks.

We fitted the models using the standard TIMIT training subset, which includes recordings from 605 speakers of American English. We compare RNADE with a mixture of Gaussians by measuring their log-likelihood on the complete TIMIT core-test data set: a held-out set of 25 speakers.

The RNADE models have 512 hidden units, ReLU activations, and a mixture of 20 one-dimensional Gaussian components per output. Given the large scale of this data set, hyperparameter choices were again made manually using validation data. The same training procedures for RNADE and mixture of Gaussians were used as for natural image patches.

The RNADE models were trained by stochastic gradient descent, with 25 data points per minibatch, for a total of 200 epochs, each comprising 1,000 minibatches. The

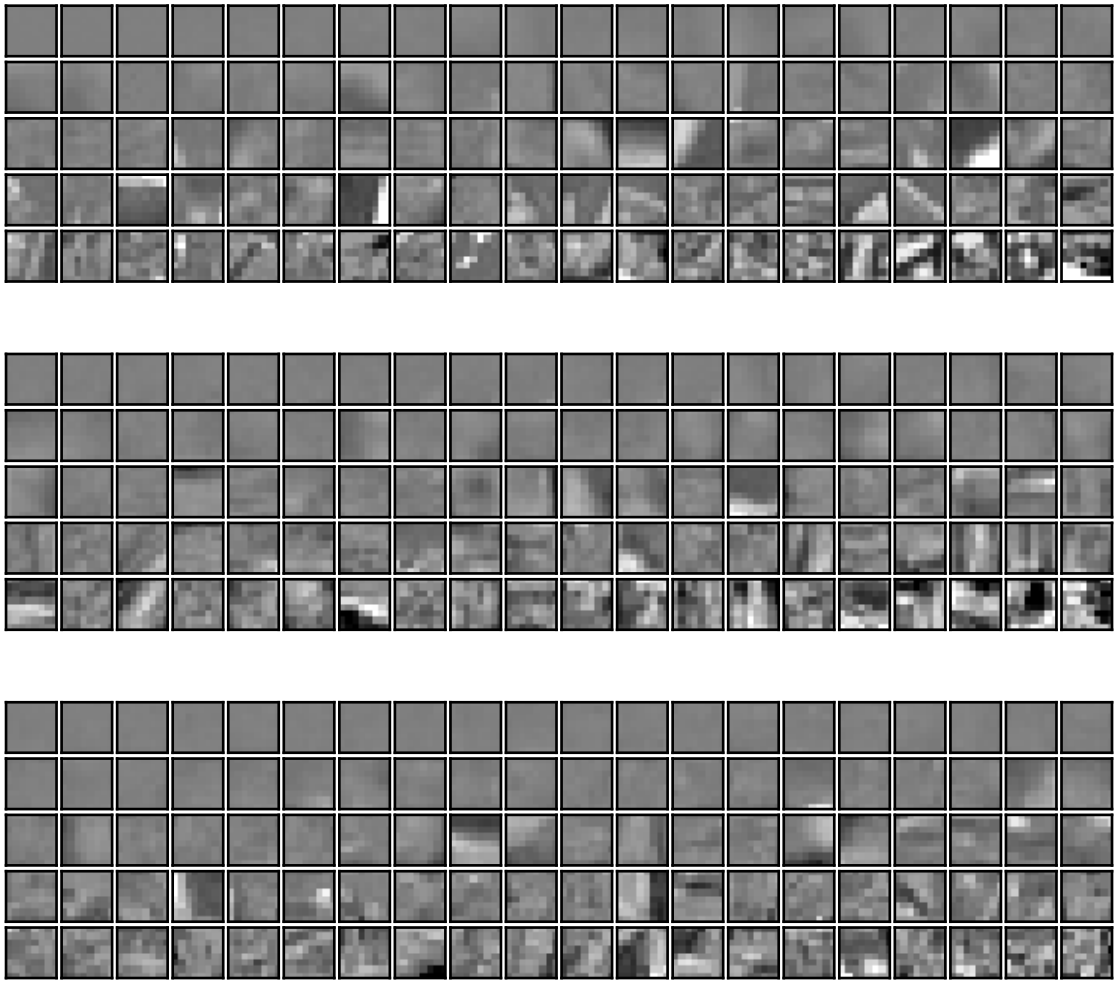


Figure 4.9: **Top:** 100 8×8 patches from the BSDS test set. **Center:** 100 samples from a mixture of Gaussians with 200 full-covariance components. **Bottom:** 100 samples from an RNADE with 1024 hidden units and 10 Gaussian components per conditional. All data and samples were drawn randomly and sorted by their density under the RNADE.

learning rate was initialized to 0.001 and linearly decreased to reach 0 after the last epoch. Gradient momentum with momentum factor 0.9 was used, but initiated after the first epoch. A weight decay rate of 0.001 was applied to the input-to-hidden weight matrix only. Again, we found that multiplying the gradient of the mean output parameters by the standard deviation improved results. RNADE training was early stopped but didn't show signs of overfitting.

As for the MoG model, it was trained exactly as in Section 4.7.3.

The results are shown in Table 4.8. The best RNADE (which would have been selected based on validation results) has 15 nats higher likelihood per test example than the best mixture of Gaussians. Examples from the test set, and samples from the MoG and RNADE-MoG models are shown in Figure 4.10. In contrast with the log-likelihood measure, there are no marked differences between the samples from each model. Both sets of samples look like blurred spectrograms, but RNADE seems to capture sharper formant structures (peaks of energy at the lower frequency bands characteristic of vowel sounds).

4.8 Conclusion

We've described the Neural Autoregressive Distribution Estimator, a tractable, flexible and competitive alternative to directed and undirected graphical models for unsupervised distribution estimation.

Since the publication of the first formulation of NADE [Larochelle and Murray, 2011], it has been extended to many more settings, other than those described in this paper. Larochelle and Lauly [2012]; Zheng et al. [2015b] adapted NADE for topic modeling of documents and images, while Boulanger-Lewandowski et al. [2012] used NADE for modeling music sequential data. Theis and Bethge [2015] and van den Oord et al. [2016b] proposed different NADE models for images than the one we presented, applied to natural images and based on convolutional and LSTM hidden units. Zheng et al. [2015a] used a NADE model to integrate an attention mechanism into an image classifier. Bornschein and Bengio [2014] showed that NADE could serve as a powerful

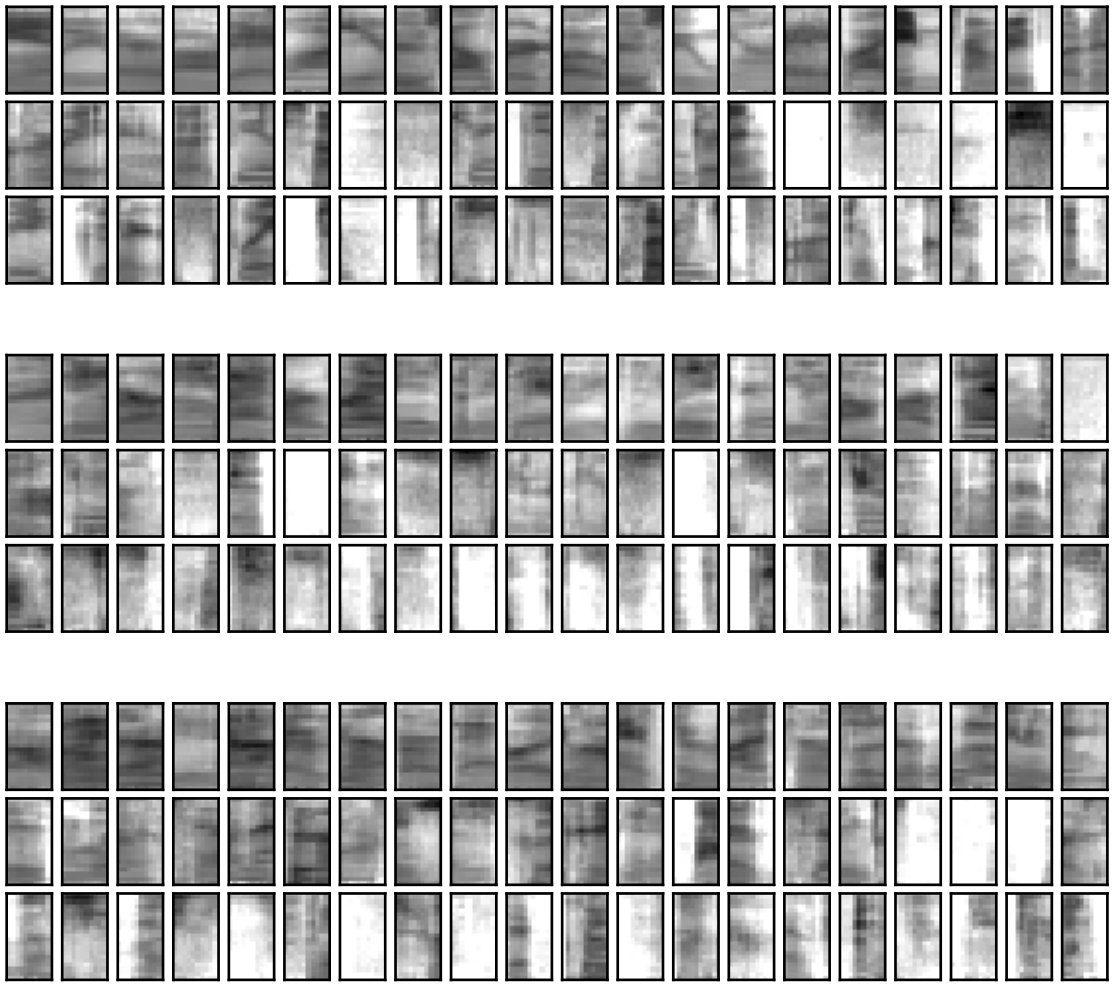


Figure 4.10: **Top:** 60 data points from the TIMIT core-test set. **Center:** 60 samples from a MoG model with 200 components. **Bottom:** 60 samples from an RNADE with 10 Gaussian output components per dimension. For each data point displayed, time is shown on the horizontal axis, the bottom row displays the energy feature, while the others display the Mel filter bank features (in ascending frequency order from the bottom). All data and samples were drawn randomly and sorted by density under the RNADE model.

