

O'REILLY®

Deep Learning avec Keras et TensorFlow

Mise en œuvre et cas concrets

2^e édition

Aurélien Géron

Traduit de l'anglais par Hervé Soulard

DUNOD

Authorized French translation of material from the English edition of
Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2E
ISBN 9781492032649
© 2019 Aurélien Géron.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

Conception de la couverture : Karen Montgomery
Illustratrice : Rebecca Demarest

© Dunod, 2017, 2020
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-080502-0

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. – Les fondamentaux du Machine Learning	1
1.1 Installer le projet handson-ml2	2
1.2 Installer Tensorflow et Jupyter	3
1.3 Qu'est-ce que le Machine Learning?	6
1.4 Comment le système apprend-il?	8
1.5 Régression linéaire	8
1.6 Descente de gradient	14
1.7 Régression polynomiale	24
1.8 Courbes d'apprentissage	26
1.9 Modèles linéaires régularisés	30
1.10 Régression logistique	38
1.11 Exercices	46
Chapitre 2. – Introduction aux réseaux de neurones artificiels avec Keras	49
2.1 Du biologique à l'artificiel	50
2.2 Implémenter des MPC avec Keras	65
2.3 Régler précisément les hyperparamètres d'un réseau de neurones	89
2.4 Exercices	97
Chapitre 3. – Entraînement de réseaux de neurones profonds	101
3.1 Problèmes de disparition et d'explosion des gradients	102
3.2 Réutiliser des couches préentraînées	116

3.3 Optimiseurs plus rapides	122
3.4 Éviter le surajustement grâce à la régularisation	134
3.5 Résumé et conseils pratiques	142
3.6 Exercices	143
Chapitre 4. – Modèles personnalisés et entraînement avec TensorFlow	145
4.1 Présentation rapide de TensorFlow	146
4.2 Utiliser TensorFlow comme NumPy	149
4.3 Personnaliser des modèles et entraîner des algorithmes	154
4.4 Fonctions et graphes TensorFlow	175
4.5 Exercices	179
Chapitre 5. – Chargement et prétraitement de données avec TensorFlow	181
5.1 L'API Data	182
5.2 Le format TFRecord	192
5.3 Prétraiter les caractéristiques d'entrée	198
5.4 TF Transform	207
5.5 Le projet TensorFlow Datasets (TFDS)	208
5.6 Exercices	210
Chapitre 6. – Vision par ordinateur et réseaux de neurones convolutifs	213
6.1 L'architecture du cortex visuel	214
6.2 Couches de convolution	215
6.3 Couche de pooling	224
6.4 Architectures de CNN	228
6.5 Implémenter un CNN ResNet-34 avec Keras	244
6.6 Utiliser des modèles préentraînés de Keras	246
6.7 Modèles préentraînés pour un transfert d'apprentissage	247
6.8 Classification et localisation	250
6.9 Détection d'objets	252
6.10 Segmentation sémantique	259
6.11 Exercices	262

Chapitre 7. – Traitement des séries avec des RNR et des CNN	265
7.1 Neurones et couches récurrents.....	266
7.2 Entraîner des RNR.....	270
7.3 Prévoir des séries chronologiques	271
7.4 Traiter les longues séries.....	279
7.5 Exercices	290
Chapitre 8. – Traitement automatique du langage naturel avec les RNR et les attentions	291
8.1 Générer un texte shakespearien à l'aide d'un RNR à caractères.....	292
8.2 Analyse de sentiments.....	301
8.3 Réseau encodeur-décodeur pour la traduction automatique neuronale	308
8.4 Mécanismes d'attention.....	314
8.5 Innovations récentes dans les modèles de langage	327
8.6 Exercices	329
Chapitre 9. – Apprentissage de représentations et apprentissage génératif avec des autoencodeurs et des GAN	331
9.1 Représentations efficaces des données.....	333
9.2 ACP avec un autoencodeur linéaire sous-complet	334
9.3 Autoencodeurs empilés	336
9.4 Autoencodeurs convolutifs	343
9.5 Autoencodeurs récurrents	344
9.6 Autoencodeurs débruiteurs	345
9.7 Autoencodeurs épars	346
9.8 Autoencodeurs variationnels.....	349
9.9 Réseaux antagonistes génératifs (GAN).....	355
9.10 Exercices	369
Chapitre 10. – Apprentissage par renforcement	371
10.1 Apprendre à optimiser les récompenses	372
10.2 Recherche de politique	374
10.3 Introduction à OpenAI Gym	375

10.4 Politiques par réseau de neurones	378
10.5 Évaluer des actions : le problème d'affectation de crédit	380
10.6 Gradients de politique	382
10.7 Processus de décision markoviens	386
10.8 Apprentissage par différence temporelle	390
10.9 Apprentissage Q	391
10.10 Implémenter l'apprentissage Q profond	394
10.11 Variantes de l'apprentissage Q profond	399
10.12 La bibliothèque TF-Agents	402
10.13 Quelques algorithmes RL populaires	422
10.14 Exercices	424
Chapitre 11. – Entraînement et déploiement à grande échelle de modèles TensorFlow	427
11.1 Servir un modèle TensorFlow	428
11.2 Déployer un modèle sur un périphérique mobile ou embarqué	444
11.3 Utiliser des GPU pour accélérer les calculs	448
11.4 Entraîner des modèles sur plusieurs processeurs	460
11.5 Exercices	475
Le mot de la fin	477
Annexe A. – Solutions des exercices	479
Annexe B. – Différentiation automatique	511
Annexe C. – Autres architectures de RNA répandues	519
Annexe D. – Structures de données spéciales	529
Annexe E. – Graphes TensorFlow	537
Index	547

Avant-propos

L'intelligence artificielle en pleine explosion

Auriez-vous cru, il y a seulement 10 ans, que vous pourriez aujourd’hui poser toutes sortes de questions à voix haute à votre téléphone, et qu'il réponde correctement ? Que des voitures autonomes sillonnaient déjà les rues (surtout américaines, pour l'instant) ? Qu'un logiciel, AlphaGo, parviendrait à vaincre Ke Jie, le champion du monde du jeu de go, alors que, jusqu'alors, aucune machine n'était jamais arrivée à la cheville d'un grand maître de ce jeu ? Que chaque jour vous utiliseriez des dizaines d'applications intelligentes, des outils de recherche à la traduction automatique en passant par les systèmes de recommandations ?

Au rythme où vont les choses, on peut se demander ce qui sera possible dans 10 ans ! Les docteurs feront-ils appel à des intelligences artificielles (IA) pour les assister dans leurs diagnostics ? Les jeunes écouteront-ils des tubes personnalisés, composés spécialement pour eux par des machines analysant leurs habitudes, leurs goûts et leurs réactions ? Des robots pleins d'empathie tiendront-ils compagnie aux personnes âgées ? Quels sont vos pronostics ? Notez-les bien et rendez-vous dans 10 ans ! Une chose est sûre : le monde ressemble de plus en plus à un roman de science-fiction.

L'apprentissage automatique se démocratise

Au cœur de ces avancées extraordinaires se trouve le Machine Learning (ML, ou *apprentissage automatique*) : des systèmes informatiques capables d'apprendre à partir d'exemples. Bien que le ML existe depuis plus de 50 ans, il n'a véritablement pris son envol que depuis une dizaine d'années, d'abord dans les laboratoires de recherche, puis très vite chez les géants du web, notamment les GAFA (Google, Apple, Facebook et Amazon).

À présent, le Machine Learning envahit les entreprises de toutes tailles. Il les aide à analyser des volumes importants de données et à en extraire les informations les plus utiles (*data mining*). Il peut aussi détecter automatiquement les anomalies de production, repérer les tentatives de fraude, segmenter une base de clients afin de mieux cibler les offres, prévoir les ventes (ou toute autre série temporelle), classer automatiquement les prospects à appeler en priorité, optimiser le nombre de conseillers de

clientèle en fonction de la date, de l'heure et de mille autres paramètres, etc. La liste d'applications s'agrandit de jour en jour.

Cette diffusion rapide du Machine Learning est rendue possible en particulier par trois facteurs :

- Les entreprises sont pour la plupart passées au numérique depuis longtemps : elles ont ainsi des masses de données facilement disponibles, à la fois en interne et *via Internet*.
- La puissance de calcul considérable nécessaire pour l'apprentissage automatique est désormais à la portée de tous les budgets, en partie grâce à la loi de Moore¹, et en partie grâce à l'industrie du jeu vidéo : en effet, grâce à la production de masse de cartes graphiques puissantes, on peut aujourd'hui acheter pour un prix d'environ 1000 € une carte graphique équipée d'un processeur GPU capable de réaliser des milliers de milliards de calculs par seconde². En l'an 2000, le superordinateur ASCI White d'IBM avait déjà une puissance comparable... mais il avait coûté 110 millions de dollars ! Et bien sûr, si vous ne souhaitez pas investir dans du matériel, vous pouvez facilement louer des machines virtuelles dans le cloud.
- Enfin, grâce à l'ouverture grandissante de la communauté scientifique, toutes les découvertes sont disponibles quasi instantanément pour le monde entier, notamment sur <https://arxiv.org>. Dans combien d'autres domaines peut-on voir une idée scientifique publiée puis utilisée massivement en entreprise la même année ? À cela s'ajoute une ouverture comparable chez les GAFA : chacun s'efforce de devancer l'autre en matière de publication de logiciels libres, en partie pour dorer son image de marque, en partie pour que ses outils dominent et que ses solutions de cloud soient ainsi préférées, et, qui sait, peut-être aussi par altruisme (il n'est pas interdit de rêver). Il y a donc pléthore de logiciels libres d'excellente qualité pour le Machine Learning.

Dans ce livre, nous utiliserons TensorFlow, développé par Google et rendu open source fin 2015. Il s'agit d'un outil capable d'exécuter toutes sortes de calculs de façon distribuée, et particulièrement optimisé pour entraîner et exécuter des réseaux de neurones artificiels. Comme nous le verrons, TensorFlow contient notamment une excellente implémentation de l'API Keras, qui simplifie grandement la création et l'entraînement de réseaux de neurones artificiels.

L'avènement des réseaux de neurones

Le Machine Learning repose sur un grand nombre d'outils, provenant de plusieurs domaines de recherche : notamment la théorie de l'optimisation, les statistiques, l'algèbre linéaire, la robotique, la génétique et bien sûr les neurosciences. Ces dernières ont inspiré les réseaux de neurones artificiels (RNA), des modèles simplifiés des réseaux de neurones biologiques qui composent votre cortex cérébral : c'était en

1. Une loi vérifiée empiriquement depuis 50 ans et qui affirme que la puissance de calcul des processeurs double environ tous les 18 mois.

2. Par exemple, 11,8 téraFLOPS pour la carte GeForce GTX 1080 Ti de NVidia. Un téraFLOPS égale mille milliards de FLOPS. Un FLOPS est une opération à virgule flottante par seconde.

1943, il y a plus de 70 ans ! Après quelques années de tâtonnements, les chercheurs sont parvenus à leur faire apprendre diverses tâches, notamment de classification ou de régression (c'est-à-dire prévoir une valeur en fonction de plusieurs paramètres). Malheureusement, lorsqu'ils n'étaient composés que de quelques couches successives de neurones, les RNA ne semblaient capables d'apprendre que des tâches rudimentaires. Et lorsque l'on tentait de rajouter davantage de couches de neurones, on se heurtait à des problèmes en apparence insurmontables : d'une part, ces réseaux de neurones « profonds » exigeaient une puissance de calcul rédhibitoire pour l'époque, des quantités faramineuses de données, et surtout, ils s'arrêtaient obstinément d'apprendre après seulement quelques heures d'entraînement, sans que l'on sache pourquoi. Dépités, la plupart des chercheurs ont abandonné le *connexionisme*, c'est-à-dire l'étude des réseaux de neurones, et se sont tournés vers d'autres techniques d'apprentissage automatique qui semblaient plus prometteuses, telles que les arbres de décision ou les machines à vecteurs de support (SVM).

