

Introduction au Deep Learning



Eugene Charniak

DUNOD

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

© Dunod, 2021
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-082578-3

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

Table des matières

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) Karramchetti, 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

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 *discréteriser* 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 sait 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

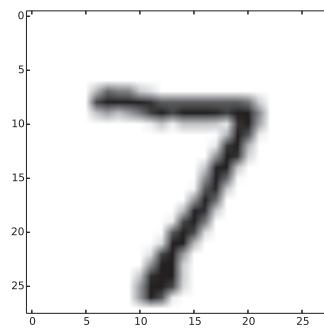


Figure 1.2 – Image en noir et blanc correspondant aux pixels de la figure 1.1

Cependant, ce n'est pas ce que nous allons faire, car dans ce livre nous nous concentrerons 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, $\Phi = \{\mathbf{w} \cup b\}$.

Chapitre 1 · Réseaux de neurones à propagation avant

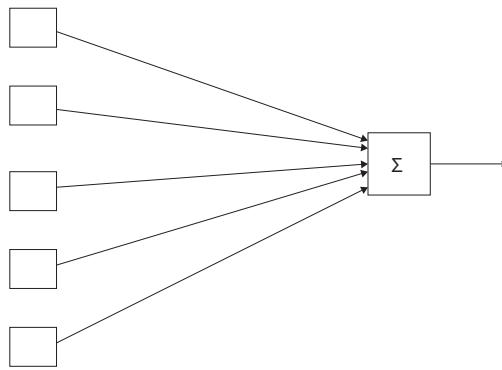


Figure 1.3 – Diagramme schématique d'un perceptron

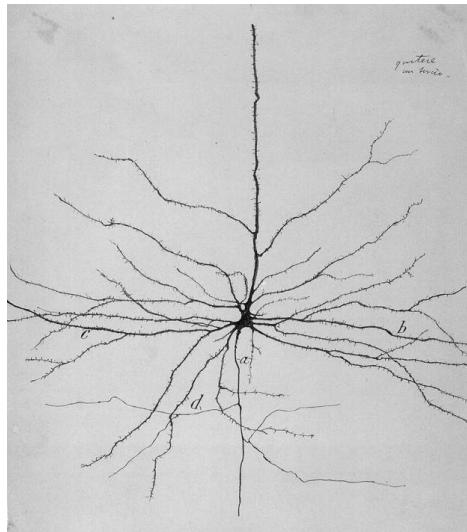


Figure 1.4 – Un neurone ordinaire

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 reconvertir 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.

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 , $\Delta w_i = (a^k - f(\mathbf{x}^k))x_i^k$

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 refaisions 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 classifie 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.

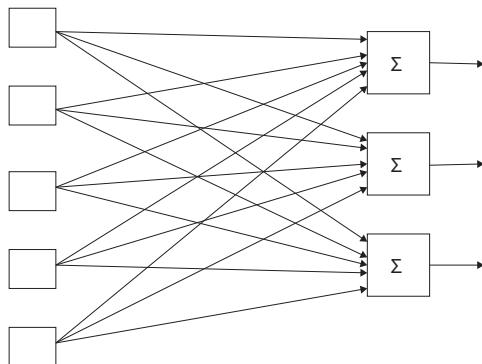


Figure 1.6 – Perceptrons multiples pour l'identification de classes multiples

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

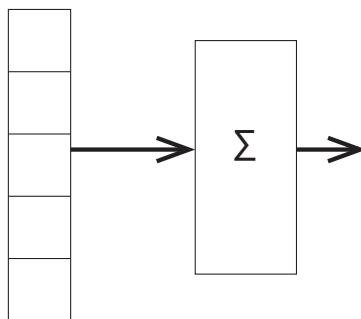


Figure 1.7 – Réseau de neurones à plusieurs couches

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 associe 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 le *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 $-$ précédent \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.

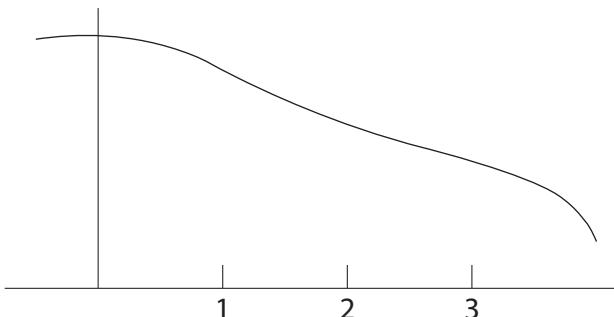


Figure 1.8 – Perte en tant que fonction de ϕ_1

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 :

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

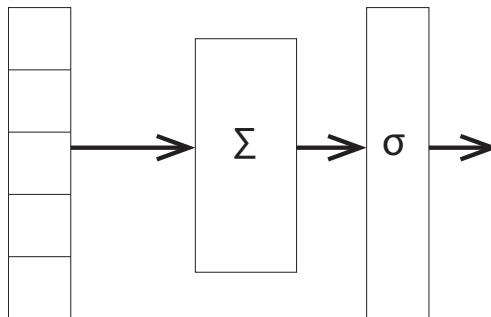


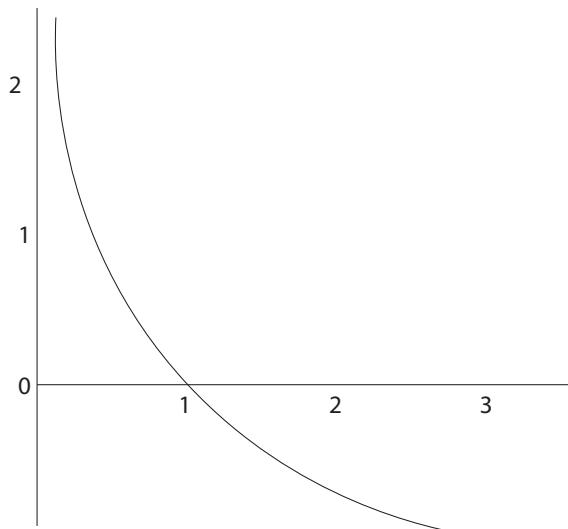
Figure 1.9 – Un réseau simple avec une couche softmax

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 dissimilarité. 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.

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 *passe avant* de l'algorithme d'apprentissage : il calcule les valeurs qui vont être utilisées dans la *passe 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

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 avions 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_a est la probabilité affectée à la classe a 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 soigneuse des termes que nous

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 :

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

L'équation de modification des paramètres de poids (autres que de biais) consiste en une variation mineure de l'équation 1.19. L'équation correspondant à l'équation 1.12 pour les poids est :

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \quad (1.20)$$

Notons tout d'abord que la dérivée la plus à gauche est la même que dans l'équation 1.12. Ceci signifie que nous pouvons sauvegarder le résultat obtenu lors de la modification des biais pour le réutiliser durant la phase d'ajustement des poids. L'évaluation de la première des deux dérivées de droite nous donne :

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}(b_j + (w_{1,j}x_1 + \dots + w_{i,j}x_i + \dots)) = x_i \quad (1.21)$$

Si nous avions pris en considération l'idée qu'un biais est simplement un poids dont la caractéristique correspondante vaut toujours 1, nous aurions pu nous contenter de dériver cette équation, après quoi l'équation 1.13 aurait découlé immédiatement de 1.21 une fois appliquée à ce nouveau pseudo-poids.

En utilisant ce résultat, on obtient notre équation pour la mise à jour des poids :

$$\Delta w_{i,j} = -\mathcal{L}x_i \frac{\partial X(\Phi)}{\partial l_j} \quad (1.22)$$

Nous avons maintenant vu comment les paramètres de notre modèle doivent être ajustés en fonction d'un seul exemple d'entraînement. L'algorithme de *descente de gradient* nous amènerait à prendre en compte successivement tous les exemples d'entraînement, en enregistrant comment chacun d'eux recommanderait de modifier les valeurs des paramètres, mais sans effectuer aucune modification effective tant que tous n'ont pas été examinés. À ce stade, nous modifions chaque paramètre de la somme des modifications associées à chacun des exemples.

Le problème, ici, c'est que cet algorithme peut être très long, en particulier si le jeu d'entraînement est volumineux. En règle générale, nous avons besoin d'ajuster souvent les paramètres, étant donné qu'ils vont interagir de différentes manières lorsque chacun d'eux augmente ou diminue selon le résultat d'exemples de test particuliers. C'est pourquoi, en pratique, nous n'utilisons pratiquement jamais une descente de gradient ordinaire mais plutôt une *descente de gradient stochastique* qui met à jour les paramètres tous les m exemples, m étant très inférieur à la taille du jeu d'entraînement. Une valeur courante de m est 20. C'est ce qu'on appelle la *taille du lot*.

En général, le taux d'apprentissage \mathcal{L} doit être choisi d'autant plus petit que la taille du lot (en anglais, *batch*) est petite. L'idée est que chacun des exemples va modifier les poids de façon à classer cet exemple correctement aux dépens des autres. Si le taux d'apprentissage est bas, cela n'a pas beaucoup d'importance, étant donné que les modifications apportées aux paramètres seront, de manière correspondante, petites. À l'inverse, avec un lot de plus grande taille nous effectuons implicitement une moyenne sur m exemples différents, ce qui diminue le risque de distordre les

paramètres en fonction des particularités d'un exemple et permet donc d'effectuer des modifications plus importantes sur ceux-ci.

1.4 ÉCRITURE DE NOTRE PROGRAMME

Nous avons maintenant préparé le terrain pour notre premier programme d'apprentissage profond. Le pseudo-code est présenté à la figure 1.11. Au début, il faut commencer par initialiser les paramètres du modèle. Parfois, il est acceptable de les initialiser tous à zéro, comme nous l'avons fait dans l'algorithme du perceptron. Même si c'est encore acceptable dans notre problème actuel, ce n'est pas toujours le cas. Par conséquent, la bonne pratique consiste en général à choisir aléatoirement des poids tous proches de zéro. Vous pouvez aussi choisir de fournir vous-même le germe (ou *seed*, en anglais) au générateur de nombres aléatoires de Python, car il sera utile lorsque vous ferez du débogage de pouvoir réinitialiser les paramètres aux mêmes valeurs initiales et par conséquent d'obtenir exactement la même sortie (si vous ne définissez pas le germe, Python utilise pour celui-ci une valeur générée à partir de l'environnement au moment de l'exécution, comme les derniers chiffres de l'horloge par exemple).

1. Pour j de 0 à 9, initialiser b_j aléatoirement (mais proche de 0)
2. Pour j de 0 à 9 et pour i de 0 à 783, initialiser $w_{i,j}$ de la même manière
3. Jusqu'à ce que l'exactitude sur le jeu de développement cesse de s'accroître
 - (a) pour chaque exemple d'entraînement k dans les lots de m exemples
 - i. effectuer la passe avant en utilisant les équations 1.9, 1.10 et 1.11
 - ii. effectuer la passe arrière en utilisant les équations 1.22, 1.19 et 1.14
 - iii. tous les m exemples, modifier tous les Φ avec la somme des mises à jour
 - (b) calculer l'exactitude du modèle en effectuant la passe avant sur tous les exemples du corpus de développement
4. Fournir en résultat le Φ de l'itération *avant* la modification de l'exactitude sur le jeu de développement.

Figure 1.11 – Pseudo-code pour une simple reconnaissance de chiffres à propagation avant

Remarquez qu'à chaque itération de l'entraînement, nous modifions d'abord les paramètres, puis utilisons le modèle sur le jeu de développement pour juger de la qualité des résultats obtenus avec le jeu de paramètres courant. Lors de l'exécution sur des exemples de développement, nous n'exécutons *pas* la passe arrière de l'apprentissage. Si nous avons l'intention d'utiliser notre programme dans une situation réelle (par exemple pour lire des codes postaux sur le courrier), les exemples que

nous voyons ne sont pas ceux sur lesquels nous avons pu effectuer l'entraînement, c'est pourquoi nous voulons savoir si notre programme se comporte bien ailleurs. Nos données de développement constituent une approche de cette situation.

Un certain nombre de connaissances empiriques s'avèrent pratiques ici. Tout d'abord, la pratique courante consiste à avoir des valeurs de pixels ne s'écartant pas trop de -1 à 1 . Dans notre cas, les valeurs originelles des pixels étant comprises entre 0 et 255 , nous nous sommes contentés de les diviser par 255 avant de les utiliser dans notre réseau. Cette étape du processus porte le nom de *normalisation des données*. Il n'existe pas de règles toutes faites, mais souvent il est raisonnable d'avoir des entrées entre -1 et 1 ou entre 0 et 1 . Pour en comprendre la raison, on peut entre autres examiner l'équation 1.22 ci-dessus, où nous avons vu que la différence entre l'équation pour ajuster le terme de biais et celle pour un poids provenant de l'une des entrées du réseau de neurones était que la dernière avait un terme multiplicatif x_i , la valeur du terme d'entrée. Tout à l'heure nous avons dit que si nous avions pris en considération l'idée qu'un biais est simplement un poids dont la caractéristique correspondante vaut toujours 1 , l'équation de mise à jour des paramètres de biais aurait découlé naturellement de l'équation 1.22. Par conséquent, si nous ne modifions pas les valeurs d'entrée et si l'un des pixels a la valeur 255 , nous modifierons son poids 255 fois plus que nous ne modifierons un biais. Étant donné que nous n'avons aucune raison a priori de penser que l'un nécessite plus de correction que l'autre, cela semble étrange.

Vient ensuite la question du choix de \mathcal{L} , le taux d'apprentissage. Cela peut être délicat. Dans notre implémentation, nous avons choisi la valeur 0.0001 . La première chose à remarquer est qu'il est beaucoup plus gênant de le choisir trop grand que de le choisir trop petit. Si vous le choisissez trop grand, la fonction softmax risque de vous renvoyer une erreur de débordement (overflow). Si vous examinez à nouveau l'équation 1.5, une des premières choses qui devrait vous frapper est la présence d'exponentielles tant au numérateur qu'au dénominateur. Élever e (≈ 2.7) à une grande puissance est un moyen infaillible d'obtenir un débordement, ce qui arrivera si l'un quelconque des logits devient grand, ce qui à son tour risque de se produire si nous avons un taux d'apprentissage trop élevé. Même si vous n'obtenez pas de message d'erreur vous indiquant qu'il y a quelque chose qui cloche, avec un taux d'apprentissage trop élevé votre programme peut perdre son temps dans une zone peu profitable de la courbe d'apprentissage.

C'est pourquoi la pratique habituelle consiste à observer comment évolue la perte sur des exemples individuels tandis que le calcul est en cours. Voyons d'abord à quoi nous attendre avec la première image d'entraînement. Les nombres traversent le réseau de neurones et sont transmis en sortie à la couche des logits. Tous nos poids et biais valent 0 plus ou moins un petit quelque chose (mettons 0.1). Ceci signifie que toutes les valeurs des logits sont très proches de 0 , et donc que toutes les probabilités sont très proches de $\frac{1}{10}$ (voir la discussion page 11). La perte est égale à l'opposé du logarithme naturel de la probabilité affectée à la réponse correcte, $-\ln(\frac{1}{10}) \approx 2.3$. En règle générale, nous nous attendons à ce que les pertes individuelles déclinent lorsque nous effectuons l'entraînement sur davantage d'exemples.

Mais naturellement, certaines images s'écartent plus de la norme que d'autres et sont par conséquent classifiées avec moins de certitude par le réseau de neurones. Par conséquent, les pertes individuelles peuvent augmenter ou diminuer, et la tendance peut être plus difficile à discerner. C'est pourquoi, au lieu d'imprimer les pertes une par une, nous les additionnons toutes durant la progression et n'imprimons la moyenne que, par exemple, tous les 100 lots. Cette moyenne devrait décroître de façon aisément observable, même si là encore vous pouvez voir des oscillations.

Pour en revenir à notre discussion sur le taux d'apprentissage et sur le risque qu'il y a à le choisir trop grand, un taux d'apprentissage trop bas peut grandement ralentir la convergence de votre programme vers un bon ensemble de paramètres. C'est pourquoi d'ordinaire le meilleur mode d'action consiste à commencer petit, puis à expérimenter avec des valeurs plus importantes.

Étant donné qu'il y a tant de paramètres qui sont modifiés en même temps, les algorithmes d'apprentissage profond peuvent être difficiles à déboguer. Comme pour toute activité de débogage, l'astuce consiste à modifier aussi peu de paramètres que possible avant que le problème lui-même se manifeste. Tout d'abord, rappelez-vous que lorsque vous modifiez des poids, si vous réexécutez immédiatement le même exemple d'entraînement, la perte sera moindre. Si ce n'est pas le cas, soit il y a un bogue, soit le taux d'apprentissage est trop élevé. Ensuite, sachez qu'il n'est pas nécessaire de modifier tous les poids pour voir la perte décroître. Vous pouvez n'en modifier qu'un, ou seulement certains d'entre eux. Par exemple, lors de la première exécution de l'algorithme, ne changez que les biais. (Cependant, si l'on y réfléchit, un biais dans un réseau à une couche va surtout capturer le fait que les différentes classes apparaissent à des fréquences différentes. Ce n'est pas vraiment le cas dans le réseau Mnist, c'est pourquoi nous n'obtenons pas beaucoup d'amélioration dans ce cas en apprenant uniquement les biais.)

Si notre programme fonctionne correctement, vous devriez obtenir sur les données de développement une exactitude de 91 à 92 %. Ce n'est pas très bon pour cette tâche. Dans les chapitres suivants, nous allons voir comment obtenir aux alentours de 99 %. Mais c'est un début.

Un bon côté des réseaux de neurones vraiment simples, c'est qu'on peut parfois interpréter directement les valeurs des paramètres individuels et décider si elles sont raisonnables ou non. Vous vous souvenez peut-être que, dans notre discussion à propos de la figure 1.1, nous avons remarqué que le pixel (8,8) était sombre — sa valeur était 254. Nous avons remarqué que c'était en quelque sorte une caractéristique du chiffre « 7 », par opposition au chiffre « 1 », pour lequel il n'y a normalement aucune marque dans le coin supérieur gauche. Nous pouvons transformer cette observation en une prédiction concernant les valeurs de notre matrice de poids $w_{i,j}$, où i est le numéro du pixel et j est la valeur de réponse. Si l'on numérote les pixels de 0 à 784, alors la position (8,8) correspondrait au pixel $8 * 28 + 8 = 232$, et le poids le connectant à la réponse 7 (la bonne réponse) serait $w_{232,7}$, alors que celui le connectant à la réponse 1 serait $w_{232,1}$. Comme vous le voyez, cela suggère que $w_{232,7}$ doit être plus grand que $w_{232,1}$. Nous avons exécuté notre programme plusieurs fois,

avec une initialisation aléatoire de faible variance de nos poids. Dans chaque cas, la première des valeurs était positive (p. ex. 0.25) tandis que la seconde était négative (p. ex. -0.17).

1.5 REPRÉSENTATION MATRICIELLE DES RÉSEAUX DE NEURONES

L'algèbre linéaire nous fournit un autre moyen de représenter ce qui se passe dans un réseau de neurones : en utilisant des matrices. Une *matrice* est un tableau à deux dimensions composé d'éléments. Dans notre cas, ces éléments sont des nombres réels. Les dimensions d'une matrice sont respectivement le nombre de ses lignes et le nombre de ses colonnes. Par conséquent, une matrice l par m ressemble à ceci :

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ \dots \\ x_{l,1} & x_{l,2} & \dots & x_{l,m} \end{pmatrix} \quad (1.23)$$

Les principales opérations sur les matrices sont l'addition et la multiplication. L'addition de deux matrices (qui doivent avoir les mêmes dimensions) s'effectue élément par élément. C'est-à-dire que si nous ajoutons deux matrices $\mathbf{X} = \mathbf{Y} + \mathbf{Z}$, alors $x_{i,j} = y_{i,j} + z_{i,j}$.

Le produit de deux matrices $\mathbf{X} = \mathbf{YZ}$ peut être effectué si \mathbf{Y} a pour dimensions l et m et si celles de \mathbf{Z} sont m et n . Le résultat est une matrice l par n , telle que :

$$x_{i,j} = \sum_{k=1}^{k=m} y_{i,k} z_{k,j} \quad (1.24)$$

À titre d'exemple :

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 9 & 12 & 15 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 16 & 20 & 24 \end{pmatrix}$$

Nous pouvons utiliser une telle combinaison de multiplications et d'additions de matrices pour définir le fonctionnement de nos unités linéaires. En particulier, les caractéristiques d'entrée constituent une matrice $1 * l \mathbf{X}$. Dans le cas de la reconnaissance des chiffres, $l = 784$. Les poids sur les unités sont représentés par \mathbf{W} où $w_{i,j}$ est le i -ième poids pour l'unité j . Par conséquent, les dimensions de \mathbf{W} correspondent au nombre de pixels et au nombre de chiffres, soit $784 * 10$. \mathbf{B} est un vecteur de biais de longueur 10, et

$$\mathbf{L} = \mathbf{XW} + \mathbf{B} \quad (1.25)$$

où \mathbf{L} est un vecteur de logits de longueur 10. Lorsqu'on rencontre une équation de ce type, il est de bonne pratique de s'assurer d'abord que les dimensions correspondent.

Nous pouvons maintenant exprimer comme suit la perte (L) pour notre modèle Mnist à propagation avant :

$$\text{Pr}(A(x)) = \sigma(\mathbf{x}\mathbf{W} + \mathbf{b}) \quad (1.26)$$

$$L(x) = -\log(\text{Pr}(A(x) = a)) \quad (1.27)$$

où la première équation donne la distribution de probabilité sur les classes possibles ($A(x)$) et où la seconde fournit la perte d'entropie croisée.

Nous pouvons aussi exprimer de manière plus compacte la passe arrière. Tout d'abord, nous définissons l'*opérateur gradient* :

$$\nabla_{\mathbf{l}} X(\Phi) = \left(\frac{\partial X(\Phi)}{\partial l_1} \dots \frac{\partial X(\Phi)}{\partial l_m} \right) \quad (1.28)$$

Le triangle inversé, $\nabla_{\mathbf{x}} f(\mathbf{x})$, identifie le vecteur obtenu en prenant la dérivée partielle de f par rapport à chacune des composantes de \mathbf{x} . Auparavant, nous parlions simplement des dérivées partielles par rapport à chacun des l_j . Ici nous définissons la dérivée par rapport à \mathbf{l} tout entier comme le vecteur des dérivées individuelles. Rapelons aussi l'opération de transposition d'une matrice — consistant à transformer ses lignes en colonnes, et inversement :

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ \dots & & & \\ x_{l,1} & x_{l,2} & \dots & x_{l,m} \end{pmatrix}^T = \begin{pmatrix} x_{1,1} & x_{2,1} & \dots & x_{l,1} \\ x_{1,2} & x_{2,2} & \dots & x_{l,2} \\ \dots & & & \\ x_{1,m} & x_{2,m} & \dots & x_{l,m} \end{pmatrix} \quad (1.29)$$

Grâce à cela, nous pouvons réécrire l'équation 1.22 sous la forme suivante :

$$\Delta \mathbf{W} = -\mathcal{L} \mathbf{X}^T \nabla_{\mathbf{l}} X(\Phi) \quad (1.30)$$

À droite, nous multiplions une matrice $784 * 1$ par une matrice $1 * 10$ pour obtenir la matrice $784 * 10$ des modifications apportées à la matrice $784 * 10$ des poids \mathbf{W} .

Ceci est une synthèse élégante de ce qui se passe lorsque la couche d'entrée alimente la couche des unités linéaires afin de produire les logits, ce qui se poursuit par la propagation arrière des dérivées de la perte afin de modifier les paramètres. Mais il existe aussi une raison pratique qui fait préférer cette nouvelle notation. Avec un très grand nombre d'unités linéaires, le calcul matriciel en général et l'entraînement par apprentissage profond en particulier peuvent prendre beaucoup de temps. Cependant, de très nombreux problèmes peuvent être exprimés en notation matricielle, et beaucoup de langages de programmation disposent de packages spéciaux vous permettant de programmer simplement certaines formules d'algèbre linéaire. De plus, ces packages sont optimisés, ce qui rend ces algorithmes plus efficaces que si vous

les aviez codés à la main. En particulier, si vous programmez en Python, il est grandement recommandé d'utiliser le package *Numpy* et ses opérations matricielles. En général, vous améliorerez vos performances de manière substantielle.

De plus, l'algèbre linéaire trouve tout particulièrement son application dans le domaine de l'informatique graphique et des programmes de jeux. Ceci a conduit à développer des matériaux spécifiques appelés *processeurs graphiques* (en anglais, *graphics processing units* ou GPU). Les GPU sont des processeurs lents par rapport aux CPU, mais ils sont en grand nombre, et disposent du logiciel permettant de les utiliser efficacement en parallèle pour les calculs d'algèbre linéaire. Certains langages conçus spécialement pour les réseaux de neurones (p. ex. *TensorFlow*) intègrent du logiciel détectant la disponibilité de GPU et les utilisant sans aucune modification de code. Ceci amène également une amélioration de performances substantielle.

Il existe encore une troisième raison poussant à adopter une notation matricielle dans ce cas. Les packages logiciels spécialisés (p. ex. *Numpy*) ainsi que les composants matériels (GPU) sont plus efficaces s'ils traitent plusieurs exemples d'entraînement en parallèle. De plus, ceci s'accorde avec notre idée de traiter m exemples d'entraînement (c'est-à-dire tout un lot) avant de mettre à jour les paramètres du modèle. Pour ce faire, on présente communément les m exemples simultanément à notre traitement matriciel qui les exécute ensemble. Dans l'équation 1.25 nous avons présenté l'image \mathbf{x} sous forme d'une matrice $1 * 784$. Il s'agissait là d'un exemple d'entraînement, comportant 784 pixels. Nous modifions maintenant ceci pour que la matrice ait pour dimensions m par 784. Il est intéressant de constater que ceci fonctionne presque sans modification de notre traitement (et les modifications nécessaires sont déjà intégrées, entre autres, dans *Numpy* et *TensorFlow*). Voyons pourquoi.

Considérons d'abord le produit matriciel \mathbf{XW} où \mathbf{X} possède maintenant m lignes plutôt qu'une seule. Bien sûr, avec une seule ligne nous obtenons une sortie de taille $1 * 784$. Avec m lignes la sortie est de taille $m * 784$. De plus, selon les principes de l'algèbre linéaire que vous pouvez d'ailleurs confirmer en consultant la définition du produit matriciel, chacune des lignes de la matrice obtenue en sortie correspond à la multiplication d'une des lignes de la matrice d'entrée, et il suffit de superposer ces lignes pour obtenir la matrice $m * 784$ résultante.

Ajouter le terme de biais dans l'équation ne fonctionne pas si bien. Nous avons vu que pour ajouter deux matrices, elles devaient avoir les mêmes dimensions. Ce n'est plus vrai pour l'équation 1.25, car \mathbf{XW} est désormais de dimension m par 10 alors que \mathbf{B} , qui regroupe les termes de biais, a pour dimensions 1 par 10. C'est là qu'il faut apporter quelques petites modifications.

Numpy et *TensorFlow* offrent une possibilité d'ajustement appelée *broadcasting*. Lorsque certaines opérations algébriques nécessitent que les tableaux aient des tailles différentes de celles qu'ils ont, leurs dimensions peuvent parfois être ajustées. En particulier, lorsqu'un tableau a pour dimensions $1 * n$ et que nous avons besoin de $m * n$, il est complété par $m - 1$ copies (virtuelles) de son unique ligne ou colonne, de façon à ce qu'il ait la bonne taille. C'est exactement ce que nous voulons ici. Ceci

permet de transformer \mathbf{B} en une matrice $m * 10$. De cette façon, nous ajoutons le biais à tous les termes de la matrice $m * 10$ obtenue après multiplication. Souvenez-vous de ce que nous avons fait de notre résultat de taille $1 * 10$. Chacune des 10 valeurs correspondait à une décision possible en ce qui concerne la bonne réponse, et nous avons ajouté le biais au nombre représentant cette décision. Maintenant nous faisons de même, mais cette fois pour chaque décision possible et pour chacun des m exemples que nous traitons en parallèle.

1.6 INDÉPENDANCE DES DONNÉES

Tous les théorèmes visant à prouver que, moyennant certaines conditions, nos modèles de réseaux de neurones convergent vers la bonne solution presupposent la *condition iid* — à savoir que nos données sont indépendantes et identiquement distribuées. Un exemple classique est celui de la mesure des rayons cosmiques — le flux des rayons entrants et les processus concernés sont aléatoires et inchangés.

Nos données ressemblent rarement (et même presque jamais) à ceci — imaginez que le jeu de données Mnist soit sans arrêt complété par de nouveaux exemples. Pour les données de notre première passe (ou époque), l'hypothèse iid paraît plutôt bonne, mais dès que nous attaquons la seconde, nos données sont identiques à celles de la première. Dans certains cas, nous données peuvent cesser d'être iid dès le deuxième exemple d'entraînement. C'est souvent le cas en apprentissage par renforcement profond (voir chapitre 6), c'est pourquoi les réseaux de ce type souffrent souvent d'*instabilité* — incapacité du réseau à converger vers la bonne solution, voire parfois même à converger vers une *quelconque* solution. Nous allons examiner ici un exemple d'assez petite taille dans lequel le simple fait de ne pas introduire les données dans un ordre aléatoire peut avoir des conséquences désastreuses.

Supposons que, pour chaque image Mnist, nous ayons ajouté une seconde image, identique à ceci près que le noir et le blanc sont inversés (c'est-à-dire que si le pixel de l'image originale a la valeur v , celui de l'image inverse a la valeur $-v$). Nous entraînons maintenant notre perceptron Mnist sur ce nouveau corpus, mais en modifiant l'ordre des exemples d'entraînement. Nous choisissons pour taille de lot un entier pair. Dans notre premier ordonnancement, chaque image numérisée du jeu Mnist est suivie immédiatement par sa version inversée. Nous pouvons affirmer (et déjà vérifier empiriquement) que notre simple réseau de neurones Mnist n'arrive pas à faire mieux que de répondre au hasard. Il suffit d'y réfléchir quelques instants pour que ceci paraisse raisonnable. Nous examinons la première image, et la passe arrière modifie les poids. Nous traitons alors la deuxième image, qui est inversée. Étant donné que l'entrée est l'opposé de la précédente, toutes choses étant égales par ailleurs, les modifications de chacun des poids annulent exactement les précédentes, ce qui fait que, à la fin du jeu d'entraînement, tous les poids sont restés inchangés. Il n'y a pas eu d'apprentissage, et les choix restent toujours aussi aléatoires qu'au début.

Pourtant, il ne devrait pas être difficile du tout d'apprendre à gérer séparément chaque jeu de données, le normal et l'inversé, et il ne serait que modérément difficile pour un seul jeu de pondérations de gérer les deux. De fait, il suffit simplement de modifier aléatoirement l'ordre des entrées pour ramener les performances pratiquement au niveau du problème originel. Si l'image inversée apparaît 10 000 exemples plus tard, les poids ont été suffisamment modifiés entre-temps pour que l'image inverse ne corrige pas exactement l'apprentissage d'origine. Si nous avions une source sans fin d'images et si nous choisissions à pile ou face de fournir au réseau de neurones l'image réelle ou son inverse, même cette petite annulation finirait par disparaître.

1.7 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

Dans cette section « Références et lectures supplémentaires » et dans toutes celles qui suivent, je poursuis plus ou moins simultanément plusieurs objectifs : (a) orienter l'étudiant vers des documents complémentaires sur le sujet développé dans le chapitre, (b) identifier quelques contributions importantes dans ce domaine et (c) citer des références que j'ai utilisées pour mon propre apprentissage. Dans les différents cas, et particulièrement en (b), je ne prétends être ni complet ni exhaustif. J'ai réalisé cela lorsque, me préparant à écrire cette section, j'ai étudié plus en profondeur l'histoire des réseaux neuronaux. J'ai lu en particulier un article de blog posté par Andrey Kurenkov [Kur15], dans l'idée de confirmer mes souvenirs (et peut-être de les compléter).

L'une des premières publications essentielles en matière de réseaux de neurones a été celle de McCulloch et Pitts [MP43], qui ont proposé ce que nous appelons ici une unité linéaire en tant que modèle formel d'un neurone. Ceci remonte à 1943. Ils n'avaient pas, cependant, d'algorithme d'apprentissage permettant d'en entraîner un ou plusieurs à effectuer une tâche. Ceci a été la grande contribution de Rosenblatt dans son article de 1958 à propos des perceptrons [Ros58]. Cependant, comme nous l'avons fait remarquer, son algorithme ne fonctionnait que pour un réseau de neurones à une seule couche.

La grande étape suivante a été l'invention de la rétropropagation qui, *elle*, fonctionne avec des réseaux de neurones multi-couches. Il s'agissait de l'une de ces situations où de nombreux chercheurs ont tous indépendamment la même idée sur une période de quelques années (ceci ne se produit, bien sûr, que lorsque les publications initiales n'ont pas suffisamment attiré l'attention pour que tout le monde réalise que le problème est désormais résolu). La publication marquant la fin de cette période a été celle de Rumelhart, Hinton et Williams, où ils mentionnent explicitement que leur travail n'est qu'une redécouverte [RHW86]. Cet article faisait partie des nombreuses publications d'un groupe de l'Université de San Diego à qui l'on doit la seconde éclosion des réseaux neuronaux sous la rubrique du *traitement parallèle distribué* (en anglais, *parallel distributed processing* ou PDP). Les deux volumes regroupant ces publications ont eu un net succès [RMG⁺87].

Quant à la façon dont j'ai forgé mon expérience en matière de réseaux de neurones, j'en parlerai plus spécifiquement dans les chapitres suivants. Pour ce qui concerne ce chapitre, je me souviens d'avoir lu très tôt un blog de Steven Miller [Mil15] qui explore très lentement les passes avant et arrière de la propagation arrière en s'appuyant sur un excellent exemple numérique. Plus généralement, laissez-moi mentionner deux ouvrages généraux sur l'apprentissage profond que j'ai consultés. L'un est intitulé *Deep Learning*, de Ian Goodfellow, Yoshua Bengio et Aaron Courville [GBC16] ; l'autre est un ouvrage d'Aurélien Géron [Gér17] intitulé *Hands-On Machine Learning with Scikit-Learn and TensorFlow*¹.

Exercices

1.1 Considérons notre programme Mnist à propagation avant avec une taille de lot égale à 1. Supposons que nous observions les variables de biais avant et après l'entraînement sur le premier exemple. Si elles ont été affectées correctement (c'est-à-dire s'il n'y a pas d'erreur dans notre programme), décrivez les modifications que vous devriez observer dans leurs valeurs.

1.2 Nous simplifions notre calcul Mnist en supposant que notre « image » a deux pixels à valeur binaire, 0 et 1, qu'il n'y a pas de paramètres de biais, et que nous effectuons une classification binaire. **(a)** Calculez les logits de la passe avant et les probabilités lorsque les valeurs des pixels sont [0,1] et que les poids sont :

$$\begin{matrix} 0.2 & -0.3 \\ -0.1 & 0.4 \end{matrix}$$

Ici $w[i, j]$ est le poids affecté à la connexion entre le i -ième pixel et la j -ième unité. Par exemple, $w[0, 1]$ vaut ici -0.3 . **(b)** Supposez que la réponse correcte est 1 (et non 0) et utilisez un taux d'apprentissage de 0.1. Quelle est la perte ? Calculez aussi $\Delta w_{0,0}$ sur la passe arrière.

1.3 Mêmes questions que dans l'exercice 1.2 excepté que l'image est [0,0].

1.4 Un autre étudiant vous demande : « En algèbre, nous pouvons trouver le minimum d'une fonction en la dérivant, puis en recherchant pour quelle valeur cette

1. NDT : Dunod a fait paraître la version française de la première édition, en deux tomes intitulés *Machine Learning avec Scikit-Learn* et *Deep Learning avec TensorFlow*, de même que celle de la deuxième édition, datant de 2019 et intitulée *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*.

Chapitre 1 · Réseaux de neurones à propagation avant

dérivée s'annule. Puisque notre fonction de perte est différentiable, pourquoi ne pas faire cela plutôt que de nous compliquer la vie avec une descente de gradient ? » Expliquez pourquoi ce n'est pas possible en pratique.

1.5 Calculez ce qui suit :

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} 4 & 5 \end{pmatrix} \quad (1.31)$$

Ajustez par broadcasting afin que le calcul soit bien défini.

1.6 Dans ce chapitre, nous nous sommes limités à des problèmes de classification, dans lesquels l'entropie croisée est en règle générale la fonction de perte de prédilection. Il existe aussi des problèmes où nous voulons que notre réseau de neurones prédise des valeurs particulières. Par exemple, de nombreuses personnes apprécieraient sans aucun doute un programme qui, étant donné le cours d'une action un jour donné ainsi que toutes sortes d'autres faits mondiaux, fournirait en sortie le cours de cette action le lendemain. Si nous entraînions un réseau de neurones monocouche pour cela, nous utiliserions en général la *perte de l'erreur quadratique* (encore appelée perte quadratique) :

$$L(\mathbf{X}, \Phi) = (t - l(\mathbf{X}, \Phi))^2 \quad (1.32)$$

où t est le cours atteint ce jour et $l(\mathbf{X}, \Phi)$ est la sortie du réseau de neurones à une couche avec $\Phi = \{\mathbf{b}, \mathbf{W}\}$. Dérivez cette fonction pour obtenir la dérivée de la perte par rapport à b_i .

TENSORFLOW

2

2.1 INTRODUCTION À TENSORFLOW

TensorFlow est un langage de programmation open source développé par Google qui a été spécialement conçu pour permettre d'écrire aisément des programmes d'apprentissage profond, ou au moins de simplifier cette programmation. Nous commencerons par le premier programme traditionnel :

```
import tensorflow as tf
x = tf.constant("Hello World")
sess = tf.Session() print(sess.run(x)) # Imprime "Hello World"
```

Si cela ressemble à du code Python, c'est parce que c'en est effectivement. De fait, TensorFlow (désormais TF) est une collection de fonctions pouvant être appelées à partir de différents langages de programmation. L'interface la plus complète est celle depuis Python, et c'est ce que nous utilisons ici.

Il faut ensuite remarquer que les fonctions TF, plutôt que d'exécuter un programme, définissent un calcul qui n'est exécuté que lorsque l'on appelle la commande `run`, comme c'est le cas à la dernière ligne du programme ci-dessus. Plus précisément, la fonction `TF Session` à la troisième ligne crée une session à laquelle est associé un graphe définissant le calcul. Les commandes telles que `constant` ajoutent des éléments à ce calcul. Dans ce cas, l'élément est simplement une donnée constante dont la valeur est la chaîne Python `Hello World`. La troisième ligne dit à TF d'évaluer la variable TF sur laquelle pointe `x` à l'intérieur du graphe associé à la session `sess`. Comme vous pouvez vous y attendre, le résultat est l'impression de `Hello World`.

Il est instructif de comparer ce comportement avec ce qui se passe si nous remplaçons la dernière ligne par `print(x)`. Ceci imprime :

```
Tensor("Const:0", shape=(), dtype=string)
```

ceci parce que la variable Python `x` n'est pas attachée à une chaîne, mais plutôt à un morceau du graphe de calcul de TensorFlow. Ce n'est que lorsque nous évaluons cette portion du graphe en exécutant `sess.run(x)` que nous accédons à la valeur de la constante TF.

Sans vouloir trop insister sur ce qui est une évidence pour certains, dans le code ci-dessus `x` et `sess` sont des variables Python et en tant que telles auraient pu être nommées à notre convenance. `import` et `print` sont des fonctions Python et doivent figurer sous ce nom exact pour que Python comprenne quelle fonction il doit exécuter. Enfin, `constant`, `Session` et `run` sont des commandes TensorFlow dont le nom doit être orthographié correctement (y compris le S majuscule de `Session`). Par ailleurs, il nous faut toujours commencer par `import tensorflow`. Ceci étant précisé, nous l'omettrons désormais dans nos exemples.

Chapitre 2 · TensorFlow

```
x = tf.constant(2.0)
z = tf.placeholder(tf.float32)
sess= tf.Session()
comp=tf.add(x,z)
print(sess.run(comp,feed_dict={z:3.0})) # Imprime 5.0
print(sess.run(comp,feed_dict={z:16.0})) # Imprime 18.0
print(sess.run(x)) # Imprime 2.0
print(sess.run(comp)) # Imprime un long message d'erreur
```

Figure 2.1 – Nœuds de substitution dans TF

Dans le code de la figure 2.1, `x` est à nouveau une variable Python dont la valeur est une constante TF, ici le nombre décimal 2.0. Ensuite, `z` est une variable Python dont la valeur est un *nœud de substitution* (ou *placeholder*, en anglais) de TensorFlow. Un nœud de substitution TF est analogue à une variable formelle dans une fonction d'un langage de programmation. Supposons que nous ayons le code Python suivant :

```
x = 2.0
def sillyAdd(z):
    return z+x
print(sillyAdd(3)) # Imprime 5.0
print(sillyAdd(16)) # Imprime 18.0
```

Ici, `z` est le nom de l'argument de `sillyAdd` et lorsque nous appelons la fonction comme dans `sillyAdd(3)`, il est remplacé par sa valeur, 3. La version TF fonctionne de manière similaire, sauf que le moyen d'affecter une valeur à un nœud de substitution TF diffère, comme le montre la cinquième ligne de la figure 2.1 :

```
print(sess.run(comp,feed_dict={z:3.0}))
```

Ici `feed_dict` est un argument nommé de `run` (donc son nom doit être orthographié correctement). Les valeurs possibles pour un tel argument sont des dictionnaires Python. Dans le dictionnaire, chaque nœud de substitution nécessaire au calcul doit avoir une valeur. Par conséquent, le premier `sess.run` imprime la somme de 2.0 et 3.0, et le deuxième 18.0. Le troisième est ici pour montrer que si le calcul ne requiert pas de valeur de substitution, alors il est inutile d'en fournir. Inversement, comme l'indique le commentaire de la quatrième commande d'impression, si le calcul requiert une valeur et qu'elle n'est pas fournie, vous obtenez une erreur.

TensorFlow est appelé **TensorFlow** car ses structures de données fondamentales sont des *tenseurs*, c'est-à-dire des tableaux multidimensionnels typés. Il y a à peu près 15 types de tenseurs. Lorsque nous avons défini le nœud de substitution `z` ci-dessus, nous lui avons donné le type `float32`. En plus de son type, le tenseur possède une *forme* (en anglais, *shape*). Considérez donc une matrice 2×3 . Elle est de forme [2, 3]. Un vecteur de longueur 4 a pour forme [4]. Il diffère d'une matrice 1×4 qui a pour

forme [1, 4], ou d'une matrice $4 * 1$ qui a pour forme [4, 1]. Un tableau 3 par 17 par 6 a pour forme [3, 17, 6]. Ce sont tous des tenseurs. Les scalaires (c'est-à-dire les nombres) ont une forme nulle, mais sont également des tenseurs. Sachez également que les tenseurs ne connaissent pas la distinction propre à l'algèbre linéaire entre vecteur ligne et vecteur colonne. Il existe des tenseurs comportant un seul élément, par ex. [5], et c'est tout. La façon dont nous les représentons sur la page importe peu pour les mathématiques. Lorsque nous fournissons une représentation d'un tenseur à deux dimensions, nous respectons toujours la règle qui consiste à placer verticalement la dimension 0, et horizontalement la dimension 1. Mais notre cohérence s'arrête là. Notez également que les composantes d'un tenseur sont numérotées à partir de zéro.

Pour en revenir à notre discussion sur les nœuds de substitution (ou placeholders), la plupart d'entre eux ne sont pas de simples scalaires comme dans nos exemples précédents, mais plutôt des tenseurs multidimensionnels. Par exemple, la section suivante commence avec un programme TensorFlow simple pour la reconnaissance des chiffres Mnist. Le code TF primaire prendra une image et exécutera la passe avant du réseau de neurones pour obtenir la supposition du réseau quant au chiffre examiné. Durant la phase d'entraînement, il exécutera également la passe arrière et modifiera les paramètres du programme. Pour transmettre l'image au programme, nous définissons un nœud de substitution. Il sera de type `float32` et de forme [28,28], ou éventuellement [784], selon que nous transmettons une liste Python à une ou deux dimensions. Par exemple :

```
img=tf.placeholder(tf.float32, shape=[28, 28])
```

Remarquez que `shape` (signifiant forme) est un argument nommé de la fonction `placeholder`.

Voyons encore une structure de données de TF avant de plonger dans un programme réel. Comme mentionné précédemment, les modèles de réseaux de neurones sont définis par leurs paramètres et par l'architecture du programme, à savoir comment les paramètres se combinent aux valeurs d'entrée pour produire la réponse. Les paramètres (par ex. les poids `w` qui connectent l'image d'entrée aux logits de réponse) sont (habituellement) initialisés aléatoirement, et le réseau de neurones les modifie pour minimiser la perte sur les données d'entraînement. Il y a trois étapes dans la création des paramètres TF. Premièrement, créer un tenseur avec les valeurs initiales. Puis transformer le tenseur en une `Variable` (qui est ce que TF appelle paramètres) et initialiser les variables/paramètres. Créons par exemple les paramètres dont nous avons besoin pour le pseudo-code Mnist de propagation avant figurant à la figure 1.11. Tout d'abord les termes de biais `b`, puis les poids `w` :

```
bt = tf.random_normal([10], stddev=0.1)
b = tf.Variable(bt)
W = tf.Variable(tf.random_normal([784,10],stddev=0.1))
sess=tf.Session()
sess.run(tf.global_variables_initializer)
print(sess.run(b))
```

La première ligne ajoute une instruction pour créer un tenseur de forme [10] dont les 10 valeurs sont des nombres aléatoires générés à partir d'une *distribution normale* d'écart type 0.1. (Une distribution normale, appelée aussi *distribution gaussienne*, présente le profil de courbe en cloche bien connu. Les valeurs choisies à partir d'une distribution normale seront centrées autour de la moyenne (μ), et leur écart par rapport à cette moyenne est régi par l'*écart type* (σ). Plus précisément, environ 68 % des valeurs choisies se trouveront à moins d'un écart type de la moyenne, et la probabilité de s'en écarter davantage diminuera rapidement.)

La deuxième ligne du code ci-dessus prend `bt` et ajoute un élément du graphe TF qui crée une variable de mêmes forme et valeurs. Étant donné que nous avons rarement besoin du tenseur d'origine une fois que nous avons créé la variable, nous combinons normalement les deux événements sans sauvegarder de pointeur vers le tenseur, comme dans la troisième ligne qui crée les paramètres `w`. Avant que nous puissions utiliser `b` ou `w`, nous devons les initialiser dans la session que nous avons créée. C'est ce qui est fait à la cinquième ligne. La sixième ligne imprime (lorsque je l'ai exécutée, car le résultat sera différent à chaque fois) :

```
[-0.05206999 0.08943175 -0.09178174 -0.13757218 0.15039739  
 0.05112269 -0.02723283 -0.02022207 0.12535755 -0.12932496]
```

Si nous avions inversé l'ordre des deux dernières lignes, nous aurions reçu un message d'erreur lorsque nous aurions tenté d'évaluer la variable pointée par `b` dans la commande d'impression.

Donc, dans les programmes TF, nous créons des variables dans lesquelles nous rangeons les paramètres du modèle. Initialement, leurs valeurs ne sont pas informatives (proches de zéro), en général choisies aléatoirement avec un faible écart type. Comme nous l'avons dit précédemment, la passe arrière de la descente de gradient les modifie. Une fois modifiées, la session pointée par `sess` conserve les nouvelles valeurs, puis les utilise la prochaine fois que nous exécutons la session.

2.2 UN PROGRAMME TF

À la figure 2.2 nous fournissons un programme TF (presque) complet pour un programme Mnist de réseau de neurones à propagation avant. Il devrait fonctionner tel qu'il est écrit. L'élément clé que vous ne voyez pas ici est le code `mnist.train.next_batch`, qui gère les détails de la lecture des données Mnist. Pour vous repérer, notez que tout ce qui se trouve avant la ligne pointillée concerne la définition du graphe de calcul de TF ; tout ce qui suit utilise le graphe tout d'abord pour entraîner les paramètres, puis pour exécuter le programme sur les données de test afin d'en vérifier l'exactitude. Nous allons maintenant examiner le code ligne par ligne.

```

0 import tensorflow as tf
1 from tensorflow.examples.tutorials.mnist import input_data
2 mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
3
4 batchSz=100
5 W = tf.Variable(tf.random_normal([784, 10],stddev=.1))
6 b = tf.Variable(tf.random_normal([10],stddev=.1))
7
8 img=tf.placeholder(tf.float32, [batchSz,784])
9 ans = tf.placeholder(tf.float32, [batchSz, 10])
10
11 prbs = tf.nn.softmax(tf.matmul(img, W) + b)
12 xEnt = tf.reduce_mean(-tf.reduce_sum(ans * tf.log(prbs),
13                                     reduction_indices=[1]))
14 train = tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
15 numCorrect= tf.equal(tf.argmax(prbs,1), tf.argmax(ans,1))
16 accuracy = tf.reduce_mean(tf.cast(numCorrect, tf.float32))
17
18 sess = tf.Session()
19 sess.run(tf.global_variables_initializer())
20 #-----
21 for i in range(1000):
22     imgs, anss = mnist.train.next_batch(batchSz)
23     sess.run(train, feed_dict={img: imgs, ans: anss})
24
25 sumAcc=0
26 for i in range(1000):
27     imgs, anss= mnist.test.next_batch(batchSz)
28     sumAcc+=sess.run(accuracy, feed_dict={img: imgs, ans: anss})
29 print "Exactitude du test : %r" % (sumAcc/1000)

```

Figure 2.2 – Code TensorFlow pour un réseau de neurones Mnist à propagation avant

Après avoir importé TensorFlow et le code permettant de lire les données Mnist, nous définissons nos deux ensembles de paramètres aux lignes 5 et 6. C'est une variante mineure de ce que nous venons juste de voir à propos des variables TF. Ensuite, nous créons des nœuds de substitution pour les données d'entrée du réseau de neurones. D'abord, à la ligne 8 nous avons un nœud de substitution pour les données de l'image. C'est un tenseur de forme [batchSz, 784] où batchSz est la taille du lot. Lorsque nous avons expliqué pourquoi l'algèbre linéaire était une bonne façon de représenter les calculs des réseaux de neurones (page 22), nous avons souligné que le temps de calcul s'améliorait lorsque nous traitions plusieurs exemples à la fois, et que de plus ceux-ci s'accordent bien avec la notion de taille de lot de la descente de gradient stochastique. Ici, nous voyons comment cela se passe dans TensorFlow, à savoir que le nœud de substitution pour l'image ne se contente pas de prendre une

ligne de 784 pixels, mais 100 d'entre elles (puisque c'est la valeur de `batchSz`, la taille du lot). De même, à la ligne 9 nous voyons que nous fournissons au programme 100 réponses (ou étiquettes) d'images à la fois.