Model	Test LogL
MoG $N=50$	110.4
MoG $N=100$	112.0
MoG $N=200$	112.5
MoG $N=300$	112.5
RNADE-Gaussian	110.6
RNADE-Laplace	108.6
RNADE-SAS	119.2
RNADE-MoG $K=2$	121.1
RNADE-MoG $K=5$	124.3
RNADE-MoG $K=10$	**127.8**
RNADE-MoL $K=2$	116.3
RNADE-MoL $K=5$	120.5
RNADE-MoL $K=10$	123.3

Table 4.8: Log-likelihood of several MoG and RNADE models on the core-test set of TIMIT measured in nats. Standard errors due to the finite test sample size are lower than 0.4 nats in every case. RNADE obtained a higher (better) log-likelihood.

prior over the latent state of directed graphical model. These are just a few examples of many possible ways one can leverage the flexibility and effectiveness of NADE models.

Chapitre 5

Apprentissage automatique appliqué à la tractograhie

“ I am my connectome.

Sebatian Seung, 2010 ”

Résumé

Dans cet article, nous démontrons que les techniques d'apprentissage profond peuvent être utilisées afin de *tractographier* les fibres du cerveau tout en évitant d'introduire des biais provenant d'experts. Pour ce faire, nous avons entrainé un réseau de neurones récurrent afin de générer des fibres directement à partir de données provenant de l'imagerie par résonance magnétique de diffusion (IRMd). Nous avons étudié le comportement du modèle sur un ensemble de données synthétiques de la matière blanche. Ces données ont été choisies pour leur réalisme et parce que l'on connait la vérité terrain. Nous démontrons que le modèle offre des performance compétitives par rapport aux techniques usuelles, même en utilisant des données différentes de celles dont on s'est servi lors de la phase d'entrainement. En fait, les résultats obtenus lors de nos expériences suggèrent que l'exploitation du passé d'une fibre durant la phase de tractographie permet de mieux prédire la prochaine direction à suivre. Étonnament, cette méthode basée sur l'apprentissage aurait terminé parmi les 15 meilleures lors de l'*ISMRM 2015 Tractography Challenge*.

Commentaires

L'article sera soumis à la conférence *Medical Image Computing & Computer Assisted Intervention* (MICCAI) en février 2017. Mes contributions pour cet article sont les suivantes:

- Développer le système permettant d'entrainer un réseau de neurones récurrent (plus particulièrement un *Gated Recurrent Unit*) à l'aide de la librairie Theano [The Theano Development Team et al., 2016]. L'implémentation comprend également le script permettant de reconstruire les fibres directement à partir d'image de diffusion d'un cerveau et d'un modèle déjà entrainé. Le code source pour reproduire les expériences sera disponible publiquement lorsque l'article sera accepté.
- Mettre en place les outils et les scripts nécessaires au traitement de données de diffusion et la manipulation des reconstructions de fibres. Entre autres, j'ai implémenté, en Theano, l'interpolation trilinéaire pour les données de diffusion. J'ai également écrit un script d'assurance qualité pour s'assurer que toutes les informations fournies au modèle se retrouvent dans le même espace.
- Évaluer la performance du modèle proposé à l'aide de l'erreur quadratique moyenne 1.4.1 et du Tractometer Côté et al. [2013] sur des données synthétiques Maier-Hein et al. [2016] utilisées lors du *ISMRM 2015 Tractography Challenge*. Il s'agit d'une tâche de régression (section 1.1.2). L'apprentissage du modèle s'est fait de façon supervisée (section 1.2.1).
- Rédiger l'article conjointement avec Philippe Poulin, Maxime Descoteaux et Hugo Larochelle.
- Créer les images illustrant l'entrainement du modèle et le processus de tractographie.

Learn to track: Deep learning for tractography

Marc-Alexandre Côté
Université de Sherbrooke
marc-alexandre.cote@usherbrooke.ca

Philippe Poulin
Université de Sherbrooke
philippe.poulin2@usherbrooke.ca

Jasmeen Sidhu
Université de Sherbrooke

Jean-Christophe Houde
Université de Sherbrooke

Eleftherios Garyfallidis
University of Indiana

Hugo Larochelle
Google Brain

Maxime Descoteaux
Université de Sherbrooke

Keywords: tractography, deep learning, neuroimaging, RNN, GRU, neural network

Abstract

We show that deep learning techniques can be applied successfully to fiber tractography. Specifically, we used a recurrent neural network to learn the generation process of streamlines directly from the diffusion-weighted imaging (DWI) data. Furthermore, we empirically study the behaviour of the proposed model on a realistic white matter phantom with known ground truth. We show that its performances are competitive to that of commonly used techniques, even when the model is used on DWI data unseen at training time. In fact, results of our experiments suggest that exploiting the past information of a streamline during tracking helps predict the following direction. Surprisingly, this learning-based method would have finished in the top 15 of the ISMRM 2015 Tractography Challenge.

5.1 Introduction

Tractography is currently at the heart of human brain connectomics studies [Van Essen et al., 2012]. However, recent biases and limitations of existing tractography pipelines have been highlighted [Côté et al., 2013], such as the reconstruction of

many non-existent connections (false positive streamlines), poor spatial extent of existing connections and the difficulty of injecting anatomical priors outside manual dissection.

Currently, tracking algorithms depend on local modeling assumptions on the nature of the underlying diffusion MRI signal. In 2015, Neher et al. [2015] proposed a machine learning (ML) approach to fiber tractography based on a random-forest classifier. They succesfully demonstrated how a purely data-driven approach can be used to reconstruct streamlines from the raw diffusion signal, thus avoiding bias introduced by experts. Their method is innovative, works well on 2D synthetic data and shows promising qualitative results on *in vivo* data. However, it has yet to be shown how well machine learning approaches can perform quantitatively on more realistic data and how well they can generalize to unseen data. In this paper, our main contribution is the first deep learning model for this problem, based on a recurrent neural network that is able to exploit the sequential nature of streamlines. We quantitatively evaluate its performances on the realistic phantom of the ISMRM 2015 Tractography Challenge and show it can also work on unseen data.

5.2 Method

The goal is to train a model to predict tracking directions to follow, from a diffusion dataset and sequences of spatial coordinates. In the context of tractography, such a model can be used in an iterative process for streamline creation. We chose to focus on deep learning models because of their well-known ability to discover and extract meaningful structures directly from raw data [LeCun et al., 2015].

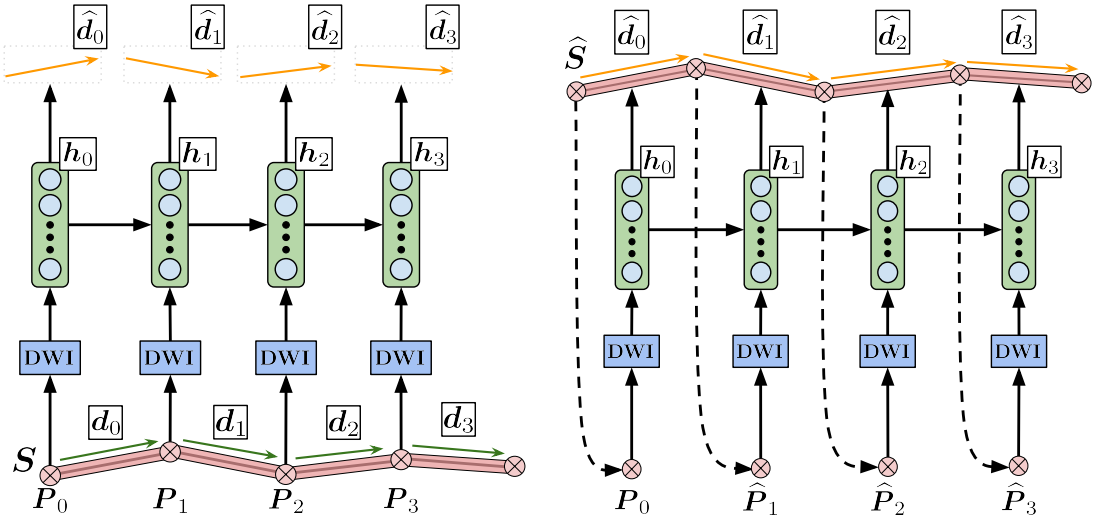
Deep Learning: *Artificial Neural Networks (ANNs)* are models now widely known for their ability to learn features while making minimal assumptions about the data. They dramatically improved the state-of-the-art in several domains [LeCun et al., 2015]. An ANN consists of multiple processing layers, composed of units, that learn representations of the data at multiple levels of abstraction. Both its depth (nb. of layers) and width (nb. units per layer) affect its expressive power. *Recurrent Neural*

Network (RNN) is an extension of the ANN to support inputs structured as sequences. The general idea behind the RNN is to model an internal state that is updated with each new observation in the input sequence and can be used to make predictions. Through its updatable internal state, the model can "remember" relevant features about the past. In this paper, we used a Gated Recurrent Unit (GRU) [Cho et al., 2014] type of RNN, with gradient norm clipping of 1.

