

Introduction au **Deep **Learning****

Chez le même éditeur

Introduction au Machine Learning

Chloé-Agathe Azencott

240 pages

Dunod, 2019

Big Data et Machine Learning

3^e édition

Pirmin Lemberger, Marc Batty, Médéric Morel, Jean-Luc Raffaëlli

272 pages

Dunod, 2019

Machine Learning avec Scikit-Learn

2^e édition

Aurélien Géron

320 pages

Dunod, 2019

Deep Learning avec Keras et TensorFlow

2^e édition

Aurélien Géron

576 pages

Dunod, 2020

Introduction au **Deep** **Learning**

Eugene Charniak

Professeur d'informatique et de sciences cognitives
à l'Université Brown

Traduit de l'anglais par **Anne Bohy**

DUNOD

Traduction de l'ouvrage d'Eugene Charniak :
Introduction to Deep Learning
Copyright © 2018, The Massachusetts Institute of Technology

Illustration de couverture : © kras99 - Adobe Stock

<p>Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.</p> <p>Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements</p>	<p>d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.</p> <p>Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).</p>
--	--



© Dunod, 2021

11 rue Paul Bert, 92240 Malakoff

www.dunod.com

ISBN 978-2-10-081926-3

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

AVANT-PROPOS	ix
CHAPITRE 1 • RÉSEAUX DE NEURONES À PROPAGATION AVANT	1
1.1 Les perceptrons	3
1.2 Fonctions de perte d'entropie croisée pour les réseaux de neurones	8
1.3 Dérivées et descente de gradient stochastique	13
1.4 Écriture de notre programme	17
1.5 Représentation matricielle des réseaux de neurones	20
1.6 Indépendance des données	23
1.7 Références et lectures supplémentaires	24
Exercices	25
CHAPITRE 2 • TENSORFLOW	27
2.1 Introduction à TensorFlow	27
2.2 Un programme TF	30
2.3 Réseaux de neurones à plusieurs couches	36
2.4 Autres particularités	39
2.5 Références et lectures supplémentaires	45
Exercices	45
CHAPITRE 3 • RÉSEAUX DE NEURONES CONVOLUTIFS	47
3.1 Filtres, pas et marge	47
3.2 Exemple simple de convolution en TF	53
3.3 Convolution à plusieurs niveaux	56
3.4 Présentation détaillée de la convolution	59
3.5 Références et lectures supplémentaires	61
Exercices	63
CHAPITRE 4 • PLONGEMENTS DE MOTS ET RÉSEAUX DE NEURONES RÉCURRENTS	65
4.1 Plongement de mots pour modèles linguistiques	65
4.2 Construction de modèles linguistiques à propagation avant	70

Table des matières

4.3	Amélioration des modèles linguistiques à propagation avant	72
4.4	Surajustement	73
4.5	Réseaux récurrents	76
4.6	Longue mémoire à court terme	82
4.7	Références et lectures supplémentaires	85
	Exercices	86
CHAPITRE 5 • APPRENTISSAGE SÉQUENCE À SÉQUENCE		89
5.1	Principe du seq2seq	90
5.2	Écriture d'un programme de traduction automatique seq2seq	92
5.3	L'attention dans seq2seq	95
5.4	Seq2Seq multi-longueurs	99
5.5	Exemples de programmation	100
5.6	Références et lectures supplémentaires	103
	Exercices	104
CHAPITRE 6 • APPRENTISSAGE PAR RENFORCEMENT PROFOND		105
6.1	Itération sur les valeurs	106
6.2	Apprentissage sans modèle	109
6.3	Les bases de l'apprentissage par renforcement profond	111
6.4	Méthodes de gradients de politique	115
6.5	Méthodes acteur-critique	121
6.6	Répétition d'expériences	124
6.7	Références et lectures supplémentaires	125
	Exercices	126
CHAPITRE 7 • MODÈLES DE RÉSEAUX DE NEURONES NON SUPERVISÉS		127
7.1	Encodage de base	127
7.2	Auto-encodage convolutif	130
7.3	Auto-encodage variationnel	134
7.4	Réseaux antagonistes génératifs	142
7.5	Références et lectures supplémentaires	146
	Exercices	147

ANNEXE : CORRIGÉ DES EXERCICES	149
A.1 Chapitre 1	149
A.2 Chapitre 2	149
A.3 Chapitre 3	150
A.4 Chapitre 4	151
A.5 Chapitre 5	151
A.6 Chapitre 6	152
A.7 Chapitre 7	152
BIBLIOGRAPHIE	153
INDEX	157

AVANT-PROPOS

Votre auteur, de longue date chercheur en intelligence artificielle, a vu son domaine d'expertise, le traitement du langage naturel, révolutionné par le Deep Learning. Malheureusement, il lui a fallu (il m'a fallu) longtemps pour admettre ce fait. Je peux le justifier en disant que c'est la troisième fois que les réseaux de neurones menacent de tout révolutionner, mais seulement la première fois qu'ils y parviennent. Quoi qu'il en soit, je me suis retrouvé soudainement à la traîne et peinant à rattraper le temps perdu. J'ai donc fait ce que ferait tout professeur qui se respecte : j'ai inclus cette matière dans mon planning d'enseignement, j'ai démarré une formation accélérée en surfant sur le Web, et finalement ce sont mes étudiants qui me l'ont enseignée. (Cette dernière affirmation n'est pas une boutade. Il me faut saluer tout particulièrement l'aide de Siddarth (Sidd) Karamcheti, assistant d'enseignement principal en premier cycle.)

Ceci explique plusieurs caractéristiques essentielles de ce livre. En premier lieu, il est court. J'apprends lentement. Ensuite, il est très orienté projet. De nombreux écrits, tout particulièrement en informatique, sont en perpétuelle tension entre l'organisation propre au sujet et l'organisation des données dans le cadre de projets spécifiques. Séparer les deux est souvent une bonne idée, mais je pense que j'apprends mieux l'informatique en m'asseyant et en écrivant des programmes, c'est pourquoi mon livre reflète largement mon mode d'apprentissage. C'était la façon la plus pratique de tout formaliser, et j'espère que nombreux seront les futurs lecteurs qui trouveront également cela utile.

Ce qui soulève la question de mes futurs lecteurs. Si j'espère que de nombreux adeptes de la programmation trouveront ce livre utile pour la raison même qui m'a poussé à l'écrire, en tant qu'enseignant mes étudiants sont ma première préoccupation, c'est pourquoi ce livre est conçu avant tout comme un support de cours sur le Deep Learning.