Encore une remarque à propos de la ligne 9. Nous représentons une réponse par un vecteur de longueur 10 avec toutes les valeurs égales à zéro sauf la a -ième, où a est le chiffre correct pour cette image. Par exemple, nous avons débuté le chapitre 1 avec une image de « 7 » (figure 1.1). La représentation correspondant à la réponse correcte est $(0, 0, 0, 0, 0, 0, 0, 1, 0, 0)$. Un vecteur de cette forme est appelé *vecteur one-hot*, expression anglaise indiquant qu'il permet de ne sélectionner qu'une seule valeur à activer.

La ligne 9 en finit avec les paramètres et les entrées de notre programme, après quoi le code consiste à placer les calculs réels dans le graphe. La ligne 11 en particulier nous montre la puissance de TF pour les calculs d'apprentissage profond. Elle définit l'essentiel de la passe avant du réseau de neurones pour notre modèle. Elle précise en particulier que nous voulons alimenter les images (un lot à la fois) dans nos unités linéaires (comme défini par `w` et `b`) puis appliquer softmax à tous les résultats pour obtenir un vecteur de probabilité.

Lorsque vous étudiez un code comme celui-ci, nous vous recommandons de vérifier d'abord les formes des tenseurs utilisés pour vous assurer qu'ils sont corrects. Ici, le premier calcul interne est une multiplication matricielle `matmul` des images d'entrée [100, 784] par `w` [784, 10], ce qui nous donne une matrice de forme [100, 10] à laquelle nous ajoutons les biais pour obtenir une matrice de forme [100, 10]. Ce qui constitue les 10 logits pour les 100 images de notre lot. Nous transmettons alors ceci à la fonction softmax et obtenons enfin une matrice [100, 10] des probabilités d'affectation des étiquettes à nos images.

La ligne 12 calcule la perte moyenne d'entropie croisée sur les 100 exemples que nous traitons en parallèle. En exécutant de l'intérieur vers l'extérieur, `tf.log(x)` renvoie un tenseur dans lequel chaque élément de `x` est remplacé par son logarithme naturel. Sur la figure 2.3 nous voyons `tf.log` opérer sur un lot de trois vecteurs, chacun d'eux correspondant à une distribution de probabilité sur cinq éléments.

0.20	0.10	0.20	0.10	0.40		-1.6	-2.3	-1.6	-2.3	-0.9
0.20	0.10	0.20	0.10	0.40	→	-1.6	-2.3	-1.6	-2.3	-0.9
0.20	0.10	0.20	0.10	0.40		-1.6	-2.3	-1.6	-2.3	-0.9

Figure 2.3 – Fonctionnement de `tf.log`

Ensuite, le symbole de multiplication ordinaire '`*`' dans `ans * tf.log(prbs)` effectue une multiplication élément par élément de deux tenseurs. La figure 2.4 montre comment la multiplication élément par élément des vecteurs one-hot de chacune des étiquettes du lot par l'opposé du logarithme naturel de la matrice créée des lignes où tous les éléments sont nuls à l'exception de l'opposé du logarithme de la probabilité de la réponse correcte.

0	0	1	0	0	1.6	2.3	1.6	2.3	0.9	0	0	1.6	0	0		
0	0	1	0	0	*	1.6	2.3	1.6	2.3	0.9	=	0	0	1.6	0	0
0	0	0	0	1	1.6	2.3	1.6	2.3	0.9	0	0	0	0	0	0.9	

Figure 2.4 – Calcul du produit des réponses par l’opposé du log des probabilités

À ce stade, pour obtenir l’entropie croisée sur les images, il ne nous reste plus qu’à ajouter toutes les valeurs dans le tableau. La première opération à effectuer pour cela est :

```
tf.reduce_sum( A, reduction_indices = [1] )
```

qui effectue une sommation sur chaque ligne de A, comme sur la figure 2.5. L’élément essentiel ici est :

```
reduction_indices = [1]
```

0	0	1.6	0	0	1.6
0	0	1.6	0	0	→ 1.6
0	0	0	0	0.9	0.9

Figure 2.5 – Résultat de tf.reduce_sum avec une réduction d’indice sur la dimension [1]

Dans l’introduction sur les tenseurs, nous avons mentionné au passage que leurs dimensions étaient numérotées à partir de zéro. Maintenant, `reduce_sum` peut effectuer une sommation sur les colonnes (le mode par défaut) avec `reduction_index=[0]` ou, comme c’est le cas ici, effectuer une sommation sur les lignes, `reduction_index=[1]`. Ceci résulte en un tableau [100, 1] avec une seule entrée sur chaque ligne correspondant au logarithme de la probabilité correcte (la figure 2.5 utilise uniquement une taille de lot égale à 3, et suppose uniquement cinq choix alternatifs au lieu de 10). Dernier élément du calcul de l’entropie croisée, `reduce_mean` à la ligne 13 de la figure 2.2 effectue une sommation sur les colonnes (option par défaut) et renvoie la moyenne (1.1 à peu près).

Nous pouvons finalement passer à la ligne 14 de la figure 2.2, et c’est là que TF montre tous ses mérites : cette seule ligne est tout ce dont nous avons besoin pour effectuer l’ensemble de la passe arrière :

```
tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
```

Celle-ci demande de calculer les modifications des poids en utilisant la descente de gradient et de minimiser la fonction de perte d’entropie croisée définie aux lignes 12 et 13. Elle spécifie également un taux d’apprentissage de 0.5. Nous n’avons pas à nous soucier de calculer des dérivées ou quoi que ce soit. Si vous décrivez dans TensorFlow

le calcul de la passe avant et la perte, alors le compilateur TF sait comment calculer les dérivées nécessaires et enchaîner les calculs dans le bon ordre pour effectuer les modifications. Nous pouvons modifier cet appel de fonction en choisissant un taux d'apprentissage différent, ou, si nous avons une fonction de perte différente, remplacer `xEnt` par quelque chose qui pointerait sur un calcul TF différent.

Naturellement, il existe des limites à la capacité de TF à déduire la passe arrière sur la base de la passe avant. Répétons-le, il n'est capable de le faire que si les calculs de la passe avant sont effectués avec des fonctions TF. Pour les débutants comme nous, ce n'est pas une trop grande limitation car TF dispose d'une large variété de fonctions intégrées qu'il sait comment différentier et connecter.

Les lignes 15 et 16 calculent l'exactitude du modèle. Pour cela, elles comptent le nombre de réponses correctes et le divisent par le nombre d'images traitées. Tout d'abord, concentrons-nous sur la fonction mathématique classique `argmax`, comme dans `arg maxxf(x)`, qui renvoie la valeur de x maximisant $f(x)$. Dans l'utilisation que nous en faisons ici, `tf.argmax(prbs, 1)` accepte deux arguments. Le premier est le tenseur auquel on applique `argmax`. Le second est l'*axe* du tenseur à utiliser pour `argmax`. Ceci fonctionne comme l'argument nommé que nous avons utilisé pour `reduce_sum` qui nous permettait d'effectuer des sommes selon différents axes du tenseur. Par exemple, si le tenseur est $((0,2,4),(4,0,3))$ et que nous utilisons l'axe 0 (celui par défaut), nous recevons en retour $(1,0,0)$. Nous avons d'abord comparé 0 à 4 et renvoyé 1 car 4 est plus grand. Puis nous avons comparé 2 à 0 et renvoyé 0 car 2 est plus grand. Si nous avions utilisé l'axe 1 nous aurions renvoyé $(2,0)$. À la ligne 15 nous avons un tableau de logits ayant la taille du lot. La fonction `argmax` renvoie un tableau de la taille du lot contenant les positions des logits maximaux. Nous appliquons ensuite `tf.equal` pour comparer les logits maximaux à la réponse correcte. Cette fonction renvoie un vecteur de la taille du lot comportant des valeurs booléennes (Vrai en cas d'égalité) que `tf.cast(tensor, tf.float32)` transforme en nombres à virgule flottante pour que `tf.reduce_mean` puisse les additionner et obtenir le pourcentage correct. Il ne faut pas forcer les valeurs booléennes au type entier, car sinon la moyenne renverrait aussi un entier qui dans ce cas serait toujours égal à zéro.

Ensuite, une fois que nous avons défini notre session (ligne 18) et initialisé les valeurs des paramètres (ligne 19), nous pouvons entraîner le modèle (lignes 21 à 23). Ici nous utilisons le code récupéré dans la bibliothèque TF du Mnist pour extraire en une fois 100 images et leurs réponses puis lancer l'exécution sur celles-ci en appelant `sess.run` sur la portion de graphe de calcul pointée par `train`. Une fois cette boucle terminée, nous avons effectué 1 000 itérations d'entraînement avec 100 images par itération, soit 100 000 images de test au total. Sur mon Mac Pro à quatre processeurs, ceci prend environ 5 secondes (un peu plus la première fois pour placer dans le cache tout ce qui est nécessaire). Je mentionne les quatre processeurs car TF vérifie la puissance de calcul disponible et l'utilise en général plutôt efficacement sans qu'on ait à le lui demander.

Remarquez une chose plutôt étrange à propos des lignes 21 à 23 : nous ne mentionnons jamais explicitement l'exécution de la passe avant ! TF comprend cela tout seul, en se basant sur le graphe de calcul. Le `GradientDescentOptimizer` lui indique qu'il doit avoir effectué le calcul pointé par `xEnt` (ligne 12), ce qui nécessite le calcul de `probs`, ce qui à son tour spécifie le calcul de la passe avant à la ligne 11.

Enfin, les lignes 25 à 29 nous permettent de juger de la qualité des résultats sur les données de test en termes de pourcentage correct (91 % ou 92 %). Tout d'abord, en étudiant l'organisation du graphe, observez que le calcul d'`exactitude` nécessite à la fin d'effectuer le calcul de la passe avant `probs` mais pas celui de la passe arrière `train`. Par conséquent, comme nous devons nous y attendre, les poids ne sont pas modifiés pour obtenir de meilleurs résultats sur les données de test.

Comme nous l'avons mentionné au chapitre 1, imprimer le taux d'erreur au cours de l'entraînement du modèle constitue une bonne pratique de débogage. En règle générale, il décroît. Pour cela, nous modifions la ligne 23 en :

```
acc, ignore= sess.run([accuracy, train],
                      feed_dict={img: imgs, ans: anss})
```

Il s'agit là d'une syntaxe Python normale pour combiner les calculs. La valeur du premier calcul (celui de l'`exactitude`) est affectée à la variable `acc`, le second à `ignore` (une subtilité propre à Python consisterait à remplacer `ignore` par un tiret bas (`_`, *underscore* en anglais), le symbole universel de Python utilisé lorsque la syntaxe requiert qu'une variable recueille une valeur mais que nous n'avons pas besoin de la mémoriser). Naturellement nous aurions aussi besoin d'ajouter une commande pour imprimer la valeur de `acc`.

Nous avons mentionné cela pour encourager le lecteur à éviter une erreur commune (ou du moins, une erreur que votre auteur et certains de ses étudiants débutants ont faite). L'erreur consiste à laisser la ligne 23 telle quelle et à ajouter une nouvelle ligne 23.5 :

```
acc= sess.run(accuracy, feed_dict={img: imgs, ans: anss})
```

Ceci est cependant moins efficace, car TF effectue maintenant deux fois la passe avant : une fois lorsque nous lui demandons d'effectuer l'entraînement, et une fois lorsque nous lui demandons de calculer l'`exactitude`. Mais il y a une raison plus importante d'éviter cette situation. Notez que le premier appel modifie les poids et rend donc plus probable d'obtenir l'étiquette correcte pour l'image. En demandant après cela d'effectuer le calcul d'`exactitude`, le programmeur obtient une idée exagérée des bons résultats du programme. Lorsque nous n'avons qu'un seul appel à `sess.run` mais demandons les deux valeurs, ceci ne se produit pas.

2.3 RÉSEAUX DE NEURONES À PLUSIEURS COUCHES

Le programme présenté en pseudo-code au chapitre 1 et maintenant en TF n'a qu'une seule couche. Il n'y a qu'une couche d'unités linéaires. La question qui se pose tout naturellement, c'est de savoir si nous pouvons faire mieux avec de multiples couches de telles unités. Très tôt, les spécialistes des réseaux de neurones ont rapidement compris que la réponse est « Non ». Cette idée s'impose presque immédiatement après avoir vu que les unités linéaires peuvent être redéfinies en matrices d'algèbre linéaire — c'est-à-dire une fois que nous avons vu qu'un réseau de neurones monocouche à propagation avant calcule simplement $y = \mathbf{XW}$. Dans notre modèle Mnist, \mathbf{W} a pour forme [784, 10] de manière à transformer les 784 valeurs de pixels en 10 valeurs de logits et ajouter un poids supplémentaire pour remplacer le terme de biais. Supposons que nous ajoutions une couche supplémentaire d'unités linéaires \mathbf{U} de forme [784, 784], qui à son tour alimente une couche \mathbf{V} de même forme que \mathbf{W} , [784, 10] :

$$\mathbf{y} = (\mathbf{xU})\mathbf{V} \quad (2.1)$$

$$= \mathbf{x}(\mathbf{UV}) \quad (2.2)$$

La seconde ligne découle de la propriété d'associativité du produit matriciel. En fait, quelles que soient les possibilités offertes dans un modèle à deux couches par la combinaison de \mathbf{U} multiplié par \mathbf{V} , celles-ci pourraient être obtenues par un réseau de neurones à une couche $\mathbf{W} = \mathbf{UV}$.

Il se trouve qu'il existe une solution simple : ajouter quelques calculs non linéaires entre les couches. Une des fonctions les plus couramment utilisées est `tf.nn.relu` (ou ρ), `relu` étant l'abréviation de *rectified linear unit* (unité linéaire rectifiée), définie par

$$\rho(x) = \max(x, 0) \quad (2.3)$$

et présentée à la figure 2.6.

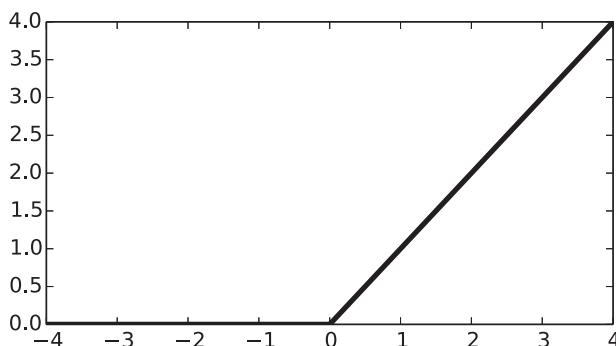


Figure 2.6 – Comportement de `tf.nn.relu`

2.3 Réseaux de neurones à plusieurs couches

Les fonctions non linéaires placées entre les couches en apprentissage profond sont appelées *fonctions d'activation*. Si *relu* est très en vogue à l'heure actuelle, d'autres sont utilisées également, comme par exemple la fonction *sigmoïde*, définie par

$$S(x) = \frac{e^{-x}}{1 + e^{-x}} \quad (2.4)$$

et présentée sur la figure 2.7. Dans tous les cas, les activations sont appliquées tour à tour à chacun des nombres réels de l'argument tenseur. Par exemple, $\rho([1, 17, -3]) = [1, 17, 0]$.

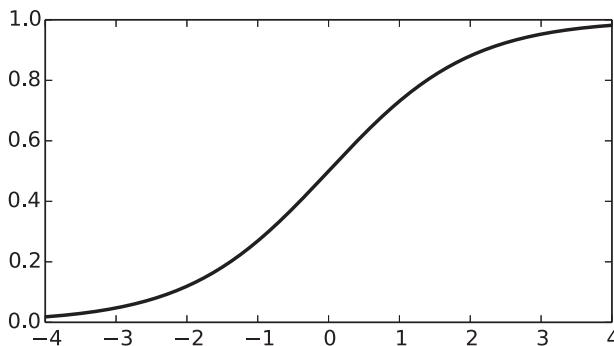


Figure 2.7 – La fonction sigmoïde

Avant qu'on ait découvert qu'une non-linéarité aussi simple que *relu* pouvait fonctionner, la sigmoïde avait beaucoup de succès. Mais la plage des valeurs de sortie d'une sigmoïde était plutôt limitée, de zéro à un, alors que *relu* va de zéro à l'infini. Il est essentiel, lorsque nous effectuerons la passe arrière, de trouver (grâce au gradient) quels sont les paramètres qui affectent le plus la perte. Une propagation arrière à travers des fonctions n'ayant qu'un petit nombre de valeurs peut faire que le gradient soit pratiquement nul, effet souvent nommé problème de la *disparition du gradient*. Les fonctions d'activation plus simples ont été d'une grande aide ici. Pour cette raison `tf.nn.lrelu`, *leaky relu*, est aussi très utilisée car elle a une plage de valeurs encore plus large que *relu*, comme on peut le voir sur la figure 2.8.

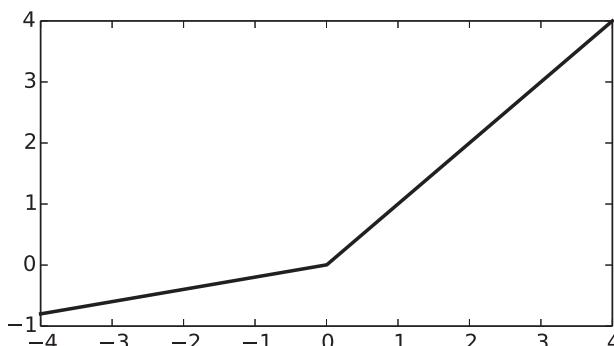


Figure 2.8 – La fonction lrelu

En assemblant les morceaux de notre réseau de neurones multi-couches, notre nouveau modèle est :

$$\text{Pr}(A(x)) = \sigma(\rho(\mathbf{x}\mathbf{U} + \mathbf{b}_u)\mathbf{V} + \mathbf{b}_v) \quad (2.5)$$

où σ et la fonction softmax \mathbf{U} et \mathbf{V} sont les poids de la première et de la deuxième couche d'unités linéaires, tandis que \mathbf{b}_u et \mathbf{b}_v sont leurs biais.

Effectuons maintenant cela en TF. Dans le code présenté sur la figure 2.9, nous remplaçons les définitions de \mathbf{W} et \mathbf{b} aux lignes 5 et 6 de la figure 2.2 par les deux couches \mathbf{U} et \mathbf{V} , lignes 1 à 4 de la figure 2.9. Nous remplaçons aussi le calcul de prbs à la ligne 11 de la figure 2.2 par les lignes 5 à 7 de la figure 2.9. Ceci transforme notre code en un réseau de neurones multi-couches. Par ailleurs, pour tenir compte du plus grand nombre de paramètres, nous devons diminuer d'un facteur 10 le taux d'apprentissage. Alors que l'ancien programme plafonnait aux alentours d'un taux d'exactitude de 92 % après un entraînement sur 100 000 images, le nouveau atteint une exactitude d'environ 94 % sur 100 000 images. De plus, si nous augmentons le nombre d'images d'entraînement, les performances sur le jeu de test continuent à augmenter jusqu'à environ 97 %. Notez que la seule différence entre ce code et celui sans la fonction non linéaire figure à la ligne 6. Si nous supprimons celle-ci, les performances retombent en fait aux alentours de 92 %. C'est assez pour nous convaincre de l'utilité des mathématiques !

```

1 U = tf.Variable(tf.random_normal([784, 784], stddev=.1))
2 bU = tf.Variable(tf.random_normal([784], stddev=.1))
3 V = tf.Variable(tf.random_normal([784, 10], stddev=.1))
4 bV = tf.Variable(tf.random_normal([10], stddev=.1))
5 L1Output = tf.matmul(img, U)+bU
6 L1Output=tf.nn.relu(L1Output)
7 prbs=tf.nn.softmax(tf.matmul(L1Output, V)+bV)

```

Figure 2.9 – Code de construction du graphe TF pour reconnaissance de chiffres multi-niveaux

Encore un point. Dans un réseau monocouche avec une matrice de paramètres \mathbf{W} , la forme de \mathbf{W} est déterminée d'une part par le nombre d'entrées (784), d'autre part par le nombre de sorties possibles (10). Avec deux couches, il y a un autre choix que nous pouvons faire, la *taille cachée*. Par conséquent \mathbf{U} a pour dimensions taille-d'entrée par taille-cachée et \mathbf{V} taille-cachée par taille-de-sortie. Sur la figure 2.9 nous avons choisi une taille cachée égale à 784, identique à la taille d'entrée, mais rien n'exigeait de faire ce choix. En règle générale, lorsqu'on choisit une valeur plus grande, on accroît les performances, mais le gain se réduit de plus en plus pour atteindre une sorte de plateau.

2.4 AUTRES PARTICULARITÉS

Dans cette section, nous couvrons des aspects de TensorFlow qui sont très utiles lorsque nous effectuons certains des exercices de programmation suggérés dans le reste de cet ouvrage (p. ex. les points de contrôle) ou que nous emploierons de diverses manières dans les chapitres suivants.

2.4.1 Point de contrôle

Il est souvent utile d'établir un *point de contrôle* (en anglais, *checkpoint*) dans un calcul TF, à savoir sauvegarder les tenseurs intervenant dans ce calcul pour pouvoir reprendre le calcul ultérieurement, ou les réutiliser dans un programme différent. Pour réaliser cela en TF, nous devons créer et utiliser des *objets Saver* :

```
saveOb= tf.train.Saver()
```

Comme précédemment, `saveOb` est une variable Python dont le nom peut être choisi librement. L'objet peut être créé à n'importe quel moment avant son utilisation, mais, pour les raisons indiquées ci-dessous, il est plus logique de le faire juste avant d'initialiser les variables (en appelant `global_variable_initialize`). Puis, toutes les n époques d'entraînement, sauvegarder les valeurs courantes de toutes vos variables :

```
saveOb.save(sess, "mylatest.ckpt")
```

La méthode `save` reçoit deux arguments : la session à sauvegarder, ainsi que le nom et l'emplacement du fichier. Dans le cas ci-dessus, l'information va dans le même répertoire que le programme Python. Si l'argument avait été `tmp/model.checkpt`, l'information aurait été placée dans un sous-répertoire de `tmp`.

L'appel à `save` crée quatre fichiers. Le plus petit, nommé `checkpoint`, est un fichier ASCII spécifiant quelques renseignements de haut niveau à propos du point de contrôle sauvegardé dans ce répertoire. Le nom `checkpoint` est imposé. Si vous aviez déjà créé un autre fichier nommé `checkpoint`, son contenu sera écrasé. Les trois autres noms de fichiers utilisent la chaîne que vous avez transmise à `save`. Dans notre exemple, ils sont nommés :

```
mylatest.ckpt.data-00000-of-00001
mylatest.ckpt.index
mylatest.ckpt.meta
```

Le premier de ceux-ci contient les valeurs des paramètres que vous avez sauvegardées. Les deux autres contiennent des métainformations que TF utilise lorsque vous voulez importer ces valeurs (comme décrit ci-après). Si votre programme appelle plusieurs fois `save`, le contenu de ces fichiers est écrasé à chaque fois par le nouveau.

Supposons que nous voulions ensuite réaliser de nouvelles époques (ou itérations) d'entraînement sur le même modèle de réseau de neurones déjà utilisé pour un premier entraînement. Le moyen le plus simple consiste à modifier le programme d'entraînement originel. Nous conservons la création de l'instance de Saver, mais il nous faut maintenant initialiser toutes les variables TF avec les valeurs sauvegardées. Pour cela, il suffit de remplacer `global_variable_initialize` par un appel à la méthode `restore` de notre objet Saver :

```
saveOb.restore(sess, "mylatest.ckpt")
```

La prochaine fois que vous appelez le programme d'entraînement, il reprend ce dernier après avoir affecté aux variables TF les valeurs qu'elles avaient lorsque vous les avez sauvegardées à la fin de votre entraînement précédent. Il n'y a pas d'autre modification. Par conséquent, si votre code d'entraînement imprimait le numéro d'époque suivi de la perte, cette fois-ci il imprimera les numéros d'époque en les numérotant à partir de 1, à moins de modifier votre code pour procéder autrement (naturellement, vous êtes libres de corriger cela ou de le reprogrammer plus élégamment, mais la qualité du code Python n'est pas notre souci principal ici).

2.4.2 `tensordot`

`tensordot` est une généralisation de la multiplication matricielle de tenseurs. Nous avons vu au chapitre précédent la multiplication standard de matrices, `matmul`. Nous pouvons appeler `tf.matmul(A, B)` lorsque A et B ont le même nombre de dimensions, par exemple n , que la dernière dimension de A est égale à l'avant-dernière dimension de B, et que les $n-2$ dimensions précédentes sont identiques. Ainsi, si A a pour dimensions [2, 3, 4] et B [2, 4, 6], alors les dimensions du produit sont [2, 3, 6]. On peut visualiser la multiplication matricielle comme une succession de produits scalaires. Par exemple, la multiplication de matrices

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \end{pmatrix} \quad (2.6)$$

peut être réalisée en effectuant le produit scalaire des vecteurs $(1, 2, 3)$ et $(-1, -3, -5)$ et en reportant le résultat dans l'élément en haut et à gauche de la matrice résultante. De la même façon, nous effectuons le produit scalaire de la i -ième ligne et de la j -ième colonne et plaçons ce résultat dans l'élément d'indice i, j de la matrice résultante. Par conséquent si A est la première des matrices ci-dessus et B la seconde, ce calcul peut aussi être exprimé comme :

```
tf.tensordot(A, B, [[1], [0]])
```

Les deux premiers arguments sont, bien sûr, les tenseurs sur lesquels nous travaillons. Le troisième argument est une liste à deux éléments : le premier élément est une liste de dimensions provenant du premier argument, le second élément est la liste

de dimensions correspondante du deuxième argument. Ceci indique à `tensordot` qu'il doit effectuer les produits scalaires selon ces deux dimensions. Naturellement, les dimensions spécifiées doivent être de même taille si nous voulons effectuer leur produit scalaire. Sachant que la dimension 0 correspond aux lignes et que la dimension 1 correspond aux colonnes, ceci nous demande d'effectuer les produits scalaires de chacune des lignes de `A` par chacune des colonnes de `B`. `tensordot` coordonne les dimensions de sortie de la gauche vers la droite, partant de celles de `A` pour finir avec celles de `B`. C'est-à-dire que nous avons ici les dimensions d'entrée [2, 3] suivies de [3, 2]. Les deux dimensions impliquées dans les produits scalaires « disparaissent » (dimensions 1 et 0) pour obtenir une réponse de dimensions [2, 2].

La figure 2.10 présente un exemple plus sophistiqué que `matmul` ne saurait pas gérer en une seule instruction. Nous l'avons emprunté au chapitre 5 où les noms de variables prendront tout leur sens. Ici, nous voulons simplement étudier ce que fait `tensordot`. Sans vous occuper des valeurs numériques, regardez simplement le troisième argument de l'appel de fonction `tensordot`, `[[1] [0]]`. Ceci signifie que nous allons effectuer les produits scalaires selon la dimension 1 de `encOut` et la dimension 0 de `AT`. C'est licite puisqu'elles sont toutes les deux de taille 4. Ce qui veut dire que nous effectuons des produits scalaires sur deux tenseurs respectivement de forme [2, 4, 4] et [4, 3] (les valeurs en italique correspondent aux dimensions sur

```

eo=  ( (( 1, 2, 3, 4),
        ( 1, 1, 1, 1),
        ( 1, 1, 1, 1),
        ( -1, 0,-1, 0)),
       (( 1, 2, 3, 4),
        ( 1, 1, 1, 1),
        ( 1, 1, 1, 1),
        ( -1, 0,-1, 0)) )
encOut=tf.constant(eo, tf.float32)

AT = ( ( 0.6, 0.25, 0.25 ),
       ( 0.2, 0.25, 0.25 ),
       ( 0.1, 0.25, 0.25 ),
       ( 0.1, 0.25, 0.25 ) )
wAT = tf.constant(AT, tf.float32)

encAT = tf.tensordot(encOut,wAT,[[1],[0]])
sess= tf.Session()

print sess.run(encAT)
[[[ 0.80000001  0.5          0.5          ]
  [ 1.50000012  1.           1.           ]
  [ 2.           1.           1.           ]
  [ 2.70000005  1.5          1.5          ]]
   ...
  ]

```

Figure 2.10 – Exemple d'utilisation de `tensordot`

lesquelles sont effectués les produits scalaires). Étant donné que ces dimensions disparaissent une fois les produits scalaires effectués, le tenseur résultant a pour forme [2, 4, 3], ce qui correspond bien à ce que nous imprimons à la fin de l'exemple. Pour en venir brièvement aux calculs effectifs, effectuons les produits scalaires entre les colonnes des deux tenseurs. Le premier produit scalaire à effectuer est entre [1, 1, 1, -1] et [0.6, 0.2, 0.1, 0.1]. Le résultat, 0.8, apparaît en tant que première valeur numérique du tenseur résultant.

Enfin, `tensordot` ne se limite pas à effectuer des produits scalaires selon une seule dimension de chaque tenseur. Si A a pour forme [2, 4, 4] et B [4, 4] alors l'opération `tensordot(A, B, [[1, 2], [0, 1]])` produit un tenseur de forme [2].

2.4.3 Initialisation des variables TF

À la section 1.4 nous avons dit qu'il était de bonne pratique d'initialiser les paramètres d'un réseau de neurones (c'est-à-dire les variables TF) de manière aléatoire, mais proches de zéro. Dans notre premier programme TF (figure 2.9), nous avons traduit cette injonction par une commande du genre :

```
b = tf.Variable(tf.random_normal([10], stddev=.1))
```

où nous avons supposé qu'un écart type de 0.1 était suffisamment « proche de zéro ».

Il y a eu, cependant, beaucoup de recherches théoriques et d'expérimentations pratiques quant au choix de l'écart type dans de tels cas. Nous donnons ici une règle appelée *initialisation de Xavier*. Celle-ci est couramment utilisée pour choisir l'écart type lorsqu'on effectue une initialisation aléatoire des variables. Soit n_i le nombre de connexions arrivant à la couche et n_o le nombre de celles qui en sortent. Pour la variable W de la figure 2.9, $n_i = 784$ (le nombre de pixels) et $n_o = 10$ (le nombre de classifications différentes). Pour l'initialisation de Xavier, nous définissons σ , l'écart type, de la façon suivante :

$$\sigma = \sqrt{\frac{2}{n_i + n_o}} \quad (2.7)$$

Pour W , par exemple, étant donné que les valeurs en question sont 784 et 10 nous obtenons $\sigma \approx 0.0502$, que nous avons arrondi à 0.1. En règle générale, les écarts types recommandés peuvent varier entre 0.3 pour une couche $10 * 10$ et 0.03 pour une de $1\,000 * 1\,000$. Plus il y a de valeurs d'entrée de sortie, plus l'écart type est petit.

L'initialisation de Xavier a été créée à l'origine pour être utilisée avec la fonction d'activation sigmoïde (voir figure 2.7). Comme nous l'avons déjà mentionné, $\sigma(x)$ ne réagit plus vraiment lorsque x devient très inférieur à -2 ou supérieur à +2. Ce qui veut dire que, si les valeurs transmises à la sigmoïde sont trop élevées ou trop basses, une modification de celles-ci n'a plus beaucoup d'effet sur la perte. Inversement, lors de la passe arrière, la modification apportée à la perte n'aura aucun effet sur les

paramètres ayant alimenté la sigmoïde si cette modification de la perte est effacée par la sigmoïde. Pour éviter cela, nous voulons que la *variance* du ratio entre l'entrée et la sortie d'un niveau soit voisine de 1. Nous utilisons ici le mot « variance » dans son sens technique : l'espérance mathématique du carré de la différence entre la valeur d'une variable aléatoire à valeurs numériques et sa moyenne. De même, l'*espérance mathématique* d'une variable aléatoire X (notée $E[X]$) est une moyenne probabiliste de toutes ses valeurs :

$$E[X] = \sum_x p(X=x) * x \quad (2.8)$$

Un exemple classique en est l'espérance du lancer d'un dé à six faces non truqué :

$$E[R] = \frac{1}{6} * 1 + \frac{1}{6} * 2 + \frac{1}{6} * 3 + \frac{1}{6} * 4 + \frac{1}{6} * 5 + \frac{1}{6} * 6 = 3.5 \quad (2.9)$$

Nous voulons donc conserver un rapport entre la variance d'entrée et la variance de sortie voisin de 1 afin que le niveau ne contribue pas à une atténuation indue du signal par la fonction sigmoïde. Ceci introduit donc des contraintes sur la façon dont on initialise. Nous énonçons donc sans démonstration (mais vous pouvez effectuer la dérivation) que, pour une unité linéaire avec une matrice de poids \mathbf{W} , la variance durant la passe avant (V_f) et celle durant la passe arrière (V_b) sont respectivement :

$$V_f(W) = \sigma^2 \cdot n_i \quad (2.10)$$

$$V_b(W) = \sigma^2 \cdot n_o \quad (2.11)$$

où σ est l'écart type des poids de \mathbf{W} (ceci est logique sachant que la variance d'une distribution gaussienne est σ^2). Si V_f et V_b valent 0 et que nous résolvons les équations pour trouver σ , nous obtenons :

$$\sigma = \sqrt{\frac{1}{n_i}} \quad (2.12)$$

$$\sigma = \sqrt{\frac{1}{n_o}} \quad (2.13)$$

Naturellement, ce système d'équations n'a de solution que si le nombre des entrées est égal à celui des sorties. Étant donné que ce n'est généralement *pas* le cas, nous prenons une « moyenne », ce qui nous donne la règle de Xavier :

$$\sigma = \sqrt{\frac{2}{n_i + n_o}} \quad (2.14)$$

Il existe des équations équivalentes pour d'autres fonctions d'activation. Avec l'émergence de `relu` et d'autres fonctions d'activation qui ne saturent pas aussi aisément que la sigmoïde, le problème n'est plus aussi important qu'il a été. Cependant,

la règle de Xavier nous donne une bonne idée de ce que devrait être l'écart type, et les versions TF de cette règle et de ses homologues sont fréquemment utilisées.

2.4.4 Simplification de la création de graphe TF

En effectuant un retour sur la figure 2.9, nous voyons qu'il nous a fallu sept lignes de code pour spécifier notre réseau à propagation avant à deux couches. Dans l'absolu, ce n'est pas beaucoup : considérez ce qu'il nous faudrait pour programmer ceci en Python sans TF ! Cependant, si vous deviez par exemple créer un réseau à huit couches — et c'est ce que vous ferez avant la fin de ce livre —, il vous faudrait pas loin de 24 lignes de code !

TF dispose d'un groupe de fonctions bien pratiques, le module `layers`, pour coder de manière plus compacte des configurations en couches très communes. Nous présentons ici :

```
tf.contrib.layers.fully_connected
```

Une couche est dite *entièrement connectée* si toutes ses unités sont connectées à toutes les unités de la couche suivante. Toutes les couches que nous avons utilisées dans les deux premiers chapitres étaient entièrement connectées, c'est pourquoi il n'a pas été nécessaire jusqu'ici de faire la distinction entre ces réseaux et ceux qui n'ont pas cette propriété. Pour définir une telle couche, nous faisons en général ce qui suit : (a) créer les poids **W**, (b) créer les biais **b**, (c) effectuer le produit des matrices et ajouter les biais, et finalement (d) appliquer une fonction d'activation. En supposant que nous ayons importé `tensorflow.contrib.layers` en tant que `layers`, tout ceci peut être réalisé en une seule ligne :

```
layerOut=layers.fully_connected(layerIn,outSz,activeFn)
```

L'appel ci-dessus crée une matrice initialisée par une initialisation de Xavier et un vecteur de biais initialisés à zéro. Il renvoie `layerIn` fois la matrice, plus les biais, après application de la fonction d'activation spécifiée par `activeFn`. Si vous ne spécifiez pas de fonction d'activation, il utilise `relu`. Si vous spécifiez `None` comme fonction d'activation, alors aucune n'est utilisée.

Avec `fully_connected` nous pouvons récrire les sept lignes de la figure 2.9 sous la forme :

```
L1Output=layers.fully_connected(img, 756)
prbs=layers.fully_connected(L1Output,10,tf.nn.softmax)
```

Remarquez que nous avons spécifié d'appliquer `tf.nn.softmax` à la sortie de la seconde couche en tant que fonction d'activation pour cette couche.

Bien sûr, si nous avons un réseau de neurones à 100 couches et que nous voulons procéder ainsi, le simple fait d'écrire 100 appels à `fully_connected` est fastidieux. Heureusement, nous pouvons utiliser Python, ou ce qui figure dans l'API TF,

pour spécifier notre réseau. Pour citer un exemple plutôt fictif, supposons que nous voulions créer 100 couches cachées, chacune comportant une unité de moins que la précédente, la taille de la première étant un paramètre du système. Nous pourrions écrire :

```
outpt = input
for i in range(100):
    outpt = layers.fully_connected(outpt, sysParam - i)
```

Cet exemple est idiot, mais le point est important : en Python, on peut transmettre et traiter des morceaux de graphes TF comme s'il s'agissait de listes ou de dictionnaires.

2.5 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

TensorFlow a été conçu par *Google Brain*, projet au sein de Google initialisé par Jeff Dean et Greg Corrado, chercheurs chez Google, et Andrew Ng, professeur à Stanford. À l'époque, le projet s'appelait DistBelief. Lorsque son utilisation a dépassé le cadre du projet initial, Google a repris en main la suite du développement et embauché Geoffrey Hinton de l'Université de Toronto, dont nous avons parlé au chapitre 1 pour ses contributions d'avant-garde à l'apprentissage profond.

L'initialisation de Xavier fait référence au prénom de Xavier Glorot, le premier auteur de [GB10], qui a présenté la technique.

De nos jours TensorFlow n'est qu'un des nombreux langages destinés à la programmation de l'apprentissage profond (p. ex. [Var17]). Du point de vue du nombre d'utilisateurs, TensorFlow est de loin le plus populaire. Ensuite vient Keras, un langage de haut niveau construit par-dessus TensorFlow, suivi par Caffe, développé à l'origine par l'Université de Berkeley, en Californie. Facebook soutient maintenant une version open source de Caffe, nommée Caffe2. Pytorch est une interface Python pour Torch, un langage qui a fait des adeptes dans la communauté du traitement du langage naturel par l'apprentissage profond.

Exercices

- 2.1** Quel aurait été le résultat si à la figure 2.5 nous avions calculé à la place `tf.reduce_sum(A)`, où A est le tableau à gauche de la figure ?
- 2.2** Qu'y a-t-il de mal à prendre la ligne 14 de la figure 2.2 et à l'insérer entre les lignes 22 et 23, de sorte que la boucle ressemble à ce qui suit ?

```
for i in range(1000):
    imgs, anss = mnist.train.next_batch(batchSz)
    train = tf.train.GradientDescentOptimizer(0.5).minimize(xEnt)
    sess.run(train, feed_dict={img: imgs, ans: anss})
```

2.3 Voici une autre variante à partir des mêmes lignes de code. Est-ce correct ? Sinon, pourquoi ?

```
for i in range(1000):
    img, anss= mnist.test.next_batch(batchSz)
    sumAcc+=sess.run(accuracy, feed_dict={img:img, ans:anss})
```

2.4 À la figure 2.10, quelle serait la forme du tenseur résultant de l'opération suivante :

```
tensordot(wAT, encOut, [[0], [1]])
```

Expliquez.

2.5 Vérifiez par le calcul l'exactitude (à trois décimales près) de la première valeur (0.8) dans le tenseur imprimé en bas de l'exemple de la figure 2.10.

2.6 Supposons qu'`input` ait pour forme [50, 10]. Combien le code suivant crée-t-il de variables TF ?

```
o1 = layers.fully_connected(input, 20, tf.sigmoid)
```

Quel sera l'écart type des variables de la matrice créée ?

RÉSEAUX DE NEURONES CONVOLUTIFS

3

Les réseaux de neurones que nous avons examinés jusqu'ici étaient tous *entièrement connectés*, ce qui signifie que toutes les unités linéaires d'une couche étaient connectées à toutes les unités linéaires de la couche suivante. Cependant, rien n'oblige à ce que tous les réseaux de neurones aient cette forme particulière. Nous pouvons certainement concevoir une passe avant dans laquelle une unité linéaire ne transmet sa sortie qu'à certaines des unités de la couche suivante. De même, il n'est pas beaucoup plus difficile d'imaginer que TensorFlow, s'il sait quelles unités sont connectées entre elles, puisse calculer correctement les dérivées des poids durant la passe arrière.

Les *réseaux de neurones convolutifs* sont un cas particulier de ces réseaux de neurones partiellement connectés. Les réseaux de neurones convolutifs sont particulièrement utiles en vision par ordinateur, c'est pourquoi nous allons reprendre notre discussion à propos du jeu de données Mnist.

Le réseau de neurones Mnist entièrement connecté à un niveau apprend à associer certaines intensités lumineuses en certaines positions de l'image avec certains chiffres — en associant par exemple de fortes valeurs à la position (8,14) avec le chiffre « 1 ». Mais ce n'est clairement pas de cette façon que les gens travaillent. Photographier les chiffres dans une pièce plus claire ajouterait peut-être 10 à chaque valeur de pixel, mais aurait peu ou pas d'influence sur notre catégorisation — ce qui importe dans la reconnaissance d'une scène, c'est la différence entre les valeurs des pixels, et non leurs valeurs absolues. De plus, les différences n'ont de sens que pour les valeurs voisines. Supposons que vous soyez dans une petite pièce avec une simple ampoule dans un coin. Ce que nous percevons comme une tache claire, par exemple sur le papier peint à l'autre bout de la pièce, pourrait refléter moins de photons qu'une tache sombre près de l'ampoule. Ce qui importe lorsqu'on cherche à comprendre ce qui se passe dans une scène, ce sont les différences d'intensité lumineuse locales, les termes les plus importants étant « locales » et « différences ». Naturellement, les spécialistes de la vision par ordinateur en sont bien conscients, et leur réponse a été l'adoption quasi universelle des méthodes convolutives.¹

3.1 FILTRES, PAS ET MARGE

Pour nos besoins, un *filtre de convolution* (également appelé *noyau de convolution*) est un tableau de valeurs numériques (en général petit). Si nous avons affaire à une image en noir et blanc, il s'agit d'un tableau à deux dimensions. Les images Mnist sont en noir et blanc, c'est donc tout ce qu'il nous faut ici. Si nous avions de la couleur, il nous faudrait un tableau à trois dimensions — ou de manière équivalente

1. Le terme « convolution » que nous utilisons ici a un sens propre à l'apprentissage profond. Il est très voisin de celui utilisé en mathématiques, où la convolution de l'apprentissage profond serait appelée *corrélation croisée*.

Chapitre 3 · Réseaux de neurones convolutifs

trois tableaux à deux dimensions — un pour chacune des trois longueurs d'onde rouge, vert et bleu (RVB) de la lumière, à partir desquelles on peut reconstruire toutes les couleurs. Pour le moment, nous ignorons la complexité de la couleur. Nous y reviendrons plus tard.

Considérons le filtre de convolution présenté à la figure 3.1. Pour *convoluer* un filtre avec une portion d'image, nous effectuons le produit matriciel du filtre et d'une portion de l'image de même taille. Rappelez-vous que pour effectuer le produit scalaire de deux vecteurs, nous multiplions les éléments correspondants puis ajoutons tous ces produits pour trouver un nombre unique. Nous généralisons ici cette notion à des tableaux à deux dimensions ou plus, c'est pourquoi nous multiplions les éléments correspondants des tableaux puis ajoutons les produits.

Pour en donner une définition plus formelle, nous considérons que le noyau de convolution est une fonction, la *fonction noyau* (en anglais, *kernel function*). Voici comment obtenir la valeur V de cette fonction à la position x, y d'une image I :

$$V(x, y) = (I \cdot K)(x, y) = \sum_m \sum_n I(x + m, y + n)K(m, n) \quad (3.1)$$

Ceci signifie que, formellement, la convolution est une opération (représentée ici par un point noir) qui prend deux fonctions, I et K , et renvoie une troisième fonction qui effectue l'opération à droite. Pour nos besoins pratiques, nous pouvons sauter la définition formelle et passer directement aux opérations de droite. Par ailleurs, nous considérons normalement un point x, y proche du centre de la portion d'image sur laquelle nous travaillons, donc, pour le noyau $4 * 4$ présenté ci-dessus, m et n peuvent tous les deux varier de -2 à $+1$ inclus.

Convoluons maintenant le filtre de la figure 3.1 avec le coin inférieur droit de l'image d'un petit carré présentée figure 3.2. Les deux lignes du bas du filtre se

$$\begin{matrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ -1.0 & -1.0 & -1.0 & -1.0 \\ -1.0 & -1.0 & -1.0 & -1.0 \end{matrix}$$

Figure 3.1 – Filtre simple pour détection de ligne horizontale

$$\begin{matrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{matrix}$$

Figure 3.2 – Image d'un petit carré

superposent avec des 0.0 de l'image. Mais les quatre éléments en haut et à gauche du filtre se superposent tous avec des 2.0 du carré, ce qui fait que la valeur de ce filtre sur cette portion d'image est égale à 8. Naturellement, si toutes les valeurs des pixels étaient 0, la valeur obtenue par application du filtre serait 0. Mais si l'ensemble de la portion d'image était remplie de 10, la valeur résultante serait aussi 0. En fait, on voit aisément que ce filtre fournit des valeurs plus élevées pour les portions d'image comportant au milieu une frontière horizontale séparant des valeurs élevées en haut et plus faibles en bas. L'important, bien sûr, c'est que les filtres peuvent être sensibles aux modifications d'intensité lumineuse plutôt qu'à leurs valeurs absolues et que, en raison de leur taille généralement beaucoup plus petite que celle des images complètes, ils peuvent se concentrer sur les modifications locales. Nous pouvons, bien sûr, concevoir un noyau de convolution affectant des valeurs élevées aux portions d'image comportant une ligne droite depuis en haut à gauche jusqu'en bas à droite, ou n'importe quoi d'autre.

Précédemment, nous avons présenté le filtre comme s'il était conçu par le programmeur pour repérer une propriété particulière de l'image, et c'est effectivement ce qui se passait avant l'émergence du filtrage convolutif profond. Mais ce qui fait la particularité des approches d'apprentissage profond, c'est que *les valeurs du filtre sont des paramètres du réseau de neurones : elles sont apprises durant la passe arrière*. Dans notre discussion sur le mode de fonctionnement de la convolution, il est plus simple d'ignorer cela : jusqu'à la section suivante, nous continuerons à présenter nos filtres comme « prédéfinis ».

Outre la convolution d'un filtre avec une *portion* d'image, on peut aussi parler de convolution d'un filtre avec une *image*. Ceci consiste à appliquer le filtre à de nombreuses portions de l'image. D'ordinaire, nous avons de nombreux filtres différents, chacun d'entre eux ayant pour objectif d'identifier une caractéristique particulière dans l'image. Cela fait, nous pouvons alors transmettre les valeurs de ces caractéristiques à une ou plusieurs couches entièrement connectées puis à softmax et par conséquent à la fonction de perte. Cette architecture est présentée à la figure 3.3.

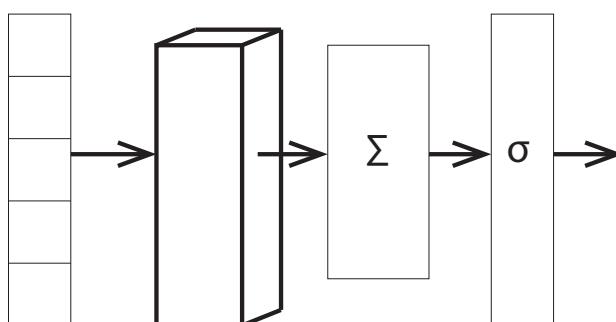


Figure 3.3 – Une architecture de reconnaissance d'image avec filtres de convolution

Ici nous représentons la couche de filtres de convolution sous forme d'une boîte à trois dimensions (parce qu'une banque de filtres est (au moins) un tenseur à trois dimensions, hauteur par largeur par nombre de filtres différents).

Vous avez peut-être remarqué le flou délibéré lorsque nous avons parlé de convoluer un filtre avec de « nombreuses » portions d'une image. Pour préciser cela, définissons d'abord le *pas* (en anglais, *stride*), c'est-à-dire la distance entre deux applications d'un filtre. Un pas de 2, par exemple, signifierait que nous appliquons un filtre un pixel sur deux. Pour rendre la chose encore plus précise, nous parlons à la fois de pas horizontal, s_h , et de pas vertical, s_v . L'idée est que quand nous balayons l'image horizontalement, nous appliquons le filtre tous les s_h pixels. Lorsque nous atteignons la fin d'une ligne, nous descendons verticalement de s_v lignes et répétons le processus. Lorsque nous appliquons un filtre en utilisant un pas de 2, nous l'appliquerons toujours à l'ensemble des pixels de la région (et non pas un pixel sur deux). Le pas définit uniquement l'endroit à partir duquel le filtre est appliqué la fois suivante.