Training, in a supervised context, requires examples of inputs and their associated target outputs, that the model should reproduce. A model is iteratively presented batches of examples, and the mean prediction error (the difference between the model's predictions and the targets) is "propagated" back into the model to adjust the parameters and minimize the model's errors. Over time, the model learns to extract the important features in order to produce good predictions. This optimization scheme is known as stochastic gradient descent, but other optimizers have improved upon it, such as Adam [Kingma and Ba, 2015], which we used.

Our model: The proposed model is derived from the GRU RNN. As in [Neher et al., 2015], to be independent of the gradient scheme, the raw diffusion signal is first resampled to have D gradient encodings evenly distributed on the sphere (we used $D = 100$). We also normalized each diffusion-weighted images by the b=0 image.

The model takes as input a sequence $\boldsymbol{S}$ of M equally-spaced spatial coordinates $P_i = (x_i, y_i, z_i)$, i.e. a streamline. The diffusion signal is then evaluated at each of these points, using trilinear interpolation. This results in a sequence of M vectors with D dimensions representing the diffusion information along the streamline. As shown in Figure 5.1, for each point P_i in the streamline, its associated diffusion information $\mathbf{DWI}(P_i)$ is used to update the current internal state $\boldsymbol{h}_i$ of the model. From there, at each step along the streamline, the model makes a prediction of the direction to follow $\widehat{\boldsymbol{d}}_i$ which is normalized. Note that the model *only* observes the diffusion information at each streamlines coordinate. Thus, it is unaware of any spatial position and has to understand the diffusion process in order to predict a direction.



(a) Training (b) Tracking

Figure 5.1: Architecture of the proposed model. (a) Given a streamline $\boldsymbol{S}$, diffusion information is evaluated at each point P_i using trilinear interpolation ($\mathbf{DWI}(P_i)$). The resulting vector is provided to the RNN to predict a direction $\widehat{\boldsymbol{d}}_i$ (orange), which is compared against its associated target direction $\boldsymbol{v}_{di}$ (green). (b) Given a starting point P_0, a new streamline $\widehat{\boldsymbol{S}}$ is generated by iteratively predicting a new direction $\widehat{\boldsymbol{d}}_i$, and feeding the estimated new position $\widehat{P}_{i+1}$ back to the model. Note how the predicted direction $\widehat{\boldsymbol{d}}_i$ gets influenced by prior information along the streamlines through $\boldsymbol{h}_{j<i}$.

Training is done by minimizing the mean squared error (MSE) between the model prediction $\widehat{\boldsymbol{d}}$ and the target $\boldsymbol{v}_d$ (i.e. the next normalized segment of the streamline) for all streamlines available for training. The more natural solution would be to minimize the mean angular error (MAE), but working with an angular error is not simple in practice. Since the direction vectors are normalized and thus are inscribed within a unit circle, it is easily seen that minimizing the MSE is equivalent to minimizing the MAE.

Tractography is performed by using a fully trained model. Streamlines generation follows an iterative and deterministic process as illustrated in Figure 5.1(b). A seed point $P_0 = (x_0, y_0, z_0)$ is first chosen. Then, a new streamline is created with the initial seed $\widehat{\boldsymbol{S}} = \{P_0\}$. Next, the model is given the DWI data at the last streamline

coordinate P_i to obtain a predicted direction $\widehat{\boldsymbol{d}}_i$. The next points $P_{i+1} = P_i + \alpha\widehat{\boldsymbol{d}}_i$ are then computed, where α is a chosen step size. Finally, generation of a streamline is stopped when a desired criterion is met. The whole process is repeated as many time as required to produce a full tractogram.

5.3 Related Work

The work of Neher et al. [2015] hypothesizes that tractography can be improved by considering features of the local neighborhood. More precisely, their model makes predictions based on a voting mechanism, using local direction proposals from multiple sample positions in the vicinity of the current location. Each direction proposal is obtained by a classification over 100 possible directions, along with a streamline termination probability. In our current approach, the problem is framed as a regression task over normalized directions instead of a vote over discretized directions. This means that to produce a prediction, fewer computations are needed at the output, compared to computing and voting over many proposals. In addition, regression allows the model to output more precise directions and thus be more suitable at exploiting smaller variations in direction.

While Neher et al. [2015] consider the neighborhood of the current position, they do not consider the full evolution of the streamline up to each point. Our hypothesis is that there are high-order dependencies between the next direction in the streamline and all previous directions. Consequently, our RNN approach has a natural mechanism for integrating past information along the streamline to predict a next direction. Note that these two approaches are not exclusive.

5.4 Experiments

We developed three experiments to measure the performance of our model to 1) learn and perform tractography, 2) generalize to unseen bundles and 3) generalize to different diffusion MRI data with complex noise and artefacts.

Realistic White Matter Software Phantom (Exp. 1): We aim to demonstrate that the proposed model successfully learns to perform tractography and is competitive with four commonly used tractography pipelines.

Materials: We use a realistic software phantom that was generated with Fiberfox [Neher et al., 2014] for the ISMRM 2015 Tractography Challenge [1] [Maier-Hein et al., 2016]. The brain and anatomy of Human Connectome Project [Van Essen et al., 2012] subject were used to manually extract 25 major long-range white matter (WM) bundles from a full brain global tractography using Gibbs MITK tracking, as shown in first row of Fig. 5.2. Extracted streamlines were provided to Fiberfox to produce the ground truth DWI data, consisting of a 2mm isotropic acquisition, with 32 gradient encodings, b-value=1000 s/mm^2 and a b=0 image.

Following a standard ML procedure, we divided the ground truth streamlines in three sets used for training (160,348 streamlines), validation (20,042 streamlines) and testing (20,042 streamlines), and made sure that proportions of streamlines coming from each bundle were maintained.

Training: We used Adam with batches of 64 examples randomly sampled from the training set. To avoid over-representation of larger bundles, we made sure a similar number of streamlines were picked from each bundle during sampling. To determine when to stop training, we monitored the MSE of the model on the validation set and stopped when there were no improvements in the last 10 epochs. We also explored different hyper-parameters values for our model. We tried two Adam learning rates (10^{-3} and 10^{-4}) and two hidden layer sizes (500 and 1000 units).

Tractography: We tested two seeding strategies: 1) seeding from the whole white matter mask (RNN_WM) and 2) seeding only from regions of interest (ROIs) composed of the endpoints of the ground-truth bundles (RNN_ENDS). For the latter strategy, we chose one or two seeds per voxel and varied the step size (0.2mm or 1mm).

1. Data is publicly available at `http://tractometer.org/ismrm_2015_challenge`

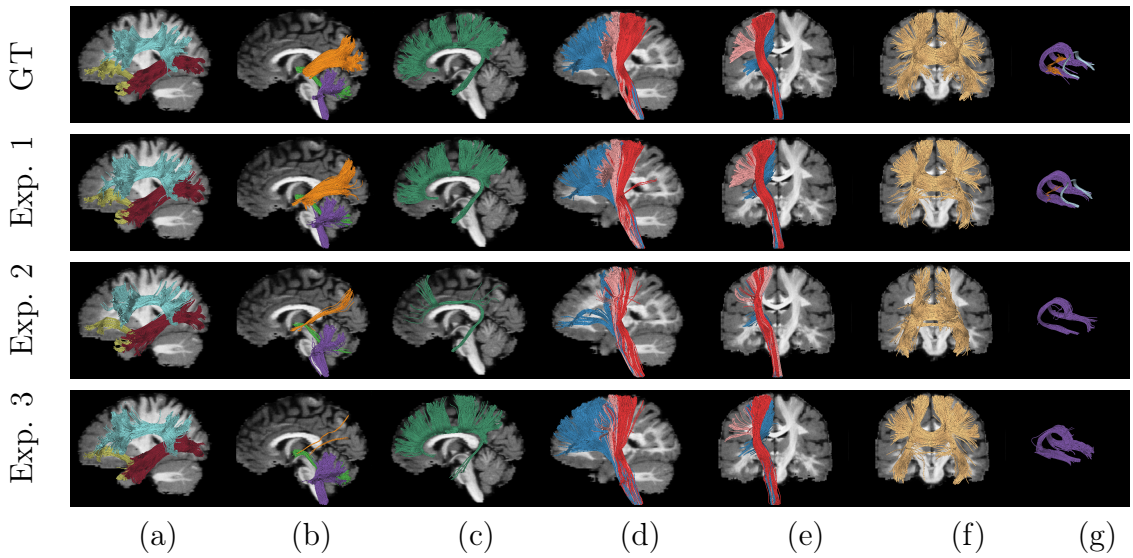


Figure 5.2: Ground truth (GT) valid bundles and those obtained for each experiment. Colored bundles (in order) : [UF, ILF, SLF], [OR, SCP, ICP], [Cingulum], [FPT, CST, POPT], [FPT, CST, POPT], [CC], [Fornix, CA, CP].