Mon enseignement à l'université de Brown s'adresse aux étudiants de premier et deuxième cycles et couvre l'ensemble des sujets développés ici, auxquels s'ajoutent quelques conférences de « culture générale ». Pour obtenir son diplôme, un étudiant doit aussi mener à bien un projet d'une certaine importance. Des connaissances en algèbre linéaire et en analyse multivariée sont requises. Bien que les notions d'algèbre linéaire utilisées ici soient relativement simples, les étudiants m'ont déclaré que sans ces bases ils auraient eu du mal à concevoir des réseaux à plusieurs niveaux et que les tenseurs qu'ils requièrent leur auraient paru difficiles. L'analyse multivariée, cependant, a constitué une plus grande difficulté. Elle n'apparaît explicitement qu'au chapitre 1, lorsque nous construisons entièrement le réseau jusqu'à la rétropropagation et je ne serais pas surpris qu'un complément sur les dérivées partielles s'avère nécessaire. Enfin, il y a un prérequis en probabilités et statistiques. Ceci aide à la

Avant-propos

bonne compréhension de l'exposé et je tiens à encourager les étudiants à suivre un tel cours. Je suppose également que le lecteur a une connaissance rudimentaire de Python. Je n'inclus pas ce sujet dans mon livre, mais mon cours comporte un TD supplémentaire sur les bases de ce langage.

Le fait que votre auteur était en train d'effectuer une formation de rattrapage tout en écrivant ce livre explique également que dans les lectures supplémentaires proposées à la fin de chaque chapitre on puisse trouver, au-delà des références habituelles aux travaux de recherche les plus marquants, de nombreuses références à des sources secondaires — les écrits à caractère pédagogique d'autres personnes. Je n'aurais jamais réussi à apprendre toutes ces choses sans eux.

Providence, Rhode Island, USA
Janvier 2018

RÉSEAUX DE NEURONES À PROPAGATION AVANT

1

Pour explorer le *deep learning* (ou *apprentissage profond*, en français), il est courant de s'intéresser tout d'abord à la vision par ordinateur. Ce secteur de l'intelligence artificielle a été révolutionné par cette technique car l'information disponible au départ — l'*intensité lumineuse* — est représentée naturellement par des nombres réels, ce que manipulent justement les réseaux de neurones utilisés en apprentissage profond.

Pour parler concrètement, considérons le problème de l'identification de chiffres manuscrits, soit les caractères 0 à 9. Si nous partions de rien, il nous faudrait tout d'abord construire un appareil photo pour concentrer les rayons lumineux afin de construire une image de ce que nous voyons. Il nous faudrait alors des capteurs de luminosité pour transformer les rayons lumineux en impulsions électriques que la machine puisse détecter. Enfin, étant donné que nous avons affaire à des ordinateurs traitant des données binaires, il nous faudrait *discrétiser* l'image — c'est-à-dire représenter les couleurs et les intensités lumineuses par des nombres dans un tableau à deux dimensions. Fort heureusement, nous disposons d'un jeu de données en ligne dans lequel tout ce travail a été effectué pour nous — le jeu de données *Mnist* (prononcer « em-nist »). La partie « nist » de ce nom désigne le *National Institute of Standards* américain (ou *NIST*), qui a rassemblé ces données. Dans ce jeu de données, chaque image est représentée par un tableau d'entiers $28 * 28$ analogue à celui de la figure 1.1. (Pour les besoins de la mise en page, les colonnes correspondant aux bordures gauche et droite de l'image ont été supprimées.)

Sur la figure 1.1, 0 peut être interprété comme du blanc, 255 comme du noir et les valeurs intermédiaires comme des nuances de gris. Ces nombres sont appelés *valeurs des pixels*, un *pixel* étant la plus petite portion de l'image que l'ordinateur sache distinguer. Dans le monde réel, la taille exacte de la zone représentée par un pixel dépend de notre appareil photographique, de sa distance à la surface de l'objet, etc. Mais dans cette étude portant sur de simples chiffres, nous n'avons pas à nous en soucier. L'image en noir et blanc correspondant à ce tableau est présentée sur la figure 1.2.

Une étude attentive de cette image peut nous suggérer des moyens simplistes d'effectuer notre tâche. Remarquons par exemple que le pixel à la position [8, 8] est noir. Sachant qu'il s'agit de l'image d'un « 7 », c'est plutôt normal. De même, les « 7 » présentent souvent une zone claire au milieu — le pixel [13, 13], par exemple, a une valeur d'intensité de 0. Comparez ceci au chiffre « 1 », qui présente des valeurs opposées dans ces deux positions, étant donné que ce chiffre écrit normalement n'occupe pas le coin supérieur gauche mais remplit la partie médiane. En y réfléchissant un peu, nous pourrions trouver de nombreuses *heuristiques* (règles fonctionnant souvent, mais pas toujours) telles que celle-ci, puis écrire un programme de classification les utilisant.

Chapitre 1 · Réseaux de neurones à propagation avant

	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	185	159	151	60	36	0	0	0	0	0	0	0	0	0
8	254	254	254	254	241	198	198	198	198	198	198	198	198	170
9	114	72	114	163	227	254	225	254	254	254	250	229	254	254
10	0	0	0	0	17	66	14	67	67	67	59	21	236	254
11	0	0	0	0	0	0	0	0	0	0	0	83	253	209
12	0	0	0	0	0	0	0	0	0	0	22	233	255	83
13	0	0	0	0	0	0	0	0	0	0	129	254	238	44
14	0	0	0	0	0	0	0	0	0	59	249	254	62	0
15	0	0	0	0	0	0	0	0	0	133	254	187	5	0
16	0	0	0	0	0	0	0	0	9	205	248	58	0	0
17	0	0	0	0	0	0	0	0	126	254	182	0	0	0
18	0	0	0	0	0	0	0	75	251	240	57	0	0	0
19	0	0	0	0	0	0	19	221	254	166	0	0	0	0
20	0	0	0	0	0	3	203	254	219	35	0	0	0	0
21	0	0	0	0	0	38	254	254	77	0	0	0	0	0
22	0	0	0	0	31	224	254	115	1	0	0	0	0	0
23	0	0	0	0	133	254	254	52	0	0	0	0	0	0
24	0	0	0	61	242	254	254	52	0	0	0	0	0	0
25	0	0	0	121	254	254	219	40	0	0	0	0	0	0
26	0	0	0	121	254	207	18	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1.1 – Version numérisée d'une image Mnist