Définissons maintenant ce que nous appelons « fin d'une ligne » lorsque nous appliquons un filtre. C'est ce qu'on fait en spécifiant le *type de marge* (en anglais, *padding*) pour la convolution. En informatique, ce terme désigne en général la marge interne d'un élément. Dans ce contexte, ce paramètre précise quel genre de complémentation on veut effectuer sur les bords de l'image. TF propose deux types de complémentation : *Valid* (valide) et *Same* (identique). Après avoir convolué les filtres avec une portion particulière de l'image, nous déplaçons s_h vers la droite. Il y a trois possibilités : (a) la portion d'image que le filtre va prendre en compte est à l'intérieur de l'image et donc nous continuons à travailler sur cette ligne ; (b) le pixel le plus à gauche de la prochaine portion d'image est à l'extérieur de l'image ; (c) le pixel le plus à gauche est dans l'image, mais le plus à droite est à l'extérieur de l'image. Si l'on a choisi une marge de type *Same*, l'algorithme s'arrête en (b), tandis qu'avec *Valid* il s'arrête en (c). Ce qui revient à dire que dans le cas *Same* l'image est complétée par des 0, alors que dans le cas *Valid* elle ne l'est pas (et certains pixels en bordure d'image peuvent être ignorés). La figure 3.4 présente le cas d'une image de 26 pixels de large complétée par des zéros, notre filtre étant de 4 pixels de large sur 2 pixels de haut, avec un pas de 1. Les pixels sont numérotés à partir de 0. Avec une marge de type *Valid*,

0	1		23	24	25	26	27
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0

Figure 3.4 – Fin de ligne avec marge de type *Valid* et *Same*

nous nous arrêtons après le pixel 23 (le 24^e pixel, puisque ceux-ci sont numérotés à partir de 0). C'est parce que notre pas nous conduirait au pixel 24, et que pour remplir un filtre de largeur 4 il faudrait aller jusqu'au pixel 27 (alors que l'image s'arrête au pixel 25). Par contre, avec une marge de type Same, nous effectuerions la convolution jusqu'au pixel 27 compris. Naturellement, nous faisons un choix analogue dans la direction verticale, lorsque nous atteignons le bas de l'image.

La valeur Same du paramètre de complétion (ou padding) indique que nous nous arrêterons après avoir complété l'image par des pixels imaginaires chaque fois que la partie gauche du filtre est à l'intérieur de l'image mais que la partie droite ne l'est pas. Dans TensorFlow, les pixels imaginaires ont une valeur nulle. Donc avec une marge de type Same nous devons éventuellement compléter la frontière de l'image avec des pixels imaginaires. Avec une complétion de type Valid, nous n'avons pas à compléter, car nous arrêtons la convolution avant qu'une quelconque partie du filtre empiète sur la bordure de l'image. Lorsqu'il y a complétion de type Same, les zéros sont répartis aussi équitablement que possible sur toutes les bordures.

Étant donné que nous en aurons besoin plus tard, voici dans le cas d'une complétion de type Same le nombre de portions d'image auxquelles sera appliquée la convolution horizontalement :

$$\lceil i_h/s_h \rceil \quad (3.2)$$

où $\lceil x \rceil$ est la fonction *ceiling* qui renvoie le plus petit entier $\geq x$. Pour comprendre pourquoi nous devons arrondir à l'entier supérieur (fonction ceiling), considérons le cas où la largeur de l'image est un nombre impair, 5 par exemple, et où le pas vaut 2. Si le filtre est de largeur 3, il est d'abord appliqué à la portion 0-2 dans le sens horizontal. Puis il se déplace de deux positions vers la droite et est appliqué en 2-4. Puis il se déplace vers la position 4, pour être appliqué en 4-6. La largeur de l'image étant 5, elle comporte des pixels de 0 à 4, mais ni pixel 5 ni pixel 6. Cependant, si nous avons choisi l'option Same, nous ajoutons deux 0 à la fin de la ligne pour permettre au filtre de travailler sur les positions 4-6, et le nombre total d'applications est donc 3. Si l'on n'ajoutait pas de pixels imaginaires (cas de l'option Valid), on aurait une itération de moins, ce qui reviendrait à arrondir à l'entier inférieur dans l'équation ci-dessus. Naturellement, le même raisonnement s'applique dans le sens vertical, ce qui nous donne $\lceil i_v/s_v \rceil$.

Pour une complétion de marge de type Valid, le nombre d'applications horizontales du filtre est :

$$\lfloor (i_h - f_h + 1)/s_h \rfloor \quad (3.3)$$

Si cette dernière équation ne vous paraît pas évidente, vérifiez que $i_h - f_h$ est bien le nombre des décalages réalisables (avant d'être à court d'espace) si le pas vaut 1. Mais le nombre d'applications du filtre est supérieur d'une unité au nombre de décalages.

Bien qu'elle utilise des pixels imaginaires, la complétion de type Same est plutôt appréciée car quand elle est combinée avec un pas de 1, elle permet d'avoir une taille

Chapitre 3 · Réseaux de neurones convolutifs

de sortie identique à la taille de l'image d'origine. On combine fréquemment de nombreuses couches de convolution, la sortie de chacune d'entre elles devenant l'entrée de la suivante. Quel que soit le pas, si la marge est de type Valid, on obtient toujours une sortie plus petite que l'entrée. Avec des couches de convolution successives, le résultat s'obtient en grignotant peu à peu à partir de l'extérieur.

Avant de passer à la programmation proprement dite, voyons comment la convolution affecte la façon dont nous représentons les images. Le cœur d'un réseau de neurones convolutif dans TF est une fonction de convolution à deux dimensions

```
tf.nn.conv2d(input, filters, strides, padding)
```

comportant optionnellement des arguments nommés supplémentaires que nous ignorons ici. Le 2d dans le nom spécifie que nous effectuons la convolution d'une image. Il existe aussi des versions 1d et 3d permettant de convoluer des objets à une dimension, tels qu'un signal audio, ou 3d, tels qu'un clip vidéo par exemple. Comme vous pouvez vous en douter, le premier argument est un lot d'images individuelles. Jusqu'ici, nous avons considéré qu'une image était un tableau 2D de valeurs numériques — chaque valeur représentant une intensité lumineuse. Si vous ajoutez le fait que nous avons un lot d'images, l'entrée est un tenseur à trois dimensions.

Mais `tf.nn.conv2d` s'attend à ce que chaque image individuelle soit un objet tridimensionnel dans lequel la dernière dimension est un vecteur de *canaux*. Comme nous l'avons mentionné précédemment, les images en couleur normales ont trois canaux — un pour le rouge, un pour le vert, un pour le bleu (RVB). À partir de maintenant, quand nous parlerons d'images, nous parlerons toujours d'un tableau 2D de pixels, mais chaque pixel sera une liste d'intensités. Cette liste comporte une valeur pour les images en noir et blanc, trois valeurs pour les images en couleur.

La même chose s'applique pour les filtres de convolution. Un filtre $m * n$ s'applique sur $m * n$ pixels, mais maintenant les pixels ainsi que le filtre peuvent avoir plusieurs canaux. Dans un projet quelque peu fantaisiste, nous créons un filtre pour identifier le bord horizontal de bouteilles de ketchup, filtre présenté à la figure 3.5. La ligne supérieure du filtre est activée au maximum lorsque la lumière est intense uniquement en rouge, et moins intense en bleu et vert. Les deux lignes suivantes requièrent moins de rouge (pour qu'il y ait du contraste) et davantage de bleu et de vert.

La figure 3.6 présente un petit exemple TF appliquant une convolution simple à une petite image inventée. Comme nous l'avons expliqué, le premier paramètre d'entrée de `conv2D` est un tenseur 4D, ici la constante `I`. Dans le commentaire qui précède la

$$\begin{array}{cccc} (1, -1, -1) & (1, -1, -1) & (1, -1, -1) & (1, -1, -1) \\ (-1, 1, 1) & (-1, 1, 1) & (-1, 1, 1) & (-1, 1, 1) \\ (-1, 1, 1) & (-1, 1, 1) & (-1, 1, 1) & (-1, 1, 1) \end{array}$$

Figure 3.5 – Filtre simple pour la détection des lignes horizontales d'une bouteille de ketchup

```

ii = [[ [[0],[0],[2],[2]],
       [[0],[0],[2],[2]],
       [[0],[0],[2],[2]],
       [[0],[0],[2],[2]] ]
''' ((0 0 2 2)
     (0 0 2 2)
     (0 0 2 2)
     (0 0 2 2))'''
I = tf.constant(ii, tf.float32)

ww = [ [[[[-1]],[[-1]],[[1]]],
       [[[[-1]],[[-1]],[[1]]]],
       [[[[-1]],[[-1]],[[1]]]] ]
'''((-1 -1 1)
    (-1 -1 1)
    (-1 -1 1))'''
W = tf.constant(ww, tf.float32)

C = tf.nn.conv2d(I, W, strides=[1, 1, 1, 1], padding='VALID')
sess = tf.Session()
print sess.run(C)
'''[[[ 6.]  [ 0.]]
 [[ 6.]  [ 0.]]]'''

```

Figure 3.6 – Simple exercice utilisant conv2D

déclaration de `I`, nous montrons à quoi cette variable ressemblerait en tant que tableau 2D, sans la dimension supplémentaire ajoutée par la taille de lot (qui vaut 1 ici) et la dimension associée au nombre de canaux (1 également ici). Le deuxième argument est un tenseur 4D de filtres, ici `W`, doté aussi d'un commentaire en montrant une version 2D, sans les dimensions supplémentaires du nombre de canaux et du nombre de filtres (1 chacun). Suit l'appel à `conv2D` avec des pas (ou `strides`) horizontaux et verticaux tous deux égaux à 1, et une marge (ou `padding`) de type `Valid`. Lorsque nous regardons le résultat, nous voyons qu'il est 4D [`batchSz(1)`, `height(2)`, `width(2)`, `channels(1)`]. Sa hauteur et sa largeur sont bien réduites par rapport à l'image initiale, ce à quoi on peut s'attendre avec une marge de type `Valid`, et par ailleurs le filtre est plutôt actif (avec une valeur de 6), comme on peut s'y attendre puisqu'il est conçu pour identifier des lignes verticales, ce qui est exactement ce qui apparaît sur l'image.

3.2 EXEMPLE SIMPLE DE CONVOLUTION EN TF

Voyons maintenant comment transformer le programme Mnist à propagation avant du chapitre 2 en un modèle de réseau de neurones convolutif. Le code que nous créons est donné à la figure 3.7.

Chapitre 3 · Réseaux de neurones convolutifs

```
1 image = tf.reshape(img, [100, 28, 28, 1])
2     # Transforme img en tenseur 4d
3 flts=tf.Variable(tf.truncated_normal([4,4,1,4], stddev=0.1))
4     # Crée les paramètres des filtres
5 convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "SAME")
6     # Crée le graphe faisant la convolution
7 convOut= tf.nn.relu(convOut)
8     # Ne pas oublier d'ajouter la non-linéarité
9 convOut=tf.reshape(convOut,[100, 784])
10    # Retour à 100 vecteurs image 1d
11 prbs = tf.nn.softmax(tf.matmul(convOut, W) + b)
```

Figure 3.7 – Code primaire permettant de transformer la figure 2.2 en réseau de neurones convolutif

Comme nous l'avons déjà remarqué, l'appel de fonction essentiel est `tf.nn.conv2d`. À la ligne 5 de la figure 3.7, nous voyons :

```
| convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "SAME") }
```

Examinons tour à tour chaque argument. Comme nous venons de le dire, `image` est un tenseur à quatre dimensions — dans ce cas un vecteur d'images tridimensionnelles. Nous choisissons une taille de lot de 100, par conséquent `tf.nn.conv2d` s'attend à recevoir 100 images 3D. Les fonctions qui lisent les données au chapitre 2 lisent en entrée des vecteurs d'images à une dimension (de longueur 784), c'est pourquoi la ligne 1 de la figure 3.7

```
| image = tf.reshape(img, [100, 28, 28, 1])
```

convertit l'entrée vers un format [100, 28, 28, 1] où le 1 final indique que nous n'avons qu'un seul canal d'entrée. `tf.reshape` fonctionne à peu près comme le `reshape` de Numpy.

L'argument d'appel suivant de `tf.nn.conv2d` à la ligne 5 est un pointeur vers les filtres à utiliser. Il s'agit aussi d'un tenseur 4D, cette fois de forme :

[hauteur, largeur, canaux, nombre]

Les paramètres des filtres sont créés à la ligne 3. Nous avons choisi des filtres de 4 par 4 ([4,4]), chaque pixel ayant un seul canal ([4,4,1]), et nous avons choisi de créer quatre filtres ([4,4,1,4]). Notez que la hauteur et la largeur de chaque filtre, ainsi que le nombre de ceux-ci, sont tous des hyperparamètres. Le nombre de canaux (1 dans ce cas) est déterminé par le nombre de canaux dans l'image, donc est fixé. Remarquez bien que nous avons enfin réalisé ce que nous avions promis au début : la ligne 3 crée les valeurs de filtre en tant que paramètres du réseau de neurones (initialisés

3.2 Exemple simple de convolution en TF

aléatoirement avec une moyenne de 0 et un écart type de 0.1), de sorte qu'ils sont appris par le modèle.

Le troisième argument de `t.f.nn.conv2d` est une liste de quatre entiers indiquant la taille du pas dans chacune des quatre dimensions de *input*. À la ligne 5 nous voyons que nous avons choisi des pas de 1, 2, 2 et 1. En pratique, le premier et le dernier valent presque toujours 1. De toute façon, il est difficile d'imaginer un cas où ils ne vaudraient pas 1. Après tout, la première dimension sépare les différentes images 3D du lot. Si le pas le long de cette dimension valait 2, nous sauterions une image sur deux ! De même, si le dernier pas était plus grand que 1, par exemple 2, et que nous ayons trois canaux de couleur, alors nous n'observerions que la lumière rouge et bleue, en sautant le vert. Par conséquent il est courant d'avoir pour valeurs de pas (1, 1, 1, 1), ou éventuellement (1, 2, 2, 1) si nous voulons convoluer sur des portions d'image espacées de deux pixels tant dans les directions horizontale que verticale. Voilà pourquoi, dans les présentations de `t.f.nn.conv2d`, il est souvent indiqué que les premier et dernier pas doivent être égaux à 1.

Le dernier argument est une chaîne de caractères égale à l'un des types de complétion reconnus par TF, comme par exemple `SAME`.

La sortie de `conv2d` ressemble beaucoup à son entrée. C'est aussi un tenseur 4D, dont la première dimension, comme pour l'entrée, est la taille du lot. Autrement dit, la sortie est un vecteur de résultats de convolution, un pour chaque image d'entrée. Les deux dimensions suivantes correspondent au nombre d'applications du filtre, horizontalement d'abord, verticalement ensuite ; ces valeurs peuvent être calculées à l'aide des équations 3.2 et 3.3. La dernière dimension du tenseur de sortie est le nombre de filtres ayant été convolus avec l'image. Nous avons dit précédemment que nous en utiliserions quatre. La forme de la sortie est donc :

[taille du lot, taille horizontale, taille verticale, nombre de filtres]

Ce qui donne dans notre cas (100, 14, 14, 4). Si nous considérons la sortie comme une sorte d'image, alors l'entrée est de 28 par 28 avec un canal, alors que la sortie est de 14 par 14 avec 4 canaux. Ceci signifie que dans les deux cas l'image est représentée par 784 nombres. Nous avons choisi cela délibérément pour que les choses restent similaires au chapitre 2, mais il n'y avait pas d'obligations à faire ainsi. Nous aurions pu par exemple choisir d'avoir 16 filtres différents et non pas 4, auquel cas nous aurions eu une image représentée par $14 * 14 * 16 = 3\,136$ nombres.

À la ligne 11, nous injectons ces 784 valeurs dans une couche entièrement connectée qui produit les logits pour chaque image, qui à leur tour sont injectés dans softmax, après quoi nous calculons la perte d'entropie croisée (non présentée sur la figure 3.7) et nous avons un réseau de neurones convolutif très simple pour les données Mnist. Le code a l'allure générale de celui de la figure 2.2. Par ailleurs, la ligne 7 ajoute de la non-linéarité entre la sortie de la convolution et l'entrée de la couche entièrement connectée. Ceci est important. Comme nous l'avons vu précédemment, sans fonction d'activation non linéaire entre les couches d'unités linéaires, on n'obtient aucune amélioration.

Les résultats de ce programme sont nettement meilleurs que ceux obtenus au chapitre 2 : 96 % voire un peu plus, selon l'initialisation aléatoire (contre 92 % pour la version à propagation avant). Le nombre de paramètres du modèle est pratiquement le même pour les deux versions. Dans les deux cas, la couche de propagation avant utilise 7 840 poids dans **W** et 100 biais dans **b** ($784 + 10$ poids par unité de la couche entièrement connectée, fois 10 unités). La convolution ajoute quatre filtres de convolution, chacun comportant $4 * 4$ poids, soit 64 paramètres de plus. C'est pourquoi nous fixons la taille de la sortie de convolution à 784. On pourrait penser que la qualité d'un réseau de neurones augmente à mesure qu'on lui ajoute des paramètres. Ici, cependant, le nombre de paramètres est essentiellement resté constant.

3.3 CONVOLUTION À PLUSIEURS NIVEAUX

Comme énoncé précédemment, nous pouvons encore améliorer l'exactitude en passant d'une couche de convolution à plusieurs. Dans cette section, nous construisons un modèle à deux couches.

Le point essentiel dans la convolution à plusieurs niveaux est celui que nous avons mentionné en passant lorsque nous avons examiné la sortie de `tf.conv2d` : elle a le même format que l'image d'entrée. Toutes deux sont constituées de vecteurs de la taille du lot, chaque élément étant une image 3D, chacune de celles-ci étant en fait une image 2D avec une dimension supplémentaire pour le nombre de canaux. Par conséquent la sortie d'une couche de convolution peut être l'entrée d'une seconde couche, et c'est exactement ce que l'on fait. Lorsque nous parlons de l'image provenant des données, la dernière dimension est le nombre de canaux de couleur. Lorsque nous parlons de la sortie de `conv2d`, nous disons que la dernière dimension est le nombre de filtres différents dans la couche de convolution. Le mot « filtre » est bien choisi ici. Après tout, pour récupérer uniquement de la lumière bleue à travers la lentille, nous plaçons effectivement un filtre coloré devant. Par conséquent, trois filtres nous permettent d'obtenir les images du spectre RVB. Maintenant nous obtenons des « images » dans un pseudo-spectre tel que le « spectre de ligne-frontière horizontale ». Il pourrait s'agir de l'image fictive produite par exemple par le filtre de la figure 3.1. De plus, de même que les filtres pour images en RVB ont des poids associés à chacun des trois spectres, la seconde couche de convolution possède des poids pour chaque canal obtenu en sortie de la première.

Le code permettant de transformer le réseau de neurones à propagation avant sur les données Mnist en un modèle convolutif à deux couches est donné à la figure 3.8. Les lignes 1-4 reproduisent les premières lignes de la figure 3.7 sauf qu'à la ligne 2 nous accroissons à 16 le nombre de filtres dans la première couche de convolution (contre 4 dans la précédente version). La ligne 2 crée les filtres `f1ts2` de la seconde couche de convolution. Notez que nous en avons créé 32. Ceci se reflète à la ligne 5, où les valeurs des 32 filtres deviennent les 32 valeurs des canaux d'entrée de la seconde couche de convolution. Lorsque nous linéarisons ces valeurs de sortie

```

1 image = tf.reshape(img, [100, 28, 28, 1])
2 flts=tf.Variable(tf.normal([4, 4, 1, 16], stddev=0.1))
3 convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "SAME")
4 convOut= tf.nn.relu(convOut)
5 flts2=tf.Variable(tf.normal([2, 2, 16, 32], stddev=0.1))
6 convOut2 = tf.nn.conv2d(convOut, flts2, [1, 2, 2, 1], "SAME")
7 convOut2 = tf.reshape(convOut2, [100, 1568])
8 W = tf.Variable(tf.normal([1568,10],stddev=0.1))
9 prbs = tf.nn.softmax(tf.matmul(convOut2, W) + b)

```

Figure 3.8 – Code de base permettant de transformer la figure 2.2 en un réseau de neurones convolutif à deux couches

à la ligne 7, il y en a $784 * 4$. Rappelez-vous que nous avons commencé avec 784 pixels, et que chaque couche de convolution utilise un pas de 2, tant horizontalement que verticalement. Par conséquent l'image 3D obtenue après la première convolution avait pour dimensions (14, 14, 16). La seconde convolution avait aussi un pas de 2 sur l'image $14 * 14$ et possédait 32 canaux, impliquant une sortie de dimensions [100, 7, 7, 32] et le fait que la version linéarisée d'une simple image à la ligne 7 a $7 * 7 * 32 = 1\,568$ valeurs scalaires, ce qui est aussi la première dimension de w qui transforme les valeurs de cette image en 10 logits.

Prenez du recul par rapport à ces détails, et remarquez le flux d'ensemble du modèle. Nous partons d'une image $28 * 28$. À la fin, nous avons une « image » $7 * 7$, mais nous avons aussi 32 valeurs de filtres différentes en chaque point du tableau 2D. Autrement dit, à la fin nous avons divisé notre image en 49 portions, chacune d'entre elles comportant initialement $4 * 4$ pixels et étant finalement caractérisée par 32 valeurs de filtres. Étant donné que ceci améliore les performances, nous sommes en droit de supposer que ces valeurs disent des choses importantes à propos de ce qui se passe dans la portion $4 * 4$ correspondante.

De fait, cela semble être le cas. Alors qu'à première vue les valeurs obtenues dans les filtres peuvent être déconcertantes, une étude approfondie peut révéler, au moins dans les premiers niveaux, une certaine logique dans leur construction. La figure 3.9 présente les poids $4 * 4$ de quatre des huit filtres de convolution de premier niveau résultant de l'apprentissage lors d'une exécution du code de la figure 3.7. Vous pouvez consacrer quelques secondes à tenter de voir si vous pouvez deviner ce qu'ils recherchent. Pour certains, vous y arriverez peut-être. Certains autres n'ont pas grand sens à mes yeux. Cependant, effectuer une corrélation croisée avec la figure 3.10 devrait vous aider. La figure 3.10 a été créée en imprimant, pour notre désormais classique image d'un « 7 », le numéro du filtre ayant la plus haute valeur pour chacun des $14 * 14$ points de l'image après la première couche de convolution. Assez rapidement, on a l'impression de voir un « 7 » émerger d'un brouillard de zéros, donc le filtre 0 est associé à une région ne comportant que des zéros. Nous pouvons alors noter que la partie droite de la diagonale du « 7 » n'est pratiquement composée que de 7, alors que le bas de la partie horizontale de l'image correspond à des 1. Observez

Chapitre 3 · Réseaux de neurones convolutifs

```
-0.152168 -0.366335 -0.464648 -0.531652  
0.0182653 -0.00621072 -0.306908 -0.377731  
0.482902 0.581139 0.284986 0.0330535  
0.193956 0.407183 0.325831 0.284819  
  
0.0407645 0.279199 0.515349 0.494845  
0.140978 0.65135 0.877393 0.762161  
0.131708 0.638992 0.413673 0.375259  
0.142061 0.293672 0.166572 -0.113099  
  
0.0243751 0.206352 0.0310258 -0.339092  
0.633558 0.756878 0.681229 0.243193  
0.894955 0.91901 0.745439 0.452919  
0.543136 0.519047 0.203468 0.0879601  
  
0.334673 0.252503 -0.339239 -0.646544  
0.360862 0.405571 -0.117221 -0.498999  
0.520955 0.532992 0.220457 0.000427301  
0.464468 0.486983 0.233783 0.101901
```

Figure 3.9 – Filtres 0, 1, 2 et 7 des huit filtres créés lors d'une exécution du réseau de neurones convolutif à deux couches

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 0 5 2 2 2 2 2 2 2 2 2 2 2 0 0  
0 0 1 1 4 4 4 4 4 2 2 2 2 0 0  
0 0 1 1 1 1 1 1 1 1 2 7 0 0  
0 0 0 0 0 0 0 5 1 4 2 7 0 0  
0 0 0 0 0 0 0 5 1 2 7 0 0 0 0  
0 0 0 0 0 5 1 4 2 7 0 0 0 0 0  
0 0 0 0 0 5 2 1 2 7 0 0 0 0 0  
0 0 0 0 0 5 1 4 2 0 0 0 0 0 0  
0 0 0 0 5 1 4 2 7 0 0 0 0 0 0  
0 0 0 0 2 1 2 2 0 0 0 0 0 0 0 0  
0 0 0 0 1 1 1 7 0 0 0 0 0 0 0 0
```

Figure 3.10 – Caractéristique la plus active pour chacun des 14 * 14 points de la couche 1 après traitement de la figure 1.1

0	0	0	0	0	0	0
17	11	31	17	17	16	16
6	16	12	6	6	5	5
17	17	17	5	24	5	10
0	0	11	26	3	5	0
0	17	11	24	5	10	0
0	6	24	8	5	0	0

Figure 3.11 – Caractéristique la plus active pour chacune des portions 7×7 de la couche 2 après traitement de la figure 1.1

à nouveau les valeurs des filtres. Pour moi, les 1, 2 et 7 semblent s'accorder aux résultats de la figure 3.9. Inversement, il n'y a rien dans le filtre 0 pour suggérer un blanc. Cependant, cela aussi paraît sensé. Nous avons utilisé la fonction arg-max de Numpy qui renvoie la position du plus grand nombre d'une liste. Dans une région blanche, toutes les valeurs des pixels sont nulles, donc tous les filtres renvoient 0. La fonction arg-max renvoie la première valeur dans le cas où toutes les valeurs sont égales, c'est ce à quoi nous devons nous attendre.

La figure 3.11 est similaire à la figure 3.10 à ceci près qu'elle montre les filtres les plus actifs dans la couche 2 du modèle. C'est moins interprétable que pour la couche 1. Il y a différentes explications à cela. Nous l'incluons surtout parce que la première couche de convolution des illustrations est bien plus interprétable que la plupart, mais il ne faudrait pas croire que ce que nous avons vu pour la couche 1 est habituel.

3.4 PRÉSENTATION DÉTAILLÉE DE LA CONVOLUTION

3.4.1 Biais

Nous pouvons aussi avoir des biais avec nos noyaux de convolution. Nous n'en avons pas parlé jusqu'ici, car ce n'est que dans le dernier exemple qu'on appliquait des filtres multiples à chaque portion : nous avons spécifié 16 filtres différents à la ligne 2 de la figure 3.8. Un biais peut faire qu'un programme accorde plus ou moins de poids à un canal de filtre plutôt qu'à un autre en ajoutant une valeur différente à la sortie de convolution de chaque canal. Par conséquent, le nombre de variables de biais d'une couche de convolution particulière est égal au nombre de canaux de sortie. Ainsi, à la figure 3.8 vous pouvez ajouter des biais à la première couche de convolution en rajoutant entre les lignes 3 et 4 :

```
bias = tf.Variable(tf.zeros [16])
convOut += bias
```

Chapitre 3 · Réseaux de neurones convolutifs

La diffusion (en anglais, *broadcasting*) est implicite. Sachant que `convOut` a pour forme [100, 14, 14, 16] et que `bias` a pour forme [16], l'addition en crée implicitement [100, 14, 14] exemplaires.

3.4.2 Couches avec convolution

Nous avons vu à la section 2.4.4 comment un composant standard des architectures à réseaux de neurones, les couches entièrement connectées, pouvait être décrit de manière concise en utilisant `layers`. Il y a des fonctions équivalentes pour les couches de convolution :

```
tf.contrib.layers.conv2d(inpt, numFlts, fltDim, strides, pad)
```

admettant également des arguments nommés optionnels. Par exemple, les lignes 2 à 4 de la figure 3.8 peuvent être remplacées par :

```
convOut = layers.conv2d(image, 16, [4, 4], 2, "Same")
```

`convOut` est un pointeur vers le résultat de la convolution. Comme précédemment, nous avons créé 16 noyaux différents, chacun de dimensions 4×4 . Le pas est de 2 dans chaque direction, et nous utilisons une marge de type `Same`. Ce n'est pas tout à fait identique à la version sans `layers` car `layers.conv2d` suppose par défaut que vous voulez des biais. Si vous n'en voulez pas, il faut le préciser en donnant la bonne valeur à l'argument nommé correspondant `use_bias=False`.

3.4.3 Pooling

Comme vous pouvez vous y attendre pour des images de plus grande taille (p. ex. $1\,000 \times 1\,000$ pixels), la réduction de taille obtenue entre l'image d'origine et les valeurs transmises à une couche entièrement connectée, suivie par softmax à la fin, est bien plus importante. Il existe des fonctions TF pouvant vous aider à gérer cette réduction. Remarquez que dans notre programme la réduction a été obtenue parce que les pas utilisés dans la convolution faisaient qu'on n'examinait qu'une portion d'image sur deux. Au lieu de cela, nous aurions pu faire ce qui suit :

```
convOut = tf.nn.conv2d(image, flts, [1, 1, 1, 1], "SAME")
convOut = tf.nn.max_pool(convOut, [1, 2, 2, 1], [1, 2, 2, 1], "SAME")
```

Ces deux lignes ont pour but de remplacer la ligne 3 de la figure 3.8. Au lieu d'une convolution avec un pas de 2, nous avons d'abord appliqué la convolution avec un pas de 1. Par conséquent, `convOut` est de forme [batchSz, 28, 28, 1] — aucune réduction de taille d'image. La ligne suivante nous donne une réduction de taille d'image exactement égale à celle produite par le pas de 2 que nous avions utilisé à l'origine.

La fonction essentielle ici, `max_pool`, détermine la valeur maximum d'un filtre sur une région de l'image. Elle admet quatre arguments, trois d'entre eux étant les mêmes

que pour `conv2d`. Le premier est notre classique tenseur 4D d'images, le troisième les pas, et le dernier le type de marge (padding). Dans le cas ci-dessus, `max_pool` examine `convOut`, la sortie 4D de la première convolution. Il le fait en utilisant des pas [1, 2, 2, 1]. Le premier élément de la liste indique qu'il faut examiner chaque image du lot, et le dernier qu'il faut examiner chaque canal. Les deux 2 indiquent qu'il faut se déplacer de deux unités avant de renouveler l'opération, et qu'il faut le faire dans le sens horizontal et dans le sens vertical. Le deuxième argument spécifie la taille de la zone sur laquelle il faut rechercher le maximum. Comme d'habitude, les 1 au début et à la fin s'imposent, tandis que les deux 2 du milieu spécifient que nous allons prendre chaque maximum sur une zone 2×2 de `convOut`.

La figure 3.12 met en contraste les deux manières différentes de réduire la dimension d'un facteur 4 dans notre programme `Mnist`, bien qu'ici nous ne le fassions que sur une image 4×4 (les valeurs numériques sont fictives). En haut, nous avons appliqué le filtre avec un pas de 2 (marge de type `Same`) et avons immédiatement obtenu le tableau 2×2 des valeurs de filtre. En bas, nous avons appliqué le filtre avec un pas de 1, ce qui crée un tableau 4×4 de valeurs. Puis, pour chacune des portions 2×2 , nous produisons en sortie la plus haute valeur, ce qui nous donne le tableau final en bas à droite de la figure.

<table border="1"><tr><td>-1</td><td>2</td><td>-3</td><td>4</td></tr><tr><td>-4</td><td>-3</td><td>2</td><td>1</td></tr><tr><td>0</td><td>-1</td><td>3</td><td>2</td></tr><tr><td>3</td><td>-2</td><td>1</td><td>0</td></tr></table>	-1	2	-3	4	-4	-3	2	1	0	-1	3	2	3	-2	1	0	\rightarrow	<table border="1"><tr><td>5</td><td>6</td></tr><tr><td>3</td><td>4</td></tr></table>	5	6	3	4												
-1	2	-3	4																															
-4	-3	2	1																															
0	-1	3	2																															
3	-2	1	0																															
5	6																																	
3	4																																	
<table border="1"><tr><td>-1</td><td>2</td><td>-3</td><td>4</td></tr><tr><td>-4</td><td>-3</td><td>2</td><td>1</td></tr><tr><td>0</td><td>-1</td><td>3</td><td>2</td></tr><tr><td>3</td><td>-2</td><td>1</td><td>0</td></tr></table>	-1	2	-3	4	-4	-3	2	1	0	-1	3	2	3	-2	1	0	\rightarrow	<table border="1"><tr><td>5</td><td>4</td><td>6</td><td>7</td></tr><tr><td>2</td><td>3</td><td>5</td><td>6</td></tr><tr><td>3</td><td>4</td><td>4</td><td>4</td></tr><tr><td>2</td><td>2</td><td>3</td><td>4</td></tr></table> \rightarrow	5	4	6	7	2	3	5	6	3	4	4	4	2	2	3	4
-1	2	-3	4																															
-4	-3	2	1																															
0	-1	3	2																															
3	-2	1	0																															
5	4	6	7																															
2	3	5	6																															
3	4	4	4																															
2	2	3	4																															
		<table border="1"><tr><td>5</td><td>7</td></tr><tr><td>4</td><td>4</td></tr></table>	5	7	4	4																												
5	7																																	
4	4																																	

Figure 3.12 – Réduction de dimension par un facteur 4, avec et sans `max_pool`

Avant de passer à la suite, notez qu'il existe aussi `avg_pool` qui travaille de manière identique à `max_pool` excepté que la valeur pour chaque région est la moyenne des valeurs individuelles et non le maximum.

3.5 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

Le premier article présentant l'apprentissage des noyaux convolutifs au travers des réseaux de neurones et de la propagation arrière a été publié par Yann LeCun *et al.* [LBD⁺90], même si une exploration beaucoup plus complète du sujet publiée plus tardivement également par LeCun *et al.* [LBBH98] constitue définitivement

Chapitre 3 · Réseaux de neurones convolutifs

la référence. J'ai puisé une partie de mes connaissances en matière de réseaux de neurones convolutifs dans le tutoriel de Google à propos des chiffres Mnist [Ten17b].

Si vous vous intéressez aux réseaux de neurones sachant reconnaître des images et souhaitez travailler sur un nouveau projet, je vous recommande le jeu de données CIFAR-10 de l'Institut canadien de recherche avancée [KH09]. Il s'agit là encore d'une tâche d'identification d'images en 10 classes, mais les objets à reconnaître sont plus compliqués (avion, chat, grenouille), les images sont en couleur, l'arrière-plan peut être compliqué et les objets à classer ne sont pas toujours au centre de l'image. Les images sont également de taille supérieure : [32, 32, 3]. Le jeu de données peut être téléchargé à partir de [Kri09]. Le nombre total d'images (60 000) est voisin de celui du Mnist, par conséquent le volume total des données reste gérable. Il existe aussi un tutoriel Google en ligne sur la façon de construire un réseau de neurones pour cette tâche [Ten17a].

Si vous êtes vraiment ambitieux, vous pouvez tenter de travailler sur le jeu de données du Challenge ImageNet de reconnaissance visuelle à grande échelle (*Imagenet Large Scale Visual Recognition Challenge* ou ILSVRC). C'est beaucoup plus difficile. Il y a 1 000 types d'images, parmi lesquelles des choses aussi ordinaires que des frites ou de la purée. C'est depuis une dizaine d'années le jeu de données utilisé dans une compétition annuelle rassemblant les meilleurs spécialistes en vision par ordinateur. La grande année des réseaux de neurones a été 2012, lorsque le programme de Deep Learning *Alexnet* a dominé la compétition — c'était la première fois qu'un programme à base de réseaux de neurones l'emportait. Le programme développé par Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton [KSH12] a obtenu un *top-5 score* de 15.5 %, c'est-à-dire que dans seulement 15.5 % des cas l'étiquette correcte ne figurait pas parmi les cinq réponses auxquelles le programme avait affecté la plus forte probabilité. Le concurrent arrivé à la deuxième place avait obtenu un score de 26.2 %. Depuis 2012, tous ceux qui arrivent premiers utilisent des réseaux de neurones.

2012	15.5
2013	11.2
2014	6.7
2015	3.6
Humain	5-10

Ici, la ligne « humain » indique que les personnes effectuant la même tâche obtiennent un score de 5 à 10 % selon leur formation.

Les informations ci-dessus sont couramment présentées dans des tableaux ou graphiques pour expliquer l'impact des réseaux de neurones sur l'intelligence artificielle au cours de la décennie écoulée.

Exercices

3.1 (a) Concevez un noyau 3×3 qui détecte des lignes verticales dans une image en noir et blanc et renvoie la valeur 8 lorsqu'on l'applique à la partie supérieure gauche de l'image de la figure 3.2. Il doit renvoyer 0 si tous les pixels de la portion d'image sont de même intensité. (b) Concevez un autre noyau similaire.

3.2 Dans notre discussion à propos de l'équation 3.2, nous avons mentionné que la taille du filtre de convolution n'avait pas d'impact sur le nombre de ses applications lorsqu'on utilise une marge de type Same. Expliquez pourquoi.

3.3 Dans notre discussion à propos de la complétion (ou padding), nous avons dit que le type Valid conduit *toujours* à une image de sortie dont les dimensions 2D sont inférieures à celles d'entrée. À proprement parler, ce n'est pas toujours vrai. Décrivez le cas (plutôt sans intérêt) où cette affirmation est fausse.

3.4 Supposons que l'entrée d'un réseau de neurones convolutif soit une image 32×32 en couleur. Nous voulons lui appliquer huit filtres de convolution, tous de forme 5×5 . Nous utilisons une marge de type Valid et un pas de 2 tant verticalement qu'horizontalement. (a) Quelle est la forme de la variable dans laquelle nous rangeons les valeurs des filtres ? (b) Quelle est la forme de la sortie de `tf.nn.conv2d` ?

3.5 Expliquez ce que le code suivant fait différemment du code presque identique au début de la section 3.4.3 :

```
convOut = tf.nn.conv2d(image, flts, [1,1,1,1], "SAME")
convOut = tf.nn.maxpool(convOut, [1,2,2,1], [1,1,1,1], "SAME")
```

En particulier, pour des valeurs arbitraires de `image` et `flts`, `convOut` a-t-il la même forme dans les deux cas ? A-t-il nécessairement les mêmes valeurs ? L'un des ensembles de valeurs est-il un sous-ensemble de l'autre ? Dans chacun des cas, pourquoi ?

3.6 (a) Combien de variables sont créées lorsque nous exécutons la commande `layers` suivante ?

```
layers.conv2d(image, 10, [2,4], 2, "Same", use_bias=False)
```

Supposons qu'`image` ait pour forme $[100, 8, 8, 3]$. Laquelle de ces valeurs de forme n'a pas d'impact sur la réponse ? (b) Combien n'ont pas d'importance si `use_bias` a pour valeur `True` (l'option par défaut) ?

PLONGEMENTS DE MOTS ET RÉSEAUX DE NEURONES RÉCURRENTS

4.1 PLONGEMENT DE MOTS POUR MODÈLES LINGUISTIQUES

Un *modèle linguistique* est une distribution de probabilité sur toutes les chaînes d'une langue. De prime abord, c'est une notion un peu difficile à appréhender. Considérez par exemple l'expression « de prime abord ». Il y a de bonnes chances que vous ne rencontriez pas souvent cette expression, et à moins de relire ce livre, vous ne l'y reverrez pas une seconde fois. Quelle que soit sa probabilité, elle doit être assez faible. Comparez-lui maintenant l'expression obtenue en inversant l'ordre des mots. C'est encore très largement moins probable. Donc, certaines chaînes de mots peuvent être plus ou moins probables. De plus, un programme souhaitant traduire, par exemple, du polonais en anglais doit être capable de faire la distinction entre des phrases qui semblent être en bon anglais et d'autres qui ne le sont pas. Un modèle linguistique est une formalisation de cette idée.

Pour appréhender encore mieux ce concept, vous pouvez découper les chaînes en mots individuels puis demander quelle est la probabilité du mot suivant compte tenu des mots qui l'ont précédé ? Soit $E_{1,n} = (E_1 \dots E_n)$ une séquence de n variables aléatoires représentant une chaîne de n mots et $e_{1,n}$ une valeur candidate. Par exemple, si n valait 6 et que $e_{1,6}$ était « Nous vivons dans un monde petit », nous pourrions utiliser la règle de la chaîne de probabilités pour obtenir :

$$\begin{aligned} P(\text{Nous vivons dans un monde petit}) \\ = P(\text{Nous})P(\text{vivons}|\text{Nous})P(\text{dans}|\text{Nous vivons}) \dots \end{aligned} \quad (4.1)$$

Plus généralement :

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{1,j-1} = e_{1,j-1}) \quad (4.2)$$

Avant de poursuivre, revenons à ce que nous avons appelé « partager une chaîne en une séquence de mots ». C'est ce qu'on appelle le *découpage en unités lexicales* (en anglais, *tokenization*), et si ce livre traitait de la compréhension d'un texte, nous pourrions consacrer un chapitre entier à ce découpage en unités lexicales. Cependant, notre objectif est autre, c'est pourquoi nous définirons pour nos besoins un « mot » comme une séquence quelconque de caractères comprise entre deux caractères d'espacement (nous considérerons le saut de ligne comme un caractère d'espacement).

Notez que ceci signifie par exemple que « 1066 » est un mot dans la phrase « Les Normands ont envahi l'Angleterre en 1066. ». En fait, ceci est faux si l'on se réfère à notre définition d'un « mot » : le mot qui apparaît dans la phrase ci-dessus est « 1066. », c'est-à-dire « 1066 » suivi d'un point. Par conséquent, nous supposons également que la ponctuation (points, virgules, deux-points, etc.) est détachée des mots, de sorte que le point final devient un mot en lui-même, séparé du mot « 1066 » qui le précède (vous commencez peut-être à comprendre comment on peut consacrer un chapitre entier à ce découpage).

Par ailleurs, nous allons limiter notre vocabulaire français, par exemple à 10 000 mots différents. Nous notons V notre vocabulaire et $|V|$ sa taille. Ceci est nécessaire car, en raison de la définition d'un « mot » que nous avons donnée ci-dessus, nous pouvons nous attendre à trouver des mots dans nos jeux de développement et de test qui n'apparaissent pas dans le jeu d'entraînement (comme par exemple « 7221 » dans la phrase « Cassis compte 7221 habitants. »). Nous résolvons ceci en remplaçant tous les mots qui ne sont pas dans V (appelés *mots inconnus*) par un mot spécial « *UNK* ». De sorte que cette phrase apparaîtra désormais dans notre corpus sous la forme : « Cassis compte *UNK* habitants. »

Les données que nous utilisons dans ce chapitre sont connues sous l'appellation *Penn Treebank Corpus*, ou PTB en bref. Le PTB est composé d'environ 1 million de mots d'articles de presse du *Wall Street Journal*. Il a été décomposé en unités lexicales, mais pas « UNKé », ce qui fait que le vocabulaire comporte près de 50 000 mots. On l'appelle « treebank » (banque d'arbres) parce que toutes les phrases ont été transformées en arbres représentant leur structure grammaticale. Ici nous ignorons les arbres, car nous ne nous intéressons qu'aux mots. Nous remplaçons également tous les mots ayant 10 occurrences ou moins par « *UNK* ».

Ceci étant dit, revenons à l'équation 4.2. Si nous avions une très grande quantité de texte, nous pourrions être capables d'estimer les deux ou trois premières probabilités du membre de droite simplement en comptant le nombre de fois que nous rencontrons « Nous vivons » et le nombre de fois que « dans » apparaît juste derrière, puis diviser la seconde par la première (c'est-à-dire utiliser l'estimateur du maximum de vraisemblance) pour obtenir par exemple une estimation de $P(\text{dans}|\text{Nous vivons})$. Mais quand n augmente, cela devient rapidement impossible par manque d'occurrences dans le corpus d'entraînement de séquences de 15 mots par exemple.

Une réponse classique à ce problème consiste à supposer que la probabilité du mot suivant ne dépend que d'un ou deux mots le précédent immédiatement, de sorte que nous pouvons ignorer tous les mots antérieurs à ceux-ci lorsque nous estimons la probabilité du mot suivant. La version où nous supposons que chaque mot ne dépend que du précédent ressemble à ceci :

$$P(E_{1,n} = e_{1,n}) = P(E_1 = e_1) \prod_{j=2}^{j=n} P(E_j = e_j | E_{j-1} = e_{j-1}) \quad (4.3)$$

C'est ce qu'on appelle un *modèle bigramme*, où « bigramme » signifie « deux mots ». Il est appelé ainsi parce que chaque probabilité dépend uniquement d'une séquence de deux mots. Nous pouvons simplifier cette équation si nous plaçons un mot imaginaire STOP au début du corpus, puis après chaque phrase. On appelle cela le *remplissage de phrase* (en anglais, *sentence padding*). Par conséquent, si le premier STOP est e_0 , l'équation 4.3 devient :

$$P(E_{1,n} = e_{1,n}) = \prod_{j=1}^{j=n} P(E_j = e_j | E_{j-1} = e_{j-1}) \quad (4.4)$$

Dorénavant, nous supposons que dans tous nos corpus linguistiques le remplissage de phrases a été effectué. Par conséquent, excepté pour le premier STOP, notre modèle linguistique prédit tous les STOP de même que tous les mots réels.

Avec les simplifications que nous avons mises en place, il devrait être clair qu'il est facile de créer un mauvais modèle linguistique. Si par exemple $|V| = 10\,000$, nous pouvons définir la probabilité de n'importe quel mot arrivant après n'importe quel autre comme $\frac{1}{10\,000}$. Mais ce que nous voulons, bien sûr, c'est un bon modèle — un dans lequel si le dernier mot est « le », la distribution affecte une très faible priorité à « un » et une priorité beaucoup plus forte à « chat » par exemple. Pour ce faire, nous utilisons l'apprentissage profond. C'est-à-dire que nous donnons au réseau profond un mot w_i et espérons en sortie une distribution de probabilité raisonnable sur les mots suivants possibles.

Pour commencer, nous devons d'une manière ou d'une autre transformer les mots en des entités que des réseaux profonds savent manipuler, c'est-à-dire en nombres à virgule flottante. La solution désormais standard consiste à associer à chaque mot un vecteur de nombres flottants. Ces vecteurs sont appelés *plongements de mots* (en anglais, *word embeddings*). Pour chaque mot, nous initialisons son plongement sous forme d'un vecteur de e nombres flottants, où e est un hyperparamètre du système. Un e égal à 20 est petit, 100 courant, et 1 000 parfois utilisé. En pratique, nous le faisons en deux étapes. Tout d'abord, chaque mot du vocabulaire V a un indice unique (un entier) de 0 à $|V| - 1$. Nous avons alors un tableau \mathbf{E} de dimensions $|V| * e$. \mathbf{E} contient tous les plongements de mots de sorte que si, par exemple, « le » a pour indice 5, la cinquième ligne de \mathbf{E} est le plongement de « le ».

Considérant ceci, un très simple réseau à propagation avant permettant d'estimer la probabilité du mot suivant est présenté à la figure 4.1. Le petit carré à gauche est la donnée d'entrée dans le réseau — l'indice du mot courant, e_i . À droite figurent les probabilités affectées aux mots suivants possibles e_{i+1} , et la fonction de perte d'entropie croisée est $-\ln P(e_c)$, l'opposé du logarithme naturel de la probabilité affectée au mot suivant correct. En revenant à gauche, le mot courant est immédiatement transformé en son plongement par la *couche de plongement* qui consulte la e_i -ième ligne de \mathbf{E} . À partir de là, toutes les opérations du réseau de neurones s'effectuent sur le plongement de mot.

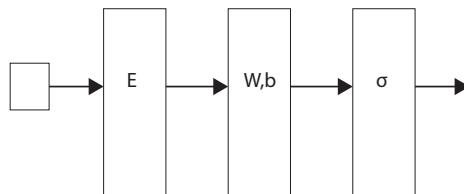


Figure 4.1 – Réseau à propagation avant pour la modélisation linguistique

Un point essentiel est que **E** est un paramètre du modèle. Ce qui veut dire qu’initialement les nombres dans **E** sont aléatoires de moyenne zéro et d’écart type petit, et leurs valeurs sont modifiées au fur et à mesure de la descente de gradient stochastique. Plus généralement, TensorFlow débute la passe arrière avec la fonction de perte et travaille en reculant, recherchant tous les paramètres qui affectent la perte. **E** est un tel paramètre, donc TF le modifie. Ce qu’il y a d’étonnant là-dedans, en dehors du fait que le processus converge vers une solution stable, c’est que la solution a la propriété que les mots se comportant de manière similaire aboutissent à des plongements qui sont proches les uns des autres. Donc si e (taille du vecteur de plongement) vaut 30 par exemple, alors les adverbes « presque » et « environ » pointent à peu près dans la même direction dans un espace à 30 dimensions, et aucun n’est très proche de « ordinateur » (qui est plus proche de « machine »).

Si l’on y réfléchit à deux fois, cependant, ce n’est peut-être pas si étonnant. Examinons plus attentivement ce qu’il advient des plongements lorsque nous tentons de minimiser la perte. Comme dit précédemment, la fonction de perte est la perte d’entropie croisée. Initialement, toutes les valeurs des logits sont presque égales puisque tous les paramètres du modèle sont presque égaux (et proches de zéro).

Supposons maintenant que nous ayons déjà effectué un entraînement sur la paire de mots « déclare que ». Ceci a provoqué une modification des paramètres du modèle de telle sorte que le plongement du mot « déclare » conduit à une plus forte probabilité que le mot « que » apparaisse juste derrière. Considérons maintenant la première fois où le modèle rencontre le mot « rappelle », et supposons de plus qu’il soit aussi suivi du mot « que ». Une façon de modifier les paramètres pour que « rappelle » prédise « que » avec une probabilité plus élevée consiste à ce que son plongement devienne plus semblable à celui de « déclare » puisque lui aussi peut prédire « que » comme mot suivant. C’est ce qui se passe en fait. Plus généralement, deux mots qui sont suivis de mots similaires produisent des plongement similaires.

La figure 4.2 montre ce qui se passe lorsque nous exécutons notre modèle sur 1 million de mots environ, avec un vocabulaire d’environ 7 500 mots et une taille de plongement de 30. La *similarité cosinus* de deux vecteurs est une mesure classique de la proximité entre deux vecteurs. Dans le cas de vecteurs à deux dimensions, elle

4.1 Plongement de mots pour modèles linguistiques

correspond à la fonction cosinus habituelle et vaut 1 si les vecteurs pointent dans la même direction, 0 s'ils sont orthogonaux et -1 s'ils sont de directions opposées. Pour des vecteurs de dimensions arbitraires, la similarité cosinus s'obtient par la formule suivante :

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{(\sqrt{\sum_{i=1}^n x_i^2})(\sqrt{\sum_{i=1}^n y_i^2})} \quad (4.5)$$

La figure 4.2 présente cinq paires de mots similaires numérotés de zéro à neuf. Nous calculons pour chaque mot sa similarité cosinus avec tous les mots qui le précédent. Par conséquent, nous nous attendons à ce que tous les mots d'indice impair aient la plus forte similarité avec le mot qui les précède immédiatement, et c'est effectivement le cas. Nous nous attendons aussi à ce que tous les mots d'indice pair (les premiers de chaque paire de mots similaires) n'aient qu'une faible similarité avec chacun des mots précédents. Dans l'ensemble, c'est vrai également.

Étant donné que la similarité de plongement reflète dans une large mesure la similarité de sens, les plongements ont été étudiés de multiples manières en tant que moyen de quantifier le sens, et nous savons désormais comment améliorer sensiblement ce résultat. Le principal facteur est simplement le nombre de mots que nous utilisons pour l'apprentissage, même s'il existe d'autres architectures qui s'avèrent utiles. Cependant la plupart des méthodes souffrent de limitations similaires. Par exemple, elles sont souvent incapables de distinguer les synonymes des antonymes (« sous » et « sur » sont par exemple des antonymes). Rappelez-vous qu'un modèle de langage cherche à deviner le mot suivant, et que par conséquent des mots suivis de mots similaires conduisent à des plongements similaires, ce qui est très souvent le cas des antonymes. Par ailleurs, il est beaucoup plus difficile d'obtenir de bons modèles pour des plongements de phrases que pour de simples mots.

Indice du mot	Mot	Similarité cosinus	Plus grande similarité
0	sous		
1	sur	0.362	0
2	le	-0.160	0
3	un	0.127	2
4	rappelle	0.479	1
5	déclare	0.553	4
6	règles	-0.066	4
7	lois	0.523	6
8	ordinateur	0.249	2
9	machine	0.333	8

Figure 4.2 – Dix mots, plus grande similarité cosinus avec les mots précédents, et indices du mot ayant la plus haute similarité

4.2 CONSTRUCTION DE MODÈLES LINGUISTIQUES À PROPAGATION AVANT

Construisons maintenant un programme TF pour calculer les probabilités de bigrammes. Il est très semblable au modèle de reconnaissance de chiffres de la figure 2.2 puisque dans les deux cas nous avons un seul réseau de neurones à propagation avant entièrement connecté aboutissant dans un softmax pour produire les probabilités nécessaires à la perte d'entropie croisée. Il n'y a que quelques différences.

Tout d'abord, au lieu d'une image, le réseau de neurones reçoit en entrée un indice de mot i tel que $0 \leq i < |V|$, et il le remplace tout d'abord par $\mathbf{E}[i]$, le plongement du mot :