Seuls quelques chercheurs particulièrement déterminés ont poursuivi leurs recherches : à la fin des années 1990, l'équipe de Yann Le Cun est parvenue à créer un réseau de neurones à convolution (CNN, ou ConvNet) capable d'apprendre à classer très efficacement des images de caractères manuscrits. Mais chat échaudé craint l'eau froide : il en fallait davantage pour que les réseaux de neurones ne reviennent en odeur de sainteté.

Enfin, une véritable révolution eut lieu en 2006 : Geoffrey Hinton et son équipe mirent au point une technique capable d'entraîner des réseaux de neurones profonds, et ils montrèrent que ceux-ci pouvaient apprendre à réaliser toutes sortes de tâches, bien au-delà de la classification d'images. L'apprentissage profond, ou *Deep Learning*, était né. Suite à cela, les progrès sont allés très vite, et, comme vous le verrez, la plupart des articles de recherche cités dans ce livre datent d'après 2010.

Objectif et approche

Pourquoi ce livre ? Quand je me suis mis au Machine Learning, j'ai trouvé plusieurs livres excellents, de même que des cours en ligne, des vidéos, des blogs, et bien d'autres ressources de grande qualité, mais j'ai été un peu frustré par le fait que le contenu était d'une part complètement éparsillé, et d'autre part généralement très théorique, et il était souvent très difficile de passer de la théorie à la pratique.

J'ai donc décidé d'écrire le livre *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (ou HOML), avec pour objectif de couvrir les principaux domaines du Machine Learning, des simples modèles linéaires aux SVM en passant par les arbres de décision et les forêts aléatoires, et bien sûr aussi le Deep Learning et même l'apprentissage par renforcement (Reinforcement Learning, ou RL). Je voulais que le livre soit utile à n'importe quelle personne ayant un minimum d'expérience de programmation (si possible en Python³), et axer l'apprentissage autour de la pratique, avec

3. J'ai choisi le langage Python d'une part parce que c'est mon langage de prédilection, mais aussi parce qu'il est simple et concis, ce qui permet de remplir le livre de nombreux exemples de code. En outre, il s'agit actuellement du langage le plus utilisé en Machine Learning (avec le langage R).

de nombreux exemples de code. Vous retrouverez ainsi tous les exemples de code de ce livre sur <https://github.com/ageron/handson-ml2>, sous la forme de notebooks Jupyter.

Notes sur l'édition française

La première moitié de HOML est une introduction au Machine Learning, reposant sur la librairie Scikit-Learn⁴. La seconde moitié est une introduction au Deep Learning, reposant sur la librairie TensorFlow. Dans l'édition française, ce livre a été scindé en deux :

- la première partie (chapitres 1 à 9) a été traduite dans le livre *Machine Learning avec Scikit-Learn*, aux éditions Dunod (2^e édition, 2019) ;
- la seconde partie (chapitres 10 à 19) a été traduite dans le livre que vous tenez entre les mains, *Deep Learning avec Keras et TensorFlow*. Les chapitres ont été renombrés de 2 à 11, et un nouveau chapitre 1 a été ajouté, reprenant les points essentiels de la première partie.

Prérequis

Bien que ce livre ait été écrit plus particulièrement pour les ingénieurs en informatique, il peut aussi intéresser toute personne sachant programmer et ayant quelques bases mathématiques. Il ne requiert aucune connaissance préalable sur le Machine Learning mais il suppose les prérequis suivants :

- vous devez avoir un minimum d'expérience de programmation ;
- sans forcément être un expert, vous devez connaître le langage Python, et si possible également ses librairies scientifiques, en particulier NumPy, pandas et Matplotlib ;
- enfin, si vous voulez comprendre comment les algorithmes fonctionnent (ce qui n'est pas forcément indispensable, mais est tout de même très recommandé), vous devez avoir certaines bases en mathématiques dans les domaines suivants :
 - l'algèbre linéaire, notamment comprendre les vecteurs et les matrices (par exemple comment multiplier deux matrices, transposer ou inverser une matrice),
 - le calcul différentiel, notamment comprendre la notion de dérivée, de dérivée partielle, et savoir comment calculer la dérivée d'une fonction.

Si vous ne connaissez pas encore Python, il existe de nombreux tutoriels sur Internet, que je vous encourage à suivre : ce langage est très simple et s'apprend vite. En ce qui concerne les librairies scientifiques de Python et les bases mathématiques requises, le site github.com/ageron/handson-ml2 propose quelques tutoriels (en anglais) sous la forme de notebooks Jupyter. De nombreux tutoriels en français sont disponibles sur Internet. Le site fr.khanacademy.org est particulièrement recommandé pour les mathématiques.

4. Cette librairie a été créée par David Cournapeau en 2007, et le projet est maintenant dirigé par une équipe de chercheurs à l'Institut national de recherche en informatique et en automatique (Inria).

Plan du livre

- Le chapitre 1 reprend les éléments du livre *Machine Learning avec Scikit-Learn* qui sont indispensables pour comprendre le Deep Learning. Il montre d'abord comment installer TensorFlow et le projet contenant les exemples de code du livre (ainsi que les librairies dont il dépend), puis il présente les bases du Machine Learning, comment entraîner divers modèles linéaires à l'aide de la descente de gradient, pour des tâches de régression et de classification, et il présente quelques techniques de régularisation.
- Le chapitre 2 introduit les réseaux de neurones artificiels et montre comment les mettre en œuvre avec Keras.
- Le chapitre 3 montre comment résoudre les difficultés particulières que l'on rencontre avec les réseaux de neurones profonds.
- Le chapitre 4 présente l'API de bas niveau de TensorFlow, utile lorsque l'on souhaite personnaliser les rouages internes des réseaux de neurones.
- Le chapitre 5 montre comment charger et transformer efficacement de gros volumes de données lors de l'entraînement d'un réseau de neurones artificiels.
- Le chapitre 6 présente les réseaux de neurones convolutifs et leur utilisation pour la vision par ordinateur.
- Le chapitre 7 montre comment analyser des séries temporelles à l'aide de réseaux de neurones récurrents, ou avec des réseaux de neurones convolutifs.
- Le chapitre 8 présente le traitement automatique du langage naturel à l'aide de réseaux de neurones récurrents, ou de réseaux de neurones dotés de mécanismes d'attention.
- Le chapitre 9 traite de l'apprentissage automatique de représentations à l'aide d'autoencodeurs ou de réseaux antagonistes génératifs (GAN). L'objectif est de découvrir, avec ou sans supervision, des motifs dans les données. Ces architectures de réseaux de neurones artificiels sont également utiles pour générer de nouvelles données semblables à celles reçues en exemple (par exemple pour générer des images de visages).
- Le chapitre 10 aborde l'apprentissage par renforcement, dans lequel un agent apprend par tâtonnements au sein d'un environnement dans lequel il peut recevoir des récompenses ou des punitions. Nous construirons en particulier un agent capable d'apprendre à jouer tout seul au jeu Atari *Breakout*.
- Le chapitre 11 présente comment entraîner et déployer à grande échelle les réseaux de neurones artificiels construits avec TensorFlow.

Conventions

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique un nouveau terme, une URL, une adresse email ou un nom de fichier.

Largeur fixe

Utilisée pour les exemples de code, ainsi qu'au sein du texte pour faire référence aux éléments d'un programme, tels que des instructions, des mots clés, des noms de variables, de fonctions, de base de données, de types de données ou encore de variables d'environnement.

Largeur fixe et gras

Affiche des commandes ou d'autres textes qui doivent être saisis littéralement par l'utilisateur.

Largeur fixe et italique

Affiche du texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.



Ce symbole indique une astuce ou une suggestion.



Ce symbole indique une précision ou une remarque générale.



Ce symbole indique une difficulté particulière ou un piège à éviter.

Remerciements

Jamais, dans mes rêves les plus fous, je n'aurais imaginé que la première édition de ce livre renconterait un public aussi vaste. J'ai reçu de nombreux messages de lecteurs, avec beaucoup de questions, certains signalant gentiment des erreurs et la plupart m'envoyant des mots encourageants. Je suis extrêmement reconnaissant envers tous ces lecteurs pour leur formidable soutien. Merci beaucoup à vous tous ! N'hésitez pas à me contacter si vous voyez des erreurs dans les exemples de code ou simplement pour poser des questions (<https://homl.info/issues2>) ! Certains lecteurs ont également expliqué en quoi ce livre les avait aidés à obtenir leur premier emploi ou à résoudre un problème concret sur lequel ils travaillaient. Ces retours sont incroyablement motivants. Si vous trouvez ce livre utile, j'aimerais beaucoup que vous puissiez partager votre histoire avec moi, que ce soit en privé (par exemple, via <https://www.linkedin.com/in/aurelien-geron/>) ou en public (par exemple dans un tweet ou par le biais d'un commentaire Amazon).

Je suis également extrêmement reconnaissant envers toutes les personnes qui ont pris le temps d'examiner mon livre avec autant de soin. En particulier, je voudrais remercier François Chollet, auteur de Keras, d'avoir passé en revue tous les chapitres

basés sur Keras et TensorFlow et de m'avoir fait des commentaires si approfondis. Comme Keras est l'un des ajouts principaux à cette deuxième édition, sa relecture critique a été inestimable. Je recommande fortement son livre⁵: il a la concision, la clarté et la profondeur de la bibliothèque Keras elle-même. Merci également à Ankur Patel, qui a relu chaque chapitre de cette deuxième édition et m'a fourni un excellent retour d'informations. Un grand merci à Olzhas Akpambetov, qui a examiné tous les chapitres, testé une bonne partie du code et proposé de nombreuses suggestions. Je suis reconnaissant envers Mark Daoust, Jon Krohn, Dominic Monn et Josh Patterson d'avoir relu ce livre de manière aussi approfondie et d'avoir offert leur expertise. Ils n'ont rien négligé et ont fourni des commentaires extrêmement utiles.

Lors de la rédaction de cette deuxième édition, j'ai eu la chance d'obtenir l'aide de nombreux membres de l'équipe de TensorFlow, en particulier Martin Wicke, qui a inlassablement répondu à des dizaines de mes questions et a envoyé le reste aux bonnes personnes, notamment Karmel Allison, Paige Bailey, Eugene Brevdo, William Chargin, Daniel « Wolff » Dobson, Nick Felt, Bruce Fontaine, Goldie Gadde, Sandeep Gupta, Priya Gupta, Kevin Haas, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Allen Lavoie, Clemens Mewald, Dan Moldovan, Sean Morgan, Tom O'Malley, Alexandre Passos, André Susano Pinto, Anthony Platanios, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Ryan Sepassi, Jiri Simska, Xiaodan Song, Christina Sorokin, Dustin Tran, Todd Wang, Pete Warden (qui a également révisé la première édition), Edd Wilder-James et Yuefeng Zhou, qui ont tous été extrêmement utiles. Un grand merci à vous tous et à tous les autres membres de l'équipe TensorFlow, non seulement pour votre aide, mais également pour avoir créé une si belle bibliothèque ! Je remercie tout particulièrement Irene Giannoumis et Robert Crowe de l'équipe TFX d'avoir révisé les chapitres 5 et 11 en profondeur.