Figure 1.2 – Image en noir et blanc correspondant aux pixels de la figure 1.1

Cependant, ce n'est pas ce que nous allons faire, car dans ce livre nous nous concentrons sur l'*apprentissage automatique* (en anglais, *machine learning*). Pour cela, nous abordons la tâche en cherchant comment permettre à un ordinateur d'apprendre en lui fournissant des exemples accompagnés de la réponse correcte. Dans le cas présent, nous voulons que notre programme apprenne à identifier des images de chiffres constituées de $28 * 28$ pixels en lui en fournissant des exemples assortis de leurs réponses (appelées également *étiquettes*). En apprentissage automatique, on appelle ceci un problème d'*apprentissage supervisé* ou, pour être plus précis, un problème d'*apprentissage totalement supervisé*, dans la mesure où, pour chaque exemple d'apprentissage, nous fournissons également à l'ordinateur la réponse correcte. Dans les chapitres qui suivent, par exemple au chapitre 6, nous n'aurons pas ce luxe. Nous y découvrirons un problème *semi-supervisé*, voire même, au chapitre 7, un *apprentissage non supervisé*. Nous verrons alors comment cela fonctionne.

Après avoir fait abstraction de l'étude détaillée des rayons lumineux et des surfaces, il nous reste un problème de *classification* : étant donné une entité (disposant d'un ensemble de données d'entrée ou *caractéristiques*) et étant donné un nombre fini de solutions possibles en sortie, associer cette entité à l'une de ces solutions (ou la *classifier* dans celle-ci). Dans notre cas les entrées sont des pixels, et la classification s'effectue entre 10 chiffres possibles. Le vecteur des l entrées (pixels) sera noté $\mathbf{x} = [x_1, x_2 \dots x_l]$ et la réponse a . En règle générale, les entrées sont des nombres réels pouvant être positifs ou négatifs, même si dans notre cas ce sont tous des entiers positifs.

1.1 LES PERCEPTRONS

Nous allons toutefois commencer par un problème plus simple, en créant un programme qui décidera si une image est un zéro ou non. C'est ce qu'on appelle un problème de *classification binaire*. L'un des tout premiers procédés d'apprentissage automatique pour la classification binaire a été le *perceptron*, présenté sur la figure 1.3.

Les perceptrons ont été conçus comme une modélisation informatique simple des neurones. Un neurone (voir figure 1.4) comporte d'ordinaire de nombreuses entrées (*dendrites*), un *corps de cellule* et une seule sortie (l'*axone*). Faisant écho à cela, le perceptron reçoit de nombreuses entrées et possède une seule sortie. Un perceptron simple permettant de décider si une image $28 * 28$ est celle d'un zéro aurait 784 entrées, une par pixel, et une sortie. Pour simplifier sa représentation, le perceptron de la figure 1.3 n'a que cinq entrées.

Un perceptron consiste en un vecteur de *poids* $\mathbf{w} = [w_1 \dots w_m]$, un pour chaque entrée, plus un poids particulier b , appelé *biais*. Nous appelons \mathbf{w} et b les *paramètres* du perceptron. Plus généralement, nous utilisons Φ pour désigner les paramètres, $\phi_i \in \Phi$ désignant le i -ième paramètre. Pour un perceptron, $\Phi = \{\mathbf{w} \cup b\}$.

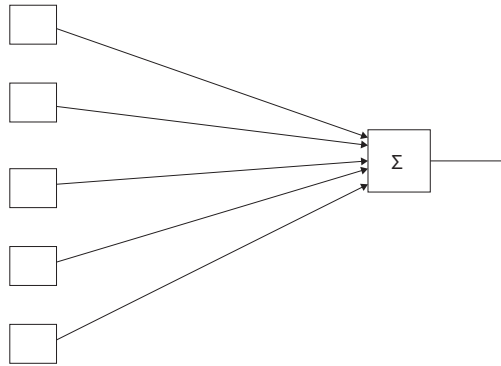


Figure 1.3 - Diagramme schématisé d'un perceptron

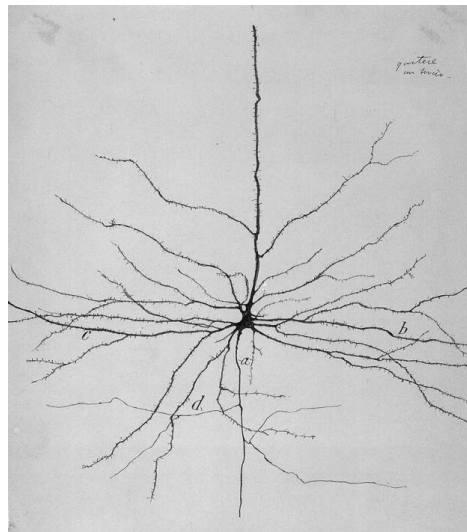


Figure 1.4 - Un neurone ordinaire

À l'aide de ces paramètres, le perceptron calcule la fonction suivante :

$$f_{\Phi}(\mathbf{x}) = \begin{cases} 1 & \text{si } b + \sum_{i=1}^l x_i w_i > 0 \\ 0 & \text{sinon} \end{cases} \quad (1.1)$$

Autrement dit, nous multiplions chaque entrée du perceptron par le poids correspondant et ajoutons le biais. Si cette valeur est supérieure à zéro nous renvoyons 1, sinon 0. Les perceptrons, souvenez-vous, sont des classificateurs binaires, donc 1 indique que \mathbf{x} appartient à la classe, et 0 qu'il ne lui appartient pas.

Le *produit scalaire* de deux vecteurs de longueur l se définit comme

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^l x_i y_i \quad (1.2)$$

ce qui nous permet de simplifier comme suit la notation du calcul effectué par le perceptron :

$$f_{\Phi}(\mathbf{x}) = \begin{cases} 1 & \text{si } b + \mathbf{w} \cdot \mathbf{x} > 0 \\ 0 & \text{sinon} \end{cases} \quad (1.3)$$

Les éléments calculant $b + \mathbf{w} \cdot \mathbf{x}$ sont appelés *unités linéaires* et comme dans la formule 1.3 nous les identifions par un Σ . Par ailleurs, lorsque nous entreprenons d'ajuster les paramètres il est pratique de reconverter le biais sous forme d'un poids supplémentaire dans \mathbf{w} , la caractéristique associée ayant toujours pour valeur 1. Ainsi, nous pouvons nous contenter de dire que nous ajustons les paramètres \mathbf{w} .