```
inpt=tf.placeholder(tf.int32, shape=[batchSz])
answr=tf.placeholder(tf.int32, shape=[batchSz])
E = tf.Variable(tf.random_normal([vocabSz, embedSz], stddev = 0.1))
embed = tf.nn.embedding_lookup(E, inpt)
```

Nous supposons qu'il existe du code non montré ici qui lit les mots en entrée et remplace les caractères par des indices de mots uniques. De plus, ce code regroupe `batchSz` d'entre eux dans un vecteur. `inpt` pointe vers ce vecteur. La réponse correcte pour chaque mot (le mot suivant du texte) est un vecteur similaire, `answr`. Ensuite nous avons créé le tableau de recherche de plongement `E`. La fonction `tf.nn.embedding_lookup` crée le code TF nécessaire et le place dans le graphe de calcul. Les manipulations futures (comme par exemple `tf.matmul`) opèrent alors sur `embed`. Naturellement, TF peut déterminer comment modifier `E` pour diminuer la perte, de la même façon que les autres paramètres du modèle.

À l'autre extrémité du réseau à propagation avant, nous utilisons une fonction intégrée de TF pour calculer la perte d'entropie croisée :

```
xEnt = tf.nn.sparse_softmax_cross_entropy_with_logits
        (logits=logits, labels=answr)
loss = tf.reduce_sum(xEnt)
```

La fonction TF `tf.nn.sparse_softmax_cross_entropy_with_logits` reçoit en entrée deux arguments nommés. Ici l'argument `logits` (que nous avons par commodité nommé `logits`) est un `batchSz` de valeurs de logits (c'est-à-dire un tableau `batchSz` par `vocabSz` de logits). L'argument `labels` est un vecteur contenant les réponses correctes. La fonction fournit les logits à `softmax` pour obtenir un vecteur colonne de probabilités `batchSz` par `vocabSz`. C'est-à-dire que $s_{i,j}$, le i,j -ième élément de softmax, est la probabilité du mot j dans le i -ième exemple de ce lot. La fonction repère alors la probabilité de la réponse correcte (à partir de `answr`) pour chaque ligne, calcule son logarithme naturel et renvoie un tableau `batchSz` par 1 (en pratique vecteur colonne) de ces logarithmes de probabilités. La seconde ligne

4.2 Construction de modèles linguistiques à propagation avant

du code ci-dessus prend ce vecteur colonne et en fait la somme pour obtenir la perte totale pour ce lot d'exemples.

Si vous êtes curieux de tout, le mot « sparse » (« creux », en français) est employé ici dans la même acception que dans « matrice creuse » (et en dérive probablement). Une matrice creuse est une matrice ne comportant que très peu de valeurs non nulles, c'est pourquoi il est plus économique de ne stocker que les positions et les valeurs de ces éléments non nuls. Pour revenir au calcul de la perte dans notre premier programme Mnist (page 31), nous avons supposé que les étiquettes correctes pour les images de chiffres étaient fournies sous forme de vecteurs one-hot où seule la position de la réponse correcte était non nulle. Dans `tf.nn.sparse_softmax` nous donnons simplement la réponse correcte. La réponse correcte peut être considérée comme une version creuse de la représentation one-hot.

Revenons au modèle linguistique armés de ce code, effectuons quelques itérations sur nos exemples d'entraînement et obtenons des plongements mettant en évidence des similarités de mots telles que celles de la figure 4.2. Par ailleurs, si nous voulons évaluer le modèle linguistique, nous pouvons imprimer à l'issue de chaque itération la perte totale sur le jeu d'entraînement. Elle devrait décroître au fur et à mesure des itérations.

Au chapitre 1 (page 19), nous vous avons suggéré d'imprimer durant les itérations la moyenne de la perte sur l'ensemble des exemples, car si nos paramètres améliorent le modèle, la perte devrait décroître (les nombres que nous voyons devraient devenir plus petits). Nous suggérons ici une amélioration mineure de cette idée. Notez tout d'abord qu'en modélisation linguistique un « exemple » affecte des probabilités aux mots suivants possibles, de sorte que le nombre d'exemples d'entraînement est le nombre de mots dans notre corpus d'entraînement. Par conséquent, plutôt que de parler de perte moyenne sur l'ensemble des exemples, nous parlons de *perte moyenne sur l'ensemble des mots*. Ensuite, plutôt que d'imprimer une perte moyenne par mot, imprimons son exponentielle, c'est-à-dire e élevé à cette puissance. C'est-à-dire que pour un corpus d comportant $|d|$ mots, si la perte totale vaut x_d , nous imprimons :

$$f(d) = e^{\frac{x_d}{|d|}} \quad (4.6)$$

C'est ce que nous appelons la *perplexité* du corpus d . À la réflexion, c'est une bonne valeur car elle a quelque chose d'intuitif : en moyenne, deviner le mot suivant est équivalent à deviner le résultat d'un lancer de dé non truqué ayant ce nombre de faces. Notez ce que cela signifie lorsqu'il s'agit de deviner le deuxième mot de notre corpus d'entraînement, étant donné le premier mot. Si notre corpus a une taille de vocabulaire de 10 000 et si nous débutons avec tous nos paramètres proches de 0, alors les 10 000 logits du premier exemple valent 0 et toutes les probabilités valent 10^{-4} . Au lecteur de confirmer qu'il en résulte une perplexité exactement égale à la taille du vocabulaire. Au fur et à mesure de l'entraînement la perplexité décroît, et pour

le corpus particulier utilisé par votre auteur avec une taille de vocabulaire d'environ 7 800 mots, après deux itérations d'entraînement sur un jeu d'entraînement d'environ 10^6 mots, le jeu de développement atteignait une perplexité de 180 à peu près. Sur un ordinateur portable doté de quatre CPU, chaque itération du modèle durait à peu près 3 minutes.

4.3 AMÉLIORATION DES MODÈLES LINGUISTIQUES À PROPAGATION AVANT

Il y a de nombreuses façons d'améliorer le modèle linguistique que nous venons de concevoir. Ainsi, nous avons vu au chapitre 2 qu'en ajoutant une couche cachée (avec une fonction d'activation entre les deux couches) les performances sur le jeu Mnist passaient de 92 % de réponses correctes à 98 %. Ici, l'ajout d'une couche cachée fait passer la perplexité sur le jeu d'entraînement de 180 à 177 environ.

Mais la manière la plus simple d'obtenir une meilleure perplexité consiste à passer d'un modèle linguistique bigramme à un *modèle trigramme*. Rappelez-vous que pour passer de l'équation 4.2 à l'équation 4.4 nous avons fait l'hypothèse que la probabilité d'un mot ne dépendait que du mot précédent. C'est manifestement faux. En général, le choix du mot suivant peut être influencé par des mots plus ou moins loin en arrière, et l'influence de l'antépénultième (situé deux places en arrière) est très importante. Par conséquent un modèle correctement entraîné basant sa prédiction sur les deux mots précédents (appelé modèle *trigramme* car les probabilités sont basées sur des séquences de trois mots) obtient une bien meilleure perplexité que les modèles bigrammes.

Dans notre modèle bigramme, nous avions une variable pour l'indice du mot précédent, `inpt`, et une pour le mot à prédire (en supposant une taille de lot de 1), `answr`. Nous introduisons maintenant une troisième variable contenant l'indice du mot à deux places en arrière, `inpt2`. Dans le graphe de calcul de TF, nous ajoutons un nœud qui trouve le plongement de `inpt2` :

```
embed2 = tf.nn.embedding_lookup(E, inpt2)
```

puis un nœud pour concaténer les deux :

```
both= tf.concat([embed, embed2], 1)
```

Ici, le deuxième argument spécifie l'axe du tenseur sur lequel on effectue la concaténation. Souvenez-vous qu'en réalité nous effectuons les plongements de tout un lot en même temps, c'est pourquoi le résultat de chacune des recherches est une matrice de taille `batchSz` par `embedSz`. Nous voulons obtenir au final une matrice de taille `batchSz` par (`embedSz * 2`), par conséquent la concaténation s'effectue le long

de l'axe 1, c'est-à-dire que les lignes sont concaténées (rappelez-vous que les colonnes correspondent à l'axe 0). Enfin, nous devons changer les dimensions de \mathbf{W} de $\text{embedSz} * \text{vocabSz}$ à $(\text{embedSz} * 2) * \text{vocabSz}$.

Autrement dit, nous fournissons en entrée les plongements des deux mots précédents, et le réseau de neurones les utilise tous les deux pour estimer la probabilité du mot suivant. De plus, la passe arrière met à jour les plongements des deux mots. Ceci ramène la perplexité de 180 à 140 environ. Ajouter encore un autre mot à la couche d'entrée la diminue davantage, jusqu'à 120 environ.

4.4 SURAJUSTEMENT

À la section 1.6, nous avons parlé de la condition iid sous-jacente derrière toutes les garanties que les méthodes d'entraînement de nos réseaux de neurones aboutissent à des poids satisfaisants. Nous avons noté en particulier que dès que nous utilisons nos données d'entraînement pour plus d'une itération, les paris sont ouverts.

Mais en dehors d'un exemple peu naturel, nous n'avons fourni aucune preuve empirique sur ce point. Ceci parce que les données Mnist que nous avons utilisées dans nos exemples du chapitre 1 sont, par rapport à la plupart des jeux de données, très bien organisées. Ce que nous demandons à des données d'entraînement, après tout, c'est qu'elles couvrent les différents cas possibles (et dans des proportions correctes), de sorte que lorsque nous regarderons les données de test, il n'y ait pas de surprise. Avec seulement 10 chiffres et 60 000 exemples d'entraînement, Mnist répond plutôt bien à ce critère.

Malheureusement, la plupart des jeux de données ne sont pas si complets, et ceux du langage écrit en général (et le Penn Treebank en particulier) s'écartent beaucoup de cet idéal. Même si nous restreignons la taille du vocabulaire à 8 000 mots et regardons uniquement des modèles trigrammes, un très grand nombre de trigrammes du jeu de test n'apparaissent pas dans les données d'entraînement. En même temps, revoir sans cesse le même jeu d'exemples (relativement petit) amène le modèle à surestimer leur probabilité. La figure 4.3 présente les résultats de perplexité pour un modèle linguistique trigramme à deux couches entraîné sur le Penn Treebank. Les lignes indiquent le nombre d'itérations d'entraînement, la perplexité moyenne à chaque itération, puis la moyenne sur l'ensemble du corpus de développement.

Regardons d'abord la ligne correspondant à la perplexité sur le jeu d'entraînement, nous voyons qu'elle décroît régulièrement au fur et à mesure des itérations. C'est

Époque	1	2	3	4	5	6	7	10	15	20	30
Entr.	197	122	100	87	78	72	67	56	45	41	35
Dév.	172	152	145	143	143	143	145	149	159	169	182

Figure 4.3 – Surajustement dans un modèle linguistique

conforme à ce qu'on attend. La ligne des perplexités sur le jeu de développement nous présente une histoire plus compliquée. Elle aussi commence par décroître, de 172 à la première itération jusqu'à 143 à la quatrième, puis demeure stable pendant deux itérations, et se met à croître à partir de la septième itération. La 20^e itération, elle, atteint 169, puis 182 à la 30^e. La différence observée entre les résultats d'entraînement et de développement à la 30^e itération, 35 contre 182, correspond à un surajustement classique des données d'entraînement.

Les modifications effectuées pour corriger le surajustement portent le nom de *régularisation*. La technique de régularisation la plus simple est l'*arrêt prématué* (en anglais, early stopping) : nous interrompons simplement l'entraînement au point où la perplexité de développement est la plus faible. Mais bien que simple, l'arrêt prématué n'est pas la meilleure méthode pour corriger un problème de surajustement. La figure 4.4 montre deux solutions bien meilleures, l'*abandon* et la *régularisation L2*.

Époque	1	2	3	4	5	6	7	10	15	20	30
Abandon	213	182	166	155	150	144	139	131	122	118	114
Rég. L2	180	163	155	148	144	140	137	130	123	118	112

Figure 4.4 – Perplexité du modèle linguistique en utilisant la régularisation

Dans la technique d'abandon, nous modifions le réseau pour abandonner de manière aléatoire des morceaux de notre calcul. Par exemple, les données abandonnées dans la figure 4.4 résultait d'un abandon aléatoire de la sortie de 50 % des unités linéaires de la première couche. Par conséquent, la couche suivante reçoit des 0 à des emplacements aléatoires de son vecteur d'entrée. On peut donc considérer que les données d'entraînement ne sont plus identiques à chaque itération, puisque à chaque fois on abandonne des unités différentes. Pour comprendre pourquoi cela aide, on peut remarquer que le classificateur ne peut pas dépendre de la coïncidence d'un grand nombre de caractéristiques des données s'alignant d'une certaine façon, et que par conséquent il devrait se généraliser mieux. Comme nous pouvons le voir à la figure 4.4, cela aide réellement à éviter le surajustement. D'une part, la première ligne de la figure 4.4 ne montre pas de renversement de la perplexité sur le corpus de développement. Même après 30 itérations, la perplexité décroît, même si c'est aussi lent qu'un glacier (environ 0.1 par itération). De plus, la valeur minimale obtenue avec dropout est bien meilleure que ce que nous pouvons obtenir avec l'arrêt prématué, soit une perplexité de 114 au lieu de 145.

La seconde technique que nous mettons en avant sur la figure 4.4 est la régularisation L2. Celle-ci s'appuie sur la constatation que, dans de nombreux types d'apprentissage automatique, le surajustement s'accompagne de paramètres d'apprentissage dont la valeur absolue devient relativement grande. Nous avons déjà fait

remarquer que le fait de revoir de nombreuses fois les mêmes données amène un réseau de neurones à surestimer les probabilités de ce qu'il a vu aux dépens des autres cas qui peuvent se produire mais ne sont pas apparus dans les données d'entraînement. Cette surestimation s'obtient par des poids de forte valeur absolue ou, ce qui est à peu près équivalent, dont les valeurs au carré sont élevées. En régularisation L2, nous ajoutons à la fonction de perte une quantité proportionnelle à la somme des carrés des poids. Ce qui veut dire que si nous utilisions jusque-là la perte d'entropie croisée, notre nouvelle fonction de perte sera :

$$\mathcal{L}(\Phi) = -\log(\Pr(c)) + \alpha \frac{1}{2} \sum_{\phi \in \Phi} \phi^2 \quad (4.7)$$

Ici α est un nombre réel qui contrôle la façon dont nous équilibrions les deux termes. Il est petit d'ordinaire ; dans les expériences ci-dessus, nous l'avons choisi égal à 0.01, une valeur classique. Lorsque nous différentions la fonction de perte par rapport à ϕ , le second terme ajoute $\alpha\phi$ au total de $\frac{\partial \mathcal{L}}{\partial \phi}$. Ceci encourage les ϕ , tant positifs que négatifs, à se rapprocher de zéro.

Les deux formes de régularisation fonctionnent à peu près aussi bien sur cet exemple, même si l'on préfère en général la méthode par abandon. Elles sont toutes deux faciles à ajouter à un réseau TF. Pour abandonner par exemple 50 % des valeurs produites par la première couche d'unités linéaires (ici `w1Out`), nous ajoutons à notre programme :

```
keepP= tf.placeholder(tf.float32)
w1Out=tf.nn.dropout(w1Out, keepP)
```

Notez que nous avons fait de `keepP` un nœud de substitution (ou `placeholder`). Nous voulons procéder ainsi car nous n'effectuons l'abandon que durant l'entraînement. En phase de test il n'est pas nécessaire, et même nocif. En faisant de cette valeur un nœud de substitution, nous pouvons y placer les valeurs 0.5 durant l'entraînement et 1.0 durant le test.

Utiliser la régularisation L2 est tout aussi simple. Si nous voulons éviter que les valeurs de `w1` par exemple, les poids de certaines unités linéaires, deviennent trop grandes, nous ajoutons simplement

```
0.01 * tf.nn.l2_loss(W1)
```

à la fonction de perte que nous utilisons durant l'entraînement. Ici 0.01 est un hyperparamètre permettant de pondérer l'importance relative de la régularisation par rapport à la perte d'entropie croisée originelle. Si votre code calcule la perplexité en calculant l'exponentielle de la perte sur l'ensemble des mots, pensez bien à séparer la perte combinée utilisée durant l'entraînement de la perte utilisée pour le calcul de la perplexité. Dans le dernier cas, nous ne voulons que la perte d'entropie croisée.

4.5 RÉSEAUX RÉCURRENTS

Un *réseau de neurones récurrent* (en anglais, *recurrent neural network* ou RNN) est, en un certain sens, l'opposé d'un réseau de neurones à propagation avant. C'est un réseau dont la sortie contribue à sa propre entrée. En terminologie de graphes, il s'agit d'un graphe cyclique orienté, par opposition au graphe acyclique orienté de la propagation avant. La version la plus simple d'un réseau de neurones récurrent est présentée à la figure 4.5. La boîte étiquetée $\mathbf{W}_r \mathbf{b}_r$ comporte une couche d'unités linéaires avec des poids \mathbf{W}_r et des biais \mathbf{b}_r plus une fonction d'activation. Les données d'entrée sont injectées en bas à gauche et les données de sortie o sortent sur la droite et se partagent en deux. Un exemplaire revient au point de départ ; c'est cette boucle qui rend le processus récurrent et non à propagation avant. L'autre exemplaire va vers la deuxième couche d'unités linéaires avec des paramètres $\mathbf{W}_o, \mathbf{b}_o$ qui est chargée de calculer la sortie du réseau de neurones récurrent et la perte. Algébriquement, nous pouvons exprimer ceci comme suit :

$$\mathbf{s}_0 = \mathbf{0} \quad (4.8)$$

$$\mathbf{s}_{t+1} = \rho((\mathbf{e}_{t+1} \cdot \mathbf{s}_t) \mathbf{W}_r + \mathbf{b}_r) \quad (4.9)$$

$$\mathbf{o} = \mathbf{s}_{t+1} \mathbf{W}_o + \mathbf{b}_o \quad (4.10)$$

Nous débutons la relation de récurrence avec l'état \mathbf{s}_0 initialisé de manière arbitraire, en général par un vecteur de zéros. La dimension du vecteur d'état est un hyperparamètre. Nous obtenons l'état suivant en concaténant l'entrée suivante (\mathbf{e}_{t+1}) avec

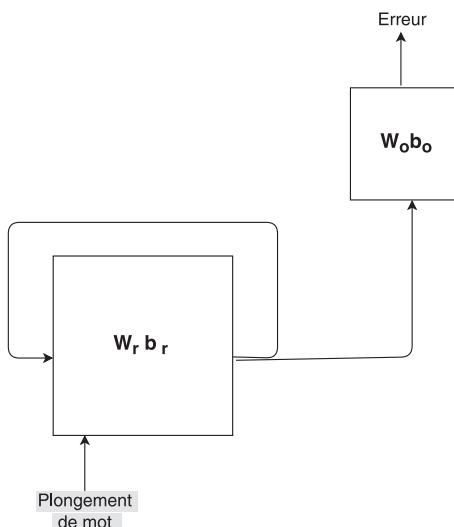


Figure 4.5 – Illustration graphique d'un réseau de neurones récurrent

l'état précédent s_t , et en transmettant le résultat à l'unité linéaire $\mathbf{W}_r, \mathbf{b}_r$. Nous transmettons alors le résultat à la fonction relu ρ (la fonction d'activation peut être choisie librement). Enfin, la sortie de l'unité \mathbf{o} du réseau de neurones récurrent est obtenue en transmettant l'état courant à une seconde unité linéaire $\mathbf{W}_o, \mathbf{b}_o$. Le choix de la fonction de perte est lui aussi libre, il s'agit le plus souvent de l'entropie croisée calculée sur \mathbf{o} .

Les réseaux récurrents sont adaptés au cas où nous souhaitons que les données d'entrée précédentes dans le réseau aient une influence pouvant s'étendre relativement loin dans le futur. Étant donné que le langage fonctionne de cette façon, les réseaux de neurones récurrents sont fréquemment utilisés dans les tâches linguistiques en général, et dans la modélisation linguistique en particulier. Mais supposons donc ici que l'entrée soit le plongement du mot courant w_i , la prédiction w_{i+1} et la perte notre perte d'entropie croisée habituelle.

Le calcul de la passe avant du réseau de neurones fonctionne à peu près comme dans le cas du réseau de neurones à propagation avant, excepté que nous mémorisons le o de l'itération précédente et le concaténons avec le plongement du mot courant au début de la passe avant. La passe arrière, cependant, n'est pas si évidente. Précédemment, lorsque nous avons expliqué comment il se fait que les paramètres dans les plongements de mots soient aussi mis à jour par TF, nous avons dit que TF travaille en sens inverse à partir de la fonction de perte, recherchant les paramètres ayant un effet sur l'erreur, puis différentie cette dernière par rapport à ces paramètres. Dans le réseau de neurones Mnist du chapitre 1, ceci nous a ramenés dans la couche avec \mathbf{W} et \mathbf{b} , puis s'est arrêté lorsque nous n'avons plus rencontré que les pixels de l'image. C'est également vrai pour les réseaux de neurones convolutifs, même si le moyen de faire rentrer les paramètres dans le calcul de la fonction d'erreur est plus compliqué. Mais maintenant, il n'y a potentiellement plus de limite à l'étendue du retour arrière à effectuer dans la passe arrière.

Supposons que nous venions de lire le 500^e mot et voulions changer les paramètres du modèle parce que nous n'avons pas prédit w_{501} avec une probabilité de 1. En revenant en arrière, nous découvrons que cette partie de l'erreur est due aux poids W_o du réseau en haut et à droite de la figure 4.5. Mais bien sûr, l'une des entrées de cette couche est la sortie de l'unité récurrente o_{500} juste après avoir traité le mot w_{500} . Et d'où venait cette valeur ? Eh bien, de $\mathbf{W}_r, \mathbf{b}_r$, mais aussi en partie de o_{499} . En bref, pour effectuer « correctement » cela, nous devons retracer l'erreur à travers 500 boucles de la couche récurrente, en ajustant sans cesse les poids $\mathbf{W}_r, \mathbf{b}_r$ grâce aux contributions de tous les mauvais choix depuis le premier mot. Ceci n'est pas pratique.

Nous résolvons ce problème par la force brutale. Nous arrêtons simplement le calcul au bout d'un nombre arbitraire d'itérations en arrière... Ce nombre d'itérations est appelé *taille de la fenêtre* et c'est un hyperparamètre du système. Cette technique qui porte le nom de *rétropropagation à travers le temps* (en anglais, *back propagation through time*) est illustrée à la figure 4.6, où nous avons choisi une taille de fenêtre de 3 (mais 20 serait une valeur plus réaliste pour cette taille de fenêtre). Pour être

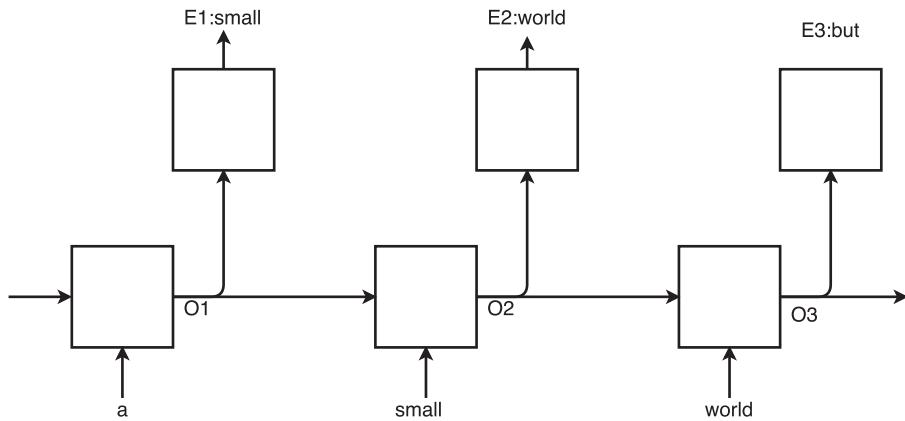


Figure 4.6 – Rétropropagation à travers le temps avec une taille de fenêtre de 3

plus précis, la figure 4.6 imagine que nous sommes en train de traiter un corpus anglais commençant par la phrase « It is a small world but I like it that way » avec un délimiteur de phrase. La rétropropagation à travers le temps traite la figure 4.6 comme s'il s'agissait d'un réseau à propagation avant prenant en compte non pas un simple mot, mais un ensemble de mots de la taille d'une fenêtre (ici trois) en calculant alors l'erreur sur les trois. Dans le cas de notre petit corpus, le premier appel d'entraînement recevrait « STOP It is » en tant que mots d'entrée et « It is a » pour les trois mots à prédire.

La figure 4.6 imagine que nous en sommes au deuxième appel, où l'on reçoit les mots « a small world » et qu'ils doivent prédire « small world but ». Au début de la seconde passe avant, la sortie de la première commande arrive à gauche (O0) et est concaténée avec le plongement de « a » puis transmise au réseau de neurones récurrent dans lequel il devient O1 alimentant la perte en E1.

Mais en plus d'aller en E1, O1 est également transmis pour être concaténé avec le deuxième mot, « small ». Nous y calculons également l'erreur. Ici nous calculons l'effet conjoint de **W** et **b** (sans parler des plongements) sur l'erreur, en prédisant « small ». Mais **W** et **b** contribuent à l'erreur de deux manières différentes — le plus directement à partir de l'erreur sur E2 provenant d'eux-mêmes, mais aussi au travers de leur contribution à O1. Naturellement, lorsque nous calculons ensuite E3, **W** et **b** affectent l'erreur de trois façons différentes : directement à partir de O3, ainsi qu'à partir de O1 et O2. Par conséquent, les paramètres dans ces variables sont modifiés six fois (ou, ce qui revient au même, le programme additionne peu à peu les corrections puis les modifie une fois).

La figure 4.6 ignore le problème de la taille du lot. Comme vous pouvez vous y attendre, les fonctions de réseaux de neurones récurrents de TensorFlow sont conçues pour permettre l'exécution d'un entraînement (ou d'un test) simultané sur tout un lot. Par conséquent, chaque appel au réseau de neurones récurrent admet en entrée un

tableau de dimension batchSz (taille du lot) par windowSz (taille de fenêtre), pour fournir un tableau de même taille des mots prédits. Comme dit précédemment, nous travaillons en parallèle sur des groupes de la taille d'un lot, par conséquent les derniers mots prédits à partir du premier appel d'entraînement sont les premiers mots introduits dans le deuxième.

Pour que ceci fonctionne, nous devons créer les lots avec soin. La figure 4.7 illustre ce qui se passe avec le corpus factice « STOP It is a small world but I like it that way STOP ». Nous supposons que la taille de lot vaut 2 et la taille de fenêtre 3. L'idée de base consiste à diviser le corpus en deux puis à remplir chaque lot avec des éléments de chacune des parties (dans notre cas, les deux moitiés) du corpus. La fenêtre supérieure de la figure 4.7 montre le corpus divisé en deux morceaux. Les fenêtres qui suivent présentent les deux lots d'entrée créés à partir de cela. Chaque lot comporte des segments de trois mots issus de chaque moitié. Il nous faut aussi organiser par lots les mots prédits pour les injecter dans le réseau. C'est exactement comme la figure précédente, mais chaque mot est un cran plus loin dans le corpus. Par conséquent, la ligne supérieure du diagramme de prédiction serait « It is a small world but ».

STOP	It	is	a	small	world
but	I	like	it	that	way

STOP	It	is
but	I	like

a	small	world
it	that	way

Figure 4.7 – Répartition des mots avec une taille de lot de 2 et une taille de fenêtre de 3

Le « corpus » n'étant que de 14 mots, chaque moitié en comporte six. Pour comprendre pourquoi six et non pas sept, intéressez-vous aux prédictions pour le deuxième lot. Déroulez l'algorithme soigneusement avec sept mots par moitié et vous constaterez que nous n'avons pas de mot de prédiction pour la dernière entrée. Par conséquent, le corpus est divisé initialement en S sections, avec, pour un corpus de taille x et une taille de lot b , $S = \lfloor (c - 1)/b \rfloor$ (où $\lfloor x \rfloor$ est la fonction *floor* — le plus grand entier inférieur à x). Ici le « -1 » permet d'associer au dernier mot d'entrée son mot de prédiction correspondant.

Nous n'avons pas parlé jusqu'à maintenant de ce que nous faisons à la fin de la phrase. La solution la plus aisée consiste simplement à passer à la suivante. Ceci signifie qu'un segment de la taille d'une fenêtre chargé dans le réseau de neurones récurrent peut contenir des morceaux de deux phrases différentes. Cependant, nous avons placé le mot de séparation STOP entre les phrases, ce qui fait que le réseau

de neurones récurrent devrait, en principe, apprendre ce que cela signifie pour les mots qui suivent, qui commenceront par exemple par une majuscule. De plus, les derniers mots de la phrase précédente peuvent fournir de bonnes indications sur les mots suivants. Si nous nous intéressons uniquement à la modélisation linguistique, séparer les phrases par STOP et ne plus se soucier de la séparation des phrases durant l'entraînement ou l'utilisation des réseaux de neurones récurrents paraît suffisant.

Reprendons les figures 4.5 et 4.6 en réfléchissant cette fois à la façon de programmer un modèle linguistique à base de réseau de neurones récurrent. Comme nous l'avons déjà remarqué, le code permettant de passer de notre corpus de mots à l'entrée du modèle doit être légèrement réorganisé. Précédemment, l'entrée (de même que les prédictions) était un vecteur de longueur batchSz (taille du lot), maintenant c'est un tableau de dimension batchSz par windowSz (taille de fenêtre). Nous devons aussi remplacer chaque mot par son propre plongement, mais ceci est inchangé par rapport au modèle à propagation avant.

Ensuite, le mot est introduit dans le réseau de neurones récurrent. Le code TF essentiel à la création du réseau de neurones récurrent est :

```
rnn= tf.contrib.rnn.BasicRNNCell(rnnSz)
initialState = rnn.zero_state(batchSz, tf.float32)
outputs, nextState = tf.nn.dynamic_rnn(rnn, embeddings,
                                       initial_state=initialState)
```

La première ligne ajoute le réseau de neurones récurrent (ou RNN) au graphe de calcul. Notez que la largeur du tableau des poids du RNN est un paramètre libre, rnnSz (vous vous rappelez peut-être que lorsque nous avons ajouté une couche supplémentaire aux unités linéaires du modèle Mnist à la fin du chapitre 2, nous avions une situation similaire). La dernière ligne est l'appel au réseau de neurones récurrent. Il accepte trois arguments, et en renvoie deux. Les entrées sont : tout d'abord le réseau de neurones récurrent proprement dit, ensuite les mots qu'il va traiter (au nombre de batchSz * windowSz) et le initialState qu'il récupère de l'exécution précédente. Étant donné que lors du premier appel à dynamic_rnn il n'y a pas d'état précédent, nous en créons un fictif avec l'appel de fonction de la deuxième ligne rnn.zero_state.

tf.nn.dynamic_rnn produit deux résultats. Le premier, que nous avons nommé outputs, constitue les informations alimentant le calcul de l'erreur. Sur la figure 4.6, il s'agit des sorties O1, O2, O3. Par conséquent outputs a pour forme [batchSz, windowSz, hiddenSz], où hiddenSz est la taille de la couche cachée. La première dimension correspond aux différents exemples du lot. Chaque exemple lui-même est composé de O1, O2 et O3, par conséquent la deuxième dimension correspond à la taille de fenêtre. Enfin, O1 par exemple est un vecteur de rnnSz nombres flottants constituant la sortie du réseau de neurones récurrent à partir d'un seul mot.

La seconde sortie de tf.nn.dynamic_rnn que nous avons appelée nextState est la dernière sortie (O3) de cette passe dans le RNN. La prochaine fois que nous

```

[[[-0.077  0.022 -0.058 -0.229  0.145]
 [-0.167  0.062  0.192 -0.310 -0.156]
 [-0.069 -0.050  0.203  0.000 -0.092]]]

[[[-0.073 -0.121 -0.094 -0.213 -0.031]
 [-0.077  0.022 -0.058 -0.229  0.145]]
 [[ 0.179  0.099 -0.042 -0.012  0.175]
 [-0.167  0.062  0.192 -0.310 -0.156]]
 [[ 0.103  0.050  0.160 -0.141 -0.027]
 [-0.069 -0.050  0.203  0.000 -0.092]]]

```

Figure 4.8 – nextState et outputs d'un RNR

appellerons `tf.nn.dynamic_rnn`, nous aurons `initialState = nextState`. Notez que `nextState` est en fait constitué d'informations qui sont présentes dans `outputs`, puisque c'est la collecte des O₃ provenant des exemples du lot. À titre d'exemple, la figure 4.8 présente `nextState` et `outputs` pour une taille de lot de 3, une taille de fenêtre de 2, et une taille de RNN de 5. Avec une taille de fenêtre de 2, une ligne sur deux de la sortie est une ligne `nextState`. Il est assez pratique d'obtenir `nextState`, l'état suivant, emballé séparément, mais nous comprendrons plus clairement la raison réelle de cette répétition à la section suivante.

La dernière partie du modèle linguistique est la perte. Celle-ci est calculée en haut et à droite de la figure 4.5. Comme nous le voyons là, la sortie du réseau de neurones récurrent est transmise d'abord à une couche d'unités linéaires permettant de calculer les logits pour softmax, puis nous calculons la perte d'entropie croisée à partir des probabilités. Comme nous venons de le dire, la sortie du réseau de neurones récurrent est un tenseur 3D de forme [batchSz, windowSz, rnnSz]. Jusqu'à maintenant, nous n'avons transmis que des tenseurs 2D, c'est-à-dire des matrices, à travers nos unités linéaires, et nous l'avons fait en utilisant la multiplication de matrices — p. ex. `tf.matmul(inpt, W)`.

La façon la plus commode de gérer cela est de changer la forme du tenseur de sortie du réseau de neurones récurrent pour en faire une matrice avec les propriétés correctes :

```

output2 = tf.reshape(output, [batchSz*windowSz, rnnSz])
logits = matmul(output2, W)

```

Ici **W** est la couche linéaire (**W_o**) qui prend la sortie du RNN et la transforme en logits à la figure 4.5. Nous pouvons alors transmettre cela à `tf.nn.sparse_softmax_cross_entropy_with_logits` qui renvoie un vecteur colonne de valeurs de perte que l'on peut réduire à une seule valeur avec

Chapitre 4 · Plongements de mots et réseaux de neurones récurrents

`tf.reduce_mean`. Il ne reste plus qu'à calculer l'exponentielle de cette valeur finale pour obtenir notre perplexité.

Modifier la forme de la sortie du réseau de neurones récurrent était pratique pour des motifs pédagogiques (cela permettait de réutiliser `tf.matmul`) et en raison de contingences informatiques (pour obtenir une forme compatible avec `sparse_softmax`). Dans d'autres situations, les calculs en aval peuvent exiger la forme d'origine. Dans ce cas, nous pouvons nous tourner vers l'une des nombreuses fonctions TF permettant de gérer des tenseurs multidimensionnels. Celui que nous allons utiliser ici est `tensordot`, déjà présenté à la section 2.4.2. Voici comment l'appeler :

```
tf.tensordot(outputs, W, [[2], [0]])
```

Ce code effectue une succession de produits scalaires (en fait, une multiplication de matrices) entre le troisième composant (numéroté à partir de zéro) de `outputs` et le premier de `W`.

Encore un point à propos de l'utilisation des réseaux de neurones récurrents en général. Dans le code Python qui accompagne le code TF pour réseaux de neurones récurrents ci-dessus, nous voyons quelque chose comme ceci :

```
inputSt = sess.run(initialSt)
for i in range(numExamps)
    ``lire les mots et effectuer les plongements''
    logts, nxts=sess.run([logits,nextState], {input=wrds,
                                                nextState=inputSt})
    inputSt=nxts
```

Rien de ceci ne doit être pris littéralement, sauf la façon dont l'état d'entrée `inputSt` du réseau de neurones récurrent est initialisé, comment nous le transmettons à TF avec le morceau du dictionnaire d'alimentation `nextState=inputState`, ainsi que la façon dont nous mettons à jour `inputSt` dans la dernière ligne ci-dessus. Jusqu'à maintenant, nous n'avons utilisé que `feed_dict` pour transmettre des valeurs aux nœuds de substitution (ou placeholders) de TF. Ici `nextState` ne pointe pas vers un tel nœud, mais plutôt vers une portion de code qui génère l'état zéro avec lequel nous démarrons le réseau de neurones récurrent. Ceci est permis.

4.6 LONGUE MÉMOIRE À COURT TERME

Un réseau de neurones de *longue mémoire à court terme* (en anglais, *long short-term memory* ou LSTM) est un type de réseau de neurones récurrent qui fournit pratiquement toujours de meilleurs résultats que le modèle RNN simple présenté à la section précédente. Le problème des réseaux de neurones récurrents ordinaires, c'est qu'alors que l'objectif est de se souvenir de choses qui se sont produites longtemps en arrière,

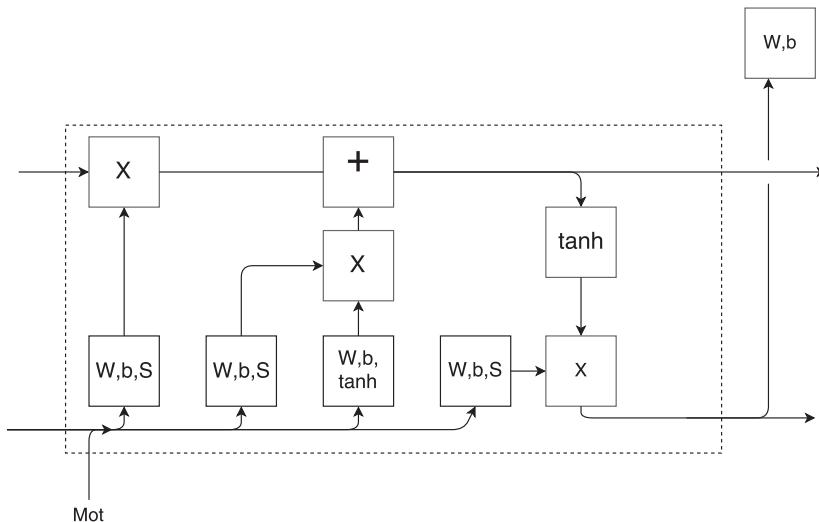


Figure 4.9 – Architecture des LSTM

en pratique ils semblent oublier rapidement. À la figure 4.9 tout ce qui se trouve à l'intérieur du cadre en pointillés correspond à une seule unité RNN. Manifestement, les LSTM complexifient significativement l'architecture. Notez tout d'abord que nous avons montré un exemplaire d'un LSTM dans un diagramme de propagation arrière à travers le temps. Sur la gauche, nous trouvons les informations provenant du traitement du mot précédent (utilisant deux tenseurs d'information plutôt qu'un seul). En bas, nous voyons l'arrivée du mot suivant. À droite, nous avons deux tenseurs sortant pour informer la prochaine unité temporelle et, comme dans les réseaux de neurones récurrents ordinaires, nous voyons ces informations « monter » dans le diagramme pour prédire le mot suivant et la perte (en haut à droite).

Le but est d'améliorer la mémorisation par le réseau de neurones récurrent des événements passés, en lui apprenant à se souvenir des choses importantes et à oublier le reste. Pour ce faire, les LSTM transmettent deux versions du passé : la mémoire sélective « officielle » en haut, et une version plus locale en bas. La chronologie de la mémoire du haut est appelée *état de cellule* (en anglais, *cell state*) et est abrégée en « c ». La ligne du bas est appelée « h ».

La figure 4.9 introduit plusieurs nouvelles connexions et fonctions d'activation. Tout d'abord, nous voyons que la ligne de mémoire est modifiée à deux endroits avant d'être transmise à la prochaine unité temporelle. Elles sont nommées fois (X) et plus (+). L'idée est d'enlever des mémoires à l'unité X, et de les ajouter à l'unité +.

Pourquoi cela ? Regardez maintenant le plongement du mot courant arrivant en bas à gauche. Il traverse une couche d'unités linéaires suivies d'une fonction d'activation

sigmoïde, comme indiqué par la notation $\mathbf{W}, \mathbf{b}, \mathbf{S}$, où \mathbf{W}, \mathbf{b} correspond à l'unité linéaire et \mathbf{S} à la fonction sigmoïde. Nous avons vu la fonction sigmoïde à la figure 2.7. Vous voudrez peut-être vous y reporter, car certaines de ses caractéristiques ont de l'importance dans la discussion qui suit. En notation mathématique, nous avons l'opération :

$$\mathbf{h}' = \mathbf{h}_t \cdot \mathbf{e} \quad (4.11)$$

$$\mathbf{f} = S(\mathbf{h}'\mathbf{W}_f + \mathbf{b}_f) \quad (4.12)$$

Nous utilisons un point centré \cdot pour indiquer la concaténation de vecteurs. Pour détailler, en bas à gauche nous concaténons la ligne \mathbf{h} précédente \mathbf{h}_t et le plongement du mot courant \mathbf{e} pour obtenir \mathbf{h}' , qui à son tour est transmise à l'unité linéaire « oubliuse » (suivie d'une sigmoïde) pour produire \mathbf{f} , le signal oublious qui remonte vers le côté gauche de la figure.

La sortie de la sigmoïde est alors multipliée terme à terme par la ligne mémoire \mathbf{c} qui arrive en haut à gauche (« terme à terme » signifie que l'élément $x[i,j]$ d'un tableau est combiné au terme $y[i,j]$ de l'autre).

$$\mathbf{c}'_t = \mathbf{c}_t \odot \mathbf{f} \quad (4.13)$$

(Ici « \odot » indique une multiplication terme à terme.) Sachant que les sigmoïdes sont bornées par 0 et 1, le résultat de la multiplication doit être une réduction de la valeur absolue en chaque point de la mémoire principale. Ceci correspond à « oublier ». Au final, cette configuration — une sigmoïde alimentant une couche multiplicative — est un modèle classique lorsque nous voulons réaliser un filtrage logiciel (en anglais, *soft gating*).

Comparez cela avec ce qui se passe dans l'unité additive que la mémoire traverse ensuite. De nouveau, le plongement de mot suivant est arrivé en bas à gauche, et cette fois il traverse séparément deux couches linéaires, l'une avec l'activation sigmoïde, et l'autre avec la *fonction d'activation tanh*, présentée à la figure 4.10. Tanh signifie *tangente hyperbolique*.

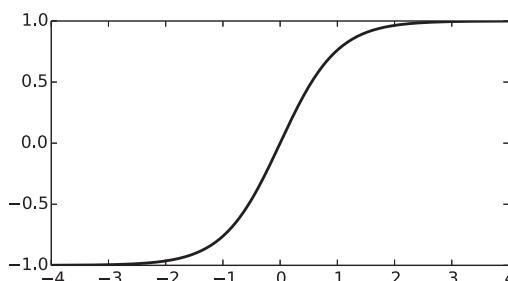


Figure 4.10 – La fonction tanh

$$\mathbf{a}_1 = S(\mathbf{h}' \mathbf{W}_{\mathbf{a}_1} + \mathbf{b}_{\mathbf{a}_1}) \quad (4.14)$$

$$\mathbf{a}_2 = \tanh((\mathbf{h}_t \cdot \mathbf{e}) \mathbf{W}_{\mathbf{a}_2} + \mathbf{b}_{\mathbf{a}_2}) \quad (4.15)$$

Il est important que, contrairement à la fonction sigmoïde, tanh puisse produire à la fois des valeurs positives et négatives, de façon à pouvoir extraire de nouvelles informations plutôt que d'effectuer simplement des changements d'échelle. Le résultat obtenu est ajouté à l'état de la cellule marquée « + » :

$$\mathbf{c}_{t+1} = \mathbf{c}'_t \oplus (\mathbf{a}_1 \odot \mathbf{a}_2) \quad (4.16)$$

Après quoi, la ligne mémoire de la cellule se partage. Un exemplaire va sur la droite et un autre exemplaire est transmis à tanh puis combiné avec une transformation linéaire de l'historique/plongement le plus local pour devenir la nouvelle ligne h en bas :

$$\mathbf{h}'' = \mathbf{h}' \mathbf{W}_h + b_h \quad (4.17)$$

$$\mathbf{h}_{t+1} = \mathbf{h}'' \odot a_2 \quad (4.18)$$

Ceci est alors concaténé avec le plongement du mot suivant, et le processus se répète. Le point à souligner ici est que la ligne de mémoire de cellule ne passe jamais directement au travers d'unités linéaires. Les choses sont minimisées (c'est-à-dire oubliées) dans l'unité X et ajoutées dans l'unité +, mais c'est tout. D'où la logique du mécanisme LSTM.

Pour ce qui concerne le programme, une seule petite modification de la version TF est nécessaire :