Un grand merci au personnel fantastique d'O'Reilly, en particulier Nicole Taché, qui m'a fait des commentaires perspicaces, toujours encourageants et utiles: je ne pouvais pas rêver d'un meilleur éditeur. Merci également à Michele Cronin, qui a été très efficace (et patiente) au début de cette deuxième édition, et à Kristen Brown, responsable de la production pour la deuxième édition, qui a suivi toutes les étapes (elle a également coordonné les correctifs et les mises à jour pour chaque réimpression de la première édition). Merci également à Rachel Monaghan et à Amanda Kersey pour leur révision complète (respectivement pour les première et deuxième éditions), et à Johnny O'Toole qui a géré la relation avec Amazon et répondu à beaucoup de mes questions. Merci à Marie Beaugureau, Ben Lorica, Mike Loukides et Laurel Ruma d'avoir cru en ce projet et de m'avoir aidé à le définir. Merci à Matt Hacker et à toute l'équipe d'Atlas pour avoir répondu à toutes mes questions techniques concernant AsciiDoc et LaTeX, ainsi qu'à Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis et tous les autres membres d'O'Reilly qui ont contribué à la rédaction de ce livre.

Je tiens également à remercier mes anciens collègues de Google, en particulier l'équipe de classification des vidéos YouTube, de m'avoir énormément appris à propos

5. François Chollet, *Deep Learning with Python* (Manning, 2017): <https://hml.info/cholletbook>.

du Machine Learning. Je n'aurais jamais pu commencer la première édition sans eux. Un merci spécial à mes gourous personnels du ML: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn et Rich Washington. Et merci à tous ceux avec qui j'ai travaillé chez YouTube et dans les formidables équipes de recherche Google de Mountain View. Grand merci également à Martin Andrews, Sam Witteveen et Jason Zaman de m'avoir accueilli dans leur groupe d'experts en développement Google à Singapour et pour toutes les excellentes discussions que nous avons eues sur le Deep Learning et TensorFlow. Toute personne intéressée par le Deep Learning à Singapour devrait absolument rejoindre son meet-up (<https://homl.info/meetupsg>). Jason mérite des remerciements spéciaux pour avoir partagé une partie de son expertise de TFLite pour le chapitre 11 !

Je n'oublierai jamais les personnes qui ont aimablement relu la première édition de ce livre, notamment David Andrzejewski, Lukas Biewald, Justin Francis, Vincent Guilbeau, Eddy Hung, Karim Matrah, Grégoire Mesnil, Salim Sémaoune, Iain Smears, Michel Tessier, Ingrid von Glehn, Pete Warden et bien sûr mon cher frère Sylvain. Un merci tout spécial à Haesun Park, qui m'a fourni d'excellents commentaires et a relevé plusieurs erreurs en écrivant la traduction coréenne de la première édition de ce livre. Il a également traduit les notebooks Jupyter en coréen, sans parler de la documentation de TensorFlow. Je ne parle pas coréen, mais à en juger par la qualité de ses retours, toutes ses traductions doivent être vraiment excellentes ! Il a en outre gentiment contribué à certaines des solutions des exercices de cette deuxième édition.

Je souhaite également remercier Jean-Luc Blanc, des éditions Dunod, pour avoir soutenu et géré ce projet, et pour ses relectures attentives. Je tiens aussi à remercier vivement Hervé Soulard pour sa traduction. Enfin, je remercie chaleureusement Brice Martin, des éditions Dunod, pour sa relecture extrêmement rigoureuse, ses excellentes suggestions et ses nombreuses corrections.

Pour finir, je suis infiniment reconnaissant envers ma merveilleuse épouse, Emmanuelle, et à nos trois enfants, Alexandre, Rémi et Gabrielle, de m'avoir encouragé à travailler dur pour ce livre. Je les remercie également pour leur curiosité insatiable : expliquer certains des concepts les plus difficiles de ce livre à ma femme et à mes enfants m'a aidé à clarifier mes pensées et à améliorer directement de nombreuses parties de ce livre. J'ai même eu droit à des biscuits et du café, comment rêver mieux ?

1

Les fondamentaux du Machine Learning

Avant de partir à l'assaut du mont Blanc, il faut être entraîné et bien équipé. De même, avant d'attaquer le Deep Learning avec TensorFlow et Keras, il est indispensable de maîtriser les bases du Machine Learning. Si vous avez lu le livre *Machine Learning avec Scikit-Learn* (A. Géron, Dunod, 2^e édition, 2019), vous êtes prêt(e) à passer directement au chapitre 2. Dans le cas contraire, ce chapitre vous donnera les bases indispensables pour la suite⁶.

Nous commencerons par installer TensorFlow et les autres bibliothèques Python dont vous aurez besoin pour exécuter les nombreux exemples de code. Dans ce premier chapitre nous utiliserons uniquement NumPy, Matplotlib et Scikit-Learn : nous attacherons TensorFlow à partir du prochain chapitre.

Ensuite, nous étudierons la régression linéaire, l'une des techniques d'apprentissage automatique les plus simples qui soient. Cela nous permettra au passage de rappeler ce qu'est le Machine Learning, ainsi que le vocabulaire et les notations que nous emploierons tout au long de ce livre. Nous verrons deux façons très différentes d'entraîner un modèle de régression linéaire : premièrement, une méthode analytique qui trouve directement le modèle optimal (c'est-à-dire celui qui s'ajuste au mieux au jeu de données d'entraînement) ; deuxièmement, une méthode d'optimisation itérative appelée *descente de gradient* (en anglais, *gradient descent* ou GD), qui consiste à modifier graduellement les paramètres du modèle de façon à l'ajuster petit à petit au jeu de données d'entraînement.

Nous examinerons plusieurs variantes de cette méthode de descente de gradient que nous utiliserons à maintes reprises lorsque nous étudierons les réseaux de

6. Ce premier chapitre reprend en grande partie le chapitre 4 du livre *Machine Learning avec Scikit-Learn* (2^e édition, 2019), ainsi que quelques éléments essentiels des chapitres 1 à 3 de ce dernier.

neurones artificiels: descente de gradient groupée (ou batch), descente de gradient par mini-lots (ou mini-batch) et descente de gradient stochastique.

Nous examinerons ensuite la régression polynomiale, un modèle plus complexe pouvant s'ajuster à des jeux de données non linéaires. Ce modèle ayant davantage de paramètres que la régression linéaire, il est plus enclin à surajuster (*overfit*, en anglais) le jeu d'entraînement. C'est pourquoi nous verrons comment détecter si c'est ou non le cas à l'aide de courbes d'apprentissage, puis nous examinerons plusieurs techniques de régularisation qui permettent de réduire le risque de surajustement du jeu d'entraînement.

Enfin, nous étudierons deux autres modèles qui sont couramment utilisés pour les tâches de classification: la régression logistique et la régression softmax.

Ces notions prises individuellement ne sont pas très compliquées, mais il y en a beaucoup à apprendre dans ce chapitre, et elles sont toutes indispensables pour la suite, alors accrochez-vous bien, c'est parti !

1.1 INSTALLER LE PROJET HANDSON-ML2

Tous les exemples de code de ce livre, ainsi que du livre *Machine Learning avec Scikit-Learn* (2^e édition, 2019) sont disponibles sur GitHub dans le projet open source suivant: <https://github.com/ageron/handson-ml2>.

Ce projet contient un notebook (bloc-notes) Jupyter pour chaque chapitre. Chaque notebook contient du code exécutable en Python, intégré avec des explications et des figures. Jupyter est une interface web permettant d'exécuter de tels notebooks de façon graphique et interactive. Les notebooks 1 à 9 correspondent aux chapitres 1 à 9 du livre *Machine Learning avec Scikit-Learn*, tandis que les notebooks 10 à 19 correspondent aux chapitres 2 à 11 du livre que vous tenez entre les mains (en effet, ces deux livres n'en font qu'un dans la version originale). Bien que ce soit facultatif, vous êtes fortement encouragé(e) à installer ce projet, à jouer avec le code, à faire les exercices pratiques et à regarder de près les corrections qui figurent à la fin de chaque notebook: le Machine Learning et le Deep Learning s'apprennent surtout par la pratique !

Pour installer ce projet, il vous faudra le logiciel git. S'il n'est pas déjà présent sur votre système, suivez les instructions d'installation disponibles sur <https://git-scm.com/>. Ensuite, ouvrez un terminal (c'est-à-dire une fenêtre de commandes), puis tapez les commandes suivantes pour créer un répertoire de travail (vous pouvez choisir un autre répertoire si vous le souhaitez) et pour y installer le projet *handson-ml2*. Par convention, le premier \$ sur chaque ligne indique que le reste de la ligne est une commande à taper dans un terminal (mais il ne faut pas taper ce premier \$).

Sous Linux ou macOS, tapez:

```
$ mkdir ~/ml # Crée le répertoire de travail  
$ cd ~/ml # Se déplace dans ce répertoire  
$ git clone https://github.com/ageron/handson-ml2.git  
$ cd handson-ml2
```

Sous Windows, remplacez les deux premières lignes par :

```
$ md% %USERPROFILE%\ml # Crée le répertoire de travail
$ cd /d% %USERPROFILE%\ml # Se déplace dans ce répertoire
```

Sous Linux et macOS, ~ est automatiquement remplacé par le chemin de votre répertoire personnel : en général /home/votre_login sous Linux, ou /Users/votre_login sous macOS. Sous Windows, %USERPROFILE% donne également le chemin de votre répertoire personnel : en général C:\Users\votre_login.

Parfait ! Vous avez créé un nouveau répertoire de travail nommé ml dans votre répertoire personnel, et il contient désormais le projet handson-ml2.

1.2 INSTALLER TENSORFLOW ET JUPYTER

TensorFlow sera bien plus rapide si votre ordinateur dispose d'une carte graphique (GPU) compatible avec TensorFlow. À l'heure actuelle, les seules cartes graphiques supportées sont les cartes Nvidia estampillées CUDA Compute Capability 3.5 ou supérieur (voir <https://www.tensorflow.org/install/gpu> pour plus de détails). Si vous choisissez d'acquérir une telle carte graphique (ou que vous utilisez une machine virtuelle équipée), la première chose à faire est de télécharger le pilote de la carte sur <https://nvidia.com>, puis de l'installer.

Ensuite, la solution la plus simple et la plus flexible pour installer TensorFlow consiste à utiliser Anaconda : il s'agit d'une distribution de Python qui inclut de nombreuses librairies scientifiques. Téléchargez la version d'Anaconda pour Python 3 sur <https://anaconda.com>, et installez-la. Tapez ensuite la commande suivante dans votre terminal pour vérifier qu'Anaconda est correctement installé et pour obtenir la toute dernière version de conda, l'outil de gestion de paquets sur lequel repose Anaconda :

```
$ conda update -n base -c defaults conda
```

Ensuite, si vous avez installé le projet handson-ml2, vous pourrez simplement taper la commande suivante pour installer TensorFlow, ainsi que toutes les librairies dont nous aurons besoin dans ce livre (sous Windows, remplacez environment.yml par environment-windows.yml) :

```
$ conda env create -f environment.yml
```

Cette commande crée un environnement conda nommé tf2 et installe toutes les librairies qui sont listées dans le fichier environment.yml : TensorFlow, Jupyter et bien d'autres. Activez ensuite cet environnement :

```
$ conda activate tf2
```

Vous pourrez (ultérieurement, si vous le souhaitez) créer d'autres environnements conda, et y installer d'autres librairies, totalement indépendamment de l'environnement tf2. Par exemple, pour créer un environnement basé sur Python 3.7, et nommé xyz, vous pourrez utiliser la commande conda create -n xyz python=3.7. Vous pourrez ensuite activer cet environnement en tapant conda activate xyz, et y installer ensuite TensorFlow 2.1 avec la commande conda install tensorflow=2.1.