Les perceptrons sont intéressants parce qu'ils utilisent un algorithme remarquablement simple et robuste, l'*algorithme du perceptron*, pour déterminer ces Φ à partir des *exemples d'entraînement*. Nous identifions l'exemple dont nous parlons à l'aide d'un exposant, par conséquent les entrées du k -ième exemple sont $\mathbf{x}^k = [x_1^k \dots x_l^k]$ et la réponse correspondante est a^k . Pour un classificateur binaire tel qu'un perceptron, la réponse est 1 ou 0 pour indiquer l'appartenance ou non à la classe. Pour effectuer une classification à m classes, la réponse serait un entier de 0 à $m-1$.

Il est parfois utile de définir l'apprentissage automatique comme un problème d'*approximation de fonction*. De ce point de vue, un perceptron unique définit une *classe paramétrée* de fonctions. Apprendre les poids du perceptron revient à choisir le membre de la classe qui constitue la meilleure approximation de la fonction solution : la « vraie » fonction qui, étant donné un ensemble de valeurs de pixels, identifie correctement si l'image est, par exemple, un zéro ou non.

Comme dans toutes les expériences d'apprentissage automatique, nous supposons que nous avons au moins deux, et de préférence trois, jeux d'exemples du problème. Le premier est le *jeu d'entraînement*. Il est utilisé pour ajuster les paramètres du modèle. Le deuxième est appelé *jeu de développement* et est utilisé pour tester le modèle tandis que nous cherchons à l'améliorer (on l'appelle aussi *jeu de réserve* ou *jeu de validation*). Le troisième est le *jeu de test*. Une fois que le modèle est établi et qu'il fournit (si nous sommes chanceux) de bons résultats, nous l'évaluons sur les exemples du jeu de test. Ceci nous évite de développer accidentellement un programme qui fonctionne sur le jeu de développement mais pas sur des données qui n'ont pas encore été vues. Ces jeux de données sont parfois appelés *corpora*, puisqu'on parle aussi de « corpus de test ». Les données Mnist que nous utilisons sont disponibles sur Internet. Les données d'entraînement comportent 60 000 images accompagnées de leurs étiquettes correctes, tandis que le jeu de développement/test comporte 10 000 images avec étiquettes.

Chapitre 1 · Réseaux de neurones à propagation avant

La propriété très intéressante de l'algorithme du perceptron est que, s'il existe un ensemble de valeurs des paramètres permettant au perceptron de classifier correctement tout le jeu d'entraînement, il est certain que l'algorithme le trouvera. Malheureusement, pour la plupart des exemples du monde réel, il n'existe pas un tel ensemble de valeurs. Par contre, même dans ce cas, les perceptrons travaillent souvent remarquablement bien en ce sens qu'ils trouvent des valeurs de paramètres permettant d'étiqueter correctement un très fort pourcentage des exemples.

L'algorithme effectue plusieurs itérations sur le jeu d'entraînement, en ajustant peu à peu les paramètres pour accroître le nombre d'exemples correctement identifiés. Une fois qu'il parcourt le jeu d'entraînement sans avoir besoin de modifier aucun paramètre, il sait qu'il a obtenu un ensemble de valeurs correctes et peut s'arrêter. Cependant, s'il n'existe pas un tel ensemble de valeurs, il pourrait poursuivre éternellement. Pour éviter cela, nous devons l'interrompre au bout de N itérations, où N est un paramètre système défini par le programmeur. En règle générale, N croît avec le nombre total de paramètres à apprendre. Dorénavant, nous prendrons soin de distinguer les paramètres système Φ et d'autres nombres associés à notre programme que nous serions tentés sinon d'appeler « paramètres » mais qui ne font pas partie de Φ , comme par exemple N , le nombre d'itérations à effectuer sur le jeu d'entraînement. Nous appellerons ces derniers *hyperparamètres*. La figure 1.5 fournit le pseudo-code pour cet algorithme. Comme c'est la norme, Δx signifie « modification de x ».

1. Initialiser b et tous les \mathbf{w} à 0
2. Pendant N itérations, ou jusqu'à ce que les poids ne changent plus
 - (a) pour chaque exemple d'entraînement \mathbf{x}^k et sa réponse a^k
 - i. si $a^k - f(\mathbf{x}^k) = 0$ continuer
 - ii. sinon pour tous les poids w_i , $\Delta w_i = (a^k - f(\mathbf{x}^k))x_i$

Figure 1.5 - L'algorithme du perceptron

Les lignes essentielles sont ici 2(a)i et 2(a)ii. Ici a^k vaut 1 ou 0, indiquant ainsi si l'image appartient à la classe ($a^k = 1$) ou non. Par conséquent, la première des deux lignes indique que si la sortie du perceptron est identique à l'étiquette, il n'y a rien à faire. La seconde spécifie comment modifier le poids w_i de telle sorte que, si nous refaisons immédiatement l'essai avec cet exemple, le perceptron trouverait la bonne réponse ou au moins une réponse moins erronée, ceci en ajoutant $(a^k - f(\mathbf{x}^k))x_i^k$ à chaque paramètre w_i .

Le meilleur moyen de voir si la ligne 2(a)ii fait ce que nous voulons est d'examiner les différents cas de figure. Supposons que l'exemple d'entraînement \mathbf{x}^k appartienne à la classe. Ceci signifie que son étiquette $a^k = 1$. Étant donné que nous n'avons pas obtenu la bonne réponse, $f(\mathbf{x}^k)$ (la sortie du perceptron pour le k -ième exemple d'entraînement) doit avoir été 0, par conséquent $(a^k - f(\mathbf{x}^k)) = 1$ et, pour tout i , $\Delta w_i = x_i$. Toutes les valeurs des pixels étant ≥ 0 , l'algorithme augmente les poids, et

la prochaine fois $f(x^k)$ renvoie une valeur plus élevée — donc « moins mauvaise ». À titre d'exercice, vous pouvez montrer que la formule fait ce que nous voulons dans la situation inverse, c'est-à-dire lorsque l'exemple n'appartient pas à la classe mais que le perceptron dit le contraire.

En ce qui concerne le biais b , nous le traitons comme le poids d'une caractéristique imaginaire x_0 dont la valeur est toujours 1. Le raisonnement ci-dessus s'applique sans modification.