We compare our approach with deterministic and probabilistic tractography algorithms that are based on the ground-truth orientations in each voxel [Garyfallidis et al., 2014]. These orientations were found using the ground-truth bundles, by extracting the main directions of bundles going through each voxel.

Additionally, the same seeding strategies were tested with each version of the algorithm leading to one of four tracking results: DET_WM, DET_ENDS, PROB_WM, PROB_ENDS.

In all cases, the WM mask was the tracking domain, a maximal angle of 45 degrees was used and streamlines shorter than 10mm or longer than 300mm were rejected.

Leave-one-out (Exp. 2): The purpose of this experiment is to measure whether the model can generalize to fully unseen bundle. To accomplish this, the models in this experiment were always trained on a training set from which one of the bundles was completely removed. All trained models had 500 hidden units and were trained using a Adam learning rate of 10^{-3}. Then, each of them was used to generate

streamlines, which were ultimately compared with those coming from a model that has seen all bundles during training. Tracking was always done from the endpoints ROIs using a step size of 1mm and two seeds per voxel. Overall, 15 different models were trained.

Noisy Phantom Data (Exp. 3): The goal of this experiment is to show that the proposed approach can generalize to a different set of DWI data. To do this, we took our best model trained on the ground truth phantom data and used it, *without retraining*, to generate streamlines on the noisy and artefacted DWI data provided for the Tractography Challenge. The data includes realisitic distortion, motion, noise and spiking artifacts, as found in a clinical-like scans [2]. As a standard preprocessing step, we applied motion correction (with a nearest neighbor interpolation), correction of the b vectors using the corresponding affine matrix and ran NLSAM for denoising and spike correction on the raw DWI [St-Jean et al., 2016]. Then, tracking was done on the resulting DWI data using the same tracking parameters as in previous experiments.

Evaluation metrics: To assess the quality of our model, we report the Mean Angular Error (MAE) on the testing set. To compute this metric, we let the model predict the next direction at every point of every streamline and compare it with the true direction. Intuitively, this metric shows how well some given streamlines are represented by the model.

Generative capability of the proposed model is evaluated using the Tractometer connectivity metrics [Côté et al., 2013]. More precisely, we report the valid connection ratio and the associated number of valid bundles (true positives), the invalid connection ratio and the associated number of invalid bundles (false negatives), the average volumetric bundle overlap and overreach in percentages. Drawing conclusions from only one of these last two metrics can be misleading (e.g. a model can have both the best overlap and the worst overreach). These metrics are related to precision and

2. `http://tractometer.org/ismrm_2015_challenge/data`

Table 5.1: Quantitative results on the ISMRM 2015 Tractography Challenge data.

Model	Step (mm)	# Seeds	MAE	Connections (%)			# Bundles		Avg. Bundle (%)		
				Valid	*Invalid*	*No*	*Valid*	*Invalid*	*Overlap*	*Overreach*	F_1
DET_ROIS	0.2	-	-	81.3	12.5	6.1	24	90	71.8	20.0	75.7
DET_WM	0.2	-	-	78.4	14.8	6.8	24	82	73.3	20.2	76.4
PROB_ROIS	0.2	-	-	78.0	15.2	6.8	24	92	69.5	20.5	74.2
PROB_WM	0.2	-	-	75.0	17.9	7.1	24	84	70.0	20.7	74.4
RNN_ROIS	0.2	1	4°	63.2	20.2	16.6	24	99	33.6	5.5	49.5
RNN_ROIS	1	1	4°	87.1	7.5	5.3	**25**	**67**	64.2	9.5	75.1
RNN_ROIS	1	2	4°	**87.6**	**7.1**	**5.2**	**25**	77	72.6	13.8	**78.8**
RNN_WM	1	1	4°	79.0	11.7	9.4	**25**	108	81.4	30.3	75.1

recall measures, which can be combined into the F_1 score[3] that we also report.

5.5 Results and Discussion

Realistic White Matter Software Phantom (Exp. 1): During the training phase, the best model obtained an MSE of 0.07 on the testing set, which translates to a MAE of 4°. Since the model only observes diffusion coefficients in order to predict a direction, the model has clearly been able to grasp the concept of diffusion to achieve such a low MSE error.

Our best model used 500 hidden units and was trained using a learning rate of 10^{-3}. The best tracking results for this model were obtained using a step size of 1mm, 2 seeds per voxel and was tracking from the endpoints ROIs. The second row of Fig. 5.2 gives a visual appreciation for the quality of the reconstructed bundles with best parameters.

Quantitative results are shown in Table 5.1. The proposed model achieves the highest valid connection ratio of 87.6%, while maintaining the invalid connections as low as 7.1%. We can see that taking local decisions on the ground truth peaks is weaker than actually using the knowledge of the pathway defined by the actual streamline being tracked.

3. $F_1 = 2*(precision*recall)/(precision+recall)$ where $precision = 1-overreach$ and $recall = overlap$ (`https://en.wikipedia.org/wiki/F1_score`).

Table 5.2: Leave-one-out Exp 2. Connection ratios are reported for all bundles while other metrics are reported only for the missing bundle. 'Found' is the nb. of generated streamlines divided by the nb. of streamlines found in Exp. 1 by the best model.

Missing	Connections (%)			Streamlines		Missing Bundle (%)		
Bundle	*Valid*	*Invalid*	*No*	*Missing*	*Found(%)*	*Overlap*	*Overreach*	F_1
CA	85.9	10.5	3.6	431	0.0	-	-	-
CC	82.6	12.6	4.8	17,993	7.9	10.8	3.2	19.4
Cingulum	84.6	11.8	3.6	35,150	3.5	8.7	1.8	15.9
CP	86.7	9.8	3.5	365	0.0	-	-	-
CST	81.5	13.2	5.3	17,449	34.6	42.9	16.2	56.8
Fornix	82.5	11.7	5.8	3,831	11.8	39.3	14.2	53.9
FPT	78.8	15.2	6.0	6,179	7.6	17.1	5.6	28.9
ICP	79.5	14.1	6.4	7,453	14.7	33.9	13.6	48.7
IOFF	83.1	10.9	6.0	28,622	13.7	34.2	15.6	48.7
MCP	86.3	10.6	3.1	21,114	7.0	24.6	3.3	39.2
OR	72.2	17.2	10.6	18,388	4.6	7.1	3.1	13.2
POPT	84.7	10.9	4.4	2,997	11.1	20.9	5.8	34.2
SCP	86.6	9.4	4.0	3,355	1.8	17.7	4.4	29.8
SLF	84.7	11.2	4.1	24,419	16.3	28.7	8.9	43.7
UF	77.8	15.0	7.2	12,687	8.3	25.6	5.1	40.3

Interestingly, we observed that using a step size of 1mm allows to retrieve around 20% more valid connections than one of 0.2mm. This might be caused by the length of the segment of the ground truth streamlines also being 1mm. Moreover, we noticed the model is better at tracking from the extremities of the bundles than from everywhere in the WM. This is not surprising since it learned to model streamlines as whole sequences (from one extremity to the other). This seeding strategy is also more intuitive from an anatomical point of view [Smith et al., 2012].

One important point to also take in account here is like any method that does data fitting for model reconstruction, the main computational cost resides in its training. Once trained the model is fairly efficient and can generalized to unseen data (i.e. so training is done only once). In the experiments, our best model took 12h using a Nvidia K20 GPU to train and can generate 80k streamlines in 5 minutes.

Leave-one-out (Exp. 2): Table 5.2 measures the ability of the proposed method to generate streamlines that were not seen during training. The third row of Fig. 5.2

also shows valid bundles where each of them was generated by the model which had not seen it during training.

This experiment highlights the fact the proposed model does learn the structure of the bundles seen during training. If the models in the first experiment were good *only* because they had learned how diffusion works, removing a few streamlines during training wouldn't change much their tracking capabilities at test time. However, this doesn't seems to be the case. According to the results in Table 5.2, removing either CA or CP, which accounts for less than 0.01% training streamlines, incapacitates the model at generating streamlines for these bundles.

These results also highlight the fact that some bundles are harder to recover than others, especially in the case they represent complex white matter geometries such as wide fannings of the corona radiata (Fig.2d) and complex branching structures as in the cingulum (Fig.2c).

Other very hard bundles to recover are the anterior and posterior commissures, CA and CP, because of their small size and location in the brain. In fact, these bundles were the most missed according to the Tractography Challenge website [4] where about only 10% of the 96 submissions found at least one of the two. Association bundles such as the uncinate, superior longitudinal fasciulus (SLF) and inferior LF (ILF) (Fig.2a), as well as a large portion of the corpus callosum (Fig.2f) are more easily reconstructed. Note this is also the case with classical tracking algorithms.

Noisy Phantom Data (Exp. 3) Even though the model was trained on the ground truth and DWI data that was as clean as possible, we used to generate streamlines from a preprocessed version of the noisy and artefacted phantom data. It obtained an average bundle overlap of 43.5% and overreach of 19.8% meaning a F_1 of 56.4%. However, it managed to score 80.8% valid and 17.8% invalid connections, which is within the top 5 according to Table 5.1 in terms of connectivity. In the challenge, it would have finished within the top 15 with 30% more valid connections than the average submission and a similar ratio of invalid.