Lorsque conda cherche à installer une librairie, il la télécharge depuis une bibliothèque de librairies, ou « chaîne » (*channel*). Par défaut, conda utilise un petit nombre de chaînes gérées par l'entreprise Anaconda Inc., notamment la chaîne nommée anaconda. Les librairies provenant des chaînes par défaut sont garanties de bien fonctionner les unes avec les autres. Mais il vous arrivera parfois de ne pas y trouver la librairie qui vous intéresse. Dans ce cas, vous pouvez la chercher dans d'autres chaînes, gérées par d'autres personnes (mais n'utilisez que des chaînes gérées par des gens en qui vous avez confiance). Par exemple, vous pouvez installer la librairie `torchvision` de la chaîne `pytorch` à l'aide de la commande suivante : `conda install -c pytorch torchvision`. L'une des plus importantes chaînes se nomme `conda-forge`. Elle est gérée par des volontaires, et contient de nombreuses librairies qui ne sont pas présentes dans les chaînes par défaut.

Si la librairie qui vous intéresse ne se trouve dans aucune chaîne conda, en dernier recours vous pouvez la chercher dans PyPI, la bibliothèque de librairies standard de Python, et l'installer à l'aide de la commande `pip`. Par exemple, la librairie `tf-agents` (que nous utiliserons dans le chapitre 10) ne se trouve actuellement dans aucune chaîne Anaconda, donc il faut l'installer avec la commande `pip install tf-agents`. Il peut parfois y avoir des problèmes de compatibilité entre librairies installées avec conda et pip, donc il vaut mieux limiter autant que possible les librairies installées avec pip. Notez que le fichier `environment.yml` indique que les librairies conda doivent être recherchées dans les chaînes par défaut, ainsi que dans `conda-forge`. Il indique également une liste de librairies à installer avec pip.

Passons maintenant à Jupyter. Il est capable d'exécuter plusieurs notebooks en parallèle, et chacun d'entre eux peut être lancé dans un environnement conda différent. Par exemple, vous pourrez lancer certains notebooks dans l'environnement `tf2`, et d'autres dans l'environnement `xyz`. Mais pour cela, il faudra au préalable déclarer chaque environnement auprès de Jupyter. Déclarons donc l'environnement `tf2` auprès de Jupyter, sous le nom `python3` :

```
$ python -m ipykernel install --user --name = python3
```

Pour utiliser l'environnement `tf2` dans Jupyter, il suffira désormais de choisir le kernel nommé `python3` dans l'interface de Jupyter. Vous pouvez le nommer autrement si vous le souhaitez (par exemple `tf2`, tout simplement), mais le nom `python3` a l'intérêt de garantir que cet environnement sera utilisé par défaut par tous les notebooks Jupyter qui reposent sur Python 3.

C'est tout, il est temps de démarrer Jupyter !

```
$ jupyter notebook
```

Cette commande démarre le serveur web de Jupyter, et ouvre votre navigateur Internet sur l'adresse `http://localhost:8888/tree`. Vous devez y voir le contenu du répertoire courant. Il ne vous reste plus qu'à cliquer sur le notebook `04_training_linear_models.ipynb`: il s'agit du notebook correspondant au chapitre actuel (pour les chapitres suivants, rajoutez 8 au numéro du chapitre).

Si vous n'avez jamais utilisé Jupyter, le principe est simple: chaque notebook est constitué d'une liste de cellules. Chacune peut contenir du texte formaté ou du

code (Python, dans notre cas). Lorsqu'on exécute une cellule de code, le résultat s'affiche sous la cellule. Cliquez sur le menu Help > User Interface Tour pour un tour rapide de l'interface (en anglais). Pour vous entraîner, insérez quelques cellules de code au début du notebook, et exécutez quelques commandes Python, telles que `print("Hello world!")` (voir la figure 1.1). Vous pouvez renommer le notebook en cliquant sur son nom (voir la flèche 1 sur la figure). Cliquez dans une cellule de code et saisissez le code à exécuter (flèche 2), puis exécutez le code de la cellule en tapant Shift-Entrée ou en cliquant sur le bouton d'exécution (flèche 3). Lorsque vous cliquez à l'intérieur d'une cellule, vous passez en mode édition (la cellule est alors encadrée en vert). Lorsque vous tapez la touche Echap (Esc) ou que vous cliquez juste à gauche de la cellule, vous passez en mode commande (la cellule est alors encadrée en bleu). Lorsque vous êtes en mode commande, tapez la touche H pour afficher les nombreux raccourcis clavier disponibles.

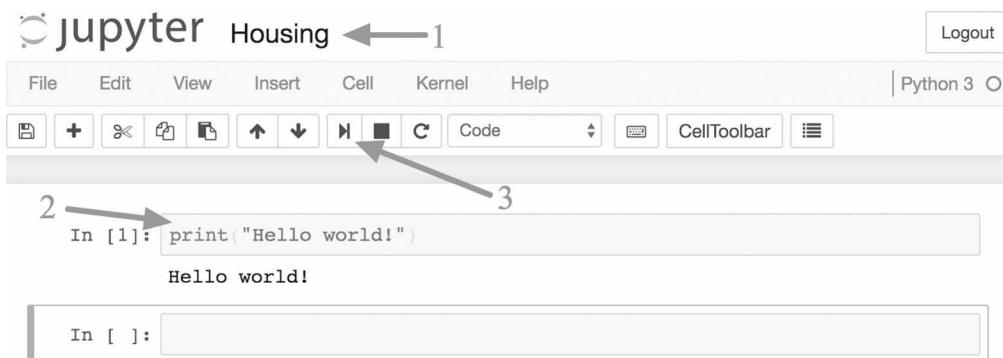


Figure 1.1 – Afficher «Hello world!» dans un notebook Jupyter

Vous pouvez exécuter toutes les cellules du notebook en cliquant sur le menu Cell > Run All.

Pour arrêter le serveur Jupyter, il suffit de taper Ctrl-C dans le terminal dans lequel vous l'avez démarré. Par la suite, à chaque fois que vous voudrez redémarrer Jupyter, il suffira d'ouvrir un terminal et de taper les commandes suivantes:

```
$ cd $HOME/ml/handson-ml2 # sous Linux ou macOS  
$ cd /d% USERPROFILE%\ml\handson-ml2 # sous Windows  
$ conda activate tf2  
$ jupyter notebook
```

Parfait ! Vous pouvez désormais exécuter tous les exemples de code de ce livre. Attaquons maintenant les bases du Machine Learning.

1.3 QU'EST-CE QUE LE MACHINE LEARNING?

Le Machine Learning (apprentissage automatique) est la science (et l'art) de programmer les ordinateurs de sorte qu'ils puissent apprendre à partir de données.

Voici une définition un peu plus générale :

« L'apprentissage automatique est la discipline donnant aux ordinateurs la capacité d'apprendre sans qu'ils soient explicitement programmés. »

Arthur Samuel, 1959

En voici une autre plus technique :

« Étant donné une tâche T et une mesure de performance P , on dit qu'un programme informatique apprend à partir d'une expérience E si les résultats obtenus sur T , mesurés par P , s'améliorent avec l'expérience E . »

Tom Mitchell, 1997

Votre filtre anti-spam, par exemple, est un programme d'apprentissage automatique qui peut apprendre à identifier les e-mails frauduleux à partir d'exemples de pourriels ou « *spam* » (par exemple, ceux signalés par les utilisateurs) et de messages normaux (parfois appelés « *ham* »). Les exemples utilisés par le système pour son apprentissage constituent le jeu d'entraînement (en anglais, *training set*). Chacun d'eux s'appelle une observation d'entraînement (on parle aussi d'échantillon ou d'instance). Dans le cas présent, la tâche T consiste à identifier parmi les nouveaux e-mails ceux qui sont frauduleux, l'expérience E est constituée par les données d'entraînement, et la mesure de performance P doit être définie. Vous pourrez prendre par exemple le pourcentage de courriels correctement classés. Cette mesure de performance particulière, appelée exactitude (en anglais, *accuracy*), est souvent utilisée dans les tâches de classification.

Pour cette tâche de classification, l'apprentissage requiert un jeu de données d'entraînement « étiqueté » (voir la figure 1.2), c'est-à-dire pour lequel chaque observation est accompagnée de la réponse souhaitée, que l'on nomme étiquette ou cible (*label* ou *target* en anglais). On parle dans ce cas d'apprentissage supervisé.

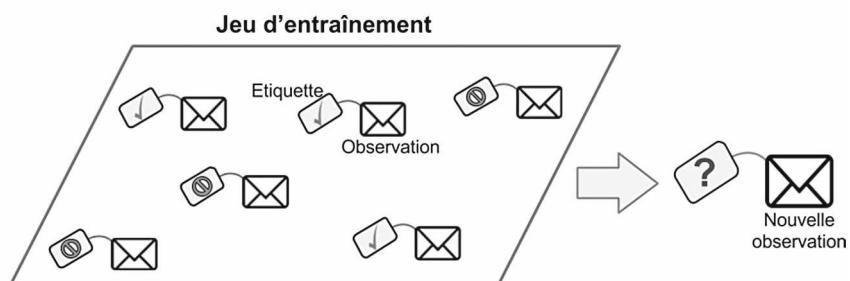


Figure 1.2 – Jeu d'entraînement étiqueté pour une tâche de classification (détection de spam)

Une autre tâche très commune pour un système d'auto-apprentissage est la tâche de « régression », c'est-à-dire la prédiction d'une valeur. Par exemple, on peut chercher à prédire le prix de vente d'une maison en fonction de divers paramètres (sa superficie, le revenu médian des habitants du quartier...). Tout comme la classification, il s'agit d'une tâche d'apprentissage supervisé : le jeu de données d'entraînement doit posséder, pour chaque observation, la valeur cible. Pour mesurer la performance du système, on peut par exemple calculer l'erreur moyenne commise par le système (ou, plus fréquemment, la racine carrée de l'erreur quadratique moyenne, comme nous le verrons dans un instant).

Il existe également des tâches de Machine Learning pour lesquelles le jeu d'entraînement n'est pas étiqueté. On parle alors d'apprentissage non supervisé. Par exemple, si l'on souhaite construire un système de détection d'anomalies (p. ex. pour détecter les produits défectueux dans une chaîne de production, ou pour détecter des tentatives de fraudes), on ne dispose généralement que de très peu d'exemples d'anomalies, donc il est difficile d'entraîner un système de classification supervisé. On peut toutefois entraîner un système performant en lui donnant des données non étiquetées (supposées en grande majorité normales), et ce système pourra ensuite détecter les nouvelles observations qui sortent de l'ordinaire. Un autre exemple d'apprentissage non supervisé est le partitionnement d'un jeu de données, par exemple pour segmenter les clients en groupes semblables, à des fins de marketing ciblé (voir la figure 1.3). Enfin, la plupart des algorithmes de réduction de la dimensionnalité, dont ceux dédiés à la visualisation des données, sont aussi des exemples d'algorithme d'apprentissage non supervisé.

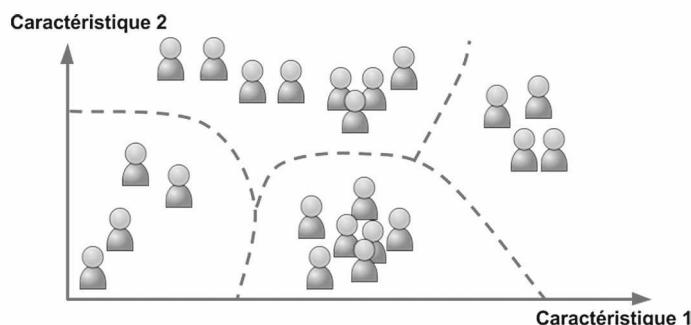


Figure 1.3 – Le partitionnement, un exemple d'apprentissage non supervisé

Résumons : on distingue les tâches d'apprentissage supervisé (classification, régression...), et les tâches d'apprentissage non supervisé (partitionnement, détection d'anomalie, réduction de dimensionnalité...). Un système de Machine Learning passe en général par deux phases : pendant la phase d'apprentissage il est entraîné sur un jeu de données d'entraînement, puis pendant la phase d'inférence il applique ce qu'il a appris sur de nouvelles données. Il existe toutes sortes de variantes de ce schéma général, mais c'est le principe à garder à l'esprit.