```
tf.contrib.rnn.BasicRNNCell(rnnSz)
```

devient

```
tf.contrib.rnn.LSTMCell(rnnSz)
```

Remarquez que cette modification affecte l'état qui est transmis d'une unité temporelle à la suivante. Précédemment, comme on pouvait le voir sur la figure 4.8, les états avaient la forme [batchSz, rnnSz]. Maintenant, c'est [2, batchSz, rnnSz], soit un tenseur [batchSz, rnnSz] pour la ligne c, un pour la ligne h.

En termes de performances, la version LSTM est bien meilleure, au prix d'un temps d'entraînement plus long. Prenez un modèle de réseau de neurones récurrent tel que celui que nous avons développé à la section précédente, donnez-lui plein de ressources (vecteur de plongement de taille 128, taille cachée de 512) et vous obtenez une perplexité respectable de 120 à peu près. Modifiez l'appel de fonction pour passer d'un réseau de neurones récurrent simple à un LSTM et votre perplexité tombe à 101.

4.7 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

La publication ayant introduit ce que nous considérons maintenant comme le modèle linguistique à propagation avant standard est celle de Bengio *et al.* [BDVJ03]. C'est

aussi là qu'a été introduit le terme « word embedding » (plongement de mot). L'idée de représenter les mots dans un espace continu, en particulier sous forme de vecteurs de nombres, est bien antérieure. C'est la publication de Bengio, cependant, qui a montré que les plongements de mots comme nous les connaissons aujourd'hui sont des sous-produits quasi automatiques des modèles de réseaux de neurones linguistiques.

Certains considèrent que ce n'est que depuis les travaux de Mikolov *et al.* que les plongements de mots sont devenus un composant quasi universel du traitement du langage naturel par les réseaux de neurones. Ils ont développé plusieurs modèles connus sous le nom de *word2vec*. La publication de référence est [MSC⁺13]. Le plus populaire des modèles *word2vec* est le *modèle skip-gram*. Dans notre présentation, les plongements étaient optimisés pour prédire le mot suivant compte tenu des mots précédents. Dans le modèle *skip-gram*, il est demandé à chacun des mots de prédire l'ensemble de ses mots voisins. Un résultat frappant des modèles *word2vec* est l'utilisation des plongements de mots pour résoudre des problèmes d'*analogie de mots* — p. ex. mâle est à roi ce que femelle est à... ? De manière inattendue, les réponses à ces problèmes ont découlé des plongements de mots créés. Il suffisait de prendre le plongement du mot « roi », de soustraire celui pour « mâle », d'ajouter « femelle » puis de rechercher le plongement de mot le plus proche du résultat. Un excellent blog sur les plongements de mots et les problèmes associés est celui de Sebastian Ruder [Rud16].

Les réseaux de neurones récurrents existent depuis le milieu des années 1980, mais ils n'ont guère donné de résultats satisfaisants jusqu'à ce que Sepp Hochreiter et Jürgen Schmidhuber créent les LSTM [HS97]. Un blog de Chris Colah donne une bonne explication des LSTM [Col15], d'ailleurs ma figure 4.9 est une reprise d'un de ses diagrammes.

Exercices

4.1 Supposons que notre corpus anglais commence par « *STOP* I like my cat and my cat likes me . *STOP* ». Supposons également que nous affectons à chaque mot un entier unique au fur et à mesure de la lecture de notre corpus, en commençant par 0. Pour une taille de lot de 5, écrivez les valeurs que nous devrions lire pour remplir les nœuds de substitution `inpt` et `answr` sur le premier jeu d'entraînement.

4.2 Expliquez pourquoi, si vous voulez avoir une chance d'entraîner un bon modèle linguistique à base de plongements, vous ne devez pas initialiser tous les **E** à 0. Assurez-vous que votre explication convient également pour valoriser tous les **E** à 1.

4.3 Expliquez pourquoi, si vous utilisez une régularisation L2, c'est clairement une mauvaise idée de calculer la perte totale courante.

4.4 Envisagez la construction d'un modèle linguistique trigramme totalement connecté. Dans notre version, nous avons concaténé les plongements des deux entrées précédentes pour former l'entrée du modèle. L'ordre dans lequel nous effectuons les concaténations a-t-il un effet sur la capacité d'apprentissage du modèle ? Expliquez.

4.5 Considérez un modèle de réseau de neurones unigramme. La perplexité de ce modèle peut-elle être meilleure qu'en tirant des mots au hasard dans une distribution uniforme ? Pourquoi ? Expliquez quels éléments du modèle bigramme sont nécessaires pour obtenir des performances optimales du modèle unigramme.

4.6 Une *unité à portes linéaire* (en anglais, *linear gated unit* ou LGU) est une variante des LSTM. En se reportant à la figure 4.9, nous voyons que ce dernier avait une couche cachée contrôlant ce qui est ôté de la ligne de mémoire principale et une seconde qui contrôle ce qui est ajouté. Dans les deux cas, les couches prennent comme entrée la ligne de contrôle du bas, et produisent un vecteur de nombres compris entre 0 et 1 qui sont multipliés par la ligne de mémoire (oubli) ou ajoutés à celle-ci (souvenir). Les LGU en diffèrent car ils remplacent ces deux couches par une seule couche ayant la même entrée. La sortie est multipliée par la ligne de contrôle, comme précédemment. Cependant, elle est aussi soustraite à 1, multipliée par la couche de contrôle et ajoutée à la ligne de mémoire. En général, les LGU fonctionnent aussi bien que les LSTM et, ayant une couche linéaire de moins, sont légèrement plus rapides. Expliquez l'idée intuitive. Modifiez la figure 4.9 de sorte qu'elle représente le fonctionnement d'un LGU.

APPRENTISSAGE SÉQUENCE À SÉQUENCE

L'*apprentissage séquence à séquence* (couramment abrégé en *seq2seq*) est une technique d'apprentissage profond permettant d'associer une séquence de symboles à une autre séquence de symboles lorsqu'il n'est pas possible (ou que nous ne voyons pas comment) effectuer cette mise en correspondance sur la base des symboles individuels eux-mêmes. L'application par excellence du seq2seq est la *traduction automatique* (en anglais, *machine translation* ou MT), qui permet à un ordinateur de traduire d'un langage naturel vers un autre, comme par exemple de l'anglais vers le français.

Depuis les années 1990, on sait qu'il est assez difficile de définir ce type de règles d'association dans un programme et qu'une approche indirecte fonctionne beaucoup mieux. Nous donnons à l'ordinateur un *corpus aligné*, à savoir de nombreux exemples de paires de phrases qui sont la traduction mutuelle l'une de l'autre, et demandons à la machine de trouver l'association par elle-même. C'est là que l'apprentissage profond entre en scène. Malheureusement, les techniques que nous avons vues jusqu'ici pour les tâches en langage naturel, par exemple les LSTM, ne sont en elles-mêmes pas suffisantes pour la traduction automatique.

Fondamentalement, la modélisation linguistique, la tâche sur laquelle nous nous sommes concentrés au chapitre précédent, procède en mot à mot. Ce qui veut dire que nous introduisons un mot et que nous prédisons le suivant. La traduction automatique ne fonctionne pas ainsi. Prenons quelques exemples tirés du *Canadian Hansard*, la base de tout ce qui s'est dit au Parlement canadien et qui, légalement, doit être publié dans les deux langues officielles du Canada, le français et l'anglais. La première paire de phrases de la section I se trouve être :

edited hansard number 1
hansard révisé numéro 1

Un premier enseignement pour un Anglais débutant l'apprentissage du français (ou l'inverse) est que les adjectifs précèdent en général le nom qu'ils modifient en anglais, et qu'ils sont souvent situés après en français. Par conséquent, ici, les adjectifs « *edited* » et « *révisé* » ne sont pas en même position dans les deux traductions. Ceci montre que nous ne pouvons pas travailler de gauche à droite dans le *langage source* (celui que nous voulons traduire) en recrachant un mot à chaque fois dans le *langage cible*. Dans ce cas, nous aurions pu traiter deux mots en entrée et en sortie, mais les inversions syntaxiques observées peuvent porter sur un bien plus grand nombre de mots. Voici ce que nous trouvons quelques lignes après l'exemple précédent. Remarquez que le texte est découpé en *unités lexicales* (en anglais *tokens*). En deux occasions la ponctuation française est séparée des mots auxquels elle serait ordinairement attachée :

this being the day on which parliament was convoked by proclamation of his excellency ...
 parlement ayant été convoqué pour aujourd' hui , par proclamation de son excellence ...

En traduisant mot à mot l'anglais, on obtiendrait « Ceci étant le jour où le Parlement a été convoqué sur proclamation de son excellence ». On constate en particulier que « this being the day » se traduit en réalité par « aujourd' hui » (à vrai dire, en général les longueurs des phrases en correspondance ne sont pas les mêmes). Par conséquent, il est nécessaire d'effectuer un apprentissage séquence à séquence, où une séquence est généralement prise comme une phrase complète.

5.1 PRINCIPE DU SEQ2SEQ

Le diagramme d'un modèle seq2seq très simple est présenté à la figure 5.1. Il présente au fil du temps (qui s'écoule comme d'habitude de la gauche vers la droite) le processus de traduction de « hansard révisé numéro 1 » en « edited hansard number 1 ». Le modèle est composé de deux réseaux de neurones récurrents (ou RNN). Contrairement aux LSTM, nous supposons un réseau de neurones récurrent ne transmettant qu'une seule ligne de mémoire. Nous pourrions utiliser BasicRNNCell ; cependant, il existe un meilleur choix avec un nouveau concurrent des LSTM, le *réseau à portes récurrent* (en anglais, *Gated Recurrent Unit* ou GRU), qui ne transmet qu'une seule ligne mémoire entre unités de temps.

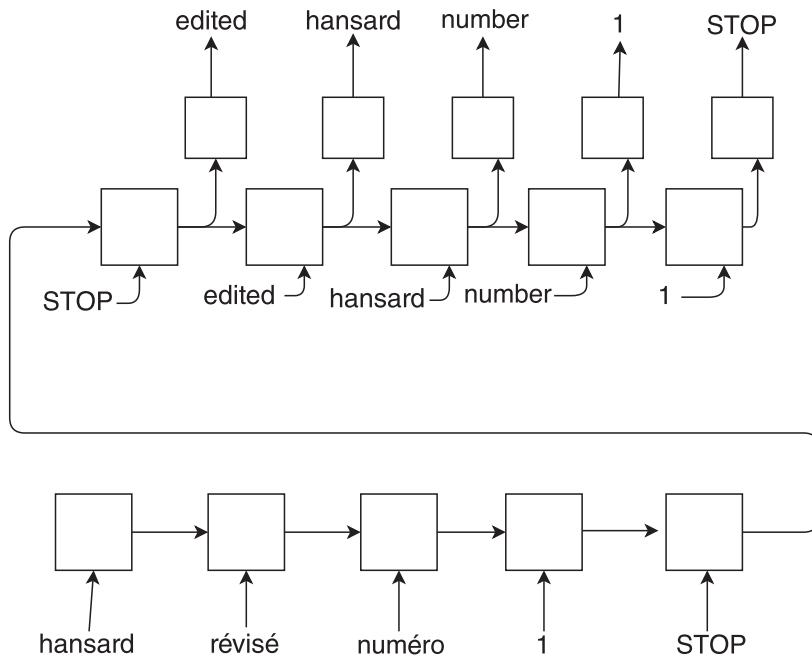


Figure 5.1 – Un modèle d'apprentissage séquence à séquence très simple

Le modèle opère en deux passes, chacune ayant son propre GRU. La première passe est appelée *passe d'encodage* et est représentée dans le bas de la figure 5.1. La passe se termine lorsque la dernière unité lexicale ou token (toujours STOP) du langage source (ici français) est traitée par le GRU du bas. L'état du GRU est alors transmis à la deuxième partie du processus. Le but de cette passe est de produire un vecteur qui « résume » la phrase. C'est ce qu'on appelle parfois *plongement de phrase* par analogie avec les plongements de mots.

La deuxième moitié de l'apprentissage seq2seq est appelée *passe de décodage*. Comme on le voit sur la figure, il s'agit d'une passe à travers la phrase du langage cible (ici l'anglais). (Il est peut-être temps de préciser que nous parlons ici de ce qui se produit durant l'entraînement, de sorte que nous connaissons la phrase anglaise.) Cette fois, le but est de prédire le prochain mot anglais après chaque mot reçu en entrée. La fonction de perte est la perte d'entropie croisée habituelle.

Les termes *encoder* et *décoder* proviennent de la théorie de la communication. Un message doit être encodé pour former le signal qui sera envoyé, puis décodé à partir du signal reçu. S'il y a du bruit sur la ligne, le signal reçu ne sera pas nécessairement identique à celui envoyé. Imaginez que le message d'origine soit en anglais et que le bruit soit la conversion en français. Alors le processus consistant à le retraduire vers l'anglais constitue le « décodage ».

Notez que le premier mot d'entrée pour la passe de décodage est le délimiteur STOP. C'est aussi le dernier mot à être restitué en sortie. Lorsque nous utilisons le modèle pour effectuer réellement de la traduction automatique du français vers l'anglais, le système ne dispose pas de la version anglaise. Mais il peut supposer que le traitement commence par STOP. Puis pour générer chacun des mots suivants, nous donnons au LSTM le mot prédict précédent. Il cesse le traitement lorsque le LSTM prédit que le « mot » suivant devrait être STOP à nouveau. Naturellement, nous devrions travailler aussi de cette façon lors du test. À ce moment-là, nous connaissons la version anglaise, mais nous ne l'utilisons que pour évaluation. Ceci signifie que dans une vraie traduction, nous prédisons le mot suivant de la traduction partiellement sur la base du dernier mot traduit *qui pourrait fort bien être une erreur*. Si c'est le cas, alors il y a beaucoup plus de chances que le mot suivant soit faux, etc.

Dans ce chapitre, nous ignorons la complexité de la véritable traduction automatique en évaluant la capacité de notre programme à prédire correctement le mot anglais suivant, connaissant le mot précédent correct. Bien sûr, dans la véritable traduction automatique il n'y a pas une seule traduction anglaise correcte pour une phrase française particulière, par conséquent ce n'est pas parce que notre programme ne prédict pas le mot exact utilisé dans la version traduite de notre corpus parallèle que cette traduction est mauvaise. L'évaluation objective de la traduction automatique est un sujet d'importance, mais un sujet que nous ignorerons ici.

Encore une dernière simplification avant de nous lancer dans l'écriture d'un programme de traduction automatique par réseau de neurones. La figure 5.1 est présentée comme un diagramme de rétropropagation à travers le temps, par conséquent toutes

les unités RNN de la ligne du bas sont en fait la même unité récurrente, mais à des instants successifs. Même chose pour les unités de la rangée du haut. Comme vous vous en souvenez peut-être, les modèles à rétropropagation à travers le temps ont un hyperparamètre de taille de fenêtre. En traduction automatique, nous voulons traiter une phrase entière à la fois, mais ces phrases sont de toutes les tailles. Dans le Penn Treebank par exemple, elles peuvent comporter de 1 à 150 mots. Nous simplifions notre programme pour ne travailler que sur des phrases ayant moins de 12 mots tant en anglais qu'en français, ou 13 en incluant le mot STOP. Nous rendons alors la taille de chacune des phrases égale à 13 en ajoutant des STOP de remplissage. Ainsi, le programme peut supposer que toutes les phrases ont une même longueur de 13 mots. Considérons donc les deux courtes phrases français–anglais alignées par lesquelles nous avons commencé notre discussion sur la traduction automatique, « edited hansard number 1 » et « hansard révisé numéro 1 ». La phrase française fournie en entrée ressemblerait à ceci :

hansard révisé numéro 1 STOP STOP

et celle en anglais :

STOP edited hansard number 1 STOP STOP STOP STOP STOP STOP STOP STOP

5.2 ÉCRITURE D'UN PROGRAMME DE TRADUCTION AUTOMATIQUE SEQ2SEQ

Commençons par passer en revue les réseaux de neurones récurrents que nous avons étudiés au chapitre 4, mais avec une tournure d'esprit légèrement différente. Jusqu'à maintenant, nous n'avons pas accordé beaucoup d'attention aux bonnes pratiques de développement logiciel. Ici, cependant, pour des raisons que nous allons expliquer, TF nous oblige à faire les choses proprement. Étant donné que nous créons deux réseaux de neurones récurrents pratiquement identiques, nous introduisons la construction `TF variable_scope`. La figure 5.2 présente le code TF pour les deux réseaux de neurones récurrents (ou RNN) dont nous avons besoin dans notre modèle `seq2seq` très simple.

Nous divisons le code en deux parties ; la première crée le RNN d'encodage, et la seconde celui de décodage. Chaque section est incluse dans une commande TF `variable_scope`. Cette fonction reçoit un argument, une chaîne de caractères servant de nom à la *portée* (en anglais, *scope*). Le but de `variable_scope` est de nous permettre de regrouper un certain nombre de commandes de manière à éviter les conflits de noms de variables. Par exemple, les segments du haut et du bas utilisent tous deux la variable `cell` de telle manière que, sans portées séparées, chaque segment aurait écrasé les résultats de l'autre.

5.2 Écriture d'un programme de traduction automatique seq2seq

```
1 with tf.variable_scope("enc") :  
2     F = tf.Variable(tf.random_normal((vfSz, embedSz), stddev=.1))  
3     embs = tf.nn.embedding_lookup(F, encIn)  
4     embs = tf.nn.dropout(embs, keepPrb)  
5     cell = tf.contrib.rnn.GRUCell(rnnSz)  
6     initState = cell.zero_state(batchSz, tf.float32)  
7     encOut, encState = tf.nn.dynamic_rnn(cell, embs,  
8                                         initial_state=initState)  
9  
10    with tf.variable_scope("dec") :  
11        E = tf.Variable(tf.random_normal((veSz, embedSz), stddev=.1))  
12        embs = tf.nn.embedding_lookup(E, decIn)  
13        embs = tf.nn.dropout(embs, keepPrb)  
14        cell = tf.contrib.rnn.GRUCell(rnnSz)  
15        decOut, _ = tf.nn.dynamic_rnn(cell, embs, initial_state=encState)
```

Figure 5.2 – Code TF pour deux RNN dans un modèle de traduction automatique

Mais même si nous avions pris soin de donner à nos variables des noms différents, ce code n'aurait *malgré tout* pas fonctionné correctement. Pour des raisons propres à la conception interne de TF, lorsque `dynamic_rnn` crée les éléments à insérer dans le graphe TF, il utilise toujours le même nom pour pointer sur celui-ci. À moins de placer les deux appels dans des portées séparées (ou si le code est fait de telle sorte qu'il ne se soucie pas que les deux appels n'en constituent en fait qu'un et un seul), nous obtenons un message d'erreur.

Considérons maintenant le code à l'intérieur de chaque portée de variable. Pour l'encodeur, nous créons d'abord l'espace pour les plongements des mots français, `F`. Nous supposons que nous avons un nœud de substitution (ou placeholder) nommé `encIn` qui accepte un tenseur d'indices de mots français de forme `[batchSz,windowSz]`, où `batchSz` est la taille de lot, et `windowSz` la taille de fenêtre. La fonction `embedding_lookup` renvoie alors un tenseur 3D de forme `[batchSz,windowSz, embedSz]` où `embedSz` est la taille de plongement (ligne 3) auquel nous appliquons `dropout` en fixant la probabilité de conserver une connexion à `keepPrb` (ligne 4). Nous créons alors la cellule RNN, cette fois en utilisant la variante GRU du LSTM. La ligne 7 utilise alors la cellule pour produire les sorties et le nouvel état.

Le second GRU est parallèle au premier, sauf que l'appel à `dynamic_rnn` reçoit en entrée la sortie d'état de l'encodeur RNN, plutôt qu'un état initialisé à 0. C'est le `state=encState` de la ligne 15. Conformément au diagramme de la figure 5.1, la sortie mot à mot du RNN décodeur est injectée dans une couche linéaire. La figure ne montre pas, mais le lecteur doit imaginer, que les logits sortant de cette couche sont injectés dans un calcul de perte. Le code ressemblerait à celui de la figure 5.3. La seule chose nouvelle, c'est l'appel à `seq2seq_loss`, une version spécialisée de la perte d'entropie croisée dans le cas où les logits sont des tenseurs 3D. Cette fonction accepte trois arguments, les deux premiers étant standard : les logits et un tenseur 2D des réponses correctes de forme `[batchSz, windowSz]` (c.-à-d. taille de lot par taille

Chapitre 5 · Apprentissage séquence à séquence

```

W = tf.Variable(tf.random_normal([rnnSz,veSz], stddev=.1))
b = tf.Variable(tf.random_normal([veSz, stddev=.1]))
logits = tf.tensordot(decOut,W,axes=[[2],[0]])+b
loss = tf.contrib.seq2seq.sequence_loss(logits, ans,
    tf.ones([batchSz, windowSz]))

```

Figure 5.3 – Code TF pour un décodeur seq2seq

de fenêtre). Le troisième argument permet d'avoir une somme pondérée, dans les cas où certaines erreurs doivent compter davantage dans la perte totale que d'autres. Dans notre cas, nous voulons donner la même importance à chaque erreur, c'est pourquoi dans le troisième argument tous les poids sont égaux à 1.

Comme nous l'avons dit précédemment, l'idée d'ensemble du modèle seq2seq très simple de la figure 5.1 est que la passe d'encodage crée un « résumé » de la phrase française en passant le français au travers du GRU puis en utilisant la sortie d'état final du GRU en tant que résumé. Il y a, cependant, beaucoup de façons différentes de créer de tels résumés de phrases ; beaucoup de travaux de recherche ont été consacrés à l'étude de ces alternatives. La figure 5.4 présente un second modèle qui s'avère légèrement supérieur pour la traduction automatique. La différence d'implémentation est minime : plutôt que de transmettre l'état final de l'encodeur pour qu'il devienne

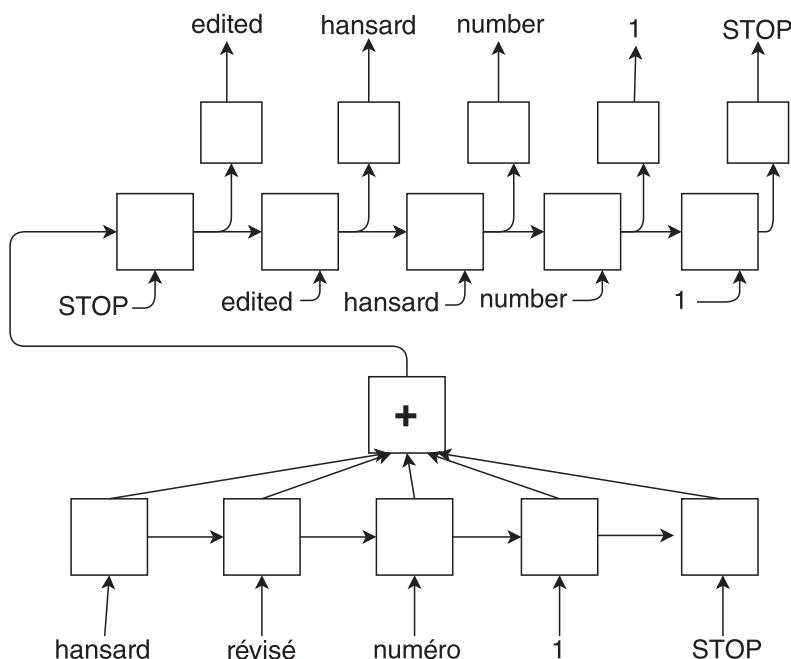


Figure 5.4 – Résumé de phrase seq2seq par addition

l'état initial du décodeur, nous prenons plutôt la somme de tous les états de l'encodeur. Puisque nous avons complété toutes les phrases françaises et anglaises jusqu'à la longueur 13, ceci signifie que nous prenons les 13 états et les additionnons. Ceci en espérant que cette somme soit plus informative que le seul vecteur final, ce qui semble effectivement être le cas.

En pratique, j'ai choisi de prendre la moyenne des vecteurs d'état plutôt que la somme. Si vous revenez au chapitre 1 et regardez le calcul de la passe avant, vous verrez que prendre la moyenne plutôt que la somme ne fait pas de différence au niveau des probabilités finales, étant donné que softmax va faire une mise à l'échelle. Par conséquent, qu'on ait ou non divisé auparavant par la taille de fenêtre (13), on aboutira au même résultat. De plus, la direction du gradient des paramètres ne change pas non plus. Ce qui change effectivement c'est l'ordre de grandeur de la modification que nous apportons aux valeurs des paramètres. Dans la situation actuelle, prendre la somme est à peu près équivalent à multiplier le taux d'apprentissage par 13. En règle générale, il est préférable dans ce type de situation de conserver des valeurs de paramètres proches de zéro et de modifier le taux d'apprentissage directement.

5.3 L'ATTENTION DANS SEQ2SEQ

La notion d'*attention* dans les modèles seq2seq découle de l'idée que, bien qu'en général nous ayons besoin de comprendre une phrase entière avant de pouvoir la traduire, en pratique, pour un groupe donné de mots traduits, certaines parties de la phrase source sont plus importantes que d'autres. En particulier, la plupart du temps, les premiers mots français informent sur les premiers mots anglais, le milieu de la phrase française détermine le milieu de la phrase anglaise, etc. Si c'est particulièrement vrai pour l'anglais et le français qui sont des langues très similaires, même les langues n'ayant pas de similitudes évidentes possèdent cette propriété. Ceci en raison de la distinction entre ce qui est donné et ce qui est nouveau. Il semble se vérifier dans toutes les langues que, lorsque nous apportons un élément nouveau à propos de choses dont on a déjà parlé auparavant (et c'est d'ordinaire ce qui se passe dans une conversation cohérente ou un écrit), nous mentionnons d'abord le « donné » (ce dont nous avons déjà parlé) et ne mentionnons qu'ensuite l'élément nouveau. Dans une conversation à propos de Jack, nous dirons « Jack a mangé un cookie », mais, si nous étions en train de parler de notre dernière fournée de cookies, « un des cookies a été mangé par Jack ».

La figure 5.5 illustre une petite variante du mécanisme de sommation de seq2seq de la figure 5.4 dans laquelle le résumé concaténé au plongement du mot anglais est injecté dans la cellule du décodeur à chaque position de fenêtre. Ceci contraste avec la figure 5.4, où seul le mot anglais est injecté. De notre point de vue, ce modèle porte une égale attention à tous les états de l'encodeur lorsqu'on travaille sur toutes les parties du texte anglais. Dans les modèles avec mécanisme d'attention, nous modifions cela de sorte que différents états soient mélangés en différentes proportions

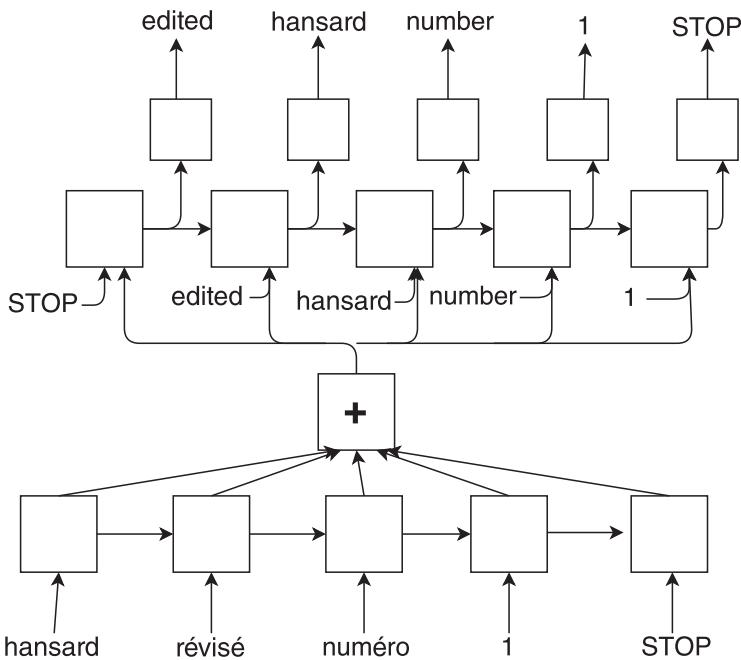


Figure 5.5 – Seq2seq où le résumé de l'encodeur est injecté directement dans chaque position de fenêtre du décodeur

avant d'être transmis au décodeur RNN. Nous appelons ceci *attention à la position uniquement* (en anglais, *position-only attention*). Les véritables modèles avec mécanisme d'attention sont plus compliqués, ils pourront faire l'objet de lectures supplémentaires.

Nous allons donc construire un procédé d'attention où, par exemple, l'attention qu'un mot anglais en position i porte à l'état de l'encodage français en position j ne dépend que de i et de j . En général, plus i et j sont proches, plus grande est l'attention. La figure 5.6 présente une matrice de poids imaginaire pour un modèle dans lequel l'encodeur (français) a une taille de fenêtre de 4, et le décodeur une taille de 3. Jusqu'ici nous avions adopté des tailles de fenêtres toutes deux égales à 13, mais rien n'impose qu'elles soient identiques. De plus, étant donné qu'un texte

$$\begin{matrix} 1/2 & 1/6 & 1/6 \\ 1/6 & 1/3 & 1/6 \\ 1/6 & 1/3 & 1/3 \\ 1/6 & 1/6 & 1/3 \end{matrix}$$

Figure 5.6 – Une possibilité de matrice de pondération permettant d'affecter plus de poids aux positions français/anglais correspondantes

français contient environ 10 % de mots de plus que sa traduction anglaise correspondante, nous avons une bonne raison d'associer une taille de fenêtre légèrement plus grande pour le français à une plus petite pour l'anglais. De plus, pour des motifs pédagogiques, rendre la matrice asymétrique nous aide à retrouver quels nombres correspondent à des positions de mots anglais, et quels nombres correspondent au français.

Ici nous définissons $W[i,j]$ comme le poids accordé au i -ième état français lorsqu'il est utilisé pour prédire le j -ième mot anglais. Le poids total pour un mot anglais quelconque est la somme de la colonne, que nous avons faite égale à 1. Ceci signifie que la première colonne donne les poids pour le premier mot anglais. En étudiant la première colonne, nous voyons que le premier état français compte pour la moitié de l'intensité (dans notre taille de fenêtre imaginaire de 4), et les trois états français restants se partagent équitablement le reste de l'intensité. Pour le moment, nous définissons dans notre programme une version $13 * 13$ de la figure 5.6 sous forme d'une constante TF.

Ensuite, étant donné cette matrice $13 * 13$ de poids, comment l'utilisons-nous pour faire varier l'attention pour une sortie anglaise particulière ? La figure 5.7 présente le code TensorFlow et des exemples de calcul numérique pour une taille de lot de 2, une taille de fenêtre de 3 et une taille de RNN de 4. En haut, nous voyons une sortie imaginaire d'encodeur `encOut`. La taille de lot est 2, et pour simplifier nous avons fait deux lots identiques. À l'intérieur de chaque lot, nous avons les trois vecteurs de longueur 4, chacun d'eux correspondant à la sortie du RNN (de taille 4) à une des positions de la fenêtre. Par exemple, dans le lot 0, le premier vecteur d'état (avec des indices débutant à 0) est (1, 1, 1, 1).

Ensuite, nous avons notre vecteur de poids pré-déterminé, `wAT`, de dimensions $4 * 3$ (tailles des fenêtres). Il croise la position d'état du français avec la position de mot de l'anglais. Par conséquent, la première colonne indique en réalité que le premier mot anglais doit recevoir un vecteur d'état égal à 60 % du premier vecteur d'état français, et 20 % de chacun des vecteurs d'état des deux autres RNN. Ce vecteur des poids est conçu de telle sorte que la somme des poids pour chaque mot anglais soit égale à 100 %.

Viennent ensuite les trois commandes TF qui nous permettent de récupérer les états non pondérés de l'encodeur et de produire les versions pondérées pour chaque décision de mot anglais. Tout d'abord, nous réorganisons le tenseur de sortie de l'encodeur, de `[batchSz, windowSz, rnnSz]` en `[batchSz, rnnSz, windowSz]`. Ceci est effectué par la commande `tf.transpose`. Cette commande accepte deux arguments, le tenseur à transposer et un vecteur d'entiers entre crochets spécifiant la transposition à effectuer. Ici nous avons demandé `[0, 2, 1]` : maintenir en place la dimension d'indice 0, prendre la dimension qui avait pour indice 2 pour en faire la dimension 1, et prendre la dimension qui avait pour indice 1 pour en faire la dernière dimension. Nous présentons le résultat de cette transformation en exécutant `print sess.run(encAT)`.

Chapitre 5 · Apprentissage séquence à séquence

```
eo=  (( ( 1, 2, 3, 4),
        ( 1, 1, 1, 1),
        ( 1, 1, 1, 1),
        ( -1, 0,-1, 0)),
       (( 1, 2, 3, 4),
        ( 1, 1, 1, 1),
        ( 1, 1, 1, 1),
        ( -1, 0,-1, 0)) )
encOut=tf.constant(eo, tf.float32)

AT = ( ( 0.6, 0.25, 0.25 ),
       ( 0.2, 0.25, 0.25 ),
       ( 0.1, 0.25, 0.25 ),
       ( 0.1, 0.25, 0.25 ) )
wAT = tf.constant(AT, tf.float32)

encAT = tf.tensordot(encOut,wAT,[[1],[0]])
sess= tf.Session()

print sess.run(encAT)
'''[[[ 0.80000001  0.5          0.5          ]
   [ 1.50000012  1.           1.           ]
   [ 2.          1.           1.           ]
   [ 2.70000005  1.5          1.5          ]]
  ...]'''

decAT = tf.transpose(encAT, [0, 2, 1])
print sess.run(decAT)
'''[[[ 0.80000001  1.50000012  2.          2.70000005]
   [ 0.5          1.           1.           1.5          ]
   [ 0.5          1.           1.           1.5          ]]
  ...]'''
```

Figure 5.7 – Calculs d'attention simplifiés avec batchSz = 2, windowSz = 3 et rnnSz = 4

Nous avons effectué cette transposition pour simplifier la multiplication de matrices de l'étape suivante (le `tensordot`). En fait, si nous n'avions pas la complication de la taille de lot, nous multiplierions des tenseurs de forme `[rnnSz, windowSz] * [windowSz, windowSz]`, et nous pourrions utiliser la multiplication matricielle standard (`matmul`). La dimension supplémentaire induite par la taille de lot interdit cette possibilité et nous en revenons à :

```
| encAT = tf.tensordot)(encOut.wAT, [[1],[0]])
```

Enfin, nous inversons la transposition faite deux étapes auparavant, pour remettre dans leur format d'origine les états de sortie de l'encodeur.

Il est bon de s'arrêter un peu pour comparer le résultat final en bas de la figure 5.7 avec la sortie de notre encodeur imaginaire en haut de la figure. La colonne (0.6, 0.2, 0.2) indique qu'il faut transmettre au premier mot anglais un vecteur composé de 60 % d'état 0, qui est [1, 2, 3, 4]. Nous nous attendons donc à ce que l'état résultant augmente à mesure que nous nous déplaçons de la gauche vers la droite, ce que fait (0.6, 1.4, 1.8, 2.6). Le deuxième état, entièrement composé de 1, n'a pas beaucoup d'effet (il ajoute 0.2 à chaque position). Mais le dernier état (0, -1, 0, -1) devrait donner un résultat avec des hauts et des bas, ce qui est à peu près le cas (les composantes de (0.6, 1.4, 1.8, 2.6) augmentent toutes, mais les augmentations de la première et de la troisième sont supérieures à celle de la seconde).

Une fois que nous avons préparé les états pondérés de l'encodeur pour alimenter le décodeur, nous complétons chacun d'eux avec le plongement du mot anglais avec lequel nous avions déjà alimenté le RNN décodeur. Ceci complète notre système de traduction automatique à mécanisme d'attention. Cependant, une chose est importante pour terminer cet exemple : notre programme pourrait facilement apprendre les poids d'attention en faisant de la matrice 13×13 d'attention une variable TF plutôt qu'une constante. L'idée est analogue à celle du chapitre 3, lorsque nous avons voulu faire apprendre au réseau de neurones les noyaux de convolution. La figure 5.8 présente une partie des poids appris de cette façon. Les nombres en gras correspondent à la valeur la plus élevée de la ligne. Ils mettent en évidence le décalage vers la droite attendu — la traduction des mots au début, au milieu, ou à la fin de l'anglais doit, en général, accorder plus d'attention au début, au milieu ou à la fin du français.

-6.3	1.3	.37	.13	.06	.04	.11	.10	.02
-.66	-.44	.64	.26	.16	.02	.03	.04	.06
-.38	-.47	-.04	.63	.18	.10	.07	.06	.12
-.30	-.44	-.35	-.15	.48	.24	.06	.13	0
-.02	-.16	-.35	-.37	-.23	.12	.32	.22	.11
.05	-.11	-.11	-.35	-.04	-.22	.05	.26	.24
.10	.02	-.04	-.23	-.32	-.33	-.25	-.01	.28
0	.03	.01	-.18	-.21	-.26	-.30	-.11	-.17

Figure 5.8 – Poids d'attention pour la portion supérieure gauche 8×9 de la matrice 13×13 de pondération d'attention

5.4 SEQ2SEQ MULTI-LONGUEURS

À la section précédente, nous avons restreint nos paires de traductions à des exemples où les deux phrases comportent 12 mots ou moins (13 y compris un STOP de délimitation). Dans une traduction automatique véritable, de telles limitations ne seraient

pas permises. D'un autre côté, si l'on avait par exemple une fenêtre de 65 qui permettait la traduction de la quasi-totalité des phrases que nous rencontrons, cela voudrait dire qu'une phrase plus normale se retrouverait complétée par une bonne quarantaine de STOP pour atteindre la taille voulue. La solution qui a été adoptée dans le monde de la traduction automatique par réseaux de neurones consiste à créer des modèles avec plusieurs tailles de fenêtre. Nous allons montrer maintenant comment cela fonctionne.

Considérons à nouveau les lignes 5-8 de la figure 5.2. De notre point de vue actuel, il est frappant qu'aucune des deux commandes TF assurant la mise en place du RNN encodeur ne mentionne la taille de fenêtre. Pour la création du GRU, il faut connaître la taille du RNN puisque, après tout, il est en charge de l'allocation d'espace pour les variables du GRU. Mais la taille de fenêtre n'entre pas en ligne de compte dans cette activité. D'un autre côté, `dynamic_rnn` a *certainement* besoin de connaître la taille de fenêtre, puisqu'elle est responsable de la création de la portion du graphe TF qui exécute la rétropropagation à travers le temps. Et celle-ci obtient l'information, dans ce cas par le moyen de la variable `embds`, qui est de taille `[batchSz, windowSz, embedSz]`. Supposons donc que nous décidions de gérer deux combinaisons différentes de taille de fenêtre. La première gère par exemple toutes les phrases où le français fait 14 mots ou moins et l'anglais 12 mots ou moins. Pour la seconde, nous doublons ces valeurs, 28 et 24. Si la phrase française fait plus de 28 mots ou si la phrase anglaise fait plus de 24 mots, nous rejetons l'exemple. Si le français ou l'anglais dépasse les 14 ou 12 mots respectivement, tout en restant dans la limite des 28 ou 24, nous mettons la paire de phrases dans le groupe le plus large. Nous créons alors un GRU à utiliser dans les deux `dynamic_rnn` comme suit :

```
cell = tf.contrib.rnn.GRUCell(rnnSz)
encOutSmall, encStateS = tf.nn.dynamic_rnn(cell, smallerEmbs, ...)
encOutLarge, encStateL = tf.nn.dynamic_rnn(cell, largerEmbs, ...)
```

Remarquez qu'alors que nous avons deux `dynamnic_rnn` (ou potentiellement cinq ou six), en fonction des intervalles de taille que nous voulons prendre en compte, ils partagent tous la même cellule GRU de sorte qu'ils apprennent et partagent la même connaissance du français. De manière similaire, nous créerions une cellule GRU pour l'anglais, etc.

5.5 EXEMPLES DE PROGRAMMATION

Ce chapitre s'étant concentré sur les technologies de réseaux de neurones utilisées en traduction automatique, nous entreprenons par conséquent ici de construire un programme de traduction. Malheureusement, dans l'état d'avancement actuel de l'apprentissage profond, c'est très difficile. Si les progrès récents ont de quoi impressionner, les programmes qui se targuent de bons résultats requièrent environ un

milliard d'exemples d'entraînement et des jours d'apprentissage, sinon plus. Cela ne peut donc pas être un sujet de travaux pratiques.

Au lieu de cela, nous allons utiliser environ un million d'exemples d'entraînement — une partie du corpus Canadian Hansard se limitant aux phrases françaises et anglaises comportant toutes deux 12 mots ou moins (13 en incluant le délimiteur STOP). Nous allons aussi donner de petites valeurs à nos hyperparamètres : taille de plongement de 30, taille de RNN de 64, et nous n'effectuons qu'une seule époque (ou itération d'entraînement). Nous fixons le taux d'apprentissage à 0.005.

Comme dit précédemment, il est difficile d'évaluer les programmes de traduction automatique, sauf à tout lire, corriger et noter à la main. Nous allons adopter un procédé particulièrement simpliste. Celui-ci consiste à parcourir la traduction anglaise correcte jusqu'à atteindre le premier STOP. Un mot généré automatiquement est considéré correct s'il est dans la même position que le mot anglais identique du Hansard. Encore une fois, nous cessons de noter après le premier STOP. Par exemple,

```
the law is very clear . STOP
the *UNK* is a clear . STOP
```

serait comptabilisé comme 5 mots corrects sur 7. À la fin, nous divisons le nombre total de mots corrects par le nombre total de mots dans les phrases anglaises du jeu de développement.

Avec cette métrique, j'ai obtenu 59 % de mots corrects sur le jeu de test après une époque (65 % après la deuxième, et 67 % après trois époques). Savoir si c'est un bon ou un mauvais résultat dépend de vos attentes antérieures. Compte tenu de nos remarques précédentes à propos de la nécessité d'un milliard d'exemples d'entraînement, vos attentes étaient probablement réduites ; les nôtres l'étaient certainement. Cependant, un examen des traductions produites montre que même 59 % constitue une évaluation faussement optimiste. Nous avons exécuté le programme en imprimant la première phrase de chaque 400^e lot, avec une taille de lot de 32. Voici les deux premiers exemples d'entraînement qui ont été correctement traduits :

Époque 0 Lot 6401 Cor : 0.432627

* * *

* * *

* * *

Époque 0 Lot 6801 Cor : 0.438996

le très hon. jean chrétien

right hon. jean chrétien

right hon. jean chrétien

Les lignes « * * * » ont été insérées entre les sessions du Parlement par le rédacteur. Sur les 351 846 lignes du fichier, 14 410 comportent uniquement ce marquage. À ce

Chapitre 5 · Apprentissage séquence à séquence

niveau de la première époque (la moitié est déjà effectuée), le programme a sans nul doute mémorisé le mot anglais correspondant (qui, bien sûr, est identique). Dans la même veine, l'identité de chaque orateur est systématiquement ajoutée devant son propos. Jean Chrétien était Premier ministre du Canada à l'époque correspondant à ce corpus, et il semble qu'il ait parlé 64 fois. C'est pourquoi la traduction de cette phrase française a été aussi mémorisée. À vrai dire, on pourrait se demander s'il y a des traductions correctes qui n'ont *pas* été mémorisées. La réponse est oui, mais il n'y en a pas tellement. Voici les six dernières sur un jeu de test de 22 000 exemples :

19154 the problem is very serious .
21191 hon. george s. baker :
21404 mr. bernard bigras (rosemont , bq) moved :
21437 mr. bernard bigras (rosemont , bq) moved :
21741 he is correct .
21744 we will support the bill .

Celles-ci proviennent d'une exécution avec double attention pour les mots correspondants, une taille de RNN de 64, un taux d'apprentissage de 0.005, et une seule époque. La mesure d'exactitude décrite précédemment était de 68.6 % pour le jeu d'entraînement. Nous avons imprimé chaque exemple de test qui était complètement correct et ne correspondait à aucune phrase anglaise de l'entraînement.

Il est intéressant et utile de se faire une idée de la façon dont l'état évolue entre les mots d'une phrase. En particulier, le premier modèle seq2seq utilisait l'état final de l'encodeur pour amorcer le décodeur anglais. Étant donné que nous avons simplement pris l'état au mot 13, quelle que soit la longueur du français d'origine (12 mots maximum), nous faisons l'hypothèse que nous n'avons pas perdu grand-chose en ne récupérant l'état qu'après huit STOP par exemple alors que le texte français original ne comportait que cinq mots. Pour tester cela, nous avons examiné les 13 états produits par l'encodeur et calculé dans chaque cas la similarité cosinus entre les états successifs. Ce qui suit provient d'une phrase d'entraînement traitée durant la troisième époque :

Version anglaise : that has already been dealt with .

Traduction : it is a . a .

Indices des mots français : [18, 528, 65, 6476, 41, 0, 0, 0, 0, 0, 0, 0, 0]

Similarité entre états : 0.078 0.57 0.77 0.70 0.90 1 1 1 1 1 1 1 1 1 1

Vous remarquerez peut-être tout d'abord la piètre qualité de la « traduction » (2 mots corrects sur 8, « . » et STOP). Les similarités entre états, par contre, semblent raisonnables. En particulier, une fois que nous avons atteint la fin de la phrase au mot f5 (en français), toutes les similarités entre états valent 1.0 — c'est-à-dire que l'état ne change plus du tout du fait du remplissage, comme nous l'espérions.

Les états entre lesquels la similarité est la plus faible sont le premier et le second. À partir de là, la similarité augmente de façon quasi monotone. En d'autres termes,

à mesure que nous progressons à travers la phrase, il y a davantage d'informations antérieures qui valent la peine d'être préservées, c'est pourquoi la plus grande partie de l'ancien état est transmis, ce qui fait que l'état suivant est similaire à l'état courant.

5.6 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

Dans les années 1980, un laboratoire de recherche d'IBM dirigé par Fred Jelinek a commencé à travailler sur un projet de création d'un programme de traduction par ordinateur visant à apprendre à la machine à traduire en repérant les régularités statistiques. Le « repérage » se faisait par apprentissage automatique bayésien et les données, comme dans ce chapitre, provenaient du corpus Canadian Hansard [BCP⁺88]. Cette approche, après quelques années de railleries, est devenue l'approche dominante et l'est restée jusqu'à récemment. Désormais, les approches d'apprentissage profond gagnent rapidement en popularité et il ne faudra plus longtemps pour que tous les systèmes commerciaux de traduction automatique soient basés sur des réseaux de neurones, si ce n'est pas déjà le cas. Un des premiers exemples d'approche par réseaux de neurones est celui de Kalchbrenner et Blunsom [KB13].

L'alignement dans les modèles seq2seq a été présenté dans Dzmitry Bahdanau *et al.* [BCB14]. Ce groupe semble aussi avoir été le premier à adopter la terminologie désormais standard de *neural machine translation* (traduction automatique neuronale) pour cette approche. Dans le modèle d'attention à la position seule de ce chapitre, le modèle ne reçoit que les valeurs numériques des emplacements des mots français et anglais lorsqu'il doit décider quel poids accorder à l'état français correspondant. Dans [BCB14], le modèle a aussi des informations sur les états du LSTM aux emplacements français et anglais.

En matière de tutoriel de traduction automatique en ligne, Thad Luong *et al.* en ont mis un en ligne peu avant la finalisation de la version originale américaine de ce livre [TL]. Le précédent tutoriel de Google sur seq2seq et la traduction automatique n'était pas très bon d'un point de vue pédagogique (imaginez un livre de cuisine vous apprenant à faire des pancakes en disant « mélangez ensemble 400 000 litres de lait et 1 million d'œufs »), mais celui-là semble plutôt raisonnable et peut fort bien constituer une bonne base en vue d'explorer plus avant la traduction automatique neuronale en particulier, et seq2seq en général.

En dehors de la traduction automatique, il existe de nombreuses autres tâches pour lesquelles les modèles seq2seq ont été mis à contribution. Un domaine qui s'avère particulièrement « chaud » actuellement est celui des *chatbots*, des programmes à qui l'on transmet des expressions de la conversation courante et qui tentent de poursuivre la conversation. Il s'agit aussi d'une des caractéristiques de base des assistants vocaux, comme par exemple Alexa d'Amazon. Il existe un magazine consacré aux chatbots en ligne et un article anglais intitulé « Why Chatbots Are the Future of Marketing ». Un article en ligne de Surlyadeepan Ram décrit un projet envisageable sur ce sujet [Ram17].

Exercices

5.1 Supposons que nous utilisions seq2seq avec des longueurs multiples pour un programme de traduction automatique et que nous ayons choisi deux tailles de phrases, l'une jusqu'à 7 mots (et délimiteurs STOP) pour l'anglais et 10 pour le français, l'autre jusqu'à 10 mots pour l'anglais et 13 pour le français. Écrivez l'entrée si la phrase française est « A B C D E F » et la phrase anglaise « M N O P Q R S T ».

5.2 Nous avons choisi d'illustrer l'attention présentée à la section 5.3 avec une forme particulièrement simple, une qui ne base la décision d'attention que sur l'emplacement de l'attention à la fois dans le français et l'anglais. Une version plus sophistiquée base la décision sur les vecteurs d'état d'entrée pour la position du texte anglais sur laquelle nous travaillons et sur le vecteur d'état proposé dont nous décidons de l'influence. Quoique pouvant permettre des jugements plus sophistiqués, cela requiert une complication significative du modèle. En particulier, nous ne pouvons plus utiliser la version standard TF de rétropropagation à travers le temps dans un réseau récurrent pour le décodeur. Expliquez pourquoi.

5.3 On a observé fréquemment qu'en introduisant le langage source dans le décodeur seq2seq *en marche arrière* (mais en laissant le décodeur travailler vers l'avant) on améliorait les performances de la traduction automatique d'un montant faible mais constant. Trouvez une explication plausible à cela.

5.4 En principe, nous pourrions avoir un modèle seq2seq avec deux pertes que nous additionnerions pour trouver la perte totale. L'une serait la perte de l'actuel système de traduction automatique pour n'avoir pas prédit le mot cible suivant avec une probabilité de 1. La seconde pourrait être une perte dans l'encodeur, en demandant à l'encodeur de prédire le mot source (p. ex. français) suivant, c'est-à-dire une perte du modèle linguistique. **(a)** Expliquez de manière plausible pourquoi ceci dégraderait les performances. **(b)** Expliquez de manière plausible pourquoi ceci améliorerait les performances.

APPRENTISSAGE PAR RENFORCEMENT PROFOND

L'apprentissage par renforcement (en anglais, *reinforcement learning*, ou RL) est la branche de l'apprentissage automatique qui consiste à apprendre comment un agent doit se comporter dans un *environnement* de manière à maximiser une *récompense*. Naturellement, l'apprentissage par renforcement *profond* restreint la méthode d'apprentissage à l'apprentissage profond.

En règle générale, l'environnement est défini mathématiquement comme un *processus décisionnel de Markov* (en anglais, *Markov decision process*, ou MDP). Les MDP sont constitués d'un ensemble d'états $s \in S$ que peut prendre l'agent (p. ex. des emplacements sur une carte), un ensemble fini d'actions ($a \in A$), une fonction de transition $T(s, a, s') = \Pr(S_{t+1} = s' | S_t = s, A = a)$ qui est la probabilité que l'agent passe d'un état à un autre en effectuant l'action a , une fonction de récompense associant à un état, une action, et un état suivant des récompenses $R(s, a, s')$ et un *rabais* $\gamma \in [0, 1]$ (qui sera expliqué un peu plus loin). En général les actions ne sont pas déterministes, c'est pourquoi T définit, pour chaque action effectuée à partir d'un état donné, une distribution sur l'ensemble des états résultants possibles. Ces modèles sont appelés processus décisionnels de Markov parce qu'ils font l'*hypothèse de Markov* : si nous connaissons l'état courant, l'historique (la façon dont nous sommes parvenus à cet état) n'a pas d'importance.

Dans les MDP, le temps est une variable discrète (et non continue). À chaque instant, l'agent est dans un certain état, effectue une action qui le conduit à un nouvel état, et reçoit une récompense, souvent zéro. L'objectif est de maximiser sa *récompense future avec rabais* (en anglais, *discounted future reward*) définie par :

$$\sum_{t=0}^{t=\infty} \gamma^t R(s_t, a_t, s_{t+1}) \quad (6.1)$$

Si $\gamma < 1$ alors cette somme est finie. Si γ n'est pas indiqué (ce qui équivaut à valoir 1) alors la somme peut croître jusqu'à l'infini, ce qui complique le calcul. Il est courant de donner à γ la valeur 0.9. La quantité figurant dans l'équation 6.1 est appelée *récompense future avec rabais* car les multiplications répétées par une quantité inférieure à 1 amènent le modèle à « rabaisser » les récompenses dans le futur par rapport à celles obtenues à l'instant présent.

Notre objectif est de *résoudre* le processus décisionnel de Markov, c'est-à-dire de trouver une *politique* optimale. Une politique est une fonction $\pi(s) = a$ qui, pour chaque état s , spécifie l'action a que l'agent devrait effectuer. Une politique est dite

optimale, et notée $\pi^*(s)$, si les actions spécifiées conduisent à une espérance de récompense future avec rabais maximale. Le terme d'*espérance* renvoie à la définition donnée à la section 2.4.3. Les actions n'étant pas déterministes, une même action peut conduire à des récompenses nettement différentes.

Ce chapitre s'intéresse donc à l'apprentissage de politiques MDP optimales : tout d'abord en utilisant des méthodes dites *tabulaires*, puis en utilisant leurs équivalentes en apprentissage profond.

6.1 ITÉRATION SUR LES VALEURS

Avant de parler de résolution de MDP, il faut répondre à une question de base : l'agent a-t-il « connaissance » des fonctions T et R ou doit-il errer dans l'environnement pour les apprendre en même temps qu'il crée sa politique ? Cela simplifie grandement les choses de connaître T et R , c'est pourquoi nous commencerons par ce cas. Dans cette section, nous supposerons également qu'il n'y a qu'un nombre fini d'états s .

L'*itération sur les valeurs* est à peu près aussi simple que l'est l'apprentissage de politique dans les MDP. En fait, on peut dire que ce n'est même pas un algorithme d'apprentissage, du fait qu'il n'a besoin ni d'exemples d'entraînement ni d'interactions avec l'environnement. L'algorithme est donné à la figure 6.1. V est une *fonction de valeur*, un vecteur de taille $|s|$ où chaque entrée $V(s)$ est la meilleure espérance de récompense avec rabais que nous puissions obtenir lorsque nous partons de l'état s . Q (appelée simplement *fonction Q*) est un tableau de taille $|s|$ par $|a|$ dans lequel nous rangeons l'estimation courante de la récompense avec remise lorsqu'on prend l'action a dans l'état s . La fonction de valeur V associe une valeur qui est un nombre réel à chaque état : plus ce nombre est élevé, plus il est souhaitable d'atteindre cet état. Q est plus précis : il fournit la valeur que nous pouvons espérer pour chaque couple état-action. Si nos valeurs dans V sont correctes, alors la ligne 2(a)i affectera $Q(s, a)$ correctement. Celle-ci indique que la valeur de $Q(s, a)$ se compose de la récompense immédiate $R(s, a, s')$ plus la valeur pour l'état dans lequel nous aboutissons, telle que spécifiée par V . Les actions n'étant pas déterministes, nous devons effectuer une sommation sur tous les états possibles. Ceci nous donne l'espérance.

1. Pour tout s initialiser $V(s) = 0$

2. Répéter jusqu'à convergence :

(a) Pour tout s :

- i. Pour tout a , calculer $Q(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(s'))$
- ii. $V(s) = \max_a Q(s, a)$

3. Renvoyer Q

Figure 6.1 – Algorithme d'itération sur les valeurs

Une fois que nous avons calculé correctement Q , nous pouvons déterminer la politique optimale π en choisissant toujours l'action $a = \arg \max_{a'} Q(s, a')$. Ici $\arg \max_x g(x)$ renvoie la valeur de x pour laquelle $g(x)$ est maximum.

Pour donner un exemple concret, étudions un MDP très simple : le *problème du lac gelé*. Ce jeu fait partie du *Open AI Gym*, un groupe de jeux informatiques avec des API uniformes, pratiques pour expérimenter l'apprentissage par renforcement. Nous avons un quadrillage $4 * 4$ (le lac) présenté à la figure 6.2. Le jeu consiste à aller du point de départ (état 0 en haut à gauche) jusqu'au but (en bas à droite) sans tomber dans un trou. Nous recevons une récompense de 1 lorsque nous effectuons une action et arrivons au but. Toutes les autres combinaisons état-action-état reçoivent une récompense de 0. Si nous tombons dans un trou (ou atteignons le but), le jeu s'arrête et si nous rejoignons nous revenons au point de départ. Sinon, nous pouvons choisir, en indiquant un nombre de 0 à 3 respectivement, d'aller à gauche (g), vers le bas (b), à droite (d) ou vers le haut (h) avec une certaine probabilité de glisser et de ne pas aller dans la direction choisie. En réalité, le mode de programmation de ce jeu Open AI Gym fait qu'une action (p. ex. aller à droite) nous conduit avec une égale probabilité dans chacun des états adjacents, à l'exception de l'exact opposé (ici, aller à gauche), le terrain est donc *très* glissant. Si une action aboutissait à nous faire sortir du lac, elle nous laisserait au lieu de cela à l'emplacement de départ.

0 : S	1 : F	2 : F	3 : F
4 : F	5 : H	6 : F	7 : H
8 : F	9 : F	10 : F	11 : H
12 : H	13 : F	14 : F	15 : G

S	point de départ
F	zone gelée
H	trou
G	but

Figure 6.2 – Le problème du lac gelé

Pour calculer V et Q pour le lac gelé, nous parcourons l'ensemble des états de s et recalculons $V(s)$. Considérons l'état 1. Ceci nécessite de calculer $Q(1, a)$ pour chacune des quatre actions puis d'affecter à $V(1)$ le maximum des quatre valeurs Q . Commençons donc par calculer ce qui se passe si nous choisissons de nous déplacer vers la gauche, $Q(1, g)$. Pour cela, nous devons effectuer une sommation sur tous les s' , c'est-à-dire sur tous les états du jeu. Il y a 16 états dans le jeu, mais en commençant à l'état 1 nous pouvons en atteindre seulement trois avec une probabilité non nulle, les états 0, 5 et 1 lui-même (en essayant de se déplacer vers le haut, étant bloqué par le bord du lac, et par conséquent en ne bougeant pas du tout). Donc, en ne prenant en compte que les états s' ayant des valeurs $T(1, g, s')$ non nulles, nous calculons la somme suivante :

$$Q(1, g) = 0.33 * (0 + 0.9 * 0) + 0.33 * (0 + 0.9 * 0) + 0.33 * (0 + 0.9 * 0) \quad (6.2)$$

$$= 0 + 0 + 0 \quad (6.3)$$

$$= 0 \quad (6.4)$$

Le premier terme de la somme indique que, lorsque nous tentons de nous déplacer vers la gauche, avec une probabilité de 0.33, nous terminons à l'état 0. Nous obtenons pour cela une récompense de 0, et l'estimation de notre future récompense est de $0.9 * 0$. Cette valeur est nulle, et il en aurait été de même si, au lieu d'aller à gauche, nous avions glissé et terminé à l'état 5 (en glissant vers le bas) ou à l'état 1 (en glissant vers le haut). Donc $Q(1, g) = 0$. Étant donné que les valeurs V pour les trois états que nous pouvons atteindre à partir de l'état 1 sont toutes nulles, $Q(1, b)$ et $Q(1, h)$ sont tous deux nuls également, et la ligne 2(a)ii de la figure 6.1 aboutit à $V(1) = 0$.

En fait, lors de la première itération, V continue à valoir 0 jusqu'à ce que nous parvenions à l'état 14, où nous obtenons finalement des valeurs non nulles pour $Q(14, b)$, $Q(14, d)$ et $Q(14, h)$:

$$Q(14, b) = 0.33 * (0 + 0.9 * 0) + 0.33 * (0 + 0.9 * 0) + 0.33 * (1 + 0.9 * 0) = 0.33$$

$$Q(14, d) = 0.33 * (0 + 0.9 * 0) + 0.33 * (0 + 0.9 * 0) + 0.33 * (1 + 0.9 * 0) = 0.33$$

$$Q(14, h) = 0.33 * (0 + 0.9 * 0) + 0.33 * (0 + 0.9 * 0) + 0.33 * (1 + 0.9 * 0) = 0.33$$

et $V(14) = 0.33$.

La partie gauche de la figure 6.3 présente le tableau des valeurs V après la première itération. L'itération sur la valeur est un des algorithmes qui recherchent une politique optimale en maintenant des tableaux des meilleures estimations des valeurs de la fonction. D'où le nom de *méthodes tabulaires*.

À la deuxième itération, de nouveau la plupart des valeurs restent nulles, mais cette fois les états 10 et 13 obtiennent aussi des valeurs non nulles dans Q et V car à partir de celles-ci on peut passer à l'état 14 et, comme nous venons de l'observer, maintenant $V(14) = 0.33$. Les valeurs de V après la seconde itération sont présentées sur la partie droite de la figure 6.3.

Une autre manière de concevoir l'itération sur la valeur est que chaque modification apportée à V (et à Q) incorpore l'information précise sur ce qui va se passer lors du prochain mouvement (la récompense R que nous obtenons) tout en conservant les informations initialement inexactes déjà incorporées dans ces fonctions. Finalement, les fonctions incluent de plus en plus d'informations sur les états que nous n'avons pas encore atteints.

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0.33	0

0	0	0	0
0	0	0	0
0	0	0.1	0
0	0.1	0.46	0

Figure 6.3 – Valeurs des états après la première itération et après la deuxième

6.2 APPRENTISSAGE SANS MODÈLE

L’itération sur la valeur suppose que l’apprenant ait accès à tous les détails de l’environnement du modèle. Voyons maintenant le cas opposé : l’*apprentissage sans modèle* (en anglais, *model-free learning*). L’agent peut explorer l’environnement en effectuant un déplacement et reçoit en retour des informations concernant la récompense et l’état suivant, mais il ne connaît ni les probabilités de déplacement T ni la fonction de récompense R .

Supposons que notre environnement soit un processus décisionnel de Markov, alors la façon la plus évidente de planifier dans un environnement sans modèle consiste à errer au hasard dans cet environnement, à collecter des statistiques sur T et R , puis à créer une politique basée sur le tableau Q comme décrit dans la section précédente. La figure 6.4 présente les grandes lignes d’un programme effectuant cela.

La ligne 1 crée le jeu du lac gelé. Pour démarrer une partie (à partir de l’état initial), nous appelons `reset()`. Une partie de jeu du lac gelé se termine soit par une chute dans un trou, soit lorsque le but est atteint. La boucle externe (ligne 2) spécifie que nous allons jouer 1 000 parties. La boucle interne (ligne 4) indique que tout jeu est interrompu après 99 coups (en pratique, cela n’arrive jamais : nous tombons dans un trou ou atteignons le but bien avant). La ligne 5 signifie qu’à chaque étape nous générerons d’abord au hasard l’action suivante (il y a quatre actions possibles, gauche, bas, droite, haut associées dans cet ordre aux valeurs 0 à 3). La ligne 6 est l’étape critique. La fonction `step(act)` reçoit un argument (l’action à effectuer) et renvoie quatre valeurs. La première est l’état après l’action (un entier de 0 à 15), la deuxième est la valeur de la récompense reçue (en général 0, parfois 1). La troisième valeur de retour, rangée dans la variable `dn` dans cet exemple, est un booléen indiquant si le jeu est terminé (c’est-à-dire si nous sommes tombés dans un trou ou avons atteint le but) ou non. Le dernier argument nous informe sur les véritables probabilités des transitions, ce que nous ignorons dans le cas d’un apprentissage sans modèle.