Voyons un petit exemple. Ici nous ne considérons (et ajustons) que les poids de quatre pixels, les pixels [7, 7] (au centre de l'angle supérieur gauche), [7, 14] (au centre de la partie haute), [14, 7] et [14, 14]. Il est pratique en général de diviser les valeurs des pixels pour qu'elles soient toutes comprises entre 0 et 1. Supposons que notre image soit un zéro, alors $a = 1$, et les valeurs des pixels pour ces quatre emplacements sont respectivement 0.8, 0.9, 0.6 et 0. Étant donné qu'à l'origine tous nos paramètres sont nuls, lorsque nous évaluons $f(x)$ sur la première image $\mathbf{w} \cdot \mathbf{x} + b = 0$, donc $f(\mathbf{x}) = 0$, et donc notre image n'est pas classifiée correctement et $a(1) - f(\mathbf{x}_1) = 1$. Par conséquent le poids $w_{7,7}$ devient $(0 + 0.8 * 1) = 0.8$. De même, les deux w_j suivants deviennent 0.9 et 0.6. Le poids du pixel central reste zéro (car la valeur de ce pixel est zéro). Le biais devient 1.0. Remarquez en particulier que si nous fournissons à nouveau cette image au perceptron, avec ces nouveaux poids elle sera classifiée correctement.

Supposons que l'image suivante ne soit pas un zéro, mais plutôt le chiffre « 1 », et que les deux pixels du centre aient la valeur 1 et les autres 0. Tout d'abord, $b + \mathbf{w} \cdot \mathbf{x} = 1 + 0.8 * 0 + 0.9 * 1 + 0.6 * 0 + 0 * 1 = 1.9$, donc $f(x) > 0$ et le perceptron classe par erreur cet exemple comme un zéro. Par conséquent $f(x) - l_x = 0 - 1 = -1$ et nous ajusterons chaque poids conformément à la ligne 2(a)ii. $w_{0,0}$ et $w_{14,7}$ sont inchangés parce que les valeurs des pixels sont nulles, tandis que $w_{7,14}$ devient maintenant $0.9 - 0.9 * 1 = 0$ (la valeur précédente moins le poids multiplié par la valeur du pixel). À vous de calculer les nouvelles valeurs de b et $w_{14,14}$!

Nous parcourons plusieurs fois le jeu d'entraînement. Chaque parcours des données est appelé *époque* (en anglais, *epoch*). Notez aussi que si les données d'entraînement sont présentées au programme dans un ordre différent, l'évolution des poids est différente. Il est de bonne pratique de présenter les données de chaque époque dans un ordre aléatoire. Nous reviendrons sur ce point à la section 1.6. Cependant, pour les lecteurs abordant ce sujet pour la première fois, nous nous accordons quelque latitude et omettons cette subtilité.

Nous pouvons étendre les perceptrons à des problèmes de *décision multi-classes* en ne créant pas un seul perceptron, mais un pour chacune des classes que nous voulons reconnaître. Pour notre problème original à 10 classes, il nous faudrait 10 perceptrons, un pour chaque chiffre, et renvoyer la classe dont le perceptron a fourni la plus grande valeur. Ceci est représenté graphiquement sur la figure 1.6, où nous voyons trois perceptrons permettant d'identifier l'appartenance d'une image à l'une des trois classes d'objets.

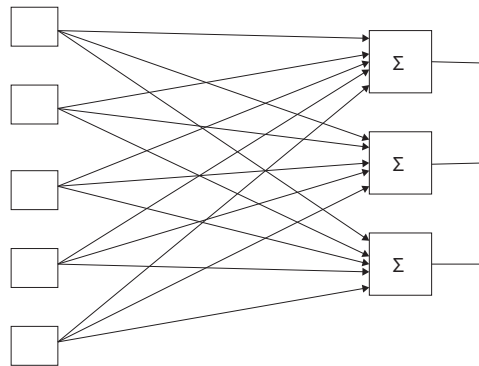


Figure 1.6 – Perceptrons multiples pour l'identification de classes multiples

Même si les interconnexions de la figure 1.6 paraissent compliquées, il ne s'agit en réalité que de trois perceptrons distincts partageant les mêmes entrées. En dehors du fait que la réponse renvoyée par le perceptron multi-classe est le numéro de l'unité linéaire renvoyant la plus grande valeur, tous les perceptrons sont entraînés indépendamment les uns des autres, en appliquant exactement le même algorithme vu précédemment. Par conséquent, étant donné une image et son étiquette, nous exécutons dix fois l'itération (a) de l'algorithme du perceptron pour chacun des 10 perceptrons. Si l'étiquette est par exemple cinq mais que le perceptron ayant fourni la plus haute valeur est le six, alors les perceptrons de zéro à quatre ne changent pas leurs paramètres (étant donné qu'ils ont correctement répondu qu'il ne s'agissait pas d'un zéro, d'un « 1 », etc.). Même chose pour les perceptrons de sept à neuf. Par contre, les perceptrons cinq et six modifient leurs paramètres car ils ont fourni des réponses incorrectes.

1.2 FONCTIONS DE PERTE D'ENTROPIE CROISÉE POUR LES RÉSEAUX DE NEURONES

Dans les tout débuts de cette technique, une discussion au sujet des *réseaux de neurones* (en anglais *neural networks* ou NN) aurait été accompagnée de diagrammes très semblables à ceux de la figure 1.6 où l'accent est mis sur les éléments de calcul individuels (appelés *unités linéaires*). De nos jours, on utilise un très grand nombre de tels éléments organisés en *couches*, chacune d'elles étant constituée par un groupe d'unités de stockage ou de calcul travaillant en parallèle et transmettant des valeurs à une autre couche. La figure 1.7 est une autre version de la figure 1.6 illustrant cette conception. Elle présente une couche d'entrée alimentant une couche de calcul.

Le terme de « couche » sous-entend qu'il peut y en avoir plusieurs, chacune d'entre elles alimentant la suivante. C'est effectivement le cas, et cet empilement de couches

1.2 Fonctions de perte d'entropie croisée pour les réseaux de neurones

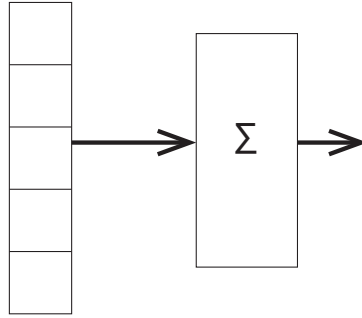


Figure 1.7 – Réseau de neurones à plusieurs couches

correspond au « profond » de l'« apprentissage profond » (ou au « deep » du « deep learning »).