4. `http://tractometer.org/ismrm_2015_challenge/results`

5.6 Conclusion

We proposed a method based on recurrent neural networks that learns fiber tractography by exploiting its sequential nature. We showed it can generate meaningful streamlines while producing few invalid connections, even when used on a different diffusion signal. We quantitatively measured its performances on a realisic white matter phantom and compared it with classic tractography methods. In future work, we are interested in using the surroundings of the streamlines to guide their progression during tracking. Moreover, the flexibility of the model could allow us to easily combine multiple modalities as input and include meaningful anatomical priors.

Conclusion

La popularité sans cesse grandissante de l'apprentissage automatique s'explique en partie par l'accès à une abondance de données numériques. Il est maintenant plus aisé de construire des modèles à partir de ces données plutôt que de les concevoir à la main. Plus particulièrement, l'apprentissage profond est en train de façonner la reconnaissance vocale, le traitement d'images, le traitement de la langue, et en est à ses débuts en neuroimagerie. De nombreux domaines utilisent les réseaux de neurones et dépendent donc maintenant de leur avancée.

La majorité des percées de l'apprentissage automatique au cours des dernières années ont nécessité une quantité gigantesque de données étiquetées. Or, tel que mentionné à la section 1.2.1, acquérir et étiqueter ces données est souvent couteux en temps et argent. Ceci limite considérablement les avancées technologiques dont sont théoriquement capables les algorithmes d'apprentissage. L'alternative est de se tourner vers les données non étiquetées et développer de meilleurs modèles capables de les exploiter pleinement malgré leurs imperfections. La recherche théorique en apprentissage automatique reste essentielle et a d'ailleurs été abordée en partie dans cette thèse.

D'abord, un premier article expliquait comment améliorer la machine de Boltzmann restreinte pour obtenir un modèle à capacité variable. En effet, ce nouveau modèle est capable d'adapter sa structure en fonction des exemples vus lors de l'entrainement. Cette stratégie d'adaptation est essentielle si l'on désire s'attaquer aux problèmes de l'apprentissage continu (*continous learning ou lifelong learning*) où l'agent apprend tout au long de son « existence ».

De plus, les caractéristiques que l'iRBM apprend à extraire sont ordonnancées en fonction de leur importance. Ceci pourrait être d'une grande utilité dans un contexte de recherche d'information. En effet, cet ordonnancement pourrait accélérer la comparaison de deux éléments en comparant d'abord leurs caractéristiques extraites les plus importantes pour ensuite, au besoin, comparer celles moins importantes.

Présentement, la représentation apprise par une iRBM consiste en une séquence li-

néaire de caractéristiques puisque la couche cachée est un vecteur. Il serait intéressant d'explorer des structures alternatives pour la couche cachée permettant d'obtenir des représentations plus complexes. À titre d'exemple, avec une structure en arbre, le modèle aurait la possibilité d'activer des sous-arbres, c'est-à-dire un ensemble de caractéristiques partageant des similarités.

Un point faible de l'iRBM, telle que présentée dans cette thèse, est qu'elle ne fonctionne qu'avec des données binaires. Or, il serait avantageux de pouvoir traiter des données réelles, ce qui augmenterait grandement la portée du modèle. Comme mentionné à la section 2.4, la RBM a déjà été appliquée à des problèmes comportant des données réelles grâce à la *Gaussian RBM*. Une approche similaire pourrait être employée afin de généraliser l'iRBM à ce type de données.

Le second article présentait un modèle autorégressif ainsi que plusieurs de ses extensions, dont une qui est en mesure d'exploiter la structure 2D des images. Grâce à la flexibilité du processus d'apprentissage du modèle *Deep NADE*, il a été possible de combiner un réseau à convolution avec un réseau multicouche pour créer *Convolutional NADE*, tout en gardant la propriété autorégressive du modèle original. Les récents progrès obtenus par PixelRNN [van den Oord et al., 2016b] et PixelCNN [van den Oord et al., 2016a], tous deux des modèles autorégressifs appliqués aux images, ont renouvelé un intérêt envers ce type de modèle.

Un avantage certain de *Convolutional NADE* est son invariance à l'ordre des dimensions des données observées. Grâce à cela, il est envisageable de l'utiliser dans un contexte de restauration d'images détériorées : retrait du bruit, d'occlusions, etc. En effet, il suffit de considérer un ordonnancement dans lequel les pixels non obstrués (ou non altérés) servent de conditionnement afin de régénérer les pixels manquants et ainsi rétablir l'image.

Finalement, le troisième article montrait comment exploiter la structure séquentielle des fibres neuronales à l'aide d'un réseau de neurones récurrent. Il s'agit d'une preuve de concept permettant de voir si l'apprentissage profond pourrait être utilisé pour tractographier un cerveau à partir d'images de diffusion brutes. Même s'il reste beaucoup de recherche à faire, les résultats obtenus jusqu'à présent sont concluants et

semblent indiquer que cette approche est prometteuse. La flexibilité des réseaux de neurones pourrait permettre de fournir au modèle des informations supplémentaires pour mieux guider le processus de tractographie.

D'ailleurs, voici quelques avenues de recherche intéressantes pour ce projet. D'abord, on pourrait intégrer l'information provenant du voisinage d'une fibre à l'aide d'un réseau à convolution. Ceci permettrait au modèle de mieux situer, dans le cerveau, la fibre que l'on est en train de tracer et de décider quand arrêter le tracé, c'est-à-dire apprendre à reconnaitre les points terminaux des fibres. Une autre avenue potentielle est de fournir au modèle l'information provenant de diverses modalités (tomographie, électroencéphalographie, etc.) et le laisser trouver la façon optimale de les combiner pour mieux performer. Une dernière avenue à explorer est de concevoir un modèle qui prédit les paramètres d'une distribution sur la sphère plutôt qu'une seule direction à suivre. Ceci permettrait une approche probabiliste de la tractographie.

Pour conclure, l'apprentissage automatique est un domaine en pleine effervescence et les applications y sont déjà nombreuses. L'arrivée des assistants personnels intelligents (Siri de Apple, Cortana de Microsoft, Alexa d'Amazon et OK de Google) risque d'augmenter encore plus l'intérêt général envers cette technologie.

“ The future starts today, not tomorrow.

Pape Jean-Paul II ”

Annexe A

Contributions de l'auteur reliées à cette thèse

Articles de journaux

- **M.-A. Côté**, and H. Larochelle (2016). Infinite Restricted Boltzmann Machine. *Neural Computation*, 28 (7).
- B. Uria, **M.-A. Côté**, K. Kregor, I. Murray, and H. Larochelle (2016). Neural Autoregressive Distribution Estimation. *Journal of Machine Learning Research*, 17 (205).
- **M.-A. Côté**, G. Girard, A. Boré, E. Garyfallidis, J.-C. Houde, and M. Descoteaux (2013). Tractometer: towards validation of tractography pipelines. *Medical Image Analysis*, 17 (7).

Soumis

- E. Garyfallidis, **M.-A. Côté**, F. Rheault, J. Sidhu, J. Hau, L. Petit, S. Cunanne, and M. Descoteaux (Soumis). Recognition of anatomical bundles using local and global streamline-based registration and clustering. *Neuroimage*.
- A. Chekir, S. Hassas, M. Descoteaux, **M.-A. Côté**, E. Garyfallidis, and F. Oulebsir-Boumghar. 3D-SSF: A Bio-inspired Approach for Dynamic Multi-Subject Clustering of White Matter Tracts. *Computers in Biology and Medicine*.
- K. Maier-Hein, P. Neher, J.-C. Houde, **M.-A. Côté**, et *al.* (2016). Tractography-based connectomes are dominated by false-positive connections. bioRxiv `http://biorxiv.org/content/early/2016/11/07/084137`

Articles de conférences

- A. Chekir, M. Descoteaux, E. Garyfallidis, **Marc-Alexandre Côté**, and F. Oulebsir-Boumghar (2014). A Hybrid Approach for Optimal Automatic Segmentation of White Matter Tracts in HARDI. *Biomedical Engineering and Sciences (IECBES)*.

Résumés de conférences

- E. Garyfallidis, **M.-A. Côté**, F. Rheault, and M. Descoteaux (2016). QuickBundlesX: Sequential Clustering of Millions of Streamlines in Multiple Levels of Detail at Record Execution Time. *International Society of Magnetic Resonance in Medicine – ISMRM 2016*.
- M. Cousineau, E. Garyfallidis, **M.-A. Côté**, P.-M. Jodoin, and M. Descoteaux (2016). Tract-profiling and bundle statistics: a test-retest validation study. *International Society of Magnetic Resonance in Medicine – ISMRM 2016*.
- **M.-A. Côté**, E. Garyfallidis, H. Larochelle, and M. Descoteaux (2015). Cleaning up the mess: tractography outlier removal using hierarchical QuickBundles clustering. *International Society of Magnetic Resonance in Medicine – ISMRM 2015*.
- E. Garyfallidis, **M.-A. Côté**, J. Hau, G. Perchey, L. Petit, S. Cunanne, and M. Descoteaux (2015). Recognition of bundles in healthy and severely diseased brains. *International Society of Magnetic Resonance in Medicine – ISMRM 2015*.
- J.-C. Houde, **M.-A. Côté**, and M. Descoteaux (2015). How to avoid biased streamlines-based metrics for streamlines with variable step sizes. *International Society of Magnetic Resonance in Medicine – ISMRM 2015*.