```

0 import gym
1 game = gym.make('FrozenLake-v0')
2 for i in range(1000):
3     st = game.reset()
4     for stps in range(99):
5         act=np.random.randint(0,4)
6         nst,rwd,dn,_=game.step(act)
7         # mise à jour de T et R
8         if dn: break
9

```

Figure 6.4 – Collecte de statistiques pour un jeu Open AI Gym

À bien y réfléchir, se déplacer au hasard dans un jeu n'est franchement pas une bonne façon de collecter des statistiques. Ce qui se passe le plus souvent, c'est que nous tombons dans un trou et revenons au point de départ, puis continuons à collecter des statistiques sur ce qui se passe dans des états proches de l'état de départ. Une bien meilleure idée consiste à apprendre et à explorer au hasard en même temps, en permettant que notre apprentissage influe sur l'endroit où nous allons. Si nous collectons ce faisant des informations utiles, alors nous pouvons progresser de plus en plus loin dans le jeu, et par conséquent en apprendre davantage sur des états plus différents. C'est ce que nous faisons dans cette section, en choisissant, avec une probabilité ϵ , d'effectuer un déplacement au hasard, ou sinon (avec une probabilité $(1 - \epsilon)$) de baser notre décision sur les connaissances glanées jusqu'ici. Si ϵ est fixé, on parle de stratégie *epsilon-gourmande* (en anglais, *epsilon-greedy*).

Il est aussi courant que ϵ décroisse avec le temps : on parle alors de stratégie *epsilon-décroissante*. Une façon simple d'y parvenir est d'avoir un hyperparamètre associé E et de définir $\epsilon = \frac{E}{i+E}$, où i est le nombre de parties déjà jouées (et donc E est le nombre de parties nécessaires pour passer d'un jeu essentiellement au hasard à une stratégie essentiellement apprise). Comme vous pouvez vous y attendre, la façon dont nous choisissons d'explorer ou au contraire de baser notre choix sur notre compréhension actuelle du jeu peut avoir un effet considérable sur notre rapidité d'apprentissage du jeu. Ceci porte le nom de *compromis exploration-exploitation* (en anglais, *exploration-exploitation tradeoff*), et, lorsque nous utilisons notre connaissance du jeu, on dit que nous *exploitons* les connaissances déjà collectées.

Un autre moyen en vogue pour combiner exploration et exploitation est de toujours utiliser les valeurs fournies par la fonction Q mais de les transformer en une distribution de probabilités et de choisir une action conformément à cette distribution, plutôt que de toujours choisir l'action ayant la plus forte valeur (ce dernier algorithme s'appelle *algorithme gourmand* ou *greedy algorithm* en anglais). Si nous avions trois actions et que leurs valeurs Q soient $[4, 1, 1]$, nous choisirions la première les deux tiers du temps, etc.

Cet algorithme dit de *Q-learning* figure parmi les algorithmes les plus utilisés et les plus populaires pour l'apprentissage sans modèle combinant exploration et exploitation. L'idée de base ne consiste pas à apprendre R et T mais à apprendre les tableaux Q et V directement. Pour cela, à la figure 6.4, nous devons modifier les lignes 5 (nous n'agissons plus totalement au hasard) et la ligne 7, afin de modifier Q et V , et non R et T .

Nous avons déjà expliqué ce qu'il fallait faire à la ligne 5, voyons maintenant la ligne 7. Voici nos équations de mise à jour pour le Q-learning :

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R(s, a, n) + \gamma V(n)) \quad (6.5)$$

$$V(s) = \max_{a'} Q(s, a') \quad (6.6)$$

où s est l'état que nous occupons, a est l'action que nous avons choisie, et a' est l'état que nous occupons maintenant, juste après avoir joué un nouveau coup dans la partie à la ligne 6 de la figure 6.4.

La nouvelle valeur de $Q(s, a)$ est un mélange contrôlé par α de son ancienne valeur et de l'information nouvelle — par conséquent α est en quelque sorte un taux d'apprentissage. En règle générale, α est petit. Pour comprendre pourquoi c'est nécessaire, il est utile de comparer ces équations aux lignes 2(a)i et 2(a)ii de l'algorithme d'itération sur les valeurs de la figure 6.1. Là, étant donné que l'algorithme recevait R et T , nous pouvions faire la somme de toutes les issues possibles pour l'action que nous avions choisie. Dans un apprentissage sans modèle, nous ne pouvons pas le faire. Tout ce que nous avons, c'est le résultat du dernier coup joué. La nouvelle information ne se base que sur un déplacement dans notre exploration de l'environnement. Supposons que nous soyons à l'état 14 de la figure 6.2 mais que, à notre insu, la probabilité soit très faible (0.0001) que, si nous nous déplaçons vers le bas à partir de cet état, nous obtenions une « récompense » de -10. Il y a de très fortes chances que cela ne se produise pas, mais si cela se produit tout risque d'être complètement bouleversé. L'enseignement à en tirer est que l'algorithme ne doit pas mettre trop l'accent sur un seul déplacement. En itération sur les valeurs, nous connaissons T et R et, pour faire la part des deux, l'algorithme prend en compte à la fois la possibilité d'une récompense négative et la faible probabilité que cela se produise.

6.3 LES BASES DE L'APPRENTISSAGE PAR RENFORCEMENT PROFOND

Après avoir vu les méthodes tabulaires d'apprentissage par renforcement, nous sommes maintenant en état de comprendre l'*apprentissage par renforcement profond* (en anglais, *deep-Q learning*). Comme dans le modèle tabulaire, nous commençons avec le schéma de la figure 6.4. La grande nouveauté cette fois est que nous ne représentons plus la fonction Q sous forme d'un tableau mais en utilisant un réseau de neurones.

Au chapitre 1, nous avons brièvement mentionné que l'apprentissage automatique pouvait être caractérisé comme un *problème d'approximation de fonction*, à savoir trouver une fonction en correspondance étroite avec certaines fonctions cibles. Une telle fonction cible pourrait par exemple associer un ensemble de pixels à des entiers de 0 à 9, où les pixels seraient ceux d'une image du chiffre correspondant. On nous fournit la valeur de la fonction pour certaines entrées et le but est de créer une fonction qui s'approche au plus près des résultats obtenus pour toutes ces valeurs, ce qui nous permet de la sorte d'obtenir les valeurs de la fonction là où nous n'avons aucune valeur d'entrée. Dans le cas de l'apprentissage par renforcement profond, l'analogie avec l'approximation de fonction est tout à fait pertinente : nous allons rechercher une approximation de notre fonction Q (inconnue) à l'aide de réseaux de

neurones en explorant au hasard un processus décisionnel de Markov, et en effectuant l'apprentissage en cours de route.

Il faut souligner que le passage du modèle tabulaire à un modèle d'apprentissage profond n'est *pas* motivé par l'exemple du lac gelé, qui est exactement le type de problème pour lequel un apprentissage de type tabulaire est adapté. L'apprentissage par renforcement profond est nécessaire lorsqu'il y a trop d'états différents pour les rassembler dans un tableau.

L'un des événements ayant conduit à la réémergence des réseaux de neurones a été la création d'un modèle de réseau de neurones unique permettant d'appliquer l'apprentissage par renforcement profond à de nombreux jeux Atari. Ce programme a été créé par *DeepMind*, une start-up rachetée en 2014 par Google. DeepMind a été capable de faire apprendre un grand nombre de jeux à un programme unique, en représentant ces jeux sous forme des pixels d'images que ces jeux génèrent. Chaque combinaison de pixels est un état. Je n'ai plus en tête la taille d'image qu'ils ont utilisée, mais même si celles-ci étaient aussi petites que les images $28 * 28$ que nous avons utilisées pour le Mnist, et si chaque pixel ne pouvait qu'être allumé ou éteint, cela ferait quand même 2^{784} combinaisons possibles de valeurs de pixels, et il faudrait donc en principe ce nombre d'états dans le tableau Q . De toute façon, c'est largement trop pour une approche tabulaire. (J'ai finalement vérifié : une fenêtre de jeu Atari fait $210 * 160$ pixels RVB, et le programme DeepMind a réduit ceci à $84 * 84$ noir et blanc.) Nous reviendrons plus tard discuter de cas plus compliqués que le lac gelé.

Remplacer le tableau Q par une fonction s'appuyant sur un réseau de neurones se ramène à ceci : pour obtenir une recommandation de déplacement, plutôt que de se référer au tableau Q , nous appelons en pratique le tableau Q en injectant l'état dans un réseau de neurones à une couche, comme le montre la figure 6.5. Le code TF de création des paramètres du modèle de fonction Q est donné à la figure 6.6. Nous fournissons en entrée l'état courant (le scalaire `inptSt`), que nous transformons en vecteur one-hot `oneH` qui est alors transformé par une unique couche d'unités linéaires Q . Cette couche Q a pour forme $16 * 4$ où 16 est la taille du vecteur one-hot des états et 4 est le nombre d'actions possibles. La sortie `qVals` est constituée par les entrées dans $Q(s)$, et `outAct`, la valeur maximum parmi les entrées du tableau Q , est la recommandation.

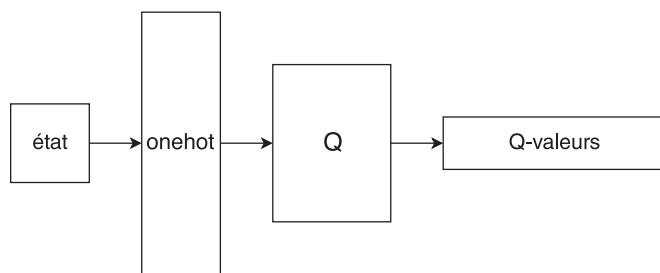


Figure 6.5 – Réseau de neurones d'apprentissage par renforcement profond pour le lac gelé

6.3 Les bases de l'apprentissage par renforcement profond

```

inptSt = tf.placeholder(dtype=tf.int32)
oneH = tf.one_hot(inptSt, 16)
Q = tf.Variable(tf.random_uniform([16, 4], 0, 0.01))
qVals = tf.matmul([oneH], Q)
outAct = tf.argmax(qVals, 1)

```

Figure 6.6 – Paramètres du modèle TF pour la fonction d'apprentissage de Q

Dans la figure 6.6, il est implicite que nous ne jouons qu'une partie à la fois, et par conséquent lorsque nous fournissons un état d'entrée (et obtenons une recommandation), il n'y en a qu'une. Dans notre gestion habituelle des réseaux de neurones, ceci correspond à une taille de lot égale à 1. Par exemple, l'état d'entrée, `inptSt`, est un scalaire : c'est l'indice de l'état dans lequel le joueur se trouve. Il en découle que `oneH` est un vecteur. Puis, étant donné que `matmul` attend deux matrices, nous l'appelons avec `[oneH]`. Ceci signifie par conséquent que `qVals` va être une matrice de forme $[1, 4]$, c'est-à-dire qu'elle ne contiendra les valeurs Q que pour une action (haut, bas, etc.). Enfin, `outAct` est de forme $[1]$, donc la recommandation d'action est `outAct[0]` (vous comprendrez mieux la raison de ces explications très détaillées lorsque nous vous présenterons le reste du code de l'apprentissage par renforcement profond à la figure 6.7).

Comme dans le modèle tabulaire, l'algorithme choisit soit une action au hasard (au début du processus d'apprentissage), soit une action sur la base de la recommandation du tableau Q (lorsqu'on approche de la fin). En apprentissage par renforcement profond, nous obtenons la recommandation du tableau Q en injectant l'état courant dans le réseau de neurones de la figure 6.5 et en choisissant l'action correspondant à la plus forte valeur. Une fois que nous avons l'action, nous appelons `step` pour obtenir le résultat et apprendre à partir de celui-ci. Naturellement, comme il s'agit d'un réseau de neurones, il nous faut une fonction de perte.

Mais à propos, quelle est la fonction de perte pour l'apprentissage par renforcement profond ? Il s'agit d'une question-clé car, de manière évidente, lorsque nous effectuons un déplacement, particulièrement au début de l'apprentissage, nous ne savons pas si c'est bon ou mauvais ! Cependant, nous savons ceci : en moyenne,

$$R(s, a) + \gamma \max_{a'} Q(s', a') \quad (6.7)$$

(où comme précédemment, s' est l'état auquel nous aboutissons après avoir effectué a depuis l'état s) est une meilleure estimation $Q(s, a)$ que la valeur actuelle, car nous regardons un déplacement plus en avant. Par conséquent nous choisissons pour perte

$$(Q(s, a) - (R(s, a) + \gamma \max_{a'} Q(s', a')))^2 \quad (6.8)$$

soit le carré de la différence entre ce qui vient de se passer (lorsque nous avons effectué le déplacement) et les valeurs prédites (par la table/fonction Q). C'est ce qu'on

appelle la *perte quadratique* (en anglais, *squared-error loss* ou *quadratic loss*). La différence entre le Q calculé par le réseau (le premier terme) et la valeur que nous pouvons calculer en observant la récompense effective pour l'action suivante plus la valeur Q une étape dans le futur (le second terme) est appelée *erreur de différence temporelle* (en anglais, *temporal difference error* ou TD(0)). Si nous regardions deux étapes en avant dans le futur, ce serait TD(1).

La figure 6.7 fournit le reste du code TF (la suite de celui de la figure 6.6). Les cinq premières lignes construisent le reste du graphe TF. Survolons maintenant le reste du code en nous attardant sur les lignes 7, 11, 13, 14, 19 et 25. Elles implémentent la « promenade au hasard » dans AI Gym, c'est-à-dire tout ce qui correspond à la figure 6.4. Nous créons le jeu (ligne 7) et jouons 2 000 parties individuelles (ligne 11), chacune d'entre elles commençant par `game.reset()` (ligne 13). Chaque partie comporte 99 déplacements au maximum (ligne 14). Le déplacement réel est effectué à la ligne 19. C'est l'indicateur nommé `dn` (ligne 24) qui signale que le jeu est terminé.

```

1 nextQ = tf.placeholder(shape=[1,4], dtype=tf.float32)
2 loss = tf.reduce_sum(tf.square(nextQ - qVals))
3 trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
4 updateMod = trainer.minimize(loss)
5 init = tf.global_variables_initializer()

6 gamma = .99
7 game=gym.make('FrozenLake-v0')
8 rTot=0
9 with tf.Session() as sess:
10     sess.run(init)
11     for i in range(2000):
12         e = 50.0/(i + 50)
13         s=game.reset()
14         for j in range(99):
15             nActs,nxtQ=sess.run([outAct,qVals],feed_dict={inptSt: s})
16             nAct=nActs[0]
17             if np.random.rand(1)<e: nAct= game.action_space.sample()
18             s1,rwd,dn,_ = game.step(nAct)
19             Q1 = sess.run(qVals,feed_dict={inptSt: s1})
20             nxtQ[0,nAct] = rwd + gamma*(np.max(Q1))
21             sess.run(updateMod,feed_dict={inptSt:s, nextQ:nxtQ})
22             rTot+=rwd
23             if dn: break
24             s = s1
25     print "Taux de parties réussies : ", rTot/2000

```

Figure 6.7 – Reste du code d'apprentissage par renforcement profond

Il nous reste à explorer les lignes 15-17 (choisir la prochaine action) et 20-22. La ligne 15 est la passe avant dans laquelle nous donnons au réseau de neurones l'état actuel et obtenons en retour un vecteur de longueur 1 (que la ligne suivante transforme en scalaire : le numéro de l'action). Nous donnons aussi toujours au programme une petite probabilité d'effectuer une action au hasard (ligne 18). Ceci nous garantit d'arriver au bout du compte à explorer l'ensemble de l'espace de jeu. Les lignes 20-22 concernent le calcul de la perte et l'exécution de la passe arrière pour mettre à jour les paramètres du modèle. Ceci concerne aussi les lignes 1-5, qui ont créé le graphe TF pour le calcul et la mise à jour de la perte.

Les performances de ce programme ne sont pas aussi bonnes que celles de la méthode tabulaire mais, comme nous l'avons dit, cette dernière était plutôt adaptée au processus de décision markovien du lac gelé.

6.4 MÉTHODES DE GRADIENTS DE POLITIQUE

Examinons maintenant un autre problème d'Open AI Gym ne pouvant pas être géré par des méthodes tabulaires standard, *cart-pole*, et une nouvelle méthode d'apprentissage par renforcement profond, les *gradients de politique*. Le « cart-pole » représenté sur la figure 6.8 est un chariot se déplaçant sur une piste à une dimension. Une perche y est attachée par un joint élastique de sorte que quand le chariot est lancé dans une direction ou l'autre, le haut de la perche se déplace vers la gauche ou vers la droite conformément aux lois de Newton. Un état est constitué de quatre valeurs : la position du chariot et l'angle de la perche, après le déplacement précédent et après le déplacement courant. Nous fournissons des valeurs à des intervalles de temps successifs pour permettre au programme de comprendre la direction du déplacement. Le joueur peut effectuer deux actions : pousser le chariot vers la droite ou vers la gauche. L'intensité de l'impulsion est toujours la même. Si le chariot se déplace trop loin à

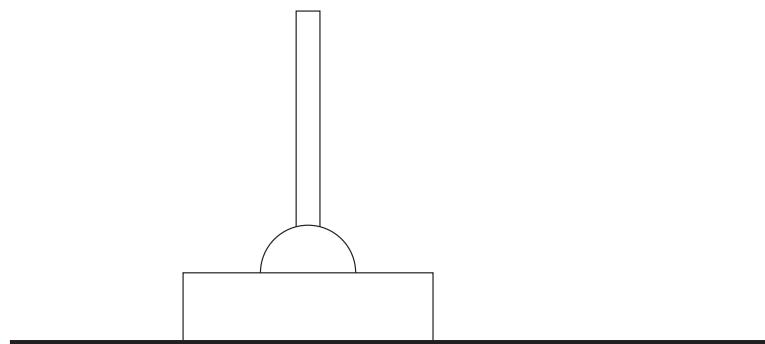


Figure 6.8 – Un « cart-pole »

droite ou à gauche, ou si la perche s'écarte trop de la verticale, step signale que la partie en cours est terminée, et nous devons effectuer une réinitialisation pour en commencer une nouvelle. Nous obtenons une unité de récompense pour chaque déplacement effectué avant d'échouer. Naturellement, le but est de conserver le bon positionnement du chariot et de sa perche le plus longtemps possible. Étant donné que l'état correspond à un quadruplet de valeurs réelles, le nombre d'états possibles est infini, c'est pourquoi les méthodes tabulaires ne sont pas envisageables.

Jusqu'ici nous avons utilisé nos modèles de réseaux de neurones pour obtenir une approximation de la fonction Q pour notre MDP. Dans cette section, nous présentons une méthode dans laquelle le réseau de neurones modélise directement la fonction de politique. Là encore, nous nous intéressons à un apprentissage sans modèle, et nous adoptons à nouveau le paradigme du vagabondage dans l'environnement de jeu, initialement en choisissant les actions à peu près au hasard puis en adoptant progressivement la recommandation du réseau de neurones. Comme un peu partout dans ce chapitre, le problème crucial est de trouver une fonction de perte appropriée, étant donné que nous ne connaissons pas les actions correctes à utiliser.

En Deep-Q Learning nous effectuons un déplacement à la fois, et nous comptons sur le fait que, après avoir effectué un déplacement, reçu une récompense et terminé dans un nouvel état, notre connaissance de l'environnement local actuel s'est améliorée. Notre perte était la différence entre ce qui était prévu (c'est-à-dire la fonction Q) compte tenu des connaissances antérieures et ce qui s'est réellement passé.

Ici, nous essayons quelque chose d'autre. Supposons que nous jouions une partie complète sans apporter aucune modification à notre réseau — par exemple, nous effectuons 20 déplacements (ordres donnés au chariot) avant que la perche tombe. Cette fois nous gérons l'exploration/exploitation en choisissant des actions conformément à une distribution de probabilité découlant de la fonction Q , plutôt qu'en prenant le maximum de la fonction Q .

Selon ce scénario, nous pouvons calculer la récompense avec rabais pour le premier état ($D_0(\mathbf{s}, \mathbf{a})$) lorsqu'il est suivi par tous les états et toutes les actions que nous venons d'essayer :

$$D_0(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} \gamma^t R(s_t, a_t, s_{t+1}) \quad (6.9)$$

Si nous avions effectué n étapes, nous pourrions calculer la future récompense avec rabais pour chacune des combinaisons état-action s_i, a_i grâce à la relation de récurrence :

$$D_n(\mathbf{s}, \mathbf{a}) = 0 \quad (6.10)$$

$$D_i(\mathbf{s}, \mathbf{a}) = R(s_i, a_i, s_{i+1}) + \gamma D_{i+1}(\mathbf{s}, \mathbf{a}) \quad (6.11)$$

où D_n est la future récompense avec rabais pour le n -ième état dans la séquence d'états que nous parcourons (en effectuant l'action a). Remarquez encore que nous

avons glané des informations ici. Par exemple, avant d'avoir essayé la première séquence de déplacements au hasard, nous n'avions aucune idée de ce que pouvait être la récompense. Après, nous savons par exemple que 10 est possible (et même raisonnable pour une séquence d'actions au hasard). Et nous savons aussi maintenant, si nous avons échoué lors du déplacement 10, que $Q(s_9, a_9) = 0$.

Voici une bonne fonction de perte qui tient compte de ces faits et de beaucoup d'autres :

$$L(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} D_t(\mathbf{s}, \mathbf{a})(-\log \Pr(a_t \mid s)) \quad (6.12)$$

Pour décrypter cela, remarquez que le terme le plus à droite est la perte d'entropie croisée, et que par elle-même elle a l'effet d'encourager à répondre par l'action a_t lorsqu'on est dans l'état s_t . Bien sûr, c'est plutôt inutile en soi étant donné que, surtout au début de l'apprentissage, les actions ont été choisies au hasard.

Considérons ensuite la façon dont les valeurs D_t ont une incidence sur cela. Supposons en particulier que a_0 ait été une mauvaise réaction dans l'état s_0 . Supposons par exemple que le chariot soit centré, que la perche penche vers la droite au départ, et que nous choisissons d'aller à gauche, ce qui ferait pencher la perche encore davantage vers la droite. On voit bien que, toutes choses égales par ailleurs, la valeur de D_0 est plus petite dans ce cas que si nous avions choisi de nous déplacer vers la droite — la raison en étant que (toutes choses égales par ailleurs) si le premier déplacement est bon, la perche et le chariot demeureront dans les limites plus longtemps (n est plus grand) et les valeurs D seront plus grandes. Par conséquent l'équation 6.12 associe une perte plus élevée à un mauvais a_0 qu'à un bon, entraînant par conséquent le réseau de neurones à préférer le bon.

Cette combinaison d'une architecture et d'une fonction de perte est connue sous le nom de *REINFORCE*. La figure 6.9 en présente l'architecture de base. Il est important de remarquer ici que le réseau de neurones est utilisé de deux manières différentes. Tout d'abord, si l'on regarde la partie gauche, nous lui transmettons un état unique qui, comme indiqué précédemment, est un quadruplet de nombres réels indiquant la position et la vitesse du chariot et de l'extrémité de la perche. Dans ce mode, nous obtenons des probabilités de prendre les deux actions possibles, comme indiqué au milieu droit de la figure. Pendant que nous sommes dans ce mode, nous ne fournissons pas de valeurs au nœud de substitution pour les récompenses ou actions parce que nous ne les connaissons pas et que nous n'en avons pas besoin puisque nous ne calculons pas la perte à cet endroit. Après avoir joué l'ensemble d'une partie, nous utilisons le réseau de neurones dans l'autre mode. Cette fois, nous lui donnons la séquence d'actions et les récompenses, en lui demandant alors de calculer la perte et d'effectuer la rétropropagation. En mode d'entraînement, nous sommes en un certain sens en train de calculer les actions de deux manières différentes. Tout d'abord, nous donnons au réseau de neurones les états que nous traversons et, pour chacun

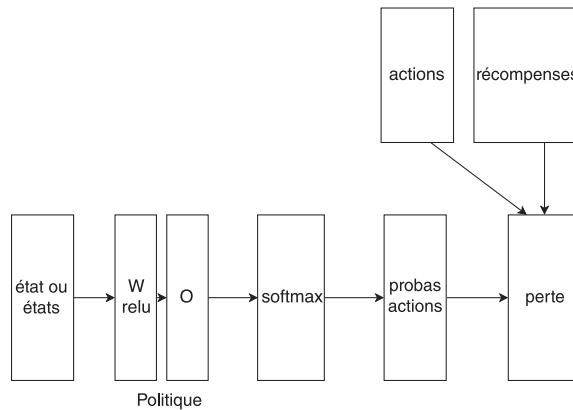


Figure 6.9 – Architecture d'apprentissage profond pour REINFORCE

d'eux, les couches de détermination de la politique calculent les probabilités des actions. Ensuite, nous injectons directement les actions effectuées à l'aide d'un nœud de substitution. Ceci parce que lorsqu'il faut décider d'une action en mode jeu, nous ne choisissons pas forcément l'action ayant la plus haute probabilité, mais sélectionnons plutôt au hasard en nous basant sur les probabilités des actions. Pour calculer la perte conformément à l'équation 6.12, nous avons besoin des deux.

La figure 6.10 fournit le code TF permettant de créer un réseau de neurones à gradient de politique utilisant la fonction de perte de l'équation 6.12, et la figure 6.11 fournit le pseudo-code utilisant un réseau de neurones pour apprendre une politique et l'appliquer dans un environnement de jeu. Examinons d'abord le pseudo-code. Remarquez que la boucle externe (ligne 2) nous fait jouer 3 001 parties. La ligne

```

state = tf.placeholder(shape=[None, 4], dtype=tf.float32)
W = tf.Variable(tf.random_uniform([4, 8], dtype=tf.float32))
hidden = tf.nn.relu(tf.matmul(state, W))
O = tf.Variable(tf.random_uniform([8, 2], dtype=tf.float32))
output = tf.nn.softmax(tf.matmul(hidden, O))

rewards = tf.placeholder(shape=[None], dtype=tf.float32)
actions = tf.placeholder(shape=[None], dtype=tf.int32)
indices = tf.range(0, tf.shape(output)[0]) * 2 + actions
actProbs = tf.gather(tf.reshape(output, [-1]), indices)
aloss = -tf.reduce_mean(tf.log(actProbs)*rewards)
trainOp = tf.train.AdamOptimizer(.01).minimize(aloss)
  
```

Figure 6.10 – Graphe TF pour le réseau de neurones à gradient de politique du jeu cart-pole

1. totRs=[]
2. Pour i dans range(3001) :
 - (a) st=réinitialiser le jeu
 - (b) pour j dans range(999) :
 - i. actDist = sess.run(output, feed_dict=state:[st])
 - ii. sélectionner aléatoirement act à l'aide de actDist
 - iii. st1,r,dn,_=game.step(act)
 - iv. rassembler st,a,r dans hist
 - v. st=st1
 - vi. si dn :
 - A. disRs = $[D_i(\text{states, actions de hist}) \mid i = 0 \text{ à } j - 1]$
 - B. créer feed_dict avec state=st, actions de hist et rewards=disRs
 - C. sess.run(trainOp,feed_dict=feed_dict)
 - D. ajouter j à la fin de totRs
 - E. interrompre
 - vii. si $i \% 100 = 0$: imprimer la moyenne des 100 dernières entrées dans totRs

Figure 6.11 – Pseudo-code pour l’entraînement par gradient de politique d’un réseau de neurones pour le jeu cart-pole

interne (ligne 2b) fait durer la partie jusqu'à ce que `step` nous informe que c'est terminé (ligne D) ou jusqu'à ce que nous ayons effectué 999 déplacements. Nous choisissons aléatoirement une action en nous basant sur les probabilités calculées par notre réseau de neurones (lignes i, ii) puis exécutons celle-ci. Nous sauvegardons les résultats dans la liste `hist` afin de garder une trace de ce qui s'est produit. Si l'action conduit à un état final, nous mettons alors à jour les paramètres du modèle.

Nous voyons sur la figure 6.10 que la sortie `output` est calculée en prenant les valeurs de l'état courant `state` et en leur faisant parcourir les deux couches d'unités linéaires `w` et `o` du réseau de neurones séparées naturellement par `tf.relu`, puis en les injectant dans un `softmax` pour transformer les logits en probabilités. Grâce aux exemples précédents de réseaux de neurones multicouches, il devrait être clair pour vous que la première couche a pour dimensions `[inputSz, hiddenSz]` et la seconde `[hiddenSz, outputSz]` où `inputSz` et `outputSz` représentent la taille d'entrée et de sortie et où `hiddenSz`, la taille cachée, est un hyperparamètre (que nous avons choisi égal à 8).

Étant donné que nous avons conçu une nouvelle fonction de perte et n'avons pas utilisé celle fournie en standard par TensorFlow, le calcul de la perte doit être construit à partir de fonctions TF plus élémentaires (seconde moitié de la figure 6.10).

Par contre, dans tous nos réseaux de neurones précédents, la passe avant et la passe arrière étaient inextricablement liées, étant donné qu'elles ne faisaient intervenir aucun calcul extérieur à TF. Ici, nous récupérons les valeurs de la récompense `reward` depuis l'extérieur — `reward` est un nœud de substitution (placeholder), alimenté en données conformément aux lignes A, B et C de la figure 6.11. De même, `actions` est un placeholder.

Les trois dernières lignes de la figure 6.10 semblent plus familières. `aloss` reçoit le résultat du calcul correspondant à l'équation 6.12. Pour l'optimisation, nous avons choisi la *méthode Adam* (nom dérivé de l'anglais *adaptive moment estimation*). Nous aurions pu utiliser à la place l'algorithme de base de descente de gradient en doublant le taux d'apprentissage, et nous aurions obtenu des résultats presque aussi bons, mais pas tout à fait. L'optimiseur Adam est légèrement plus compliqué et considéré généralement comme meilleur. Il diffère de la descente de gradient de plusieurs manières, la plus fondamentale étant l'utilisation de la *vitesse* (en anglais, *momentum*). Comme le nom le suggère, un optimiseur utilisant la vitesse tend à continuer à déplacer une valeur de paramètre vers le haut ou vers le bas si elle a été déplacée vers le haut ou vers le bas récemment — davantage que ne le ferait une descente de gradient ordinaire.

Il nous reste les deux lignes du milieu de la figure 6.10, celles qui calculent `indices` et `actProbs`. Dans un premier temps, ignorons la façon dont elles fonctionnent, et concentrons-nous sur ce qu'elles doivent faire. Ce dont nous avons besoin, c'est la transformation présentée à la figure 6.12. Sur la gauche, nous voyons la sortie d'une passe avant calculant la probabilité que chacune des actions possibles r et 1 soit la meilleure à prendre. Si nous en étions au chapitre 1 et que nous avions une totale supervision, nous multiplierions ceci par un tenseur de la taille du lot constitué de vecteurs one-hot pour obtenir les probabilités des actions que nous devrions effectuer en accord avec la supervision. C'est en fait ce que nous montrons à droite de la figure 6.12.

Pour effectuer cette transformation, nous avons besoin de `gather`, une fonction à deux arguments :

```
tf.gather(tensor, indices)
```

qui extrait les éléments du tenseur spécifiés par les indices et les regroupe dans un nouveau tenseur. Par exemple, si `tensor` vaut $((1,3), (4,6), (2,1), (3,3))$ et `indices`

$$\begin{array}{ccc} \Pr(1 \mid s_1) & \Pr(r \mid s_1) & \Pr(a_1 \mid s_1) \\ \Pr(1 \mid s_2) & \Pr(r \mid s_2) & \rightarrow \Pr(a_2 \mid s_2) \\ \\ \Pr(1 \mid s_n) & \Pr(r \mid s_n) & \Pr(a_n \mid s_n) \end{array}$$

Figure 6.12 – Extraction des probabilités des actions à partir du tenseur de toutes les probabilités

vaut (3,1,3), alors la sortie vaut ((3,3), (4,6), (3,3)). Dans notre cas, nous transformons la matrice de probabilités d’actions à gauche de la figure 6.12 en un vecteur de probabilités, et, grâce à la ligne précédente où nous avons associé `indices` à la liste correcte, `tf.gather` collecte uniquement les probabilités des actions spécifiées par le vecteur `actions`. Le lecteur pourra montrer qu’`indices` est affecté correctement (c’est le sujet de l’exercice 6.5).

Il est utile de revenir en arrière et de voir plus soigneusement quelle relation existe entre les modèles Q-learning et REINFORCE. D’abord, ils diffèrent dans la façon de collecter les informations sur l’environnement pour informer le réseau de neurones. Q-learning effectue un déplacement, puis regarde si la prédiction du résultat par le réseau de neurones est proche de ce qui s’est produit réellement. En reprenant l’équation 6.8, la fonction de perte du Q-learning, nous voyons que si la prédiction est identique au résultat, alors il n’y a rien à mettre à jour. Avec REINFORCE, au contraire, nous effectuons une partie complète avant de changer un quelconque paramètre du réseau de neurones, depuis l’état initial jusqu’à ce que le jeu signale que c’est terminé. Remarquez que nous aurions pu faire quelque chose comme le Q-learning mais en utilisant le calendrier de modification des paramètres de REINFORCE. Ceci ralentit l’apprentissage dans la mesure où nous modifions beaucoup moins souvent les paramètres, mais en compensation nous effectuons de meilleures modifications car nous calculons la récompense avec rabais réelle.

6.5 MÉTHODES ACTEUR-CRITIQUE

Après avoir examiné les différences entre Q-learning et REINFORCE, nous nous concentrerons maintenant sur les similitudes. Dans les deux cas, le réseau de neurones calcule soit une politique, soit, pour le Q-learning, une fonction dont on peut trivialement déduire une politique (pour tout état s , toujours choisir l’action a qui maximise $Q(s, a)$). Par conséquent, dans les deux cas notre réseau de neurones cherche à définir par approximations successives une fonction qui nous dit comment agir. Nous appelons ces programmes d’apprentissage avec renforcement des *méthodes acteur*¹. Dans cette section, nous allons voir des programmes ayant deux sous-composants, deux réseaux de neurones ayant chacun sa propre fonction de perte : l’un est comme précédemment un programme acteur, et le second un programme *critique*. Comme vous vous en doutez peut-être, les méthodes d’apprentissage par renforcement de ce type sont appelées *méthodes acteur-critique*. Nous parlerons en particulier dans cette section de la méthode *avantage acteur-critique*, ou *a2c*. C’est un bon choix pour nous parce qu’elle fonctionne plutôt bien et que nous pouvons nous approcher par approximations successives de la solution en partant de REINFORCE. La première version (incrémentale) s’appelle *a2c-*. Nous l’appliquons à nouveau au jeu cart-pole.

1. On parle aussi de méthodes acteur-seul, pour les distinguer des méthodes acteur-critique.

Cette méthode est appelée *avantage* acteur-critique parce qu'elle utilise la notion d'« avantage ». L'avantage d'un couple état-action est la différence entre la valeur Q de ce couple état-action et la valeur de l'état :

$$A(s, a) = Q(s, a) - V(s) \quad (6.13)$$

Intuitivement, nous nous attendons à ce que l'avantage soit un nombre négatif parce que, par exemple dans l'itération sur les valeurs, $V(s)$ est calculée en appliquant un $\arg \max_a$ à l'ensemble des actions possibles. Cependant, pour les bonnes actions, A est plutôt grand par rapport aux nombres négatifs, par conséquent A mesure l'efficacité d'une action dans un état particulier en comparant à l'ensemble des états.

Ensuite nous définissons comme suit la perte subie par a2c pour avoir exploré une séquence d'actions depuis un état initial jusqu'à la fin de la partie :

$$L_A(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} A(s_t, a_t)(-\log \Pr(a_t | s_t)) \quad (6.14)$$

C'est très proche de la perte de REINFORCE de l'équation 6.12 mais nous avons remplacé $D_t(s, a)$, la récompense avec rabais, par $A_t(s, a)$. Nous avons appelé cette perte L_A pour la différencier de la perte totale pour a2c qui, comme nous allons le voir, englobe une seconde perte L_C en rapport avec le critique.

Nous nous souvenons que la perte de REINFORCE a pour but d'encourager les actions conduisant à une récompense plus importante. Maintenant, nous encourageons les actions qui sont meilleures que les autres actions pouvant être choisies à partir du même état. Bien que cela semble plutôt raisonnable, pourquoi serait-ce meilleur que d'encourager directement les actions à forte récompense ?

La réponse a à voir avec la variance de $A(s, a)$. Comme mentionné à la section 2.4.3, la variance d'une fonction est l'espérance du carré de la différence entre la valeur de la fonction et sa valeur moyenne. Intuitivement, ceci signifie que les fonctions variant beaucoup ont une variance élevée, et A devrait avoir une variance beaucoup plus faible que Q . Regardez le jeu cart-pole. En supposant que le jeu nous donne une réponse raisonnable en termes de déplacement vers la gauche ou la droite en fonction de la vitesse de déplacement de la perche, la différence entre un déplacement vers la droite et un déplacement vers la gauche sera petite et par conséquent A est petit pratiquement partout dans l'espace des états. Comparez ceci avec Q . Après avoir effectué 100 parties d'apprentissage, une session de cart-pole effectue environ 20 déplacements avant d'échouer, alors qu'une politique même modérément bonne arrive à 200 ou plus.

Ajoutez maintenant un second fait : toutes choses égales par ailleurs, il est plus facile d'obtenir une approximation d'une fonction lorsque sa variance est faible que lorsqu'elle est forte. Une fonction constante avec une variance nulle est ce qu'il y a de plus simple. Donc si A est bien plus facile à estimer, cela pourrait compenser le désavantage représenté par la maximisation de A plutôt que de Q directement.

Cela semble être le cas. Bien sûr, nous ne savons pas encore comment calculer A . Venons-en à ce point.

Comme vous vous en souvenez, avec la méthode REINFORCE nous suivons jusqu'à la fin du jeu un cheminement basé sur notre politique actuelle et utilisons la récompense avec rabais $D_t(s,a)$ de l'équation 6.11 pour estimer $Q(s,a)$. Nous en faisons maintenant un deuxième usage pour notre estimation de Q lors du calcul de A (équation 6.13). Comme pour $V(s)$, nous construisons un sous-réseau à l'intérieur de notre réseau de neurones juste pour le calculer.

La figure 6.13 présente le code TF de construction de réseau s'ajoutant à celui requis par REINFORCE (figure 6.10). Nous avons créé un réseau de neurones à deux couches entièrement connectées pour calculer V , la fonction de valeur — le critique. Il est entraîné à produire de bonnes estimations de V en utilisant une perte quadratique mesurant la disparité entre la récompense effective obtenue et la sortie de l'approximation du réseau de neurones ($cLoss$). La perte de l'acteur correspond ici à l'équation 6.14 et utilise donc la fonction d'avantage. Ces modifications relativement modestes transforment notre REINFORCE en a2c-.

Dépassant ainsi a2c-, le véritable a2c incorpore deux autres améliorations. L'un des problèmes de REINFORCE (dont hérite a2c-) est qu'il lui faut jouer une partie entière avant de pouvoir réaliser le moindre apprentissage. Au début d'un apprentissage de cart-pole, lorsqu'une partie ne dure que 10 à 20 déplacements, ce n'est pas une véritable limitation. Mais les parties avec REINFORCE finissent par compter 100 ou 200 coups, et les parties a2c- peuvent durer encore plus longtemps. Cependant, a2c peut améliorer cela en mettant à jour les paramètres du modèle beaucoup plus tôt, et beaucoup plus souvent.

L'astuce consiste à faire une pause dans le déroulement de la partie toutes les 50 actions par exemple (c'est un hyperparamètre) pour mettre à jour les paramètres du modèle. Nous ne pouvions pas faire cela avec REINFORCE. À vrai dire, l'intérêt de surveiller l'ensemble des actions d'une partie était d'obtenir une bonne estimation des valeurs de Q pour les actions que nous avions effectuées. Mais a2c nous permet de faire une estimation en additionnant simplement les récompenses effectives accumulées sur les 50 derniers déplacements et la valeur V de l'état dans lequel nous arrivons. Nous remettons alors à zéro l'historique $hist$ et redémarrons complètement avec le 51^e déplacement, afin de répéter la même chose 50 déplacements plus tard