1.4 COMMENT LE SYSTÈME APPREND-IL ?

L'approche la plus fréquente consiste à créer un modèle prédictif et d'en régler les paramètres afin qu'il fonctionne au mieux sur les données d'entraînement. Par exemple, pour prédire le prix d'une maison en fonction de sa superficie et du revenu médian des habitants du quartier, on pourrait choisir un modèle linéaire, c'est-à-dire dans lequel la valeur prédictive est une somme pondérée des paramètres, plus un *terme constant* (en anglais, *intercept* ou *bias*). Cela donnerait l'équation suivante :

Équation 1.1 – Un modèle linéaire du prix des maisons

$$\text{prix} = \theta_0 + \theta_1 \times \text{superficie} + \theta_2 \times \text{revenu médian}$$

Dans cet exemple, le modèle a trois paramètres : θ_0 , θ_1 et θ_2 . Le premier est le terme constant, et les deux autres sont les *coefficients de pondération* (ou poids) des variables d'entrée. La phase d'entraînement de ce modèle consiste à trouver la valeur de ces paramètres qui minimisent l'erreur du modèle sur le jeu de données d'entraînement.⁷

Une fois les paramètres réglés, on peut utiliser le modèle pour faire des prédictions sur de nouvelles observations : c'est la phase d'inférence (ou de test). L'espoir est que si le modèle fonctionne bien sur les données d'entraînement, il fonctionnera également bien sur de nouvelles observations (c'est-à-dire pour prédire le prix de nouvelles maisons). Si la performance est bien moindre, on dit que le modèle a « surajusté » le jeu de données d'entraînement. Cela arrive généralement quand le modèle possède trop de paramètres par rapport à la quantité de données d'entraînement disponibles et à la complexité de la tâche à réaliser. Une solution est de réentraîner le modèle sur un plus gros jeu de données d'entraînement, ou bien de choisir un modèle plus simple, ou encore de contraindre le modèle, ce qu'on appelle la *régularisation* (nous y reviendrons dans quelques paragraphes). À l'inverse, si le modèle est mauvais sur les données d'entraînement (et donc très probablement aussi sur les nouvelles données), on dit qu'il « sous-ajuste » les données d'entraînement. Il s'agit alors généralement d'utiliser un modèle plus puissant ou de diminuer le degré de régularisation.

Formalisons maintenant davantage le problème de la régression linéaire.

1.5 RÉGRESSION LINÉAIRE

Comme nous l'avons vu, un modèle linéaire effectue une prédiction en calculant simplement une somme pondérée des variables d'entrée, en y ajoutant un terme constant :

Équation 1.2 – Prédiction d'un modèle de régression linéaire

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

7. Le nom « terme constant » peut être un peu trompeur dans le contexte du Machine Learning car il s'agit bien de l'un des paramètres du modèle que l'on cherche à optimiser, et qui varie donc pendant l'apprentissage. Toutefois, dès que l'apprentissage est terminé, ce terme devient bel et bien constant. Le nom anglais *bias* porte lui aussi à confusion car il existe une autre notion de biais, sans aucun rapport, présentée plus loin dans ce chapitre.

Dans cette équation :

- \hat{y} est la valeur prédictée,
- n est le nombre de variables,
- x_i est la valeur de la $i^{\text{ème}}$ variable,
- θ_j est le $j^{\text{ème}}$ paramètre du modèle (terme constant θ_0 et coefficients de pondération des variables $\theta_1, \theta_2, \dots, \theta_n$).

Ceci peut s'écrire de manière beaucoup plus concise sous forme vectorielle :

**Équation 1.3 – Prédiction d'un modèle de régression linéaire
(forme vectorielle)**

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

Dans cette équation :

- $\boldsymbol{\theta}$ est le *vecteur des paramètres* du modèle, il regroupe à la fois le terme constant θ_0 et les coefficients de pondération θ_1 à θ_n (ou poids) des variables. Notez que, dans le texte, les vecteurs sont représentés en minuscule et en gras, les scalaires (les simples nombres) sont représentés en minuscule et en italique, par exemple n , et les matrices sont représentées en majuscule et en gras, par exemple \mathbf{X} .
- \mathbf{x} est le *vecteur des valeurs* d'une observation, contenant les valeurs x_0 à x_n , où x_0 est toujours égal à 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ est le produit scalaire de $\boldsymbol{\theta}$ et de \mathbf{x} , qui est égal à $\theta_0x_0 + \theta_1x_1 + \dots + \theta_nx_n$, et que l'on notera dans ce livre $\boldsymbol{\theta}^T \mathbf{x}$.⁸
- h_{θ} est la fonction hypothèse, utilisant les paramètres de modèle $\boldsymbol{\theta}$.

Par souci d'efficacité, on réalise souvent plusieurs prédictions simultanément. Pour cela, on regroupe dans une même matrice \mathbf{X} toutes les observations pour lesquelles on souhaite faire des prédictions (ou plus précisément tous leurs vecteurs de valeurs). Par exemple, si l'on souhaite faire une prédition pour 3 observations dont les vecteurs de valeurs sont respectivement $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}$ et $\mathbf{x}^{(3)}$, alors on les regroupe dans une matrice \mathbf{X} dont la première ligne est la transposée de $\mathbf{x}^{(1)}$, la seconde ligne est la transposée de $\mathbf{x}^{(2)}$ et la troisième ligne est la transposée de $\mathbf{x}^{(3)}$:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ (\mathbf{x}^{(3)})^T \end{pmatrix}$$

8. En Machine Learning, on manipule beaucoup de vecteurs et de matrices, et il est parfois plus pratique de représenter les vecteurs sous la forme de matrices à une seule colonne (ce qu'on appelle des vecteurs colonnes). Cela permet notamment d'avoir le même code pour gérer indifféremment des vecteurs ou des matrices. Si $\boldsymbol{\theta}$ et \mathbf{x} sont des vecteurs colonnes, alors $\boldsymbol{\theta}^T$ est la transposée de $\boldsymbol{\theta}$ (c'est-à-dire une matrice à une seule ligne, ce qu'on appelle un vecteur ligne), et $\boldsymbol{\theta}^T \mathbf{x}$ représente le produit matriciel de $\boldsymbol{\theta}^T$ et de \mathbf{x} : le résultat est une matrice contenant une seule cellule dont la valeur est le produit scalaire $\boldsymbol{\theta} \cdot \mathbf{x}$. Voilà pourquoi nous noterons le produit scalaire $\boldsymbol{\theta}^T \mathbf{x}$, bien qu'en réalité ce ne soit pas rigoureusement la même chose que $\boldsymbol{\theta} \cdot \mathbf{x}$.

Pour réaliser simultanément une prédiction pour toutes les observations, on peut alors simplement utiliser l'équation suivante :

Équation 1.4 – Prédictions multiples d'un modèle de régression linéaire

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$$

- $\hat{\mathbf{y}}$ est le vecteur des prédictions. Son $i^{\text{ème}}$ élément correspond à la prédiction du modèle pour la $i^{\text{ème}}$ observation.
- Plus haut, l'ordre n'importait pas car $\boldsymbol{\theta} \cdot \mathbf{x} = \mathbf{x} \cdot \boldsymbol{\theta}$, mais ici l'ordre est important. En effet, le produit matriciel n'est défini que quand le nombre de colonnes de la première matrice est égal au nombre de lignes de la seconde matrice. Ici, la matrice \mathbf{X} possède 3 lignes (car il y a 3 observations) et $n+1$ colonnes (la première colonne est remplie de 1, et chaque autre colonne correspond à une variable d'entrée). Le vecteur colonne $\boldsymbol{\theta}$ possède $n+1$ lignes (une pour le terme constant, puis une pour chaque poids de variable d'entrée) et bien sûr une seule colonne. Le résultat $\hat{\mathbf{y}}$ est un vecteur colonne contenant 3 lignes (une par observation) et une colonne. De plus, si vous vous étonnez qu'il n'y ait plus de transposée dans cette équation, rappelez-vous que chaque ligne de \mathbf{X} est déjà la transposée d'un vecteur de valeurs.

Voici donc ce qu'on appelle un modèle de régression linéaire. Voyons maintenant comment l'entraîner. Comme nous l'avons vu, entraîner un modèle consiste à définir ses paramètres de telle sorte que le modèle s'ajuste au mieux au jeu de données d'entraînement. Pour cela, nous avons tout d'abord besoin d'une mesure de performance qui nous indiquera si le modèle s'ajuste bien ou mal au jeu d'entraînement. Dans la pratique, on utilise généralement une mesure de l'erreur commise par le modèle sur le jeu d'entraînement, ce qu'on appelle une *fonction de coût*. La fonction de coût la plus courante pour un modèle de régression est la racine carrée de l'*erreur quadratique moyenne* (en anglais, *root mean square error* ou RMSE), définie dans l'équation 1.5 :

Équation 1.5 – Racine carrée de l'erreur quadratique moyenne (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m [h(\mathbf{x}^{(i)}) - y^{(i)}]^2}$$

- Notez que, pour alléger les notations, la fonction d'hypothèse est désormais notée h plutôt que $h_{\boldsymbol{\theta}}$, mais il ne faut pas oublier qu'elle est paramétrée par le vecteur $\boldsymbol{\theta}$. De même, nous écrirons simplement $\text{RMSE}(\mathbf{X})$ par la suite, même s'il ne faut pas oublier que la RMSE dépend de l'hypothèse h .
- m est le nombre d'observations dans le jeu de données.

Pour entraîner un modèle de régression linéaire, il s'agit de trouver le vecteur $\boldsymbol{\theta}$ qui minimise la RMSE. En pratique, il est un peu plus simple et rapide de minimiser l'*erreur quadratique moyenne* (MSE, simplement le carré de la RMSE), et ceci conduit au même résultat, parce que la valeur qui minimise une fonction positive minimise aussi sa racine carrée.

1.5.1 Équation normale

Pour trouver la valeur de θ qui minimise la fonction de coût, il s'avère qu'il existe une *solution analytique*, c'est-à-dire une formule mathématique nous fournissant directement le résultat. Celle-ci porte le nom d'*équation normale*.

Équation 1.6 – Équation normale

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Dans cette équation :

- $\hat{\theta}$ est la valeur de θ qui minimise la fonction de coût.
- L'exposant -1 indique que l'on calcule l'inverse de la matrice $\mathbf{X}^T \mathbf{X}$. En théorie, cet inverse n'est pas forcément défini, mais dans la pratique il l'est quasi toujours pourvu qu'il y ait bien davantage d'observations que de variables.
- \mathbf{y} est le vecteur des valeurs cibles $y^{(1)}$ à $y^{(m)}$ (une par observation).