Présentations

- **M.-A. Côté** (2016) Infinite Restricted Boltzmann Machine. *CIFAR - Deep Learning Summer School*, Montréal, Canada.
- **M.-A. Côté** (2015) Tract-Analysis - Computational tools in the space of streamlines. *Brainhack MTL 2015*, Montréal, Canada.
- **M.-A. Côté** (2013) Tractometer. *CNS - Journée scientifique*, Sherbrooke, Canada.

Bibliothèques de programmes

Nibabel 2.1

Nibabel est une bibliothèque Python permettant de lire et d'écrire les formats de fichiers standards en neuroimagerie. Cette bibliothèque est essentielle à tout chercheur en neuroimagerie travaillant en Python.

Ma principale contribution réside en l'intégration d'une interface permettant de manipuler, de lire et d'écrire des tractogrammes (fichiers contenant un ensemble de reconstructions de fibres de la matière blanche). Il s'agit de l'aboutissement de deux ans d'efforts au cours desquels j'ai élaboré et raffiné plusieurs prototypes pour l'interface. En ce moment, l'interface supporte la lecture et l'écriture que d'un format de fichier de fibres (soit les TRKs), mais l'ajout d'un second format (TCKs) est présentement en révision. Le support pour d'autres formats est également prévu en 2017.

Dipy 0.8–0.11

Dipy est une librarie Python conçue pour faire de la neuroanatomie computationnelle et se concentrant surtout sur l'analyse de l'imagerie par résonance magnétique de diffusion. Cette bibliothèque est essentielle à tout chercheur en neuroimagerie traitant des images de diffusion et travaillant en Python.

Ma première contribution est l'ajout d'une interface modulaire facilitant le *clustering* de séquences finies de points en trois dimensions. Plus précisément, cette plateforme est utilisée pour la segmentation automatique de *streamlines* (reconstructions de fibres de la matière blanche dans le cerveau). Par la même occasion, j'ai généralisé l'algorithme QuickBundles [Garyfallidis et al., 2012] afin qu'il fonctionne avec différentes mesures de distance. Cela permet à un utilisateur de facilement regrouper ses *streamlines* en fonction de leur longueur, orientation, position spatiale, forme, etc. Le coeur de la plateforme de *clustering* a été codé en Cython permettant de traiter des millions de *streamlines* en l'espace de quelques minutes.

Ma deuxième contribution consiste en l'amélioration et l'ajout d'outils pour la visualisation 3D des *streamlines*. À titre d'exemple, il arrive fréquemment en neuroimagerie d'avoir à comparer les *streamlines* provenant de plusieurs sujets ou de différents faisceaux de fibres. À l'aide de mes outils, les chercheurs peuvent facilement afficher leurs données sous forme de mosaïque.

Pour moi, l'aspect interactif de ces outils est très important et beaucoup d'efforts ont été mis dans le développement d'un système de gestion d'évènements pour la manipulation de *streamlines*. D'ailleurs, dans le cadre du Google Summer of Code 2016, j'ai eu l'opportunité de superviser un étudiant qui a travaillé sur le projet de visualisation dans Dipy.

Ma troisième contribution est l'ajout d'outils permettant la manipulation efficace de *streamlines*. Notamment, j'ai intégré à Dipy l'algorithme de compression de *streamlines* tel que proposé par [Presseau et al., 2015].

Theano

Theano est une bibliothèque Python permettant de définir, d'optimiser et d'évaluer des expressions mathématiques représentées sous forme de graphe symbolique. Les noeuds du graphe, appelé *Ops*, représentent les opérations à appliquer aux données qui traversent le graphe. Theano est conçu pour gérer efficacement les tableaux multidimensionnels autant sur CPU que sur GPU. Elle est l'une des trois bibliothèques les plus utilisées en apprentissage automatique.

Ma contribution à Theano consiste en l'ajout de trois *Ops* : CumsumOp, CumprodOp et SearchSortedOp. Pour chaque *Op*, j'ai eu à fournir l'implémentation CPU et GPU permettant d'obtenir le résultat attendu, ainsi que la partie du graphe responsable de la propagation du gradient à travers cet *Op*.

Parmi les bogues que j'ai rapportés, j'ai découvert que la bibliothèque cuDNN v4 de NVIDIA produisait des résultats incohérents pour certains types de convolutions sur certaines cartes graphiques, notamment sur la K40. J'ai également réglé quelques bogues comme celui du goulot d'étranglement qui survenait lors de l'initialisation des flux aléatoires.

TractConverter

Le TractConverter est un outil et une bibliothèque permettant de convertir les *streamlines* provenant d'un certain format de fichier à un autre. Je suis l'unique développeur de cette bibliothèque utilisée dans quelques labos à travers le monde : Sherbrooke Connectivity Imaging Lab, Centre de recherche de l'Institut universitaire de gériatrie de Montréal (CRIUGM), Intelligent Systems Engineering de l'Université d'Indiana, École polytechnique fédérale de Lausanne, Institut des Maladies Neurodégénératives de l'Unversité de Bordeaux, etc. Ceci dit, je travaille activement à l'intégrer à Nibabel afin de rejoindre une plus grande communauté.

SmartDispatch

SmartDispatch est un outil Python permettant de lancer facilement des calculs sur une grappe de calcul de haute performance. Je suis l'un des développeurs responsables du bon fonctionnement et de l'évolution de l'outil. SmartDispatch supporte présentement quatre grappes de calcul : Mammouth (Université de Shebrooke), Colosse et Helios (Université de Laval), et Guillimin (Université McGill).

Smart-Learner

Smart-Learner est une bibliothèque Python encapsulant celle de Theano dans le but de faciliter le développement de nouveaux modèles à base de réseaux de neurones et de simplifier la gestion des expérimentations en apprentissage automatique. Je suis l'un des principaux développeurs de cette bibliothèque. Elle a été utilisée dans tous les projets présentés dans cette thèse.

Tractometer

Le Tractometer est un système permettant l'évaluation quantitative, basée sur la connectivité, de reconstructions des fibres de la matière blanche. Je suis l'un des deux développeurs qui l'ont conçu. Cet outil a servi, entre autres, à évaluer les soumissions reçues lors du *2013 ISBI Hardi Reconstruction Challenge* ainsi que lors du *ISMRM 2015 Challenge*.

Scilpy

Scilpy est la bibliothèque Python contenant les méthodes, les outils et les scripts utilisés au sein du Sherbrooke Connectivity Imaging Lab. Mes contributions se limitent surtout à des revues de codes et des corrections de bogues.

Bibliographie

D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

K. Bache and M. Lichman. UCI Machine Learning Repository, 2013.

F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, nov 2012. URL `http://arxiv.org/abs/1211.5590`.

Y. Bengio. Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009.

Y. Bengio and S. Bengio. Modeling High-Dimensional Discrete Data with Multi-Layer Neural Networks. In *Advances in Neural Information Processing Systems 12*, pages 400–406. MIT Press, 2000.

J. Bergstra, O. Breuleux, F. F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math compiler in Python. *Proceedings of the Python for Scientific Computing Conference (SciPy)*, (Scipy):1–7, 2010.

J. Besag. Statistical analysis of non-lattice data. *The Statistician*, 24(3):179–195, 1975.

C. Bishop. Mixture density networks. Rapport technique, Neural Computing Research Group, Aston University, Birmingham, 1994.

C. Bishop. *Pattern Recognition and Machine Learning*. Information science and statistics. Springer-Verlag New York, Inc., 2006.

J. Bornschein and Y. Bengio. Reweighted Wake-Sleep. 1:1–12, jun 2014.

BIBLIOGRAPHIE

N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent. Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription. In *Proceedings of the 29th International Conference on Machine Learning*, pages 1159–1166. Omnipress, 2012.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*, volume 19. 1984.

Y. Burda, R. Salakhutdinov, and R. Grosse. Importance Weighted Autoencoders. In *Proceedings of the 4th International Conference on Learning Representations*, pages 1–8. arXiv:1509.00519v3, 2016.

K. Cho, T. Raiko, and A. Ilin. Parallel Tempering is Efficient for Learning Restricted Boltzmann Machines. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–8. IEEE, 2010.

K. Cho, T. Raiko, and A. Ilin. Enhanced gradient for training restricted Boltzmann machines. *Neural computation*, 25:805–831, 2013. ISSN 1530-888X.

K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, sep 1995. ISSN 0885-6125.

M.-A. Côté, G. Girard, A. Boré, E. Garyfallidis, J.-C. Houde, and M. Descoteaux. Tractometer: Towards validation of tractography pipelines. *Medical image analysis*, 17(7):857–844, apr 2013. ISSN 1361-8423.

M. A. Cueto, J. Morton, and B. Sturmfels. Geometry of the Restricted Boltzmann Machine. *arXiv e-prints*, pages 1–18, 2009.

BIBLIOGRAPHIE

G. Dahl, R. Adams, and H. Larochelle. Training restricted boltzmann machines on word observations. *arXiv e-prints*, 2012.

G. Dahl, T. Sainath, and G. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8609–8613, 2013.

N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, I:886–893, 2005. ISSN 1063-6919.

P. Dayan, G. Hinton, R. Neal, and R. Zemel. The Helmholtz machine. *Neural Computation*, 7:889–904, 1995.

E. L. Denton, S. Chintala, A. Szlam, and R. Fergus. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. In *Advances in Neural Information Processing Systems 28*, pages 1486–1494. Curran Associates, Inc., 2015.