```
V1 = tf.Variable(tf.random_normal([4, 8], dtype=tf.float32, stddev=.1))
v1Out = tf.nn.relu(tf.matmul(state, V1))
V2 = tf.Variable(tf.random_normal([8, 1], dtype=tf.float32, stddev=.1))
v2Out = tf.matmul(v1Out, V2)
advantage = rewards-v2Out
aLoss = -tr.reduce_mean(tf.log(actProbs) * advantage)
cLoss = tf.reduce_mean(tf.square(rewards-vOut))
loss = aLoss + cLoss
```

Figure 6.13 – Code TF ajouté aux figures 6.10 et 6.11 pour a2c

(en poussant les choses à l'extrême, ceci pourrait aussi libérer a2c de la contrainte imposée par REINFORCE de ne l'utiliser que dans des jeux avec réinitialisation explicite).

La deuxième amélioration dans a2c complet est l'utilisation d'environnements multiples. Nous avons déjà remarqué que l'exécution simultanée de tout un lot d'exemples d'entraînement est avantageux dans la mesure où cela permet de mieux exploiter les capacités d'optimisation des multiplications de matrices. Jouer une seule partie à la fois ne le permet pas lorsqu'on calcule le prochain coup (action) à jouer. De ce point de vue, jouer plusieurs parties à la fois est équivalent à regrouper des exemples par lots.

6.6 RÉPÉTITION D'EXPÉRIENCES

Nous avons remarqué précédemment qu'un catalyseur majeur de la renaissance des réseaux de neurones avait été la réussite de DeepMind avec un programme pouvant jouer de multiples jeux Atari à un niveau expert. La technologie utilisée là est connue sous le nom de DQN (*Deep Q Network*). Cette conception particulière d'apprentissage par renforcement a été largement remplacée par les méthodes acteur-critique, mais le programme a aussi introduit plusieurs améliorations indépendantes de l'utilisation des méthodes acteur ou acteur-critique. L'une d'entre elles est la *répétition d'expériences* (en anglais, *experience replay*).

Comme on pouvait s'y attendre, l'apprentissage par renforcement est une composante importante dans les efforts actuels de développement d'un véhicule autonome. Lorsqu'on applique l'apprentissage par renforcement à ce domaine, l'acquisition des données d'entraînement constitue un problème important. Les systèmes actuels d'apprentissage par renforcement en nécessitent énormément et par rapport aux ordinateurs le monde réel évolue très lentement, en particulier dans les rues et sur les autoroutes. À vrai dire, si vous commencez à minuter les jeux Open AI Gym, même les simulations par ordinateur peuvent être lentes : une grande partie du temps passé en apprentissage par renforcement est consacrée à l'exécution du jeu. Si nous pouvions accélérer le monde réel, nous pourrions apprendre encore plus vite, mais ce n'est pas possible.

Dans la répétition d'expérience, nous utilisons de nombreuses fois les mêmes données d'entraînement. Le plus simple est de l'expliquer dans le contexte d'Open AI Gym. Pour revenir à REINFORCE, lorsque nous avons joué nous avons utilisé une variable `hist` pour enregistrer l'historique de la partie : chaque état occupé, l'action effectuée, l'état qui en a résulté et la récompense reçue. Nous en avions besoin à la fin de la partie pour calculer les D_t , mais une fois ceux-ci calculés, nous jetions l'historique. Avec la répétition d'expérience, pour chaque instant t nous sauvegardons $\langle s_t, a_t, s_{t+1}, D_t \rangle$. Avec ces valeurs, nous pouvons effectuer une autre passe avant et une autre passe arrière sur nos données, afin d'en tirer davantage. Mais il existe aussi un deuxième avantage : nous pouvons jouer, puis rejouer, chaque étape temporelle dans un ordre aléatoire. Vous vous souvenez peut-être de la condition iid (données

indépendantes et identiquement distribuées) mentionnée à la section 1.6 lorsque nous avons constaté combien l'apprentissage par renforcement pouvait être problématique lorsque les exemples d'entraînement étaient corrélés depuis le départ. Choisir des actions au hasard dans plusieurs parties réduit significativement ce problème.

Bien sûr, cela a un prix. Un vieil exemple d'entraînement n'est pas aussi informatif qu'un nouveau. De plus, les données peuvent en quelque sorte se gâter. Supposons que nous ayons des données datant du début de notre entraînement, avant de savoir par exemple qu'il ne faut pas se déplacer vers la gauche lorsque la perche penche vers la droite. Supposons qu'ensuite nous ayons amélioré notre apprentissage. Ceci signifie que nous réapprenons sans nécessité à partir d'anciennes données ce qu'il faut faire dans l'état s_{old} , alors qu'en fait notre politique actuelle ne nous permet jamais d'atteindre cet état. Au lieu de cela, nous procédons comme suit : nous conservons une mémoire tampon de 50 parties correspondant par exemple à 5 000 quadruplets état-action-état-récompense (nous effectuons en moyenne 100 déplacements avant d'échouer). Nous choisissons alors au hasard 400 états par exemple pour s'entraîner. Nous remplaçons alors la plus ancienne partie dans la mémoire tampon par la nouvelle partie jouée en utilisant la nouvelle politique basée sur des paramètres mis à jour.

6.7 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

Bien avant l'avènement de l'apprentissage profond, l'apprentissage par renforcement faisait déjà l'objet d'un grand nombre d'études théoriques et pratiques, et ce dernier arrivé ne l'a pas supplanté. Après tout, le problème majeur en apprentissage par renforcement est de déterminer comment apprendre quand vous n'avez que des informations indirectes pour savoir quelles actions sont bonnes ou mauvaises, et l'apprentissage profond ne fait que déplacer ce problème vers la question de la définition de la fonction de perte. Il ne dit pas grand-chose, voire même rien du tout, sur la nature de la solution. Le texte de référence sur l'apprentissage par renforcement et celui de Richard Sutton et Andrew Barto [SB98]. Je me suis moi-même largement inspiré d'un article didactique précurseur de Kaelbling *et al.* [KLM96] pour le contenu pré-apprentissage profond présenté ici.

Après l'émergence de l'apprentissage profond, alors que je commençais à m'intéresser à l'apprentissage par renforcement, j'ai découvert le blog d'Arthur Juliani sur ce sujet. Si vous consultez ce blog, en particulier les parties 0 [Jul16a] et 2 [Jul16b], vous observerez que mes présentations du jeu cart-pole et de REINFORCE ont été significativement influencées par les siennes, et que son code a servi de point de départ au mien. Et n'oublions pas que la première publication sur REINFORCE est celle de Ronald Williams [Wil92].

L'algorithme a2c d'apprentissage par renforcement a été proposé en tant que variante de l'algorithme *Asynchronous Advantage Actor-Critic* (*a3c*). Nous avons remarqué à la page 123 que a2c permet à de multiples environnements de mieux utiliser l'accélération logicielle et matérielle de la multiplication de matrices. Dans a3c, ces environnements sont évalués de manière asynchrone, sans doute pour mieux

mélanger les combinaisons état-action que l'apprenant peut observer [MBM⁺16]. Ce même article a également proposé l'algorithme a2c en tant que sous-composant de a3c. Finalement, il s'est avéré qu'il fonctionnait tout aussi bien et qu'il était beaucoup plus simple.

Exercices

6.1 Montrez que le tableau V présenté à droite de la figure 6.3 donne les valeurs correctes (jusqu'à deux chiffres significatifs) pour les valeurs d'état après la seconde passe d'itération sur les valeurs.

6.2 L'équation 6.5 possède un paramètre α , mais notre implémentation TF des figures 6.6 et 6.7 ne semble pas faire mention de α . Expliquez où il se cache et quelle valeur nous lui avons donnée.

6.3 Supposons que dans la phase d'entraînement au jeu cart-pole de l'algorithme REINFORCE, celui-ci n'ait effectué que trois actions (l,l,r) avant d'échouer, et que $\Pr(l \mid s_1) = 0.2$, $\Pr(l \mid s_2) = 0.3$, $\Pr(r \mid s_3) = 0.9$. Calculez les valeurs de output, actions, indices et actProbs.

6.4 Dans REINFORCE, nous sélectionnons d'abord les actions qui nous mènent du début de la partie de cart-pole jusqu'à ce que (en général) la perche tombe ou le chariot sorte des limites. Nous le faisons sans mettre à jour les paramètres. Nous sauvegardons ces actions et, en réalité, recommençons l'ensemble du scénario, cette fois en calculant la perte et en mettant à jour les paramètres. Remarquez que si nous avions sauvegardé les actions *et leurs probabilités softmax* alors nous pourrions calculer la perte sans refaire tous les calculs servant à alimenter la fonction de perte une seconde fois. Expliquez pourquoi néanmoins cela ne fonctionne pas, c'est-à-dire pourquoi REINFORCE n'apprendrait rien si nous le faisions sans réitérer les calculs.

6.5 La fonction TF `tf.range`, lorsqu'on lui donne deux arguments

```
tf.range(start, limit)
```

crée un vecteur d'entiers commençant à `start` et allant jusqu'à `limit` (mais sans l'inclure). À moins que la variable nommée `delta` soit définie, les entiers diffèrent d'une unité. Par conséquent, son utilisation dans la figure 6.10 produit une liste d'entiers dans la plage de valeurs allant de 0 à `batchSz`, la taille du lot. Expliquez comment ceci se combine avec la ligne TF suivante pour effectuer la transformation de la figure 6.12.

MODÈLES DE RÉSEAUX DE NEURONES NON SUPERVISÉS

Ce livre a suivi un cheminement en grande partie original depuis les problèmes d'*apprentissage supervisé* tels que celui du Mnist jusqu'aux problèmes d'apprentissage faiblement supervisé tels que l'apprentissage seq2seq et l'apprentissage par renforcement. Notre problème de reconnaissance de chiffres est dit totalement supervisé parce que chaque exemple d'entraînement est fourni avec la réponse correspondante (ou étiquette). Dans nos exemples d'apprentissage par renforcement, les données d'entraînement n'étaient pas étiquetées. Au lieu de cela, nous obtenions une forme faible d'étiquetage dans la mesure où les récompenses distribuées par Open AI Gym guidaient le processus d'apprentissage. Dans ce chapitre, nous nous intéressons à l'*apprentissage non supervisé*, dans lequel nous n'avons ni étiquettes ni aucune autre forme de supervision. Nous voulons apprendre la structure de nos données uniquement à partir de celles-ci. Nous allons examiner en particulier les *auto-encodeurs* (abrégés en AE) et les *réseaux antagonistes génératifs* (en anglais, *generative adversarial networks*, ou GAN).

7.1 ENCODAGE DE BASE

Un auto-encodeur est une fonction dont la sortie est, s'il fonctionne correctement, pratiquement identique à l'entrée. Dans notre cas, cette fonction est un réseau neuronal. Pour compliquer la chose, nous plaçons des obstacles sur le chemin, la méthode la plus courante étant la *réduction de dimension*. La figure 7.1 présente un auto-encodeur simple à deux couches. L'entrée (supposons une image Mnist de 28×28 pixels) est injectée dans une couche d'unités linéaires et transformée en un vecteur

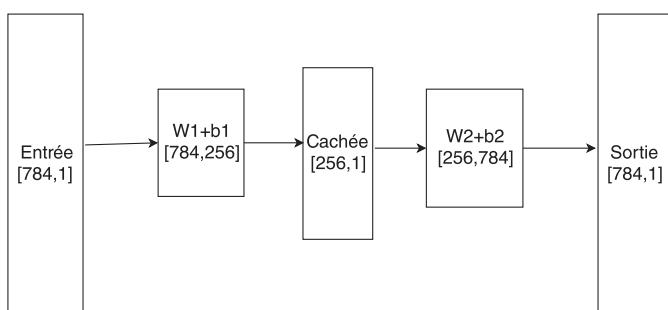


Figure 7.1 – Un AE simple à deux couches

intermédiaire nettement plus petit que l'entrée d'origine, avec par exemple 256 éléments contre 784 à l'origine. Ce vecteur est alors transmis à une deuxième couche, dont le but est d'obtenir une sortie identique à l'entrée de la première couche. Le raisonnement est que, dans la mesure où nous pouvons réduire les dimensions de la couche intermédiaire par rapport à l'entrée, le réseau de neurones a encodé des informations dans cette couche intermédiaire à propos de la structure des images Mnist. De manière plus abstraite, nous avons :

entrée → encodeur → couche cachée → décodeur → entrée

où l'*encodeur* ressemble à un réseau de neurones orienté tâche et le *décodeur* ressemble à l'encodeur en sens inverse. Le processus d'encodage est aussi appelé *sous-échantillonnage* (en anglais, *downsampling*) dans la mesure où il réduit la taille de l'image, et le processus de décodage *reconstruction* (en anglais, *upsampling*).

Nous avons plusieurs raisons de nous intéresser aux auto-encodeurs. L'une d'elles est simplement théorique, et peut-être psychologique. Excepté à l'école, notre apprentissage ne se fait pas par supervision, et lors de notre entrée à l'école nous avons déjà acquis nos compétences les plus importantes : vision, expression orale, tâches motrices et ce qui constitue la base, le sens de l'organisation. Vraisemblablement, ceci n'est possible que par apprentissage *non supervisé*.

Une raison plus pratique en est l'utilisation du *pré-apprentissage* ou *co-apprentissage*. Les données d'entraînement avec étiquettes ne sont pas nombreuses en général, et nos modèles fonctionnent pratiquement toujours d'autant mieux qu'ils ont davantage de paramètres à manipuler. Le pré-entraînement est une technique consistant à entraîner d'abord certains des paramètres sur une tâche associée, puis à commencer les cycles de l'entraînement principal non pas avec notre classique initialisation aléatoire, mais plutôt à partir des valeurs atteintes lors du pré-entraînement. Le co-entraînement fonctionne de manière assez analogue, à ceci près que nous effectuons simultanément l'entraînement et le pré-entraînement. Ce qui signifie que le modèle de réseau de neurones a deux fonctions de perte qu'on additionne pour obtenir la perte totale. L'une est la perte « réelle » qui minimise le nombre d'erreurs dans le problème que nous voulons effectivement résoudre. L'autre concerne un problème associé qui pourrait être simplement de reproduire tout ou partie des données que nous utilisons, c'est-à-dire un problème d'auto-encodage.

Une troisième raison d'étudier l'auto-encodage, c'est une variante appelée *auto-encodeur variationnel*. Ces auto-encodeurs sont identiques au modèle standard, à ceci près qu'ils sont conçus pour envoyer des images aléatoires dans le style de celles sur lesquelles l'auto-encodeur a été entraîné. Un concepteur de jeux vidéo peut choisir une ville comme cadre de l'action, mais sans souhaiter perdre de temps à concevoir séparément les quelques centaines d'immeubles dans lesquels se déroulera l'action. On peut de nos jours confier cette tâche à de bons encodeurs variationnels. Dans un avenir plus lointain, on peut en espérer de bien meilleurs qui pourraient produire des romans se lisant comme du Hemingway ou comme nos romans policiers favoris.

	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	3	14	15	1	0	0	0	0	0	0	0	0	0	0
7	187	221	205	151	74	11	1	0	0	2	8	23	55	69
8	237	249	251	250	249	239	221	225	197	214	216	236	237	228
9	92	194	232	219	217	225	245	251	251	249	241	237	249	250
10	1	8	7	17	31	49	100	126	106	45	37	81	242	251
11	0	0	0	0	1	9	13	7	2	0	1	43	239	247
12	0	0	0	0	0	2	2	0	0	0	2	151	247	215
13	0	0	0	0	0	0	0	0	0	1	61	246	248	57
14	0	0	0	0	0	0	0	0	1	32	207	253	185	10
15	0	0	0	0	0	0	0	0	9	176	251	237	31	1
16	0	0	0	0	0	0	0	0	47	237	252	67	2	0
17	0	0	0	0	1	0	1	9	171	249	237	9	0	0
18	0	0	2	7	1	1	5	100	243	251	138	1	0	0
19	0	0	0	2	1	0	19	217	253	222	19	0	0	0
20	0	0	0	0	0	2	107	246	241	44	1	0	0	0
21	0	0	0	0	1	48	220	247	168	4	0	0	0	0
22	0	0	0	0	18	196	251	233	42	0	0	0	0	0
23	0	0	0	1	98	249	250	140	2	0	0	0	0	0
24	0	0	0	14	237	254	242	40	0	0	0	0	0	0
25	0	0	5	116	252	254	205	8	0	0	0	0	0	0
26	0	0	16	158	253	249	56	4	2	0	1	0	0	0
27	0	0	0	0	2	1	0	0	0	0	0	0	0	0

Figure 7.2 – Reconstruction des exemples Mnist de test de la figure 1.1

Nous commençons par l’auto-encodage de base dans lequel nous reconstruisons simplement l’entrée — les chiffres Mnist. La figure 7.2 présente la sortie d’un auto-encodeur à quatre couches lorsque l’entrée est l’image d’un « 7 » figurant au tout début du chapitre 1. L’auto-encodeur peut être exprimé sous la forme :

$$\mathbf{h} = S(S(\mathbf{x}\mathbf{E}_1 + \mathbf{e}_1)\mathbf{E}_2 + \mathbf{e}_2) \quad (7.1)$$

$$\mathbf{o} = S(\mathbf{h}\mathbf{D}_1 + \mathbf{d}_1)\mathbf{D}_2 + \mathbf{d}_2 \quad (7.2)$$

$$L = \sum_{i=1}^n (x_i - o_i)^2 \quad (7.3)$$

Nous avons deux couches d’encodage entièrement connectées, la première avec des poids $\mathbf{E}_1, \mathbf{e}_1$, la seconde avec $\mathbf{E}_2, \mathbf{e}_2$. Dans la création de la reconstruction du « 7 » de la figure 7.2, la première couche et de forme [784, 256] et la seconde [256,128], de sorte que l’image finale comporte 128 « pixels » c’est-à-dire que c’est en quelque sorte une image de hauteur et de largeur égales à $\sqrt{128}$. L’équation 7.3 indique que nous utilisons une perte d’erreur quadratique, ce qui est logique puisque nous essayons de prédire des valeurs de pixels, et non l’appartenance à une classe. Nous utilisons S , la fonction sigmoïde, pour notre non-linéarité.

Vous vous demandez peut-être pourquoi nous utilisons une fonction d’activation sigmoïde plutôt que notre quasi-standard `relu`. C’est parce que jusqu’ici nous ne

nous sommes pas beaucoup souciés des valeurs réelles qui étaient transmises à travers le réseau : à la fin, elles passent toutes dans la fonction softmax, et l'essentiel de ce qu'il en reste, ce sont leurs valeurs relatives. Un auto-encodeur, par contre, compare les valeurs absolues de l'entrée à celles de la sortie. Vous vous souvenez peut-être que pour normaliser les données de nos images Minst (page 18), nous avons divisé les valeurs brutes des pixels par 255 pour obtenir des valeurs comprises entre 0 et 1. Comme nous l'avons vu à la figure 2.7, la fonction sigmoïde prend des valeurs comprises entre 0 et 1. Comme celles-ci correspondent exactement à la plage de valeurs des pixels, cela signifie que le réseau de neurones n'a pas besoin d'apprendre à normaliser ces données. Naturellement, cela rend l'apprentissage pour produire ces valeurs plus simple qu'avec `relu`, qui a la même borne inférieure, mais pas la borne supérieure.

Étant donné que les auto-encodeurs possèdent de multiples couches assez semblables (dans ce cas, elles sont toutes entièrement connectées), le module `layers` présenté à la section 2.4.4 est particulièrement utile ici. Remarquez en particulier que le modèle présenté dans les équations 7.1-7.3 peut être codé succinctement comme suit :

```
E1=layers.fully_connected(img,256,tf.sigmoid)
E2=layers.fully_connected(E1,128,tf.sigmoid)
D2=layers.fully_connected(E2,256,tf.sigmoid)
D1=layers.fully_connected(D2,784,tf.softmax)
```

où nous supposons que notre image `img` arrive sous forme d'un simple vecteur à 784 éléments.

Une autre façon d'empêcher un auto-encodeur de se contenter de recopier l'entrée vers la sortie consiste à ajouter du *bruit*. Nous utilisons ici le mot « bruit » dans son sens technique d'événements aléatoires corrompant l'image originale que, dans ce même contexte, on appellerait le *signal*. Dans un *auto-encodeur de débruitage*, nous ajoutons du bruit à l'image sous la forme de pixels aléatoirement remis à zéro. En règle générale, près de 50 % des pixels peuvent être dégradés de cette façon. La fonction de perte d'un auto-encodeur est encore une fois la perte d'erreur quadratique, cette fois entre les pixels de l'image non corrompue et ceux de l'image produite en sortie du décodeur.

7.2 AUTO-ENCODAGE CONVOLUTIF

À la section précédente, nous avons construit un auto-encodeur pour les chiffres du Mnist utilisant un encodeur pour réduire l'image initiale de 784 pixels tout d'abord à 256, puis à 128. Cela fait, le décodeur a inversé le processus, en ce sens qu'en partant de 128 « pixels », nous avons obtenu tout d'abord une image de 256, puis de 784. Tout ceci a été effectué par des couches entièrement connectées. La première avait une matrice de poids de forme [784, 256], la seconde [256, 128], puis, pour

le décodeur, [128, 256] suivie par [256, 784]. Cependant, comme nous l'avons vu précédemment lorsque nous avons exploré l'apprentissage profond dans le domaine de la vision par ordinateur, les meilleurs résultats sont obtenus lorsqu'on utilise la convolution. Dans cette section, nous allons construire un auto-encodeur utilisant des méthodes convolutives.

L'encodeur convolutif permettant de réduire les dimensions de l'image ne pose pas problème. Au chapitre 3, nous avons remarqué comment un pas horizontal et vertical de 2 par exemple permettait de réduire la taille de l'image d'un facteur 2 dans chaque dimension. Au chapitre 3, nous ne nous intéressions pas à la compression d'image, c'est pourquoi, en comptant la taille du canal (le nombre de filtres appliqués à chaque portion d'image), nous nous sommes retrouvés à la fin du processus de convolution avec davantage de nombres décrivant l'image qu'au début (une image 7 par 7 multipliée par 32 filtres différents, ce qui donne 1 568). Cette fois, nous voulons absolument que la couche intermédiaire encodée ait beaucoup moins de valeurs que l'image d'origine, c'est pourquoi nous pourrions, par exemple, faire trois couches de convolution, la première couche nous amenant à $14 * 14 * 10$, la deuxième à $7 * 7 * 10$ et la troisième à $4 * 4 * 10$ (les valeurs exactes sont, bien sûr, des hyperparamètres).

Le décodage avec convolution est bien moins évident. La convolution n'accroît jamais la taille de l'image, c'est pourquoi le mode de fonctionnement de la reconstruction n'est pas évident. La solution consiste littéralement à étendre l'image d'entrée avant de la convoluer avec une banque de filtres. À la figure 7.3 nous considérons le cas où la couche cachée de l'auto-encodeur est une image $4 * 4$ que nous voulons étendre jusqu'à une image $8 * 8$. Pour ce faire, nous entourons chaque pixel réel par suffisamment de zéros pour créer une image $8 * 8$ (les valeurs réelles des pixels n'ont qu'un but illustratif). Ceci nécessite d'ajouter à chaque valeur de pixel réel des zéros à gauche, en diagonale à gauche, et vers le haut. Comme vous vous en doutez, en ajoutant suffisamment de zéros nous pouvons étendre l'image jusqu'à n'importe quelle taille. Ensuite, si nous effectuons la convolution de cette nouvelle image avec conv2d, un pas de 20, une marge de type Same, nous obtenons une nouvelle image $8 * 8$.

					0	0	0	0	0	0	0	0	0
					0	1	0	2	0	3	0	4	
1	2	3	4		0	0	0	0	0	0	0	0	0
4	3	2	1		0	4	0	3	0	2	0	1	
2	1	4	3	→	0	0	0	0	0	0	0	0	0
3	4	1	2		0	2	0	1	0	4	0	3	
					0	0	0	0	0	0	0	0	0
					0	3	0	4	0	1	0	2	

Figure 7.3 – Complétion d'une image avant son décodage dans un auto-encodeur convolutif

Chapitre 7 · Modèles de réseaux de neurones non supervisés

```
mnist = input_data.read_data_sets("MNIST_data")

orgI = tf.placeholder(tf.float32, shape=[None, 784])
I = tf.reshape(orgI, [-1, 28, 28, 1])
smallI = tf.nn.max_pool(I, [1, 2, 2, 1], [1, 2, 2, 1], "SAME")
smallerI = tf.nn.max_pool(smallI, [1, 2, 2, 1], [1, 2, 2, 1], "SAME")
feat = tf.Variable(tf.random_normal([2, 2, 1, 1], stddev=.1))
recon = tf.nn.conv2d_transpose(smallerI, feat, [100, 14, 14, 1],
                               [1, 2, 2, 1], "SAME")
loss = tf.reduce_sum(tf.square(recon - smallI))
trainop = tf.train.AdamOptimizer(.0003).minimize(loss)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(8001):
    batch = mnist.train.next_batch(100)
    fd={orgI:batch[0]}
    oo,ls,ii,_ = sess.run([smallI, loss, recon, trainop], fd)
```

Figure 7.4 – Convolution transposée sur un chiffre Mnist

Nous obtenons donc une image aux dimensions appropriées, mais comporte-t-elle des valeurs appropriées ? Pour montrer comment c'est possible, la figure 7.4 fournit du code TF illustrant la reconstruction avec convolution. Nous y sous-échantillonnons une image Mnist puis la reconstruisons en utilisant la convolution. Le sous-échantillonnage s'effectue en deux étapes :

```
smallI=tf.nn.max_pool(I, [1, 2, 2, 1], [1, 2, 2, 1], "SAME")
smallerI=tf.nn.max_pool(smallI, [1, 2, 2, 1], [1, 2, 2, 1], "SAME")
```

La première commande crée une version 14×14 de l'image où chaque portion 2×2 distincte de l'image d'origine est représentée par la valeur de pixel la plus élevée dans cette portion. Reportez-vous à l'image de gauche de la figure 7.5 où vous trouverez un exemple d'une image 14×14 d'un « 7 ». La seconde commande crée une version 7×7 encore plus petite. Les deux lignes suivantes de la figure 7.4 (feat, recon) créent recon, une reconstruction de l'image 7×7 en 14×14 , comme illustré sur l'image de droite de la figure 7.5. La figure 7.4 n'illustre par la façon dont nous utilisons normalement la reconstruction convolutive, conçue pour faire suite à un encodage convolutif. Nous avons préféré commencer par une image compréhensible, afin que vous puissiez mieux voir ce qui lui arrive. Sur la figure 7.5 nous voyons que si la reconstruction n'est pas vraiment parfaite, elle a fonctionné grossièrement (nous avons remplacé les zéros par des blancs sur la figure pour que les contours du « 7 » ressortent mieux).

9	9		4	5	9	9		6	6	2	2	5	5	6	6
9	9	9	9	9	9	9		5	6	2	2	5	5	5	6
9	9	9	9	9	9	9	7	5	6	5	6	5	6	5	6
8	8		9	9	9			5	5	5	5	6	6	6	6
			9	9	5			5	5	5	5	5	6	5	6
			6	9	9						6	6	6	6	
			8	9	9						5	6	5	6	
			9	9	7						1	1	6	6	4
			1	9	9	1					1	1	5	6	4
			4	9	9						2	2	6	6	
			1	9	4						2	2	5	6	

Figure 7.5 – 7 Mnist 14 * 14 et version reconstruite à partir de la version 7 * 7

La ligne de code essentielle de la figure 7.4 est l'appel à `conv2d_transpose`. Comme nous venons de le mentionner, `conv2d_transpose` est d'ordinaire utilisée pour « défaire » ce qu'a fait `conv2d`, par exemple dans :

```
tf.nn.conv2d(img,feat,[1,2,2,1],"SAME")
```

Cet appel encodait l'image que `conv2d_transpose` pourra reconstituer. Si nous ignorons le troisième argument de la version transposée, les arguments des deux fonctions sont exactement les mêmes. Cependant, ils ne sont pas interprétés de la même façon. Dans les deux cas, le premier argument est le tenseur 4D à manipuler, et le second est le jeu de filtres convolutifs à utiliser. Mais `conv2d_transpose`, quels que soient les arguments de pas et de compléction, va utiliser un pas de 1 et une compléction de type Same. Le but de ces arguments est plutôt de déterminer comment ajouter tous les zéros supplémentaires, comme sur la figure 7.3 — par exemple, pour défaire la contraction due à un pas de 2, il nous faudrait généralement compléter chaque pixel véritable par trois pixels nuls supplémentaires, comme sur la figure 7.3.

Malheureusement, il n'est pas possible de déterminer complètement la taille de l'image de sortie de `conv2d_transpose` sur la seule base de ces informations. Par conséquent, le troisième argument de `conv2d_transpose` est la taille de l'image de sortie désirée. Sur la figure 7.4, c'est [100, 14, 14, 1] : 100 est la taille du lot, nous voulons une image de sortie 14 * 14 et seulement un canal. La situation créant l'ambiguité est celle où le pas est supérieur à 1 avec une compléction de type Same. Considérez par exemple deux images, l'une 7 * 7 et l'autre 8 * 8. Dans les deux cas, si nous effectuons la convolution avec un pas de 2 et une compléction de type Same, nous obtenons une image de taille 4. Par conséquent, en sens inverse, `conv2d_transpose` avec un pas de 2 et une compléction de type Same ne peut pas savoir laquelle de ces images de sortie l'utilisateur attend.

0	0	0	0	1	-1	1	-1	1	-1
0	0	0	0	-1	0	-1	0	-1	0
0	-1	1	-1	1	-1	0	-1	0	0
-1	0	-1	0	-1	0	-1	0	0	0
0	0	1	-1	1	-1	1	-1	0	-1
0	0	-1	0	-1	0	-1	0	-1	0
0	-1	0	-1	0	0	1	-1	1	-1
-1	0	-1	0	0	0	-1	0	-1	0
1	-1	1	-1	1	-1	1	-1	0	0
-1	0	-1	0	-1	0	-1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 7.6 – Petit chiffre Mnist reconstruit à partir de l'exemple d'entraînement 0

Jusqu'à maintenant, nous nous sommes surtout souciés de compléter l'entrée de manière à obtenir l'effet de reconstruction désiré. Nous nous ne nous sommes pas occupés de la façon dont les filtres géraient cette tâche. À vrai dire, au début de l'entraînement, ils ne le font pas. La figure 7.6 présente l'image reconstruite à partir de l'exemple d'entraînement 0. L'effet prédominant visible dans la figure est l'alternance de 0 et de -1, qui est sans nul doute un artéfact découlant de l'alternance de valeurs nulles de complémentation et de valeurs non nulles des véritables pixels dans l'image transmise à `conv2d_transpose`. Il existe une théorie mathématique quant à la façon dont la convolution transposée peut trouver les valeurs de noyau correctes, mais il nous suffit de savoir que des valeurs de noyau variables et une rétropropagation s'en occupent pour nous.

Pour ce qui est de l'utilisation de `conv2d_transpose` pour l'auto-encodage, c'est tout à fait analogue à l'auto-encodage entièrement connecté. Nous avons une ou plusieurs couches d'encodage utilisant `conv2d` et éventuellement `max_pool` ou `avg_pool`, suivies en général par un même nombre de couches de reconstruction utilisant `conv2d_transpose`.

7.3 AUTO-ENCODAGE VARIATIONNEL

Un *auto-encodeur variationnel* (en anglais, *variational autoencoder*, ou VAE) est une variante d'auto-encodeur dont le but n'est pas de reproduire exactement l'image de départ, mais plutôt de créer une nouvelle image qui soit membre de la même classe d'images mais manifestement nouvelle. Il tire son nom des *méthodes variationnelles*, utilisées en apprentissage automatique bayésien. Revenons encore une fois à notre sympathique jeu de données Mnist. L'objectif initial de notre VAE consiste à fournir en entrée une image numérisée du Mnist et à obtenir en sortie une nouvelle image dont on puisse reconnaître qu'elle est à la fois similaire et différente.

Figure 7.7 – Une image originale et deux « reconstructions »

Si nous n'attachons pas d'importance à ce qu'on puisse reconnaître que la nouvelle image est différente, alors l'auto-encodage standard résout plutôt bien la question. Il suffit d'observer à nouveau sur la figure 7.2 la reconstruction par auto-encodeur du chiffre « 7 » du début du livre (figure 1.1). À l'époque, nous étions fiers de la similarité, mais bien sûr les deux images ne sont pas identiques. Cependant, elles sont si proches que si nous avions imprimé cette dernière version en échelle de gris, il aurait été très difficile de la distinguer de la figure 1.2. De plus, à bien y réfléchir, un auto-encodeur standard avec une perte d'erreur quadratique n'est pas réellement ce que nous voulons. Considérons les trois images 7×7 de la figure 7.7. Celle du haut est censée être une petite image d'un « 1 », et les deux autres sont censées être des reconstructions de la première. La première d'entre elles ressemble à l'original, mais étant donné que le chiffre est décalé de deux pixels vers la droite, les valeurs ne se recouvrent pas et son erreur quadratique lorsqu'on la compare à l'image du haut de la figure 7.7 est de 10. De plus, l'image en bas à droite est en fait plus semblable si l'on se réfère à notre fonction de perte, car elle ne diffère que par deux pixels. Par conséquent, l'auto-encodage standard et la perte d'erreur quadratique ne sont pas vraiment adaptés à notre tâche. Mettons cependant cette objection de côté pour l'instant, afin de nous concentrer sur le mode de fonctionnement du VAE, c'est-à-dire de l'auto-encodeur variationnel. Nous reviendrons sur ce problème plus tard et montrerons comment les VAE y apportent une solution.

Si nous examinons un peu la mécanique, notre programme va prendre en entrée une image puis faire apparaître un vecteur de nombres aléatoires, et ce sont ces nombres aléatoires qui vont contrôler la différence entre l'image originale et la nouvelle — en mettant le même couple image/nombres aléatoires, nous obtenons exactement la même variante de l'image d'entrée. Plus loin nous voyons que nous

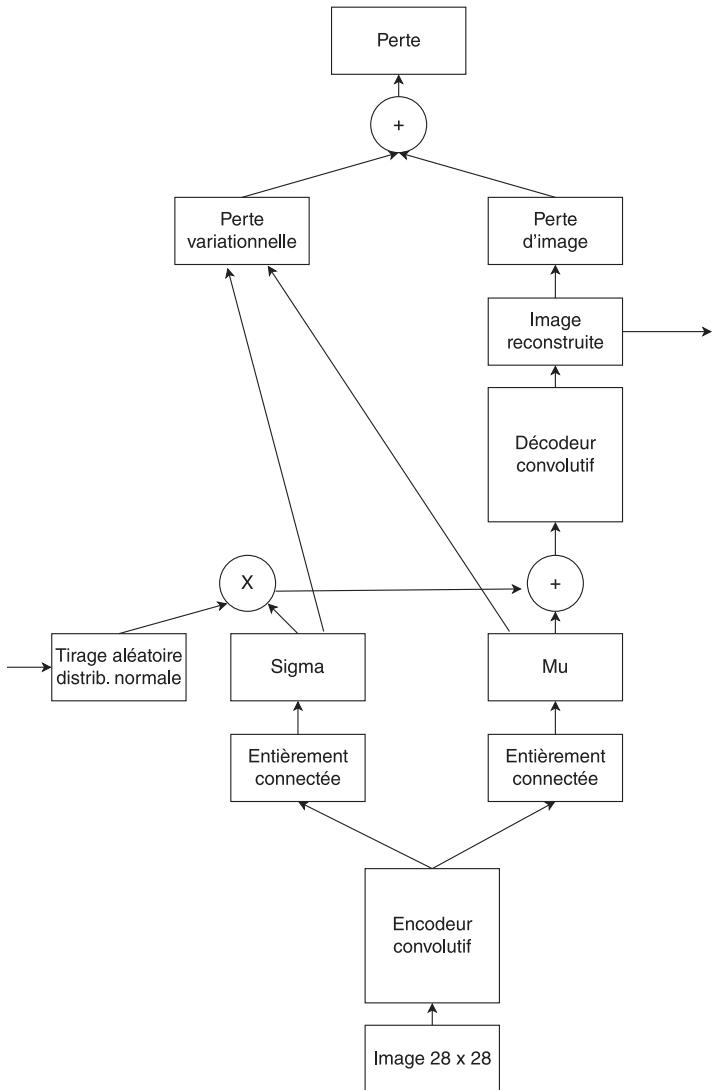


Figure 7.8 – Modèle structurel d'un auto-encodeur variationnel

pouvons omettre l'image, auquel cas nous obtenons en sortie non pas une variation d'une image particulière, mais plutôt une image totalement nouvelle dans le style général de l'ensemble des images. Celle-ci ressemblera en général à l'un des chiffres possibles, mais selon la qualité du travail du VAE, cela peut ne pas être le cas. Vous pourrez en trouver quelques exemples à la figure 7.10.

Un diagramme de l'architecture d'un VAE est présenté à la figure 7.8. Une image est introduite en bas du diagramme et nous pouvons suivre le cheminement du calcul

vers le haut à travers l'encodeur. Les informations encodées sont alors utilisées pour créer non pas un seul plongement d'image, mais deux vecteurs de nombres réels, σ et μ . Nous construisons alors un nouveau plongement d'image en générant un vecteur de nombres aléatoires \mathbf{r} , et en calculant $\mu + \sigma\mathbf{r}$. Ici, on peut considérer μ comme le plongement original de l'image que nous perturbons avec $\sigma\mathbf{r}$ pour obtenir un plongement différent qui est proche, mais pas trop, de l'original. Ce plongement modifié traverse alors un décodeur standard pour produire une nouvelle image. En phase d'exploitation du programme (après l'entraînement), l'utilisateur reçoit en sortie cette nouvelle image. En phase d'entraînement, la nouvelle image est injectée dans une couche « perte d'image ». La perte d'image est simplement la somme des carrés des différences entre les valeurs des pixels de l'original et de la nouvelle image. Donc la source de la variation de l'image de sortie dans un VAE est \mathbf{r} . Si nous fournissons en entrée la même image et le même vecteur de nombres aléatoires, nous obtenons la même image en sortie.

Pour le reformuler un peu différemment, μ est la version encodée de base de l'image d'entrée, comme toujours dans ce chapitre, tandis que σ spécifie de combien nous pouvons faire varier l'image en ayant toujours une version « reconnaissable ». Rappelez-vous que σ et μ sont des vecteurs de nombres réels, par exemple de taille 10. Si $\mu[0] = 1.12$, cela signifie que l'image encodée doit comporter comme premier nombre réel une valeur plus ou moins proche de 1.12, tandis que la variation autour de 1.12 est contrôlée par $\sigma[0]$. Si $\sigma[0]$ est grand (et si notre réseau de neurones fonctionne correctement), cela signifie qu'il est possible de faire amplement varier la première composante de l'image encodée sans que la version de sortie devienne méconnaissable. Si $\sigma[0]$ est petit, c'est impossible.

Mais ceci pose un gros problème. Nous avons supposé que d'une certaine façon les σ et les μ que nous récupérons de l'encodeur sont des nombres tels que $\mu + \mathbf{r} * \sigma$ soit l'encodage d'une image raisonnable. Si la perte est simplement la perte de l'erreur quadratique, nous n'obtenons *pas* le résultat voulu.

C'est ici que la théorie des auto-encodeurs variationnels s'aventure dans des calculs mathématiques compliqués, mais nous préférions demander au lecteur d'accepter certaines modifications semi-plausibles de notre fonction de perte. Les VAE possèdent deux pertes qui sont additionnées pour obtenir la perte totale. Nous avons déjà vu l'une d'elles, la perte quadratique entre les pixels de l'image originale et ceux de la reconstruction. Nous l'appelons la *perte d'image*.

La seconde de ces pertes est la *perte variationnelle* :

$$L_v(\mu, \sigma) = - \sum_i \frac{1}{2} (1 + 2\sigma[i] - \mu[i]^2 - e^{2\sigma[i]}) \quad (7.4)$$

Celle-ci est pour un seul exemple, et il s'agit d'un calcul point à point sur σ et μ (souvenez-vous que ce sont tous deux des vecteurs). Pour rendre ceci plus compréhensible, considérez au préalable ce qui se passe si σ ou μ sont nuls :

$$L_v(\mu, 0) = \mu^2 \quad (7.5)$$

$$L_v(0, \sigma) = -\frac{1}{2} - \sigma + \frac{e^{2\sigma}}{2} \quad (7.6)$$

La première de ces équations nous montre que le réseau de neurones est incité à conserver les valeurs moyennes $\mu = 0$. Bien sûr, la perte d'image va contrecarrer un peu cela, mais toutes choses égales par ailleurs, L_v exige $\mu \approx 0$.

La seconde des équations ci-dessus va conserver σ assez proche de 1. Lorsque σ est inférieur à 1, le second terme de $L_v(0, \sigma)$ domine et nous diminuons la perte en augmentant σ , alors que quand σ est plus grand que 1, le troisième terme se met à croître très rapidement.

Ceci ressemble à une distribution normale standard, et ce n'est pas une coïncidence. Si nous avions effectué des calculs mathématiques, nous aurions compris pourquoi c'est une bonne idée d'encourager l'encodage de l'image à s'approcher d'une distribution normale standard et pourquoi le minimum de la perte variationnelle n'est pas exactement en $\sigma = 1$. Au lieu de cela, nous demandons au lecteur d'admettre qu'en ajoutant cette seconde fonction de perte à notre calcul global, nous obtenons ce que nous voulons : un réseau de neurones qui, lorsqu'on lui donne σ et μ multidimensionnels calculés par nos réseaux de neurones sur la base d'une image Mnist réelle, produit l'encodage d'une image nouvelle et légèrement différente I' telle que :

$$I' = \mu + \mathbf{r}\sigma \quad (7.7)$$

où \mathbf{r} est un vecteur de nombres aléatoires eux-mêmes issus d'une distribution normale standard. Une fois que nous avons cette certitude, nous avons notre VAE.

Mais revenons au problème soulevé précédemment (page 135), à savoir qu'un auto-encodeur standard avec une perte quadratique ne répondait pas réellement à l'objectif fixé pour les auto-encodeurs variationnels : produire des versions notamment différentes d'une image qui soient, en même temps, notamment similaires. Notre exemple du problème de la figure 7.7 comportait deux reconstructions imaginaires d'un « 1 », l'une translatée horizontalement de deux pixels, l'autre avec deux pixels manquants au milieu du trait vertical. Selon le critère de la perte quadratique, le deuxième était plus similaire, mais le premier serait une bonne reconstruction VAE, contrairement au second. Il s'avère que les VAE, lorsqu'ils fonctionnent correctement, surmontent cette difficulté.

Notez d'abord qu'effectivement, lors de l'entraînement des VAE, on utilise la perte quadratique, mais de par leur nature les VAE doivent accepter une perte quadratique plus élevée car ils ne peuvent reconstruire l'image originale que jusqu'à un certain degré d'incertitude délibérément calculé. Notez ensuite que cette incertitude se situe dans l'encodage de l'image, au milieu de l'architecture VAE. Les chercheurs affirment que c'est le meilleur endroit pour cela, si les VAE fonctionnent correctement.

Souvent implicitement mais parfois explicitement dans notre discussion à propos des AE, nous avons pu observer qu'il s'effectue une réduction de dimension en relevant les points communs entre les entrées. Ils ajustent le plongement pour prendre en compte les caractéristiques communes, en ne « mentionnant » que les différences. Supposez, comme c'est raisonnable, que les chiffres Mnist diffèrent légèrement quant à leur position sur la page. Alors une façon d'exploiter ce fait pour avoir un encodage de petite taille consiste, par exemple, à ce qu'un des nombres réels de l'encodage spécifie la position horizontale globale du chiffre. Mais il se pourrait aussi qu'avec seulement 20 réels dans lesquels encoder l'information moyenne pour un chiffre « 1 », nous ne puissions pas nous permettre de consacrer un réel à cela, et que notre auto-encodeur « décide » qu'une autre variante est plus importante pour une bonne reconstruction de notre image : tout ceci n'est qu'une illustration. Le fait est que, s'il y a un réel qui encode la position horizontale globale, alors l'image en bas et à gauche de la figure 7.7 est, en pratique, très proche de l'original du haut — elle n'en diffère qu'en une position d'encodage.

De toute façon, les VAE fonctionnent. La figure 7.9 présente un « 4 » originel du Mnist et une nouvelle version. Il vous suffit de quelques secondes pour être convaincu qu'ils sont différents. De plus, ils diffèrent d'une manière assez typique des VAE (ou du moins pour les VAE qui ne sont pas extraordinaires) — l'image de droite, la reconstruction, est moins caractéristique. C'est un « 4 » un peu terne, si l'on veut. Tout particulièrement, le « 4 » de gauche (l'original) possède un petit crochet en bas du principal trait vertical qui manque totalement à droite.

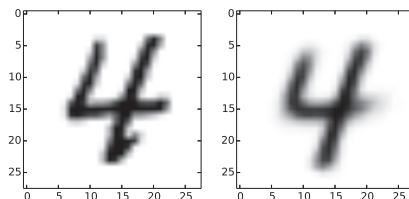


Figure 7.9 – Une image originale et une reconstruction par VAE

Jusqu'ici, nous nous sommes intéressés au problème de la génération d'une image similaire à une image de départ, mais dont on puisse reconnaître qu'elle est différente. Cependant, nous avons noté précédemment que les VAE pouvaient aussi travailler plus librement : étant donné une classe générale d'images, produire un autre membre de cette classe. C'est une tâche beaucoup plus compliquée à effectuer correctement, et pourtant le VAE reste pratiquement inchangé. L'entraînement est en fait exactement le même. La seule différence réside dans la façon dont nous utilisons le VAE. Pour produire une nouvelle image qui ne soit pas basée sur une image existante, le VAE génère un nombre aléatoire (toujours à partir d'une distribution normale standard) mais l'insère cette fois (via `feed_dict`) afin de l'utiliser comme encodage de l'image tout entière. La figure 7.10 montre quelques exemples de résultats obtenus

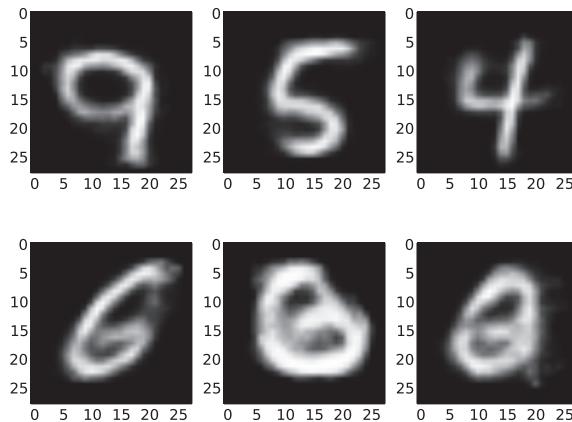


Figure 7.10 – Chiffres Mnist entièrement générés par VAE

avec le même programme que celui qui a généré les exemples de la figure 7.9, mais cette fois-ci sans lui fournir d'image à émuler. Quatre des images sont de toute évidence des chiffres semblables à ceux du Mnist, mais celle en bas à droite ressemble à un « 3-8 » et celle qui la précède est un bien piètre 8. Un modèle plus robuste, de nombreuses itérations d'entraînement supplémentaires et davantage d'attention portée aux hyperparamètres produiraient un bien meilleur résultat.

Avant de quitter les VAE, encore quelques mots destinés à ceux qui aimeraient mieux comprendre la perte variationnelle de l'équation 7.4. Dans notre formulation d'origine, nous générions une nouvelle image I' à partir d'une image I d'origine. Pour ce faire, nous utilisons un encodeur convolutif C afin de produire une représentation réduite $z = C(I)$. Nous nous concentrons ici sur deux distributions de probabilités, $\text{Pr}(z | I)$ et $\text{Pr}(z)$. En particulier, nous faisons d'abord l'hypothèse que toutes deux sont des distributions normales.

Si vous êtes passés placidement de la phrase précédente à celle-ci, soit vous ne réfléchissez pas trop à ce que je viens de dire, soit vous êtes beaucoup plus intelligent que moi. Comment pouvons-nous tout simplement faire l'hypothèse qu'une distribution de quelque chose d'aussi compliqué qu'une image du monde réel, ou même des chiffres du Mnist, est aussi simple qu'une distribution normale ? À dire vrai, ce sera une distribution normale n -dimensionnelle, où n est la dimension de la représentation de l'image produite par l'encodeur convolutif C . Mais même les distributions normales à n dimensions sont des choses relativement simples : une distribution normale 2D est en forme de cloche.

L'idée clé à retenir est que $\text{Pr}(z)$ est une distribution de probabilités non pas basée sur l'image d'origine, mais sur sa représentation $C(I)$. Vous vous souvenez peut-être que précédemment nous avions envisagé d'avoir un élément de la représentation consacré à enregistrer la distance entre le centre du chiffre et le centre de l'image, de sorte que le chiffre « 1 » en haut de la figure 7.7 ait une représentation très similaire

à celle de la version en bas à droite. Supposons que nous soyons capables de mener ce programme à sa fin logique, à savoir que chaque paramètre du vecteur de représentation soit un nombre comme cela, une quelconque spécification des caractéristiques fondamentales par lesquelles une image d'un nombre peut différer d'une autre. D'autres exemples de caractéristiques pourraient être « emplacement du trait vertical principal », « diamètre de la boucle supérieure » (pour les « 8 »), etc. Comme vous pouvez vous en douter, le diamètre de la boucle supérieure d'un « 8 » pourrait être en général de 12 pixels par exemple, mais pourrait être aussi nettement inférieur ou supérieur. Dans un tel cas, il serait très logique de décrire la variabilité de ce paramètre par une loi normale de moyenne 12 et d'écart type 3 par exemple. Pour $\Pr(z | I)$, le même type de raisonnement peut s'appliquer quant à sa distribution normale. Étant donné une image, nous voulons que le VAE produise de nombreuses images qui soient à la fois similaires et différentes. La probabilité de chacune d'entre elles pourrait raisonnablement suivre une distribution normale selon des facteurs clés tels que le diamètre des boucles dans un « 8 ».

Il reste encore de nombreuses étapes avant de parvenir à la perte variationnelle de l'équation 7.4, mais nous n'allons en effectuer qu'une ou deux. Nous supposons désormais que $\Pr(z)$ a une distribution normale standard, c'est-à-dire une distribution normale telle que $\mu = 0$ et $\sigma = 1$, que nous noterons $N(0, 1)$. Par ailleurs, nous supposons que la sortie de l'encodeur a une distribution normale dont la moyenne et l'écart type dépendent de l'image elle-même, ce que nous noterons $N(\mu(I), \sigma(I))$. Ceci explique pourquoi l'encodeur de la figure 7.8 aboutit à deux valeurs μ et σ , et pourquoi, pour obtenir une variation aléatoire, nous avons choisi des nombres selon une distribution normale standard.

Il nous reste encore une dernière étape. Il n'est pas possible en général de satisfaire l'hypothèse que l'une des distributions normales est standard et que l'autre ne l'est pas. Au lieu de cela, nous essayons simplement de minimiser l'écart. Nous pouvons modéliser plus étroitement une image particulière si nous pouvons choisir librement des μ et σ appropriés et si nous sommes poussés dans cette direction par la perte d'image. D'un autre côté, nous voulons que ces valeurs soient comme ailleurs aussi proches de 0 et de 1 que possible, et c'est là que la perte variationnelle intervient.

Autrement dit, nous voulons minimiser la différence entre deux distributions de probabilités $N(0, 1)$ et $N(\mu(I), \sigma(I))$. Une mesure standard de la différence entre deux distributions est la *divergence de Kullback-Leibler* :

$$D_{KL}(P || Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \quad (7.8)$$

Ainsi, si quel que soit i , $P(i) = Q(i)$, alors leur rapport est toujours égal à 1, et $\log 1 = 0$. Nous pouvons donc maintenant donner pour objectif à un VAE de minimiser la perte d'image tout en minimisant simultanément $D_{KL}(N(\mu(I), \sigma(I)) || N(0, 1))$. Par chance, il existe une solution analytique pour minimiser cette dernière, et avec encore quelques calculs algébriques (et une ou deux brillantes idées) ceci conduit à la fonction de perte variationnelle présentée ci-dessus.

7.4 RÉSEAUX ANTAGONISTES GÉNÉRATIFS

Les *réseaux antagonistes génératifs* (en anglais, *generative adversarial networks* ou GAN) sont des modèles de réseaux de neurones non supervisés qui fonctionnent en mettant en concurrence deux modèles de réseaux de neurones. Dans le cas du jeu de données Mnist, le premier réseau (appelé le *générateur*) générera ex nihilo un chiffre Mnist. Le second, le *discriminateur*, reçoit la sortie du générateur ou un exemple d'un véritable chiffre Mnist. Sa sortie est sa propre estimation de la probabilité que l'entrée qui lui a été fournie provienne d'un exemple réel, et non d'un exemple créé par le générateur. La décision du discriminateur agit alors comme un signal d'erreur adressé aux deux modèles mais dans des « directions opposées », en ce sens que s'il est certain que sa décision est correcte, cela implique une erreur importante pour le générateur (pour ne pas avoir réussi à tromper le discriminateur), tandis que si le discriminateur a été largement trompé, la perte importante est pour le discriminateur.

Comme tous les auto-encodeurs, un GAN peut être utilisé pour apprendre la structure des données d'entrée sans étiquettes. Par ailleurs, comme les VAE en particulier, les GAN peuvent générer de nouvelles variantes d'une classe, à titre de prototype une classe d'images, mais en théorie à peu près n'importe quoi. Les GAN sont un sujet brûlant en apprentissage profond, parce qu'en un certain sens un GAN constitue une *fonction de perte universelle*. N'importe quel ensemble d'images, de textes, de décisions de planification, etc., pour lequel nous possédons des données peut être utilisé pour l'apprentissage non supervisé d'un GAN avec la même perte basique.

L'architecture de base d'un GAN est présentée à la figure 7.11. Pour comprendre comment il fonctionne, prenons un exemple trivial : le discriminateur reçoit tour à tour deux nombres. L'un, la donnée « réelle », est généré à partir d'une distribution normale de moyenne 5 et d'écart type 1 par exemple, ce qui fait que les nombres produits sont essentiellement compris entre 3 et 7. Le nombre « factice » est généré

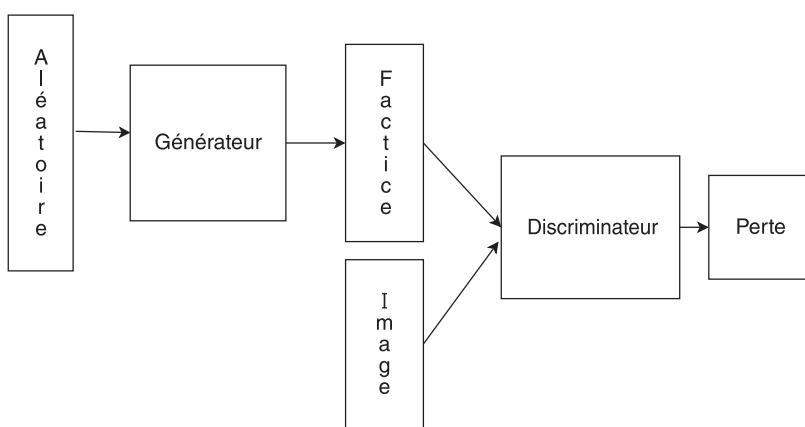


Figure 7.11 – La structure d'un réseau antagoniste génératif

par le générateur qui est un réseau de neurones monocouche (dans cette section uniquement, un nombre « réel » est opposé à un nombre factice, et non à un nombre complexe). Le générateur reçoit un nombre aléatoire, ayant la même probabilité de se trouver n'importe où entre -8 et +8. Pour pouvoir tromper le discriminateur, il doit apprendre à modifier le nombre aléatoire, pour s'efforcer de le faire ressortir entre 3 et 7. Initialement, le réseau de neurones générateur a des paramètres proches de zéro, de sorte qu'en fait il produit surtout des nombres proches de zéro en sortie.

Les GAN mettent en œuvre plusieurs aspects de TensorFlow dont nous avons déjà parlé, et nous donnons le code complet d'un GAN simple à la figure 7.12. Nous avons numéroté chaque petite section du code pour pouvoir nous y référer.

Observez d'abord la section 7 où nous entraînons le GAN. Repérez la boucle d'entraînement principale qui va effectuer 5 001 itérations. Nous générerons tout d'abord une donnée réelle — un nombre aléatoire proche de 5 — et un nombre aléatoire compris entre -8 et 8 pour alimenter le générateur. Nous mettons ensuite à jour, séparément, d'abord le discriminateur puis le générateur. Finalement, toutes les 500 itérations, nous imprimons des données de suivi.

Nous reviendrons bientôt à cette section, mais prenons d'abord de la hauteur pour comprendre comment l'ensemble doit fonctionner. Nous voulons que le discriminateur fournit en sortie un seul nombre (o) qui doit être la probabilité que le nombre qu'il vient de voir provienne de la distribution réelle. Observez brièvement la section 3 du code. Lorsque nous exécutons la fonction `discriminator`, celle-ci construit un réseau neuronal à propagation avant à quatre couches totalement connectées. Les trois premières couches ont des fonctions d'activation `relu`, tandis que la dernière utilise une sigmoïde. Étant donné que les sigmoïdes fournissent en sortie des nombres compris entre 0 et 1, comme nous l'avons remarqué précédemment ils sont appropriés pour produire des nombres censés être des probabilités (la probabilité que l'entrée provienne de la distribution réelle). Plus précisément, la sortie est interprétée comme la probabilité que l'entrée soit un membre d'une classe (la classe des entrées réelles). Ceci détermine notre choix de fonction de perte : nous utilisons la perte d'entropie croisée.

Lorsque le discriminateur reçoit un nombre provenant de la vraie distribution normale (n_r), la perte est l'opposé du log de la sortie du réseau de neurones discriminateur (o_r). Lorsque le discriminateur reçoit le nombre factice généré, la perte est $\ln(1 - o_f)$. Remarquez bien que, dans la section 7 de la figure 7.12, nous créons d'abord dans la boucle d'entraînement quelques nombres réels, puis quelques nombres aléatoires à fournir au générateur. Juste après le commentaire « mise à jour du discriminateur », nous le faisons en fournissant l'optimiseur d'Adam aux deux. La fonction de perte du discriminateur L_d est donnée par :

$$L_d = \frac{1}{2}(-\ln(o_r) - \ln(1 - o_f)) \quad (7.9)$$

Si vous examinez la section 5 où nous préparons le code de la perte et de l'entraînement, vous voyez la perte du discriminateur définie comme dans l'équation 7.9.

Chapitre 7 · Modèles de réseaux de neurones non supervisés