Générons maintenant des données à l'allure linéaire sur lesquelles tester cette équation (figure 1.4). Nous utiliserons pour cela la librairie NumPy :

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

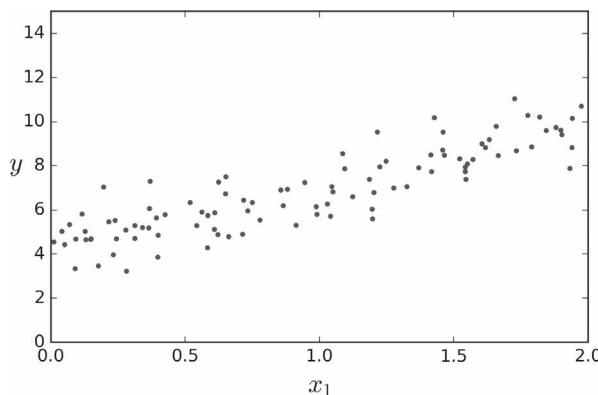


Figure 1.4 – Jeu de données généré aléatoirement

Calculons maintenant $\hat{\theta}$ à l'aide de l'équation normale. Nous allons utiliser la fonction `inv()` du module d'algèbre linéaire `np.linalg` de NumPy pour l'inversion de matrice, et la méthode `dot()` pour les produits matriciels⁹:

```
X_b = np.c_[np.ones((100, 1)), X] # Ajouter x0 = 1 à chaque observation
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

9. Un opérateur `@` représentant la multiplication matricielle a été introduit à partir de Python 3.5. Il est supporté par NumPy à partir de la version 1.10. Cela permet donc d'écrire `A @ B` plutôt que `A.dot(B)` si A et B sont des matrices NumPy, ce qui peut rendre le code beaucoup plus lisible.

Nous avons utilisé la fonction $y = 4 + 3x_1 + \text{bruit gaussien}$ pour générer les données. Voyons ce que l'équation a trouvé :

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

Nous aurions aimé obtenir $\theta_0 = 4$ et $\theta_1 = 3$, au lieu de $\theta_0 = 4,215$ et $\theta_1 = 2,770$. C'est assez proche, mais le bruit n'a pas permis de retrouver les paramètres exacts de la fonction d'origine.

Maintenant nous pouvons faire des prédictions à l'aide de $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # Ajouter x0 = 1 à chaque obs.
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

Représentons graphiquement les prédictions de ce modèle :

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

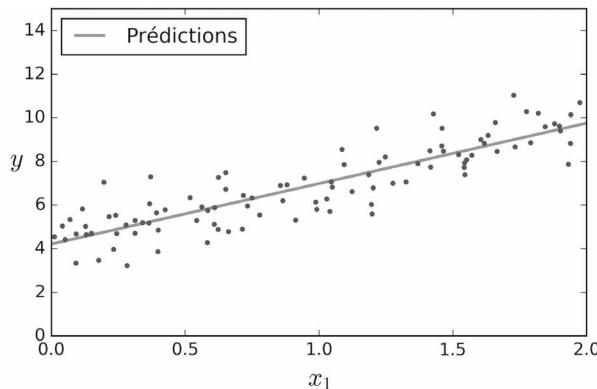


Figure 1.5 – Prédictions du modèle de régression linéaire

Effectuer une régression linéaire avec Scikit-Learn est très simple¹⁰ :

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

10. Notez que Scikit-Learn sépare le terme constant (`intercept_`) des coefficients de pondération des variables (`coef_`).

La classe `LinearRegression` repose sur la fonction `scipy.linalg.lstsq()` (le nom signifie « *least squares* », c'est-à-dire « méthode des moindres carrés »). Vous pourriez l'appeler directement ainsi :

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

Cette fonction calcule $\hat{\theta} = X^T y$, où X^T est le pseudo-inverse de X (plus précisément le pseudo-inverse de Moore-Penrose). Vous pouvez utiliser `np.linalg.pinv()` pour calculer ce pseudo-inverse, si vous le souhaitez :

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

Ce pseudo-inverse est lui-même calculé à l'aide d'une technique très classique de factorisation de matrice nommée « décomposition en valeurs singulières » (ou SVD). Cette technique parvient à décomposer le jeu d'entraînement X en produit de trois matrices U , Σ et V^T (voir `numpy.linalg.svd()`). Le pseudo-inverse se calcule ensuite ainsi : $X^T = V \Sigma^T U^T$. Pour calculer la matrice Σ^T , l'algorithme prend Σ et met à zéro toute valeur plus petite qu'un seuil minuscule, puis il remplace les valeurs non nulles par leur inverse, et enfin il transpose la matrice. Cette approche est bien plus rapide que de calculer l'équation normale, et elle gère bien les cas limites : en effet, l'équation normale ne fonctionne pas lorsque la matrice $X^T X$ n'est pas inversible (notamment lorsque $m < n$ ou quand certains attributs sont redondants), alors que le pseudo-inverse est toujours défini.

1.5.2 Complexité algorithmique

L'équation normale calcule l'inverse de $X^T X$, qui est une matrice $(n+1) \times (n+1)$ (où n est le nombre de variables). La complexité algorithmique d'une inversion de matrice se situe entre $O(n^{2.4})$ et $O(n^3)$, selon l'algorithme d'inversion utilisé. Autrement dit, si vous doublez le nombre de variables, le temps de calcul est multiplié par un facteur compris entre $2^{2.4} = 5.3$ et $2^3 = 8$.

L'approche SVD utilisée par la classe `LinearRegression` de Scikit-Learn est environ $O(n^2)$. Si vous doublez le nombre de caractéristiques, vous multipliez le temps de calcul par environ 4.



L'équation normale et l'approche SVD deviennent toutes deux très lentes lorsque le nombre de caractéristiques devient grand (p. ex. 100 000). En revanche, les deux sont linéaires vis-à-vis du nombre d'observations dans le jeu d'entraînement (algorithmes en $O(m)$), donc elles peuvent bien gérer un gros volume de données, à condition qu'il tienne en mémoire.

Par ailleurs, une fois votre modèle de régression linéaire entraîné (en utilisant l'équation normale ou n'importe quel autre algorithme), obtenir une prédiction est extrêmement rapide : la complexité de l'algorithme est linéaire par rapport au

nombre d'observations sur lesquelles vous voulez obtenir des prédictions et par rapport au nombre de variables. Autrement dit, si vous voulez obtenir des prédictions sur deux fois plus d'observations (ou avec deux fois plus de variables), le temps de calcul sera grossièrement multiplié par deux.

Nous allons maintenant étudier une méthode d'entraînement de modèle de régression linéaire très différente, mieux adaptée au cas où il y a beaucoup de variables ou trop d'observations pour tenir en mémoire.

1.6 DESCENTE DE GRADIENT

La *descente de gradient* est un algorithme d'optimisation très général, capable de trouver des solutions optimales à un grand nombre de problèmes. L'idée générale de la descente de gradient est de corriger petit à petit les paramètres dans le but de minimiser une fonction de coût.

Supposons que vous soyez perdu en montagne dans un épais brouillard et que vous puissiez uniquement sentir la pente du terrain sous vos pieds. Pour redescendre rapidement dans la vallée, une bonne stratégie consiste à avancer vers le bas dans la direction de plus grande pente. C'est exactement ce que fait la descente de gradient : elle calcule le gradient de la fonction de coût au point θ , puis progresse en direction du gradient descendant. Lorsque le gradient est nul, vous avez atteint un minimum !

En pratique, vous commencez par remplir θ avec des valeurs aléatoires (c'est ce qu'on appelle l'*initialisation aléatoire*). Puis vous l'améliorez progressivement, pas à pas, en tentant à chaque étape de faire décroître la fonction de coût (ici la MSE), jusqu'à ce que l'algorithme converge vers un minimum (voir la figure 1.6).

Un élément important dans l'algorithme de descente de gradient est la dimension des pas, que l'on détermine par l'intermédiaire de l'hyperparamètre `learning_rate` (taux d'apprentissage).



Un hyperparamètre est un paramètre de l'algorithme d'apprentissage, et non un paramètre du modèle. Autrement dit, il ne fait pas partie des paramètres que l'on cherche à optimiser pendant l'apprentissage. Toutefois, on peut très bien lancer l'algorithme d'apprentissage plusieurs fois, en essayant à chaque fois une valeur différente pour chaque hyperparamètre, jusqu'à trouver une combinaison de valeurs qui permet à l'algorithme d'apprentissage de produire un modèle satisfaisant. Pour évaluer chaque modèle, on utilise alors un jeu de données distinct du jeu d'entraînement, appelé le *jeu de validation*. Ce réglage fin des hyperparamètres s'appelle le *hyperparameter tuning* en anglais.

Si le taux d'apprentissage est trop petit, l'algorithme devra effectuer un grand nombre d'itérations pour converger et prendra beaucoup de temps (voir la figure 1.7).

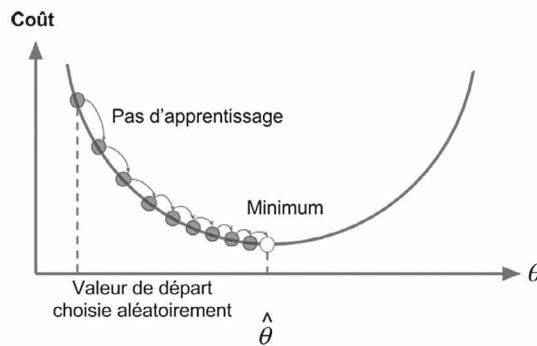


Figure 1.6 – Dans cette représentation de la descente de gradient, les paramètres du modèle sont initialisés de manière aléatoire et sont modifiés de manière répétée afin de minimiser la fonction de coût. La taille des étapes d'apprentissage est proportionnelle à la pente de la fonction de coût, de sorte que les étapes deviennent plus petites à mesure que les paramètres s'approchent du minimum

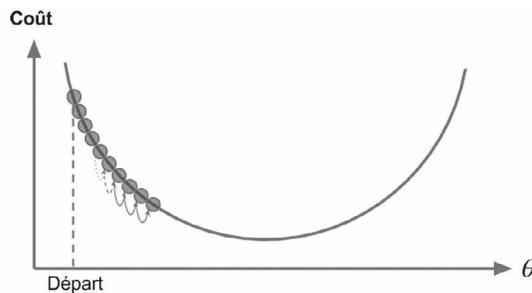


Figure 1.7 – Le taux d'apprentissage est trop petit

Inversement, si le taux d'apprentissage est trop élevé, vous risquez de dépasser le point le plus bas et de vous retrouver de l'autre côté, peut-être même plus haut qu'avant. Ceci pourrait faire diverger l'algorithme, avec des valeurs de plus en plus grandes, ce qui empêcherait de trouver une bonne solution (voir la figure 1.8).

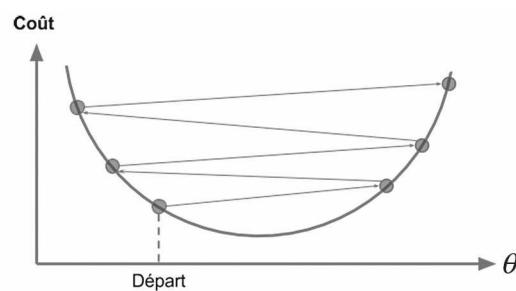


Figure 1.8 – Le taux d'apprentissage est trop élevé

Enfin, toutes les fonctions de coût n'ont pas la forme d'une jolie cuvette régulière. Il peut y avoir des trous, des crêtes, des plateaux et toutes sortes de terrains irréguliers, ce qui complique la convergence vers le minimum. La figure 1.9 illustre les deux principaux pièges de la descente de gradient. Si l'initialisation aléatoire démarre l'algorithme sur la gauche, alors l'algorithme convergera vers un *minimum local*, qui n'est pas le *minimum global*. Si l'initialisation commence sur la droite, alors il lui faut très longtemps pour traverser le plateau. Et si l'algorithme est arrêté prématurément, vous n'atteindrez jamais le minimum global.

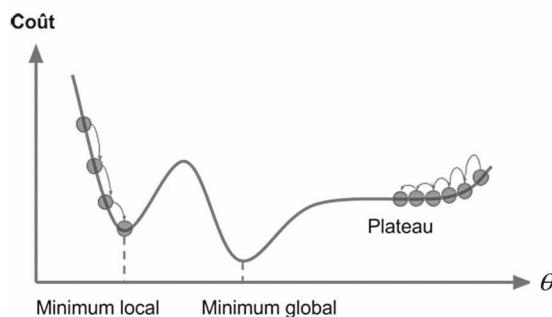


Figure 1.9 – Pièges de la descente de gradient

Heureusement, la fonction de coût MSE du modèle de régression linéaire est par chance une *fonction convexe*, ce qui signifie que si vous prenez deux points quelconques de la courbe, le segment de droite les joignant ne coupe jamais la courbe. Ceci signifie qu'il n'y a pas de minima locaux, mais juste un minimum global. C'est aussi une fonction continue dont la pente ne varie jamais abruptement¹¹. Ces deux faits ont une conséquence importante : il est garanti que la descente de gradient s'approchera aussi près que l'on veut du minimum global (si vous attendez suffisamment longtemps et si le taux d'apprentissage n'est pas trop élevé).

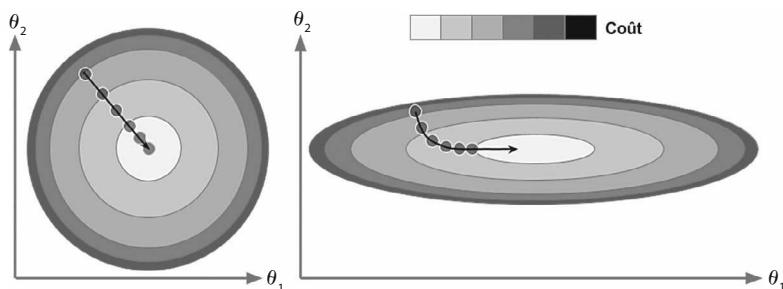


Figure 1.10 – Descente de gradient avec (à gauche) et sans (à droite) normalisation des variables

11. Techniquement parlant, sa dérivée est *lipschitzienne*, et par conséquent uniformément continue.

En fait, la fonction de coût a la forme d'un bol, mais il peut s'agir d'un bol déformé si les variables ont des échelles très différentes. La figure 1.10 présente deux descentes de gradient, l'une (à gauche) sur un jeu d'entraînement où les variables 1 et 2 ont la même échelle, l'autre (à droite) sur un jeu d'entraînement où la variable 1 a des valeurs beaucoup plus petites que la variable 2.¹²

Comme vous pouvez le constater, à gauche l'algorithme de descente de gradient descend directement vers le minimum et donc l'atteint rapidement, tandis qu'à droite il part dans une direction presque orthogonale à la direction du minimum global et se termine par une lente progression dans le fond d'une vallée pratiquement plate. Au bout du compte il atteindra le minimum, mais cela prendra très longtemps.



Lorsque vous effectuez une descente de gradient, vous devez veiller à ce que toutes les variables aient la même échelle, sinon la convergence sera beaucoup plus lente. Pour cela, une solution est de normaliser les variables d'entrée, c'est-à-dire soustraire à chaque variable sa moyenne, puis diviser le résultat par l'écart-type.

Voici comment mettre en œuvre la normalisation avec NumPy:

```
X_norm = (X - X.mean(axis=0)) / X.std(axis=0)
```

Ou bien vous pouvez utiliser la classe `StandardScaler` de Scikit-Learn:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_norm = scaler.fit_transform(X)
```

Une fois le modèle entraîné, il est important de bien penser à appliquer exactement la même transformation aux nouvelles observations (en utilisant la moyenne et l'écart-type mesurés sur le jeu d'entraînement). Si vous utilisez Scikit-Learn, il s'agit de réutiliser le même `StandardScaler` et d'appeler sa méthode `transform()`, et non sa méthode `fit_transform()`.

La figure 1.10 illustre aussi le fait que l'entraînement d'un modèle consiste à rechercher une combinaison de paramètres du modèle minimisant une fonction de coût (sur le jeu d'entraînement). C'est une recherche dans l'espace de paramètres du modèle : plus le modèle comporte de paramètres, plus l'espace comporte de dimensions et plus difficile est la recherche : rechercher une aiguille dans une botte de foin à 300 dimensions est plus compliqué que dans un espace à trois dimensions. Heureusement, sachant que la fonction de coût MSE est convexe dans le cas d'une régression linéaire, l'aiguille se trouve tout simplement au fond de la cuvette.

1.6.1 Descente de gradient ordinaire

Pour implémenter une descente de gradient, vous devez calculer le gradient de la fonction de coût par rapport à chaque paramètre θ_j du modèle : autrement dit, vous devez calculer quelle est la modification de la fonction de coût lorsque vous modifiez

12. La variable 1 étant plus petite, il faut une variation plus importante de θ_1 pour affecter la fonction de coût, c'est pourquoi la cuvette s'allonge le long de l'axe θ_1 .

un petit peu θ_j . C'est ce qu'on appelle une *dérivée partielle*. Cela revient à demander « quelle est la pente de la montagne sous mes pieds si je me tourne vers l'est ? », puis de poser la même question en se tournant vers le nord (et ainsi de suite pour toutes les autres dimensions, si vous pouvez vous imaginer un univers avec plus de trois dimensions). La dérivée partielle de la fonction de coût par rapport à θ_j , notée $\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta})$ se calcule comme suit :

Équation 1.7 – Dérivées partielles de la fonction de coût

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Au lieu de calculer individuellement chaque dérivée partielle, vous pouvez utiliser la formulation vectorielle suivante pour les calculer toutes ensemble. Le vecteur gradient, noté $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$, est composé de toutes les dérivées partielles de la fonction de coût (une pour chaque paramètre du modèle) :

Équation 1.8 – Vecteur gradient de la fonction de coût

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$



Notez que cette formule implique des calculs sur l'ensemble du jeu d'entraînement \mathbf{X} , à chaque étape de la descente de gradient ! C'est pourquoi l'algorithme s'appelle *batch gradient descent* en anglais (*descente de gradient groupée*) : il utilise l'ensemble des données d'entraînement à chaque étape. À vrai dire, *full gradient descent* (descente de gradient complète) serait un nom plus approprié. De ce fait, il est extrêmement lent sur les jeux d'entraînement très volumineux (mais nous verrons plus loin des algorithmes de descente de gradient bien plus rapides). Cependant, la descente de gradient s'accorde bien d'un grand nombre de variables : entraîner un modèle de régression linéaire lorsqu'il y a des centaines de milliers de variables est bien plus rapide avec une descente de gradient qu'avec une équation normale ou une décomposition SVD.

Une fois que vous avez le vecteur gradient (qui pointe vers le haut), il vous suffit d'aller dans la direction opposée pour descendre. Ce qui revient à soustraire $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$ de $\boldsymbol{\theta}$. C'est ici qu'apparaît le taux d'apprentissage η ¹³ : multipliez le vecteur gradient par η pour déterminer le pas de la progression vers le bas.

13. Èta (η) est la 7^e lettre de l'alphabet grec.

Équation 1.9 – Un pas de descente de gradient

$$\theta^{(\text{étape suivante})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Voyons une implémentation rapide de cet algorithme :

```
eta = 0.1 # taux d'apprentissage
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # initialisation aléatoire

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

Facile ! Voyons maintenant le `theta` qui en résulte :

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

C'est exactement ce que nous avions obtenu avec l'équation normale ! La descente de gradient a parfaitement fonctionné. Mais que se serait-il passé avec un taux d'apprentissage différent ? La figure 1.11 présente les 10 premiers pas de la descente de gradient en utilisant trois taux d'apprentissage différents (la ligne hachurée représente le point de départ).

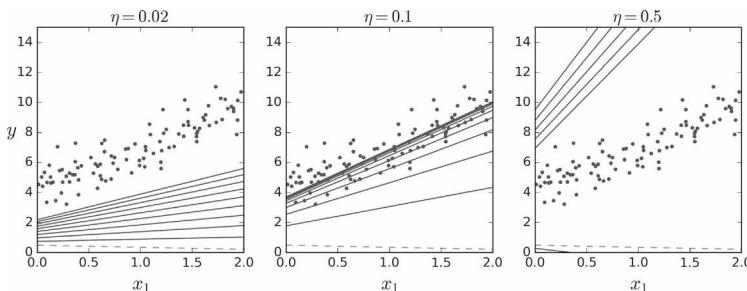


Figure 1.11 – Descente de gradient avec divers taux d'apprentissage

À gauche, le taux d'apprentissage est trop faible : l'algorithme aboutira au bout du compte à la solution, mais cela prendra très longtemps. Au milieu, le taux d'apprentissage semble assez bon : en quelques itérations seulement, l'algorithme a déjà convergé vers la solution. À droite, le taux d'apprentissage est trop haut, l'algorithme diverge, sautant ici et là et s'éloignant finalement de plus en plus de la solution à chaque étape.

Si les données sont normalisées, un taux d'apprentissage de 0,01 sera souvent correct. Si vous constatez que l'algorithme met trop de temps à converger il faudra l'augmenter, et inversement si l'algorithme diverge, il faudra le diminuer.

Vous vous demandez peut-être comment choisir le nombre d'itérations. Si ce nombre est trop faible, vous serez encore très loin de la solution optimale lorsque l'algorithme s'arrêtera ; mais s'il est trop élevé, vous perdrez du temps alors que les paramètres du

modèle n'évolueront plus. Une solution simple consiste à choisir un très grand nombre d'itérations, mais à interrompre l'algorithme lorsque le vecteur de gradient devient très petit, c'est-à-dire quand sa norme devient inférieure à un très petit nombre ϵ (appelé *tolérance*), ce qui signifie que la descente de gradient a (presque) atteint son minimum. Bien sûr, plus vous choisirez un ϵ petit, plus le résultat sera précis, mais plus vous devrez attendre avant que l'algorithme ne s'arrête.

1.6.2 Descente de gradient stochastique

Le principal problème de la descente de gradient ordinaire, c'est qu'elle utilise à chaque étape l'ensemble du jeu d'entraînement pour calculer les gradients, ce qui la rend très lente lorsque le jeu d'entraînement est de grande taille. À l'extrême inverse, la *descente de gradient stochastique* choisit à chaque étape une observation prise au hasard dans l'ensemble d'entraînement et calcule les gradients en se basant uniquement sur cette seule observation. Bien évidemment, travailler sur une seule observation à la fois rend l'algorithme beaucoup plus rapide puisqu'il n'a que très peu de données à manipuler à chaque itération. Cela permet aussi de l'entraîner sur des jeux de données de grande taille, car il n'a besoin que d'une seule observation en mémoire à chaque itération.

Par contre, étant donné sa nature stochastique (c'est-à-dire aléatoire), cet algorithme est beaucoup moins régulier qu'une descente de gradient ordinaire : au lieu de décroître doucement jusqu'à atteindre le minimum, la fonction de coût va effectuer des sauts vers le haut et vers le bas et ne décroîtra qu'en moyenne. Au fil du temps, elle arrivera très près du minimum, mais une fois là elle continuera à effectuer des sauts aux alentours sans jamais s'arrêter (voir la figure 1.12). Par conséquent, lorsque l'algorithme est stoppé, les valeurs finales des paramètres sont bonnes, mais pas optimales.

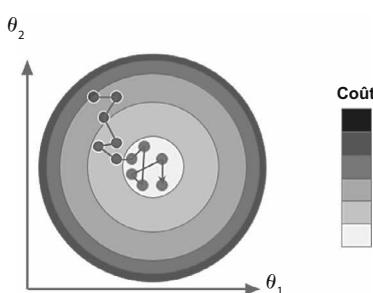


Figure 1.12 – Avec la descente de gradient stochastique, chaque étape de l'entraînement est beaucoup plus rapide mais aussi beaucoup plus stochastique qu'avec la descente de gradient par lots

Lorsque la fonction de coût est très irrégulière (comme sur la figure 1.9), ceci peut en fait aider l'algorithme à sauter à l'extérieur d'un minimum local, et donc la descente de gradient stochastique a plus de chances de trouver le minimum global que la descente de gradient ordinaire.

Par conséquent, cette sélection aléatoire est un bon moyen d'échapper à un minimum local, mais n'est pas satisfaisante car l'algorithme ne va jamais s'arrêter au minimum. Une solution à ce dilemme consiste à réduire progressivement le taux d'apprentissage : les pas sont grands au début (ce qui permet de progresser rapidement et d'échapper aux minima locaux), puis ils réduisent progressivement, permettant à l'algorithme de s'arrêter au minimum global. Ce processus est semblable à l'algorithme portant en anglais le nom de *simulated annealing* (ou *recuit simulé*) parce qu'il est inspiré du processus métallurgique consistant à refroidir lentement un métal en fusion. La fonction qui détermine le taux d'apprentissage à chaque itération s'appelle l'*échéancier d'apprentissage* (en anglais, *learning schedule*). Si le taux d'apprentissage est réduit trop rapidement, l'algorithme peut rester bloqué sur un minimum local ou même s'immobiliser à mi-chemin du minimum. Si le taux d'apprentissage est réduit trop lentement, l'algorithme peut sauter pendant longtemps autour du minimum et finir au-dessus de l'optimum si vous tentez de l'arrêter trop tôt.

Ce code implémente une descente de gradient stochastique en utilisant un calendrier d'apprentissage très simple :