Cependant, les couches multiples ne fonctionnent pas bien avec les perceptrons, c'est pourquoi il nous faut une autre méthode pour apprendre à modifier les poids. Dans cette section, nous allons voir comment le faire en utilisant une autre configuration de réseau très simple, les *réseaux de neurones à propagation avant* (en anglais, *feed-forward neural networks*), et une technique d'apprentissage relativement simple, la *descente de gradient*. (Par contre, certains experts donnent à un réseau de neurones à propagation avant entraîné avec une descente de gradient l'appellation de *perceptron multicouche*.)

Avant de parler de descente de gradient, cependant, il nous faut d'abord parler de la *fonction de perte*. Une fonction de perte associée à chaque résultat une valeur indiquant à quel point ce résultat est « mauvais » pour nous. Dans l'apprentissage des paramètres du modèle, notre objectif est de minimiser la perte. La fonction de perte pour un perceptron prend la valeur 0 pour chaque exemple d'entraînement pour lequel la bonne réponse a été trouvée, et la valeur 1 s'il n'a pas trouvé la bonne valeur. On parle de *perte zéro-un*. La perte zéro-un a l'avantage d'être plutôt évidente, si évidente que nous ne nous sommes jamais souciés de justifier son utilisation. Pourtant, elle présente quelques désavantages. En particulier, elle ne fonctionne pas bien pour un apprentissage par descente de gradient, où l'idée de base est de modifier un paramètre selon la règle suivante :

$$\Delta\phi_i = -\mathcal{L} \frac{\partial L}{\partial \phi_i} \quad (1.4)$$

Ici \mathcal{L} est la *taux d'apprentissage* (en anglais, *learning rate*), un nombre réel qui détermine de combien nous voulons changer un paramètre à chaque étape. La partie importante est la dérivée partielle de la perte L par rapport au paramètre que nous ajustons. Ce qui revient à dire que si nous ne pouvons trouver comment la perte est affectée par le paramètre en question, nous devrions modifier le paramètre pour

diminuer la perte (d'où le signe $-$ précédant \mathcal{L}). Dans notre perceptron, et plus généralement dans les réseaux de neurones, le résultat est déterminé par Φ , les paramètres du modèle, de sorte que dans de tels modèles la perte est une fonction $L(\Phi)$.

Pour en simplifier la visualisation, supposons que notre perceptron n'ait que deux paramètres. Nous pouvons matérialiser cela par un plan euclidien avec deux axes ϕ_1 et ϕ_2 , et, pour chaque point dans le plan, la valeur de la fonction de perte se situe au-dessus ou en dessous du point. Supposons que les valeurs courantes de nos paramètres soient respectivement 1.0 et 2.2. Observez le plan en (1.0, 2.2) et voyez comment L se comporte en ce point. La figure 1.8, une coupe selon le plan $\phi_2 = 2.2$, montre comment une perte imaginaire évolue en fonction de ϕ_1 . Observez la perte lorsque $\phi_1 = 1$. Nous voyons que la tangente a une pente d'environ $-\frac{1}{4}$. Si le taux d'apprentissage $\mathcal{L} = 0.5$, alors l'équation 1.4 nous fait ajouter $(-0.5) * (-\frac{1}{4}) = 0.125$ — ce qui revient à nous déplacer de 0.125 unité vers la droite, ce qui fait effectivement décroître la perte.

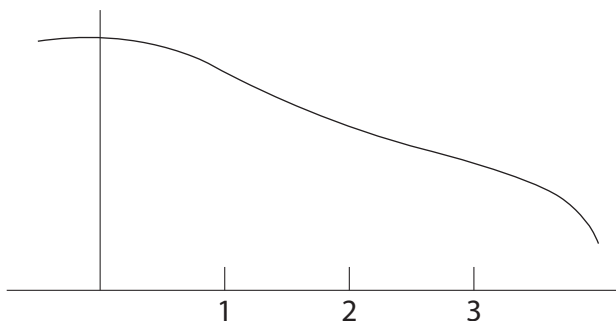


Figure 1.8 - Perte en tant que fonction de ϕ_1

Pour que l'équation 1.4 fonctionne, la perte doit être une fonction différentiable des paramètres, propriété que ne possède pas la perte zéro-un. Pour le visualiser, imaginez le graphe du nombre d'erreurs que nous commettons en fonction d'un certain paramètre, ϕ . Supposons que nous venons juste d'évaluer notre perceptron sur un exemple et qu'il n'a pas fourni le bon résultat. Eh bien, si nous continuons à accroître ϕ (ou peut-être le faisons décroître) et si nous poursuivons cela assez longtemps, en fin de compte $f(x)$ changera de valeur et nous obtiendrons la réponse correcte sur cet exemple. Lorsque nous observons le graphe, c'est celui d'une fonction en escalier. Mais les fonctions en escalier ne sont pas différentiables.

Il existe cependant d'autres fonctions de perte. La fonction de perte la plus courante, ce qu'il y a de plus « standard » en quelque sorte, est la fonction de *perte d'entropie croisée* (en anglais, *cross-entropy loss*). Dans cette section, nous expliquons de quoi il s'agit et comment notre réseau va la calculer. La section suivante présente son utilisation pour l'apprentissage des paramètres.

1.2 Fonctions de perte d'entropie croisée pour les réseaux de neurones

Pour l'instant, notre réseau de la figure 1.6 produit un vecteur de valeurs, une pour chaque entrée linéaire, et nous choisissons la classe ayant la valeur de sortie la plus élevée. Nous modifions maintenant notre réseau afin que les nombres obtenus soient (une estimation de) la distribution de probabilité sur l'ensemble des classes, dans notre cas la probabilité que $C = c$ pour $c \in [0, 1, 2, \dots, 9]$. Une *distribution de probabilité* est un ensemble de valeurs positives ou nulles dont la somme est égale à 1. Pour l'instant, notre réseau fournit des valeurs qui sont d'ordinaire à la fois positives et négatives. Heureusement, il existe une fonction bien pratique pour transformer des ensembles de nombres en distribution de probabilité, *softmax* :

$$\sigma(\mathbf{x})_j = \frac{e^{x_j}}{\sum_i e^{x_i}} \quad (1.5)$$

Softmax renvoie forcément une distribution de probabilité car même si x est négatif, e^x est positif, et la somme des valeurs est égale à 1 car en les ajoutant on obtient la même valeur au numérateur et au dénominateur. Par exemple, $\sigma([-1, 0, 1]) \approx [0.09, 0.244, 0.665]$. Un cas particulier se produit lorsque toutes les sorties du réseau de neurones transmises à softmax valent 0. Mais $e^0 = 1$, donc s'il y a 10 options, elles reçoivent toutes la probabilité $\frac{1}{10}$, soit plus généralement une probabilité de $\frac{1}{n}$ s'il y a n options.