G. Desjardins, A. Courville, Y. Bengio, P. Vincent, and O. Delalleau. Tempered Markov Chain Monte Carlo for training of Restricted Boltzmann Machine. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, JMLR W&CP*, 9:145–152, 2010a.

G. Desjardins, A. C. Courville, Y. Bengio, P. Vincent, and O. Delalleau. Parallel Tempering for Training of Restricted Boltzmann Machines. In Y. W. Teh and D. M. Titterington, editors, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, JMLR W&CP*, volume 9, pages 145–152. JMLR.org, 2010b.

J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011. ISSN 15324435.

V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv e-prints*, pages 1–28, mar 2016.

S. E. Fahlman and C. Lebiere. The Cascade-Correlation Learning Architecture. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2 (NIPS 1989)*, pages 524–532. Morgan-Kaufmann, 1990.

Y. Freund and D. Haussler. Unsupervised learning of distributions on binary vectors using two layer networks. In *Advances in Neural Information Processing Systems 4*, pages 912–919. Morgan-Kaufmann, 1992.

B. J. Frey, G. E. Hinton, and P. Dayan. Does the wake-sleep algorithm learn good density estimators ? In *Advances in Neural Information Processing Systems 8*, pages 661–670. MIT Press, 1996.

A. Gaidon, Q. Wang, Y. Cabon, and E. Vig. Virtual Worlds as Proxy for Multi-Object Tracking Analysis. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4340–4349, 2016.

J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue. TIMIT acoustic-phonetic continuous speech corpus. NIST, 1993.

E. Garyfallidis, M. Brett, M. M. Correia, G. B. Williams, and I. Nimmo-Smith. QuickBundles, a Method for Tractography Simplification. *Frontiers in neuroscience*, 6: 1–175, jan 2012. ISSN 1662-453X.

E. Garyfallidis, M. Brett, B. Amirbekian, A. Rokem, S. Van Der Walt, M. Descoteaux, and I. Nimmo-Smith. Dipy, a library for the analysis of diffusion MRI data. *Frontiers in Neuroinformatics*, 8, 2014. ISSN 1662-5196.

M. Germain, K. Gregor, I. Murray, and H. Larochelle. MADE: Masked Autoencoder for Distribution Estimation. *Proceedings of the 32nd International Conference on Machine Learning, JMLR W&CP*, 37:881–889, 2015.

Z. Ghahramani and G. E. Hinton. The EM algorithm for mixtures of factor analyzers. Rapport technique, University of Toronto, 1996.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *arXiv e-prints*, pages 1–9, jun 2014. ISSN 10495258.

BIBLIOGRAPHIE

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

A. Graves. Practical Variational Inference for Neural Networks. In *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011.

A. Graves and J. Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. *Advances in Neural Information Processing Systems 21, NIPS'21*, pages 545–552, 2008.

A. Graves, A.-r. Mohamed, and G. Hinton. Speech Recognition with Deep Recurrent Neural Networks. *arXiv e-prints*, pages 1–5, mar 2013.

K. Gregor and Y. LeCun. Learning Representations by Maximizing Compression. Rapport technique, arXiv:1108.1169, 2011.

K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. DRAW: A Recurrent Neural Network For Image Generation. *Proceedings of the 32nd International Conference on Machine Learning, JMLR W&CP*, 37:1462–1471, 2014a.

K. Gregor, A. Mnih, and D. Wierstra. Deep AutoRegressive Networks. *International Conference on Machine Learning*, 32:1242–1250, 2014b.

A. Gretton, K. Borgwardt, M. Rasch, B. Schölkopf, and A. Smola. A Kernel Method for the Two-Sample-Problem. In *Advances in Neural Information Processing Systems 19*, pages 513–520. MIT Press, 2007.

M. Gutmann and A. Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *International Conference on Machine Learning*, pages 297–304, 2010.

S. Harmeling and C. K. I. Williams. Greedy learning of binary latent trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(6):1087–1097, 2011.

K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *Arxiv.Org*, 7(3):171–180, 2015.

BIBLIOGRAPHIE

K. He, X. Zhang, S. Ren, and J. Sun. Identity Mappings in Deep Residual Networks. pages 1–15, 2016.

G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002. ISSN 0899-7667.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1161–1558, 1995.

G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

G. E. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, P. Nguyen, and T. Sainath. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, 29(November):82–97, nov 2012a.

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv e-prints*, pages 1–18, jul 2012b.

A. Hyvärinen. Estimation of Non-Normalized Statistical Models by Score Matching. *Journal of Machine Learning Research*, 6:695–709, 2005.

A. Hyvärinen. Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables. *IEEE Transactions on Neural Networks*, 18:1529–1531, 2007a.

A. Hyvärinen. Some extensions of score matching. *Computational Statistics & Data Analysis*, 51(5):2499–2512, feb 2007b. ISSN 01679473.

D. P. Kingma and J. L. Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

BIBLIOGRAPHIE

D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *Proceedings of the 2nd International Conference on Learning Representations*, pages 1–14. arXiv:1312.6114v10, dec 2013.

A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS*, pages 1–9, 2012.

H. Larochelle and S. Lauly. A neural autoregressive topic model. In *Advances in Neural Information Processing Systems 25*, pages 2708–2716. Curran Associates, Inc., 2012.

H. Larochelle and I. Murray. The Neural Autoregressive Distribution Estimator. In G. J. Gordon, D. B. Dunson, and M. Dudzik, editors, *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, volume 15, pages 29–37. AISTATS, 2011.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015. ISSN 0028-0836.

H. Lee, A. Battle, R. Raina, and A. Y. Ng. Efficient sparse coding algorithms. *Advances in neural information processing systems*, 19:801, 2007.

Y. Li, K. Swersky, and R. Zemel. Generative Moment Matching Networks. *Proceedings of the 32nd International Conference on Machine Learning, JMLR W&CP*, 37: 1718–1727, 2015.

J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015. ISSN 10636919.

D. Lowe. Object recognition from local scale-invariant features. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, pages 1150–1157 vol.2, 1999.

K. Maier-Hein, P. Neher, J.-C. Houde, M.-A. Cote, E. Garyfallidis, J. Zhong, M. Chamberland, F.-C. Yeh, Y. C. Lin, Q. Ji, W. E. Reddick, J. O. Glass, D. Q.

Chen, Y. Feng, C. Gao, Y. Wu, J. Ma, H. Renjie, Q. Li, C.-F. Westin, S. Deslauriers-Gauthier, J. O. O. Gonzalez, M. Paquette, S. St-Jean, G. Girard, F. Rheault, J. Sidhu, C. M. W. Tax, F. Guo, H. Y. Mesri, S. David, M. Froeling, A. M. Heemskerk, A. Leemans, A. Bore, B. Pinsard, C. Bedetti, M. Desrosiers, S. Brambati, J. Doyon, A. Sarica, R. Vasta, A. Cerasa, A. Quattrone, J. Yeatman, A. R. Khan, W. Hodges, S. Alexander, D. Romascano, M. Barakovic, A. Auria, O. Esteban, A. Lemkaddem, J.-P. Thiran, H. E. Cetingul, B. L. Odry, B. Mailhe, M. Nadar, F. Pizzagalli, G. Prasad, J. Villalon-Reina, J. Galvis, P. Thompson, F. Requejo, P. Laguna, L. Lacerda, R. Barrett, F. Dell'Acqua, M. Catani, L. Petit, E. Caruyer, A. Daducci, T. Dyrby, T. Holland-Letz, C. Hilgetag, B. Stieltjes, and M. Descoteaux. Tractography-based connectomes are dominated by false-positive connections. *bioRxiv*, 1-23, 2016.

J. Mairal, F. Bach, J. Ponce, G. Sapiro, and A. Zisserman. Supervised dictionary learning. In *Advances in Neural Information Processing Systems (NIPS)*, number September, pages 1–8, 2008.

B. Marlin, K. Swersky, B. Chen, and N. de Freitas. Inductive principles for Restricted {B}oltzmann Machine learning. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010a.

B. M. Marlin, K. Swersky, B. Chen, and N. de Freitas. Inductive Principles for Restricted Boltzmann Machine Learning. *Proc. Intl. Conference on Artificial Intelligence and Statistics*, 9:305–306, 2010b.

D. Martin, C. Fowlkes, D. Tal, and J. Malik. A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics. In *International Conference on Computer Vision*, volume 2, pages 416–423. IEEE, jul 2001.

T. Mikolov, A. Deoras, and S. Kombrink. Empirical Evaluation and Combination of Advanced Language Modeling Techniques. *Interspeech*, (August):605–608, 2011.

G. Montavon and K.-R. Müller. Deep Boltzmann Machines and the Centering Trick.

In *Neural Networks: Tricks of the Trade, Second Edition*, pages 621–637. Springer, 2012.

G. Montufar and N. Ay. Refinements of Universal Approximation Results for Deep Belief Networks and Restricted Boltzmann Machines. *Neural Computation*, 23(5): 1306–1319, 2011. ISSN 0899-7667.

K. Murphy. *Machine Learning: a Probabilistic Perspective*. 2012.

I. Murray and R. Salakhutdinov. Evaluating probabilities under high-dimensional latent variable models. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1137–1144. Curran Associates, Inc., 2009.

V. Nair and G. E. Hinton. Implicit Mixtures of Restricted Boltzmann Machines. *Advances in Neural Information Processing Systems*, pages 1145–1152, 2008.