```
bSz, hSz, numStps, logEvery, genRange = 8, 4, 5000, 500, 8

1 def log(x): return tf.log(tf.maximum(x, 1e-5))

2 with tf.variable_scope('GEN'):
    gIn = tf.placeholder(tf.float32, shape=(bSz, 1))
    g0 = layers.fully_connected(gIn, hSz, tf.nn.softplus)
    G = layers.fully_connected(g0, 1, None)
gParams = tf.trainable_variables()

3 def discriminator(input):
    h0 = layers.fully_connected(input, hSz*2, tf.nn.relu)
    h1 = layers.fully_connected(h0, hSz*2, tf.nn.relu)
    h2 = layers.fully_connected(h1, hSz*2, tf.nn.relu)
    h3 = layers.fully_connected(h2, 1, tf.sigmoid)
    return h3

4 dIn = tf.placeholder(tf.float32, shape=(bSz, 1))
with tf.variable_scope('DIS'):
    D1 = discriminator(dIn)
with tf.variable_scope('DIS', reuse=True):
    D2 = discriminator(G)
dParams = [v for v in tf.trainable_variables()
           if v.name.startswith('DIS')]

5 gLoss = tf.reduce_mean(-log(D2))
dLoss = 0.5*tf.reduce_mean(-log(D1) - log(1-D2))
gTrain = tf.train.AdamOptimizer(.001).minimize(gLoss, var_list=gParams)
dTrain = tf.train.AdamOptimizer(.001).minimize(dLoss, var_list=dParams)

6 sess = tf.Session()
sess.run(tf.global_variables_initializer())

7 gmus, gstds = [], []
for i in range(numStps+1):
    real = np.random.normal(5, 0.5, (bSz, 1))
    fakeRnd = np.random.uniform(-genRange, genRange, (bSz, 1))
    # mise à jour du discriminateur
    lossd, gout, _ = sess.run([dLoss, G, dTrain], {gIn:fakeRnd, dIn:real})
    gmus.append(np.mean(gout))
    gstds.append(np.std(gout))
    # mise à jour du générateur
    fakeRnd = np.random.uniform(-genRange, genRange, (bSz, 1))
    lossg, _ = sess.run([gLoss, gTrain], {gIn:fakeRnd})
    if i % logEvery == 0:
        frm = np.max(i-5, 0)
        cmu = np.mean(gmus[frm:(i+1)])
        cstd = np.mean(gstds[frm:(i+1)])
        print('{}:\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}'.
              format(i, lossd, lossg, cmu, cstd))
```

Figure 7.12 – GAN pour apprendre la moyenne d'une distribution normale

Ici o_r est le degré de crédibilité que le discriminateur accorde à l'entrée. Inversement, la perte inclut aussi un terme qui pénalise le discriminateur selon le degré de crédibilité qu'il accorde à un nombre factice (généré) (o_f).

Examinons donc ce qui pourrait se passer sur le premier exemple d'entraînement. Comme nous l'avons déjà noté, le générateur produit quelque chose proche de zéro, mettons 0.01. L'échantillon réel de notre distribution normale pourrait être 3.8. Cependant les paramètres du discriminateur ont été initialisés proches de 0, donc o_r et o_f sont tous deux proches de 0. Initialement, L_d va être dominée par le terme $-\ln(o_r)$ qui s'approche de $-\infty$ lorsque o_r se rapproche de 0, mais le programme apprend rapidement à ne pas affecter de probabilités trop proches de 0.

Ce qui est plus important, c'est la manière dont la dérivée de la perte affecte les paramètres du discriminateur. En revenant au réseau à une couche du chapitre 1, nous voyons que les dérivées des paramètres de poids sont proportionnelles aux entrées pour les poids en question (équation 1.22). Si nous effectuons le calcul, nous trouvons que lorsque nous donnons au discriminateur l'exemple réel, le poids évolue vers le haut proportionnellement à la valeur de l'échantillon (ici 3.8). Inversement, le même poids évolue vers le bas lorsque nous l'entraînons sur la valeur factice du générateur, mais ce n'est que de 0.01, donc le discriminant acteur évolue lentement vers l'association de valeurs d'entrée plus élevées à des valeurs réelles.

Ensuite, voyons comment le programme devrait noter les performances du générateur. Le générateur veut tromper le discriminateur en ce sens que quand ce dernier reçoit la sortie du premier, le générateur veut que le discriminateur pense qu'il a reçu un vrai nombre, et non un factice. Par conséquent, la perte du générateur L_g devrait être :

$$L_g = -\ln(o_f) \quad (7.10)$$

Le lecteur pourra vérifier que la première ligne de la section 5 du code du GAN définit exactement de cette manière la perte du générateur.

Pour résumer : la boucle principale d'entraînement de la section 7 entraîne d'abord le discriminateur en lui fournissant des nombres, certains véritables et certains factices, avec une fonction de perte qui pénalise les erreurs dans les deux directions, les véritables nombres jugés factices, et inversement. Puis elle entraîne le générateur avec une perte basée simplement sur la qualité de l'identification par le discriminateur des valeurs factices transmises par le générateur.

Malheureusement, les choses sont un peu plus compliquées. Notez d'abord que la section 4 comporte deux appels au code construisant un discriminateur. Ceci parce que dans TF vous ne pouvez pas alimenter séparément un réseau unique depuis deux sources de données différentes (alors que vous pouvez par exemple alimenter un réseau avec la concaténation de deux tenseurs). Nous créons donc, en un certain sens, deux discriminateurs. Le premier, D1, est alimenté à partir de la véritable distribution, alors que le second reçoit en entrée les valeurs factices du générateur D2.

Bien sûr, nous ne voulons pas réellement séparer les réseaux, de sorte que, pour les unifier, nous insistons pour que les deux réseaux partagent les mêmes paramètres, et par conséquent calculent exactement la même fonction sans prendre beaucoup plus de place qu'une version unique. C'est le but des appels à `tf.variable_scope` que nous voyons à la section 4. Nous avons déjà utilisé cette fonction TF au chapitre 5 (voir 93). À ce moment-là, nous avions besoin de deux modèles LSTM et nous voulions éviter les conflits de noms, nous en avons donc défini un avec une portée de variable `enc` (encodeur) et un autre avec une portée `dec` (décodeur). Au sein du premier d'entre eux, les noms de toutes les variables se retrouvaient préfixés par `enc`, et dans le deuxième par `dec`. Ici nous avons la préoccupation opposée. Nous voulons éviter d'avoir deux ensembles de variables séparées, donc nous adoptons le même nom pour les deux portées, et dans le deuxième appel nous demandons explicitement à TF de réutiliser les mêmes variables en ajoutant `reuse = True`.

Il nous reste une dernière difficulté à résoudre avant de poursuivre. La section 7 de la figure 7.12 nous montre d'abord l'entraînement du discriminateur, puis du générateur, à chaque étape d'apprentissage. De plus, pour entraîner le discriminateur, nous fournissons à TF deux nombres, un échantillon réel et un nombre aléatoire pour alimenter le générateur. Nous exécutons alors le générateur pour créer un nombre factice, et la probabilité que le discriminateur considère que ce nombre provienne de la véritable distribution of . Cette dernière apparaît dans la perte définie par l'équation 7.9. Cependant, en l'absence de traitement particulier, lors de la passe arrière les paramètres du générateur sont aussi modifiés et, pire, sont modifiés de manière à rendre plus petite la perte du discriminateur. Comme nous l'avons vu, ce n'est *pas* ce que nous voulons. Nous voulons modifier les paramètres du générateur de façon à ce que la tâche du discriminateur devienne plus difficile. Par conséquent, lorsque nous exécutons la rétropropagation et modifions les paramètres du discriminateur à la section 5 (voir la ligne qui définit `dTrain`), nous ordonnons à TF de ne modifier qu'un seul des deux ensembles de paramètres. Notez en particulier l'argument nommé `dAdamOptimizer`. Cet argument prévient l'optimiseur de ne modifier (pour cet appel particulier) que les variables TF dans la liste.

Pour ce qui concerne la façon dont les classes de paramètres `gParams` et `dParams` sont définies, reportez-vous à la fin des sections 2 et 4. La fonction TF `trainable_variables` renvoie un tenseur de toutes les variables dans le graphe TF définies jusqu'à ce point.

7.5 RÉFÉRENCES ET LECTURES SUPPLÉMENTAIRES

Les origines de l'auto-encodage dans les réseaux de neurones sont quelque peu incertaines. L'ouvrage de Goodfellow *et al.* [GBC16] cite la thèse de doctorat de Yann LeCun [LeC87] comme plus ancienne référence sur le sujet.

Parmi les thèmes mentionnés dans ce chapitre, le premier à avoir acquis une certaine notoriété parmi les spécialistes des réseaux de neurones est celui des auto-encodeurs variationnels. La référence en la matière est une publication de Diederik Kingma et Max Welling [KW13]. J'ai trouvé le blog de Felix Mohr [Moh17] très utile, et mon code s'en inspire. Si vous avez les bases nécessaires en statistiques et si vous vous intéressez à la théorie derrière les auto-encodeurs variationnels (ou VAE), j'ai trouvé que le tutoriel VAE de Carl Doersch constituait une bonne référence [Doe16].

L'histoire des réseaux antagonistes génératifs (ou GAN), par contre, est totalement claire en ce sens que l'idée de base est apparue quasiment dans sa forme actuelle dans une publication de Ian Goodfellow *et al.* [GPAM⁺14]. J'ai beaucoup appris du blog de John Glover [Glo16]. Mon code de GAN pour l'apprentissage d'une distribution normale est basé sur le sien, et lui-même en attribue le crédit à [Jan16].

Le commentaire indiquant que les GAN sont des fonctions de perte « universelles » (page 142) est repris d'une conférence de Phillip Isola [Iso] présentant plusieurs façons intéressantes d'effectuer un apprentissage non supervisé, tout particulièrement en utilisant des GAN.

Exercices

7.1 Considérons un encodage avec des couches entièrement connectées. Les chiffres Mnist ont toujours des rangées de zéros sur le pourtour. Compte tenu de notre commentaire indiquant que les auto-encodeurs ont un encodage d'image qui ignore les points communs, qu'est-ce que cela implique (toutes choses égales par ailleurs) pour les valeurs des poids du premier niveau entraîné ?

7.2 Même situation et question que dans l'exercice 7.1, mais maintenant pour les poids d'entraînement de la dernière couche avant la sortie.

7.3 Fournissez un appel à `conv2d_transpose` qui effectue la transposition indiquée à la figure 7.3.

7.4 En supposant que `img` est un tableau de pixels $2 * 2$ comportant des nombres de 1 à 4, montrez-en la version complétée produite par l'appel suivant à `conv2d_transpose` :

```
tf.nn.conv2d_transpose(img, flts, [1, 6, 6, 1], [1, 3, 3, 1], "SAME")
```

Vous pouvez ignorer la première et la dernière composante de la forme.

7.5 Bien que ce ne soit pas très important, pourquoi avons-nous fixé la boucle d’entraînement du GAN à 5 001 itérations dans la figure 7.12 plutôt qu’à 5 000 ?

7.6 Le GAN de la figure 7.12 imprime à la fois une moyenne des données générées que nous avons utilisée pour juger de l’exactitude du GAN, et l’écart type des données générées que nous avons ignoré ci-dessus. En réalité, et bien que nous ayons choisi pour les nombres véritables un écart type de 0.5 (recherchez donc où cela est réalisé !), la valeur véritable de σ imprimée toutes les 500 itérations a débuté très haut, puis décrue rapidement en dessous de 0.5, et semble devoir diminuer encore. Expliquez pourquoi ce modèle GAN n’a aucune incitation à apprendre la valeur σ correcte, et pourquoi il est raisonnable que la valeur qu’il trouve effectivement soit plus petite que la valeur réelle.

ANNEXE : CORRIGÉ DES EXERCICES

A

A.1 CHAPITRE 1

Exercice 1.1. Si a est la valeur du chiffre du premier exemple d'entraînement, alors après avoir effectué l'entraînement sur ce seul exemple, seule la valeur b_a devrait augmenter, et, pour toutes les valeurs de chiffres $a' \neq a$, $b_{a'}$ devrait décroître.

Exercice 1.2. (a) Les logits de la passe avant sont respectivement $(0 * 0.2) + (1 * -0.1) = -0.1$ et $(0 * -0.3) + (1 * 0.4 * 1) = 0.4$. Pour calculer les probabilités, nous calculons d'abord le dénominateur de softmax $e^{-0.1} + e^{0.4} = 0.90 + 1.50 = 2.40$. Puis les probabilités sont $0.9/2.4 = 0.38$ et $1.5/2.4 = 0.62$. (b) La perte est $-\ln 0.62 = -1.9$. Dans l'équation 1.22 nous voyons que $\Delta(0, 0)$ va être un produit de termes incluant $x_0 = 0$, donc $\Delta(0, 0) = 0$.

Exercice 1.5. En effectuant le produit matriciel nous obtenons :

$$\begin{pmatrix} 4 & 7 \\ 8 & 15 \end{pmatrix} \quad (\text{A.1})$$

Puis, en ajoutant le vecteur de droite à chacune des lignes, nous obtenons :

$$\begin{pmatrix} 8 & 12 \\ 12 & 19 \end{pmatrix} \quad (\text{A.2})$$

Exercice 1.6. La dérivée de la perte quadratique par rapport à b_j se calcule de manière pratiquement identique à ce que nous avons montré pour la perte d'entropie croisée sauf que :

$$\frac{\partial L}{\partial l_j} = \frac{\partial}{\partial l_j} (l_j - t_j)^2 = 2(l_j - t_j) \quad (\text{A.3})$$

Étant donné que la dérivée de l_j en tant que fonction de b_j vaut 1, il en résulte que :

$$\frac{\partial L}{\partial b_j} = 2(l_j - t_j) \quad (\text{A.4})$$

A.2 CHAPITRE 2

Exercice 2.1. Si nous ne spécifions pas d'indice de réduction, il est supposé valoir 0, auquel cas nous ajoutons les colonnes. Ce qui nous donne $[0, 3.2, 9]$.

Exercice 2.2. Cette nouvelle version est nettement plus lente que celle d'origine car, à chaque itération de la boucle d'entraînement principale, le programme crée un nouvel optimiseur de descente de gradient plutôt que d'utiliser le même à chaque fois.

Exercice 2.4. Vous ne pouvez pas appliquer tensordot car les dimensions ne sont pas égales. Ce serait possible si le premier argument tenseur avait pour forme [4, 3], et le second [2, 4, 4]. Si vous les concaténez simplement, vous obtenez [4, 3, 2, 4, 4]. Comme nous effectuons le produit scalaire de la composante 0 du premier tenseur par la composante 1 du deuxième, ces dimensions disparaissent et vous obtenez un résultat de forme [3, 2, 4].

A.3 CHAPITRE 3

Exercice 3.1. (a) En voici un exemple :

$$\begin{array}{ccc} -2 & 1 & 1 \\ -2 & 1 & 1 \\ -2 & 1 & 1 \end{array}$$

Pour ce qui concerne **(b)**, le fait est qu'il y en a une infinité. En multipliant les valeurs numériques du noyau ci-dessus par un nombre positif quelconque, on obtient un nouvel exemple.

Exercice 3.5. En termes de syntaxe, la seule différence est le pas de [1, 1, 1, 1] plutôt que celui de [1, 2, 2, 1] précédemment. Par conséquent, nous appliquons maxpool sur chaque portion $2 * 2$, plutôt que de l'appliquer une fois sur deux. Lorsque le pas de maxpool vaut 1, la forme de `convOut` est identique à celle de l'image d'entrée, alors qu'avec un pas de 2 on obtient approximativement la moitié de la taille tant en hauteur qu'en largeur. Par conséquent, la réponse à la première question est « non ». On n'obtient pas non plus forcément le même ensemble de valeurs puisque, par exemple, si nous avons deux portions contiguës contenant de petites valeurs mais entourées par des valeurs plus grandes, la sortie dans le cas d'un pas de 1 inclura la plus grande de ces deux petites valeurs, tandis que dans le cas d'un pas de 2 ces valeurs se retrouveront « noyées » par les valeurs des pixels avoisinants. Enfin, la troisième réponse est « oui », en ce sens que chaque valeur de portion figurant dans le premier cas existe aussi dans le deuxième cas, mais pas l'inverse.

Exercice 3.6 (a). Chacun des noyaux que nous créons a pour forme [2, 2, 3], ce qui implique 12 variables par noyau. Étant donné que nous en avons créé 10, ceci aboutit à la création de 120 variables. Étant donné que le nombre de fois que nous appliquons un noyau ne dépend pas de sa taille/forme, ni la taille du lot (100) ni la hauteur/largeur (8/8) n'ont d'effet sur cette réponse.

A.4 CHAPITRE 4

Exercice 4.2. La différence importante, lorsqu'on initialise toutes les valeurs de **E** à 0 (ou à 1), c'est que le réseau de neurones ne voit jamais la véritable entrée, mais seulement son plongement. Donner à tous les plongements la même valeur a pour effet de rendre tous les mots identiques. Manifestement, ceci enlève toute chance d'apprendre quelque chose.

Exercice 4.3. Pour calculer la perte totale lorsqu'on utilise une régularisation L2, il faut calculer la somme des carrés de chacun des poids du modèle. Cette quantité n'est nécessaire à aucun autre endroit du graphe de calcul. Par exemple, pour calculer la dérivée de la perte totale par rapport à $w_{i,j}$, il suffit d'ajouter $w_{i,j}$ à la perte normale.

Exercice 4.5. Tout d'abord, oui, elle peut être meilleure qu'en tirant des mots au hasard dans une distribution uniforme. Le modèle devrait apprendre à affecter une probabilité supérieure aux mots les plus communs (p. ex., « le »). Notez cependant qu'un modèle unigramme n'a pas d'« entrée » et par conséquent n'a pas besoin de plongements ou de poids pour l'entrée des unités linéaires. Il a cependant besoin de biais, car c'est en modifiant ceux-ci qu'il apprend à affecter des probabilités en fonction de la fréquence des mots.

A.5 CHAPITRE 5

Exercice 5.2. Le mécanisme d'attention plus compliqué utilise l'état du décodeur après le temps t pour décider de l'attention utilisée dans le décodage au temps $t + 1$. Mais manifestement, nous ne connaîtrons cette valeur qu'après avoir traité le temps t . En rétropropagation à travers le temps, nous traitons toute une fenêtre de mots en même temps. À l'exception de la toute première position dans la fenêtre du décodeur, nous n'avons pas la valeur d'état entrante dont nous avons besoin pour le calcul. Par essence, nous avons alors besoin d'écrire un nouveau mécanisme de rétropropagation à travers le temps.

Exercice 5.4. (a) Nous voulons une bonne traduction automatique. Dans la mesure où l'autre fonction de perte affecterait les choses, ce serait probablement en déplaçant les poids, de sorte qu'ils donneraient de moins bons résultats pour cette tâche. Par conséquent, les performances seraient dégradées. **(b)** Toutefois l'affirmation (a) n'est vraie que pour les données d'entraînement. Elle ne dit rien des performances obtenues sur d'autres exemples. Ajouter une deuxième fonction de perte devrait aider le programme à en apprendre plus sur la structure du français. Ceci devrait améliorer les résultats sur les nouveaux exemples, qui constituent le plus gros des données de test.

A.6 CHAPITRE 6

Exercice 6.1. Dans une itération sur les valeurs, le seul moyen qu'un état obtienne une valeur non nulle est que : soit il y ait un moyen d'obtenir une récompense immédiate en effectuant une action à partir de cet état (possible uniquement pour l'état 14), soit on puisse effectuer depuis cet état une action conduisant à un état ayant déjà une valeur non nulle (états 10, 13 et 14). Par conséquent, tous les autres états vont conserver leur valeur nulle. L'état 10 a une valeur Q maximum pour $Q(14, l)$, $Q(14, d)$ et $Q(14, r)$. Dans chaque cas, les valeurs sont obtenues en terminant à l'état 15 et valent donc $0.33 * 0.9 * 0.33 = 0.1$, donc $V(10) = 0.1$. En raisonnant exactement de la même façon, $V(14) = 0.1$. Enfin, $V(15)$ obtient sa valeur de $Q(15, d)$ et de $Q(15, r)$. Le calcul est le même dans les deux cas : $0.33 * 0.9 * 0.1 + 0.33 * 0.9 * 0.33 + 0.33 * 1 = 0.03 + 0.1 + 0.33 = 0.47$.

Exercice 6.4. Lorsque nous avons calculé toutes les actions possibles dans REINFORCE, nous avons affirmé que si nous avions sauvegardé leurs probabilités, nous aurions été capables de calculer la perte lors de la seconde passe sans partir d'un état puis calculer les probabilités. Cependant, si nous l'avions fait SANS refaire le calcul conduisant de l'état aux probabilités des actions, alors la passe arrière de TF ne serait pas capable de retracer le calcul en arrière à travers les couches entièrement connectées qui calculent les actions à partir de l'état. Par conséquent il n'y aurait pas de mise à jour des valeurs de cette couche (ou de ces couches) et le programme n'apprendrait pas à calculer la meilleure recommandation d'action pour chacun des états.

A.7 CHAPITRE 7

Exercice 7.1. Pour ignorer la valeur de la bordure, le moyen le plus évident consiste à initialiser les valeurs connectant ses pixels à la première couche à 0. Soit x l'indice d'un pixel dans la version 1D de l'image, $0 < x < 783$. Si i, j parcourt les pixels de l'image $28 * 28$, alors pour tout x tel que $x = j + 28i$, $i < 2$ ou $i > 25$, et $j < 2$ ou $j > 25$, pour tout y tel que $0 \leq y \leq 256$:

$$\mathbf{E}_1(x, y) = 0$$

Exercice 7.3.

```
tf.nn.conv2d_transpose(smallerI, feat, [1, 8, 8, 1], [1, 2, 2, 1], "SAME")
```

Exercice 7.5. Nous voulions que les valeurs de suivi soient imprimées après la dernière itération. Si nous avions donné à range la valeur 5 000, la dernière itération aurait été 4 999 et n'aurait pas déclenché d'impression.

BIBLIOGRAPHIE

- [BCB14] Dzmitry Bahdanau, Kyunghyun Cho, et Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [BCP⁺88] Peter Brown, John Cocke, S Della Pietra, V Della Pietra, Frederick Jelinek, Robert Mercer, et Paul Roossin. A statistical approach to language translation. In *Proceedings of the 12th conference on computational linguistics*, pages 71–76. Association for Computational Linguistics, 1988.
- [BDVJ03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, et Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.
- [Col15] Chris Colah. Understanding LSTM networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, August 2015.
- [Doe16] Carl Doersch. Tutorial on variational autoencoders. *ArXiv e-prints*, August 2016.
- [GB10] Xavier Glorot et Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.
- [GBC16] Ian Goodfellow, Yoshua Bengio, et Aaron Courville. *Deep learning*. MIT Press, 2016.
- [Gér17] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017. (NDT : Cet ouvrage a été édité en français par Dunod en deux tomes, ainsi que la nouvelle édition de 2019 sous les titres *Machine Learning avec Scikit-Learn* et *Deep Learning avec Keras et TensorFlow*.)
- [Glo16] John Glover. An introduction to generative adversarial networks (with code in TensorFlow). <http://blog.aylien.com/introduction-generative-adversarial-networks-code-tensorflow/>, 2016.
- [GPAM⁺14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, et Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
- [HS97] Sepp Hochreiter et Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Bibliographie

- [Iso] Phillip Isola. Learning to see without a teacher. https://www.youtube.com/watch?v=ck3_7tVuCRs.
- [Jan16] Eric Jang. Generative adversarial nets in TensorFlow (Part I). <http://blog.evjang.com/2016/06/generative-adversarial-nets-in.html>, 2016.
- [Jul16a] Arthur Juliani. Simple reinforcement learning with TensorFlow Part 0: Q-learning with tables and neural networks. <https://medium.com/emergent-future>, 2016.
- [Jul16b] Arthur Juliani. Simple reinforcement learning with TensorFlow Part 2: Policy-based agents. <https://medium.com/emergent-future>, 2016.
- [KB13] Nal Kalchbrenner et Phil Blunsom. Recurrent continuous translation models. In *EMNLP*, volume 3, page 413, 2013.
- [KH09] Alex Krizhevsky et Geoffrey Hinton. Learning multiple layers of features from tiny images. *Technical report, University of Toronto*, 2009.
- [KLM96] Leslie Pack Kaelbling, Michael L Littman, et Andrew W Moore. Reinforcement learninga survey. *Journal of artificial intelligence research*, 4:237–285, 1996.
- [Kri09] Alex Krizhevsky. The CIFAR-10 dataset. <https://www.cs.toronto.edu/~kriz/cifar.html>, 2009.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, et Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [Kur15] Andrey Kurenkov. A ‘brief’ history of neural nets and deep learning, Parts 1–4. <http://www.andreykurenkov.com/writing/>, 2015.
- [KW13] Diederik P Kingma et Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- [LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, et Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [LBD⁺90] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, et Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.
- [LeC87] Yann LeCun. *Modèles connexionnistes de l'apprentissage (connectionist learning models)*. PhD thesis, University of Paris, 1987.
- [MBM⁺16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, et Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

- [Mil15] Steven Miller. Mind: how to build a neural network (Part One). <https://stevenmiller888.github.io/mind-how-to-build-a-neural-network>, 2015.
- [Moh17] Felix Mohr. Teaching a variational autoencoder (VAE) to draw Mnist characters. <https://towardsdatascience.com/@felixmohr>, 2017.
- [MP43] Warren S McCulloch et Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biophysics*, 5(4):115–133, 1943.
- [MSC⁺13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, et Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
- [Ram17] Suriyadeepan Ram. Scientia est potentia. <http://suriyadeepan.github.io/2016-12-31-practical-seq2seq/>, December 2017.
- [RHW86] David E Rumelhart, Geoffrey E Hinton, et Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.
- [RMG⁺87] David E Rumelhart, James L McClelland, PDP Research Group, *et al.* *Parallel distributed processing*, volume 1,2. MIT Press, 1987.
- [Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [Rud16] Sebastian Ruder. On word embeddings – Part 1. <http://ruder.io/word-embeddings-1/index.html#fnref:1>, 2016.
- [SB98] Richard S Sutton et Andrew G Barto. *Reinforcement learning : An introduction*, volume 1. MIT Press, 1998.
- [Ten17a] Google TensorFlow. Convolutional neural networks. https://www.tensorflow.org/tutorials/deep_cnn, 2017.
- [Ten17b] Google TensorFlow. A guide to TF layers: Building a neural network. <https://www.tensorflow.org/tutorials/layers>, 2017.
- [TL] Rui Zhao Thang Luong, et Eugene Brevdo. Neural machine translation (seq2seq) tutorial. <https://www.tensorflow.org/tutorials/seq2seq>.
- [Var17] Amey Varangaonkar. Top 10 deep learning frameworks. <https://hub.packtpub.com/top-10-deep-learning-frameworks/>, May 2017.
- [Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

INDEX

UNK, 66

A

a2c, 121, 125
a3c, 125
abandon, 74
abandon, 75
Adam, 120
adaptive moment estimation,
 voir Adam
AE, 127
agent, 105
aléatoire, 17
Alexa, 103
Alexnet, 62
algorithme
 du perceptron, 5, 17
 gourmand, 110
Amazon, 103
analogie de mots, 86
apprentissage
 automatique, 3
 automatique bayésien,
 103, 134
 des poids, 5
 faiblement supervisé, 127
 non supervisé, 3, 127
 par renforcement, 105,
 127
 par renforcement
 profond, 23, 105, 111
 sans modèle, 109, 116
 semi-supervisé, 3
 seq2seq, 127
 séquence à séquence, 89
 supervisé, 3, 127
 taux, 9, 16, 18, 33, 95
 totalement supervisé, 3

apprentissage profond, 1
 définition, 9
approximation, 5
 de fonction, 111
argmax, 34, 107
 axe, 34
arrêt prématuré, 74
assistant vocal, 103
Asynchronous Advantage
 Actor-Critic, 125
attention, 95
 à la position uniquement,
 96
auto-encodeur, 127
 de débruitage, 130
 variationnel, 128, 134,
 147
avantage, 122
 acteur-critique, 121
avg_pool, 61
axone, 3

B

back propagation, voir
 rétrorépropagation
Bahdanau, Dzmitry,
 103
Barto, Andrew, 125
BasicRNNCell, 80, 90
batch, voir lot
Bengio, Yoshua, 25, 85
Berkeley, 45
biais, 3, 13, 15
 noyau, 59
Blunsom, Phil, 103
broadcasting, 22, voir
 diffusion
bruit, 130

C

Caffe, 45
Caffe2, 45
calcul matriciel, 20
Canadian Hansard, 89, 103
canal, 52
caractéristique, 3, 49
cart-pole, 115
cast, 34
ceiling, 51
cell state, voir état de cellule
cellule
 état, 83
chaîne de probabilités, 65
chatbots, 103
checkpoint, voir point de
 contrôle
CIFAR-10, 62
classification, 3, 26
 binaire, 3
co-apprentissage, 128
Colah, Chris, 86
complétion, 50
 type Same, 50, 51
 type Valid, 50, 51
compromis exploration-
 exploitation, 110
concat, 72
condition iid, 23, 73
constant, 27
conv2D, 52
conv2d_transpose, 133
convergence, 19, 23
convolution
 à trois dimensions, 52
 à une dimension, 52
 filtre, 47
 noyau, 47

Index

corpora, 5
corpus
 aligné, 89
 de développement, 17
Corrado, Greg, 45
corrélation croisée, 47
couche, 8, 36
 de plongement, 67
 entièrement connectée,
 44
couleur RVB, 48, 52
Courville, Aaron, 25
critique, 121
cross-entropy, voir entropie
 croisée
cross-entropy loss, voir perte
 d'entropie croisée

D

Dean, Jeff, 45
décoder, 91
décodeur, 128
deep learning, 1
Deep Q Network, 124
deep-Q learning, voir
 apprentissage par
 renforcement profond
DeepMind, 112
délimitation
 de phrases, 78
dendrite, 3
décision multi-classes, 7
dérivation
 en chaîne, 14
 fonctions composées,
 14
descente de gradient, 9, 13,
 16, 120
 ordinaire, 16
 stochastique, 16
dictionnaires Python, 28
diffusion, 60
dimension
 réduction, 127, 139

discounted future reward, voir
 récompense future avec
 rabais
discrétisation, 1
discriminateur
 GAN, 142
disparition du gradient, 37
DistBelief, 45
distribution
 de probabilité, 11, 65
 gaussienne, 30
 normale, 30, 140–142
 normale standard, 138,
 141
divergence de
 Kullback-Leibler, 141
Doersch, Carl, 147
données
 normalisation, 18
downsampling, voir encodage
DQN, 124

E

écart type, 30, 42
embedding_lookup, 70
encoder, 91
encodeur, 128
entraînement
 en parallèle, 22
 exemple, 5
entropie croisée, 10, 12
 perte, 11
environnement, 105
environnements multiples,
 124
epoch, voir époque
époque, 7, 101
epsilon-décroissant, 110
epsilon-gourmand, 110
equal, 34
erreur de différence
 temporelle, 114
espérance, 106
 mathématique, 43

estimateur
 vraisemblance, 66
état de cellule, 83
étiquette, 3
exactitude, 17, 19
experience replay, voir
 répétition d'expériences
exploration-exploitation
 tradeoff, voir compromis
 exploration-exploitation

F

Facebook, 45
feed-forward, voir
 propagation avant
feed_dict, 28, 82, 139
fenêtre
 taille, 77, 92
filtrage logiciel, 84
filtre, 48
 de convolution, 47
fonction
 approximation, 5
 de perte, 9
 de perte universelle, 142
 de valeur, v, 106
 noyau, 48
 Q, 106
 sigmoïde, 37, 42, 84,
 129, 143
 soft, 11, 84
fonction d'activation, 37, 44,
 83
 tanh, 84
fonctions composées
 dérivation, 14
forme d'un tenseur, 28
fully_connected, 44

G

GAN, 127, 142, 147
Gated Recurrent Unit, voir
 réseau à portes récurrent

- gather, 120
 générateur
 GAN, 142
 générateur aléatoire
 germe, 17
 germe, 17
 Géron, Aurélien, 25
`global_variable_initializer`, 30
 Glover, John, 147
 Goodfellow, Ian, 25, 146, 147
 Google, 27, 62, 112
 Google Brain, 45
 GPU, voir processeur graphique
 gradient, 21
 de politique, 115
 descente, 9, 13, 16
 disparition, 37
`GradientDescentOptimizer`, 30
 greedy algorithm, voir algorithme gourmand
 GRU, 90
`gym.make`, 109
- ## H
- heuristique
 définition, 1
 Hinton, Geoffrey, 24, 45, 62
 Hochreiter, Sepp, 86
 hyperparamètre, 6, 67, 76, 77, 110
 hypothèse de Markov, 105
- ## I
- IBM, 103
 iid, 23
 ILSVRC, 62
 Imagenet Large Scale Visual Recognition Challenge,
 voir ILSVRC
 initialisation
 de variable, 30
- de Xavier, 42, 44
 des poids, 17
 instabilité, 23
 Institut canadien de recherche avancée, 62
 intelligence artificielle, 1, 62
 intensité lumineuse, 1
 Isola, Phillip, 147
 itération d'entraînement, voir
 époque
 itération sur les valeurs, 106, 122
- ## J
- Jelinek, Fred, 103
 jeu
 d'entraînement, 5
 de développement, 5
 de réserve, 5
 de test, 5
 de validation, 5
 jeux Atari, 112
- ## K
- Kaelbling, Leslie, 125
 Kalchbrenner, Nal, 103
 Keras, 45
 kernel, voir noyau
 Kingma, Diederik, 147
 Krizhevsky, Alex, 62
 Kurenkov, Andrey, 24
- ## L
- `l2_loss`, 75
 langage cible, 89
 langage source, 89
`layers`, 44, 60, 130
 leaky relu, 37
 learning rate, voir taux d'apprentissage
 LeCun, Yann, 61, 146
 LGU, 87
- linear gated unit, voir LGU
`log`, dans TF, 32
 logarithme naturel, 32
 logit, 11, 14, 21
 loi de Newton, 115
 long short-term memory, voir LSTM
 longue mémoire à court terme, voir LSTM
 lot
 taille, 16, 22
 LSTM, 82, 90, 146
 Luong, Thad, 103
- ## M
- machine learning, voir apprentissage automatique
 machine translation, voir traduction automatique
 Markov, 105
 Markov decision process, voir MDP
`matmul`, 31, 32, 40, 113
 matrices, 20
 opérations, 20
 produit, 82
 transposition, 21
`max_pool`, 60, 132
 maximum de vraisemblance
 estimateur, 66
 McCulloch, Warren B., 24
 MDP, 105
 méthode
 acteur, 121
 acteur-critique, 121
 Adam, 120
 MDP tabulaire, 106
 variationnelle, 134
 Mikolov, Thomas, 86
 Miller, Steven, 25
 Mnist, 1, 112, 127
 model-free learning, voir apprentissage sans modèle

Index

modèle

- bigramme, 67
 - linguistique, 65
 - skip-gram, 86
 - trigramme, 72, 87
- modélisation linguistique, 89
- Mohr, Felix, 147
- momentum, voir vitesse
- mot inconnu, 66
- MT, 89

N

National Institute of

Standards, 1

neural machine translation,
103

neural network, voir réseau de
neurones

neurone, 3

Ng, Andrew, 45

NIST, 1

NN, voir réseau de neurones

nœud de substitution, 28, 82

normalisation de données, 18,
130

noyau, 48

biais, 59

de convolution, 47

numérotation des
composantes d'un tenseur,
29

Numpy, 22

O

one-hot, 32

one-hot, 112

Open AI Gym, 107, 109, 127

opérateur gradient, 21

optimisation, 21

optimiseur

Adam, 120, 146

overfitting, voir surajustement

P

padding, voir complétion, voir
remplissage

paramètre, 3

pas, 50

passee

avant, 13

d'encodage, 91

de décodage, 91

passee arrière, 13

représentation

matricielle, 21

PDP, 24

Penn Treebank Corpus, 66

perceptron, 3, 24

algorithme, 5

multi-classe, 8

multicouche, 9

paramètres, 3

perplexité, 71, 82

perte, 9

d'image, 137

quadratique, 26, 114,

123, 129, 135, 138

représentation

matricielle, 21

variationnelle, 137, 140

zéro-un, 9

perte d'entropie croisée, 10,

91, 143

définition, 11

Pitts, Walter, 24

pixel, 1

valeur, 1, 18

placeholder, voir nœud de

substitution

placeholder, 28

plongement de mot, 67, 86, 91

taille, 85

plongement de phrase, 91

poids, 3, 13, 16

initialisation, 17

point de contrôle, 39

politique, 105

portée, 92

position-only attention, voir
attention à la position
uniquement

pré-apprentissage, 128

probabilité

distribution, 65

processeur graphique, 22

processus décisionnel de
Markov, 105, 112

produit

de matrices, 20

scalaire, 5

propagation avant, 9

PTB, 66

Python, 27

Pytorch, 45

Q

Q-learning, 110

quadratic loss, voir perte
quadratique

R

rabais, 105

Ram, Surlyadeepan, 103

randint, 109

random_normal, 30

range, 126

récompense, 105

future avec rabais, 105

reconstruction, 128

rectified linear unit, voir relu

recurrent neural network, voir
réseau de neurones

récurrent

reduce_mean, 30

réduction de dimension, 127,
139

régularisation, 74

L2, 74, 87

REINFORCE, 117, 125

- reinforcement learning, voir
apprentissage par
renforcement
- relu, 143
- relu, 36, 44
- remplissage
de marge, voir
complétion
de phrase, 67
- répétition d'expériences, 124
- réseau à portes récurrent, 90
- réseau de neurones, 8
à propagation avant, 9
antagoniste génératif,
127, 142
convolutif, 47
entièlement connecté, 47
historique, 24
partiellement connecté,
47
récurrent, 76, 82
représentation
matricielle, 20
taille, 80, 85
- reset dans AI Gym, 109
- reshape, 54
dans Numpy, 54
- reshape, 81
- résolution des MDP, 105
- rétropropagation, 14, 24
à travers le temps, 77,
91, 104
- reuse, 146
- RNN, voir réseau de neurones
récurrent
- Rosenblatt, Frank, 24
- Ruder, Sebastian, 86
- Rumelhart, David, 24
- run, 27
- RVB, voir couleur
- S**
- save, 39
- Saver, 39
- Schmidhuber, Jürgen, 86
- scope, voir portée
- seed, voir germe
- semi-supervisé, 3
- sentence padding, voir
remplissage de phrase
- seq2seq, 89
- seq2seq_loss, 93
- Session, 27
- sigma, 30
- sigmoïde, 37
- signal, 130
- similarité cosinus, 68, 102
- soft gating, voir filtrage
logiciel
- softmax, 11, 14, 95, 130
- somme de matrices, 20
- sous-échantillonnage, 128
- sparse_softmax_
cross_entropy,
70, 81
- squared-error loss, voir perte
quadratique
- step de AI Gym, 109
- STOP de remplissage,
92, 102
- stride, voir pas
- structure grammaticale,
66
- surajustement, 74
- Sutskever, Ilya, 62
- Sutton, Richard, 125
- T**
- taille
cachée, 38
de lot, 16, 22
plongement de mot, 85
- tangente hyperbolique, 84
- taux d'apprentissage, 9, 16,
18, 19, 33, 38, 95
- temporal difference error, voir
erreur de différence
temporelle
- tenseur, 28
forme, 28
- numérotation des
composantes, 29
- tensordot, 40, 82
- TensorFlow, 22, 27
- terme à terme
opération, 84
- TF, 27
- théorie de la communication,
91
- token, voir unité lexicale
- top-5 score, 62
- Torch, 45
- traduction automatique, 89
- trainable_variables,
146
- traitement parallèle distribué,
24
- transpose, 97
- treebank, 66
- type de marge, 50
- U**
- underscore, en Python, 35
- unité
à portes linéaire, voir
LGU
lexicale, 65, 89
linéaire, 5, 8, 11
- Université de Californie,
45
- Université de Toronto, 45
- upsampling, voir
reconstruction
- V**
- VAE, 134
- Variable, 29
- variable aléatoire, 65
- variable normale
écart type, 30
moyenne, 30

Index

variable_scope, 92,
146
variance, 43, 122
vecteur one-hot, 32, 71
véhicule autonome, 124
vitesse d'optimisation,
120

vocabulaire, 66
vraisemblance, 66
W
Welling, Max, 147
Wikédia, 14
Williams, Ronald, 24, 125

word embedding, voir
plongement de mot
word2vec, 86

X

Xavier
initialisation, 42