Signalons au passage que le nom « softmax » découle du fait qu'il s'agit d'une version « soft » (douce) de la fonction « max ». La sortie de la fonction max est complètement déterminée par la valeur d'entrée maximale, alors que la sortie de softmax est déterminée principalement mais non pas complètement par ce maximum. De nombreuses fonctions de machine learning ont des noms de la forme softX pour indiquer qu'elles diffèrent de X en ce que leurs sorties sont adoucies.

La figure 1.9 présente un réseau dans lequel on a ajouté une couche softmax. Comme précédemment, les nombres introduits à gauche sont les valeurs des pixels de l'image ; cependant, les nombres produits maintenant à droite sont des probabilités de classes. Il est aussi utile d'avoir une dénomination pour les nombres sortant des unités linéaires pour être introduits dans la fonction softmax. On les appelle en général *logits* — un terme anglais désignant des nombres non normalisés que nous nous apprêtons à transformer en probabilités en utilisant softmax. Nous utilisons \mathbf{l} pour représenter le vecteur des logits (un pour chaque classe). Nous avons donc :

$$p(l_i) = \frac{e^{l_i}}{\sum_j e^{l_j}} \quad (1.6)$$

$$\propto e^{l_i} \quad (1.7)$$

Ici la seconde ligne exprime le fait que, le dénominateur de la fonction softmax étant une constante de normalisation garantissant que la somme des nombres vaudra 1, les probabilités sont proportionnelles au numérateur de softmax.

Nous pouvons maintenant définir notre fonction de perte d'entropie croisée X :

$$X(\Phi, x) = -\ln p_{\Phi}(a_x) \quad (1.8)$$

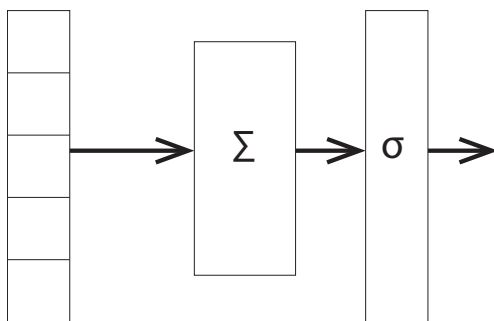


Figure 1.9 – Un réseau simple avec une couche softmax

La perte d'entropie croisée pour un exemple x est l'opposé du log de la probabilité affectée à l'étiquette de x . Pour dire les choses autrement, nous calculons la probabilité de chacune des alternatives en utilisant softmax, puis extrayons celle correspondant à la réponse correcte. La perte est l'opposé du log de cette valeur.

Voyons pourquoi ceci est raisonnable. Tout d'abord, cette valeur évolue dans la bonne direction. Si X est une fonction de *perte*, elle doit croître quand notre modèle donne de moins bons résultats. En pratique, un modèle qui s'améliore va affecter des probabilités sans cesse croissantes à la bonne réponse. C'est pourquoi nous ajoutons un signe moins devant, afin que le nombre obtenu diminue à mesure que la probabilité augmente. Ensuite, le log d'un nombre augmente ou diminue lorsque le nombre augmente ou diminue. Par conséquent, $X(\Phi, x)$ est plus grand pour les mauvais paramètres que pour les bons.

Mais pourquoi avoir introduit une fonction log ? Nous avons l'habitude de considérer que les logarithmes réduisent les distances entre nombres. La différence entre $\log(10\ 000)$ et $\log(1\ 000)$ est de 1. On pourrait s'imaginer que c'est une mauvaise propriété pour une fonction de perte : une telle fonction ferait que les mauvaises situations paraîtraient moins mauvaises. Mais cette caractérisation des logarithmes est trompeuse. Il est vrai que lorsque x augmente, $\ln x$ n'augmente pas aussi rapidement. Mais considérez le graphe de $-\ln(x)$ de la figure 1.10. Lorsque x se rapproche de zéro, la variation du logarithme est bien plus grande que la variation de x . Et comme nous avons affaire à des probabilités, c'est cette région comprise entre 0 et 1 qui nous intéresse.

Et pourquoi cette fonction est-elle appelée *perte d'entropie croisée* ? En théorie de l'information, lorsqu'une distribution de probabilité est censée être proche d'une distribution vraie, l'*entropie croisée* des deux distributions mesure leur degré de dissimilitude. La perte d'entropie croisée est une approximation de l'opposé de l'entropie croisée. Comme nous n'avons pas besoin d'en savoir davantage sur la théorie de l'information dans ce livre, nous en restons à cette explication superficielle.

1.3 Dérivées et descente de gradient stochastique

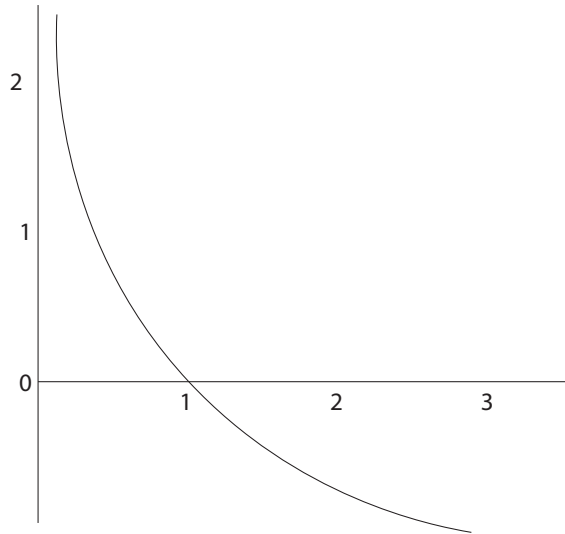


Figure 1.10 - Graphe de $-\ln x$

1.3 DÉRIVÉES ET DESCENTE DE GRADIENT STOCHASTIQUE

Nous avons maintenant notre fonction de perte et pouvons la calculer à l'aide des formules suivantes :

$$X(\Phi, x) = -\ln p(a) \quad (1.9)$$

$$p(a) = \sigma_a(\mathbf{l}) = \frac{e^{l_a}}{\sum_i e^{l_i}} \quad (1.10)$$

$$l_j = b_j + \mathbf{x} \cdot \mathbf{w}_j \quad (1.11)$$

Nous calculons d'abord les logits \mathbf{l} à partir de l'équation 1.11. Ceux-ci sont alors utilisés par la couche softmax pour calculer les probabilités (équation 1.10), après quoi nous calculons la perte, l'opposé du logarithme naturel de la probabilité de la bonne réponse (équation 1.9). Notez que précédemment les poids pour une unité linéaire étaient notés \mathbf{w} . Maintenant nous avons beaucoup de ces unités, donc \mathbf{w}_j correspond aux poids de la j -ième unité et b_j est son biais.