R. M. Neal. Connectionist learning of belief networks. *Artificial Intelligence*, 56(1): 71–113, 1992. ISSN 00043702.

P. Neher, F. B. Laun, B. Stieltjes, and K. H. Maier-Hein. Fiberfox: Facilitating the creation of realistic white matter software phantoms. *Magnetic Resonance in Medicine*, 72(5):1460–1470, 2014. ISSN 15222594.

P. Neher, G. Michael, T. Norajitra, C. Weber, and K. H. Maier-hein. A Machine Learning Based Approach to Fiber Tractography Using Classifier Voting. *Medical Image Computing and Computer-Assisted Intervention*, 9349:45–52, 2015.

A. Y. A. Ng and M. I. Jordan. On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes. In *Neural Information Processing System (NIPS)*, volume 14, pages 1–8, 2002.

J. Ngiam, Z. Chen, P. W. Koh, and A. Y. Ng. Learning Deep Energy Models. In *Proceedings of the 28th International Conference on Machine Learning*, pages 1105–1112. Omnipress, 2011.

J. Nocedal and S. J. Wright. *Numerical Optimization*. 2006.

BIBLIOGRAPHIE

P. Orbanz and Y. W. Teh. Bayesian Nonparametric Models. In *Encyclopedia of Machine Learning*. Springer, 2010.

D. Ormoneit and V. Tresp. Improved Gaussian mixture density estimates using Bayesian penalty terms and network averaging. In *Advances in Neural Information Processing Systems 8*, pages 542–548. MIT Press, 1995.

C. Presseau, P.-M. Jodoin, J.-C. Houde, and M. Descoteaux. A new compression format for fiber tracking datasets. *NeuroImage*, 109:73–83, jan 2015. ISSN 10538119.

T. Raiko and Y. Bengio. Iterative Neural Autoregressive Distribution. In *Advances in neural information processing systems*, pages 1–9, 2014.

M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun. Efficient Learning of Sparse Representations with an Energy-Based Model. *Advances in neural information processing systems*, 19:1137–1144, 2006.

D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. *Proceedings of the 31st International Conference on Machine Learning, JMLR W&CP*, 32:1278–1286, 2014.

O. Rippel, M. Gelbart, and R. Adams. Learning Ordered Representations with Nested Dropout. *Proceedings of the 31th International Conference on Machine Learning*, 32:1746–1754, 2014.

L. Rokach. *Data mining with decision trees: theory and applications*. World Scientific, 2008.

S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

R. Salakhutdinov. Learning in {M}arkov Random Fields using Tempered Transitions. In *Advances in Neural Information Processing Systems 22*, pages 1598–1606. Curran Associates, Inc., 2009.

R. Salakhutdinov. Learning Deep Boltzmann Machines using Adaptive MCMC. In *Proceedings of the 27th International Conference on Machine Learning*, pages 943–950. Omnipress, 2010.

R. Salakhutdinov and G. E. Hinton. Deep Boltzmann Machines. *Artificial Intelligence*, 5(2):448–455, 2009.

R. Salakhutdinov and H. Larochelle. Efficient Learning of Deep Boltzmann Machines. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, JMLR W&CP*, 9:693–700, 2010.

R. Salakhutdinov and I. Murray. On the quantitative analysis of Deep Belief Networks. In A. McCallum and S. Roweis, editors, *Proceedings of the 25th Annual International Conference on Machine Learning*, pages 872–879. Omnipress, 2008.

R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, pages 791–798, New York, NY, USA, 2007. ACM.

R. Silva, C. Blundell, and Y. W. Teh. Mixed Cumulative Distribution Networks. *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, JMLR W&CP*, 15:670–678, 2011.

R. Smith, J.-D. Tournier, F. Calamante, and A. Connelly. Anatomically-constrained tractography: Improved diffusion MRI streamlines tractography through effective use of anatomical information. *NeuroImage*, 2(2003):1907, jun 2012. ISSN 1095-9572.

P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing Explorations in the Microstructure of Cognition*, volume 1, chapter 6, pages 194–281. MIT Press, 1986.

P. Smyth and D. Wolpert. Linearly combining density estimators via stacking. *Machine Learning*, 36(1-2):59–83, 1999.

J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems (NIPS)*, pages 1–9, 2012.

J. Sohl-Dickstein, P. Battaglino, and M. R. DeWeese. Minimum Probability Flow Learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 905–912. Omnipress, 2011.

J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for Simplicity: the All Convolutional Net. In *Proceedings of the 3rd International Conference on Learning Representations*, pages 1–14. arXiv:1412.6806v3, 2015.

S. St-Jean, P. Coupé, and M. Descoteaux. Non Local Spatial and Angular Matching: Enabling higher spatial resolution diffusion MRI datasets through adaptive denoising. *Medical image analysis*, 32:115–30, aug 2016. ISSN 1361-8423.

K. Swersky, D. Tarlow, I. Sutskever, R. Salakhutdinov, R. Zemel, and R. Adams. Cardinality Restricted Boltzmann Machines. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, pages 3302–3310, 2012.

P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

Y. Tang, R. Salakhutdinov, and G. E. Hinton. Deep Mixtures of Factor Analysers. In *Proceedings of the 29th International Conference on Machine Learning*, pages 505–512. Omnipress, 2012.

G. W. Taylor, G. E. Hinton, and S. T. Roweis. Two Distributed-State Models For Generating High-Dimensional Time Series. *Journal of Machine Learning Research*, 12:1025–1068, 2011.

The Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng,

B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.0:1–19, may 2016.

L. Theis and M. Bethge. Generative Image Modeling Using Spatial LSTMs. In *Advances in Neural Information Processing Systems 28*, pages 1927–1935. Curran Associates, Inc., 2015.

T. Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In W. W. Cohen, A. McCallum, and S. T. Roweis, editors, *Proceedings of the 25th International Conference on Machine Learning*, volume 307, pages 1064–1071. Omnipress, 2008.

T. Tieleman and G. E. Hinton. Using Fast Weights to Improve Persistent Contrastive Divergence. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1033–1040. Omnipress, 2009.

K. Tretyakov. Machine Learning Techniques in Spam Filtering. *Data Mining Problem-oriented Seminar, MTAT.03.177*, (May):60–79, 2004.

B. Uria. *Connectionist multivariate density-estimation and its application to speech synthesis*. PhD thesis, The University of Edinburgh, 2015.

B. Uria, I. Murray, and H. Larochelle. {RNADE}: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems 26*, pages 2175–2183. Curran Associates, Inc., 2013.

B. Uria, I. Murray, and H. Larochelle. A Deep and Tractable Density Estimator. In *International Conference on Machine Learning*, volume 32, pages 467–475. JMLR.org, 2014.

A. van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, pages 4790–4798, 2016a.

A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel Recurrent Neural Networks. *Proceedings of the 33rd International Conference on Machine Learning, JMLR W&CP*, pages 1–11, jan 2016b.

D. Van Essen, K. Ugurbil, E. Auerbach, D. Barch, T. Behrens, R. Bucholz, A. Chang, L. Chen, M. Corbetta, S. Curtiss, S. Della Penna, D. Feinberg, M. Glasser, N. Harel, A. Heath, L. Larson-Prior, D. Marcus, G. Michalareas, S. Moeller, R. Oostenveld, S. Petersen, F. Prior, B. Schlaggar, S. Smith, A. Snyder, J. Xu, and E. Yacoub. The Human Connectome Project: A data acquisition perspective. *NeuroImage*, 62 (4):2222–2231, oct 2012. ISSN 10538119.

J. Verbeek. Mixture of Factor Analyzers Matlab implementation, 2005.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1096–1103. Omnipress, 2008.

M. Welling, R. Zemel, and G. Hinton. Self Supervised Boosting. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 681–688. MIT Press, 2003.

M. Welling, M. Rosen-Zvi, and G. E. Hinton. Exponential family harmoniums with an application to information retrieval. In *Advances in Neural Information Processing Systems 17*, pages 1481–1488. MIT Press, 2005.

Wikipedia. Observable universe — Wikipedia, the free encyclopedia, 2017. URL `https://en.wikipedia.org/wiki/Observable_universe#Matter_content_.E2.80.93_number_of_atoms`. [En ligne; accédé le 04 avril 2017].

BIBLIOGRAPHIE

R. J. Wilson. *Introduction to Graph Theory, fourth edition.* Addison Wesley, 1996.

L. Younes. Parameter inference for imperfectly observed Gibbsian fields. *Probability Theory Related Fields*, 82:625–645, 1989.

Y. Zheng, R. Zemel, Y.-J. Zhang, and H. Larochelle. A Neural Autoregressive Approach to Attention-based Recognition. *International Journal of Computer Vision*, 113(1):67–79, 2015a.

Y. Zheng, Y.-J. Zhang, and H. Larochelle. A Deep and Autoregressive Approach for Topic Modeling of Multimodal Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(6):1056–1069, 2015b.

G. Zhou, K. Sohn, and H. Lee. Online Incremental Feature Learning with Denoising Autoencoders. In N. D. Lawrence and M. A. Girolami, editors, *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics*, volume 22, pages 1453–1461. JMLR.org, 2012.

D. Zoran and Y. Weiss. From learning models of natural image patches to whole image restoration. In *International Conference on Computer Vision*, pages 479–486. IEEE, 2011.

D. Zoran and Y. Weiss. Natural images, Gaussian mixtures and dead leaves. In *Advances in Neural Information Processing Systems 25*, pages 1745–1753. Curran Associates, Inc., 2012.