Ce processus permettant de passer de l'entrée à la perte est appelé *passé avant* de l'algorithme d'apprentissage : il calcule les valeurs qui vont être utilisées dans la *passé arrière* — la passe d'ajustement des poids. On utilise pour cela plusieurs méthodes. Ici nous utilisons la *descente de gradient stochastique*. Le terme *descente de gradient* tire son nom du fait qu'on regarde la pente de la fonction de perte (son

Chapitre 1 · Réseaux de neurones à propagation avant

gradient) et qu'on demande au système de minimiser sa perte (descente) en suivant ce gradient. La méthode d'apprentissage tout entière est couramment dénommée *rétropropagation*.

Nous commençons par étudier le cas le plus simple d'estimation de gradient, celui pour l'un des biais b_j . Nous pouvons déduire des équations 1.9-1.11 que b_j modifie la perte en changeant d'abord la valeur du logit l_j , puis en modifiant la probabilité et par conséquent la perte. Prenons ceci étape par étape (ici nous ne prenons en compte que l'erreur introduite par un seul exemple d'entraînement, c'est pourquoi nous écrivons $X(\Phi, x)$ sous la forme $X(\Phi)$). D'abord :

$$\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j} \frac{\partial X(\Phi)}{\partial l_j} \quad (1.12)$$

Ceci utilise la règle de dérivation des fonctions composées (ou règle de dérivation en chaîne), règle que nous avons préalablement exprimée par des mots : les modifications de b_j provoquent des modifications de X en vertu des modifications qu'elles induisent sur le logit l_j .

Observons maintenant la première des dérivées partielles figurant dans le membre de droite de l'équation 1.12. Sa valeur est, en fait, tout simplement 1 :

$$\frac{\partial l_j}{\partial b_j} = \frac{\partial}{\partial b_j} (b_j + \sum_i x_i w_{i,j}) = 1 \quad (1.13)$$

où $w_{i,j}$ est le i -ième poids de la j -ième unité linéaire. Étant donné que la seule chose dans $b_j + \sum_i x_i w_{i,j}$ qui change en fonction de b_j est b_j lui-même, la dérivée vaut 1.

Voyons ensuite comment X évolue en fonction de l_j :

$$\frac{\partial X(\Phi)}{\partial l_j} = \frac{\partial p_a}{\partial l_j} \frac{\partial X(\phi)}{\partial p_a} \quad (1.14)$$

où p_i est la probabilité affectée à la classe i par le réseau. Ceci nous montre que, puisque X ne dépend que de la probabilité de la réponse correcte, l_j n'affecte X que par la modification de cette probabilité. Ensuite :

$$\frac{\partial X(\phi)}{\partial p_a} = \frac{\partial}{\partial p_a} (-\ln p_a) = -\frac{1}{p_a} \quad (1.15)$$

(calcul différentiel basique).

Il nous reste encore un terme à évaluer :

$$\frac{\partial p_a}{\partial l_j} = \frac{\partial \sigma_a(\mathbf{l})}{\partial l_j} = \begin{cases} (1 - p_j)p_a & a = j \\ -p_j p_a & a \neq j \end{cases} \quad (1.16)$$

La première égalité de l'équation 1.16 provient du fait que nous obtenons nos probabilités en appliquant softmax aux logits. La seconde égalité provient de Wikipédia. Cette dérivation requiert une manipulation soignée des termes que nous

1.3 Dérivées et descente de gradient stochastique

ne reproduirons pas ici. Cependant, nous pouvons montrer que ce résultat est raisonnable. Nous nous demandons comment les modifications du logit l_j vont affecter la probabilité produite par softmax. Si nous nous souvenons que

$$\sigma_a(\mathbf{l}) = \frac{e^{l_a}}{\sum_i e^{l_i}}$$

il devient clair qu'il y a deux cas. Supposons que le logit que nous faisons varier (j) n'est pas égal à a . C'est-à-dire, supposons qu'il s'agit de l'image d'un 6, alors que nous recherchons le biais qui détermine le logit 8. Dans ce cas, l_j apparaît au dénominateur, et la dérivée doit être négative (ou nulle) car plus l_j est grand, plus p_a est petit. C'est le second cas dans l'équation 1.16, et ce qu'a produit assurément un nombre inférieur ou égal à zéro, étant donné que les deux probabilités que nous multiplions ne peuvent être négatives.

Par contre, si $j = a$, alors l_j apparaît à la fois au numérateur et au dénominateur. Son apparition au dénominateur tend à faire décroître la sortie, mais dans ce cas, ceci est plus que compensé par l'accroissement du numérateur. Par conséquent, dans ce cas, nous nous attendons à obtenir une dérivée positive (ou nulle), ce qui correspond au premier cas de l'équation 1.16.

Avec ce résultat, nous pouvons maintenant dériver l'équation pour modifier les paramètres de biais b_j . En substituant les équations 1.15 et 1.16 dans l'équation 1.14, nous obtenons :

$$\frac{\partial X(\Phi)}{\partial l_j} = \frac{1}{p_a} \begin{cases} (1 - p_j)p_a & a = j \\ -p_j p_a & a \neq j \end{cases} \quad (1.17)$$

$$= \begin{cases} -(1 - p_j) & a = j \\ p_j & a \neq j \end{cases} \quad (1.18)$$

Le reste est plutôt simple. Dans l'équation 1.12, nous avons remarqué que

$$\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j} \frac{\partial X(\Phi)}{\partial l_j}$$

puis que la première des dérivées figurant à droite avait la valeur 1. Par conséquent, la dérivée par rapport à b_j de la perte est donnée par l'équation 1.14. Enfin, en utilisant la règle de modification des poids (équation 1.12), nous obtenons la règle de mise à jour des paramètres de biais du réseau de neurones :

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) & a = j \\ -p_j & a \neq j \end{cases} \quad (1.19)$$