```
n_epochs = 50
t0, t1 = 5, 50 # hyperparamètres d'échéancier d'apprentissage

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # initialisation aléatoire

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

Par convention, nous effectuons des cycles de m itérations : chacun de ces cycles s'appelle une *époque* (en anglais, *epoch*). Alors que le code de la descente de gradient ordinaire effectue 1 000 fois plus d'itérations sur l'ensemble du jeu d'apprentissage, ce code-ci ne parcourt le jeu d'entraînement qu'environ 50 fois et aboutit à une solution plutôt satisfaisante :

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

La figure 1.13 présente les 20 premières étapes de l'entraînement (notez l'irrégularité des pas).

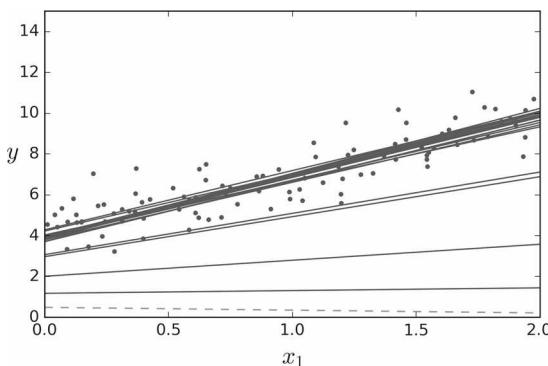


Figure 1.13 – Vingt premiers pas d'une descente de gradient stochastique

Remarquez qu'étant donné que les observations sont choisies au hasard, certaines d'entre elles peuvent être sélectionnées plusieurs fois par époque, alors que d'autres ne le seront jamais. Si vous voulez être sûr que l'algorithme parcourt toutes les observations durant chaque époque, vous pouvez adopter une autre approche qui consiste à mélanger le jeu d'entraînement comme on bat un jeu de cartes (en prenant soin de mélanger conjointement les variables d'entrée et les étiquettes), à le parcourir observation par observation, puis à mélanger à nouveau, et ainsi de suite. Cependant, la convergence est en général plus lente.



Lorsqu'on utilise une descente de gradient stochastique, les observations d'entraînement doivent être indépendantes et identiquement distribuées (IID), pour garantir que les paramètres évoluent vers l'optimum global, en moyenne. Un moyen simple d'y parvenir consiste à mélanger les observations durant l'entraînement (p. ex. en choisissant aléatoirement chaque observation, ou en battant le jeu d'entraînement au début de chaque époque). Si vous ne mélangez pas les observations, et si par exemple celles-ci sont triées par étiquette, alors la descente de gradient stochastique commencera par optimiser pour une étiquette, puis pour la suivante, et ainsi de suite, et ne convergera pas vers le minimum global.

Pour effectuer avec Scikit-Learn une régression linéaire par descente de gradient stochastique (ou SGD), vous pouvez utiliser la classe `SGDRegressor` qui par défaut optimise la fonction de coût du carré des erreurs. Le code suivant effectue au maximum 1 000 cycles ou époques ou tourne jusqu'à ce que la perte devienne inférieure à 0,001 durant une époque (`max_iter=1000, tol=1e-3`). Il commence avec un taux d'apprentissage de 0,1 (`eta0=0.1`), en utilisant le calendrier d'apprentissage par défaut (différent de celui ci-dessus). À la fin, il n'effectue aucune régularisation (`penalty=None`, expliqué un peu plus loin).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Là encore, vous obtenez une solution assez proche de celle donnée par l'équation normale :

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

1.6.3 Descente de gradient par mini-lots

Le dernier algorithme de descente de gradient que nous allons étudier s'appelle *descente de gradient par mini-lots* (en anglais, *mini-batch gradient descent*). Il se comprend aisément, une fois que vous connaissez les deux autres méthodes de descente de gradient, ordinaire et stochastique : à chaque étape, au lieu de calculer les dérivées partielles sur l'ensemble du jeu d'entraînement (DG ordinaire) ou sur une seule observation (DG stochastique), la descente de gradient par mini-lots calcule le gradient sur de petits sous-ensembles d'observations sélectionnées aléatoirement qu'on appelle *mini-lots*. Le principal avantage par rapport à la descente de gradient stochastique, c'est que vous améliorez les performances du fait de l'optimisation matérielle des opérations matricielles, particulièrement lorsque vous tirez parti de processeurs graphiques (comme nous le verrons avec TensorFlow).

La progression de l'algorithme dans l'espace des paramètres est moins désordonnée que dans le cas de la descente de gradient stochastique, tout particulièrement lorsque les mini-lots sont relativement grands. Par conséquent, la descente de gradient par mini-lots aboutira un peu plus près du minimum qu'une descente de gradient stochastique, par contre elle aura plus de mal à sortir d'un minimum local (dans le cas des problèmes où il existe des minima locaux, ce qui n'est pas le cas de la régression linéaire comme nous l'avons vu précédemment). La figure 1.14 montre les chemins suivis dans l'espace des paramètres par trois algorithmes de descente de gradient durant leur entraînement. Ils finissent tous à proximité du minimum, mais la descente de gradient ordinaire s'arrête effectivement à ce minimum, tandis que la descente de gradient stochastique et celle par mini-lots continuent à se déplacer autour. Cependant, n'oubliez pas que la descente de gradient ordinaire prend beaucoup de temps à chaque étape, et que les deux derniers algorithmes auraient aussi atteint le minimum si vous aviez utilisé un bon échéancier d'apprentissage.

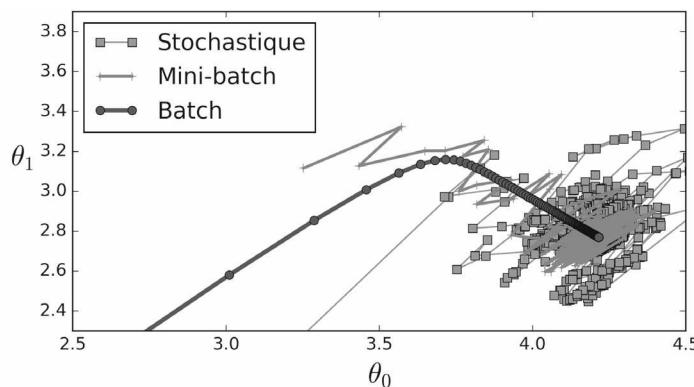


Figure 1.14 – Cheminement dans l'espace des paramètres de plusieurs descentes de gradient

Comparons les algorithmes de régression linéaire dont nous avons parlé jusqu'ici¹⁴ (rappelons que m est le nombre d'observations du jeu d'entraînement et n son nombre de variables).

Tableau 1.1 – Comparaison d'algorithmes de régression linéaire

Algorithme	m grand	Hors- mémoire possible ?	n grand	Hyper- paramètres	Normali- sation requise ?	Scikit- Learn
Équation normale	rapide	non	lent	0	non	non disponible
SVD	rapide	non	lent	0	non	LinearRegression
DG ordinaire	lent	non	rapide	2	oui	SGDRegressor
DG stochastique	rapide	oui	rapide	≥ 2	oui	SGDRegressor
DG par mini-lots	rapide	oui	rapide	≥ 2	oui	SGDRegressor



Il n'y a pratiquement aucune différence après l'entraînement : tous ces algorithmes aboutissent à des modèles très similaires et effectuent des prédictions exactement de la même manière.

1.7 RÉGRESSION POLYNOMIALE

Et si la complexité de vos données ne peut se modéliser par une ligne droite ? Étonnamment, vous pouvez aussi utiliser un modèle linéaire pour ajuster des données non linéaires. L'un des moyens d'y parvenir consiste à ajouter les puissances de chacune des variables comme nouvelles variables, puis d'entraîner un modèle linéaire sur ce nouvel ensemble de données : cette technique porte le nom de *régression polynomiale*.

Voyons-en un exemple : générons d'abord quelques données non linéaires à l'aide d'une fonction polynomiale du second degré¹⁵ (en y ajoutant des aléas) :

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

14. Si l'équation normale ne s'applique qu'à la régression linéaire, les algorithmes de descente de gradient peuvent, comme nous le verrons, permettre d'entraîner de nombreux autres modèles.

15. Une fonction polynomiale du second degré est de la forme $y = ax^2 + bx + c$.

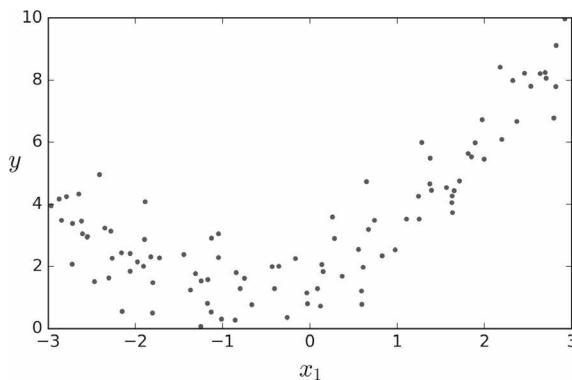


Figure 1.15 – Jeu de données non linéaire avec aléas générés

Il est clair qu'une ligne droite n'ajustera jamais correctement ces données. Utilisons donc la classe `PolynomialFeatures` de Scikit-Learn pour transformer nos données d'apprentissage, en ajoutant les carrés (polynômes du second degré) des variables aux variables existantes (dans notre cas, il n'y avait qu'une variable) :

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

`X_poly` contient désormais la variable originelle de `X` plus le carré de celle-ci. Nous pouvons alors ajuster un modèle `LinearRegression` à notre jeu d'entraînement complété :

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

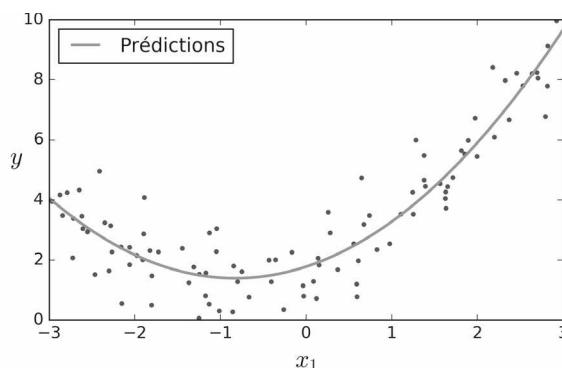


Figure 1.16 – Prédictions du modèle de régression polynomiale

Ce n'est pas mal ! Le modèle donne l'estimation $\hat{y} = 0,56 x_1^2 + 0,93 x_1 + 1,78$ alors que la fonction d'origine était $y = 0,5 x_1^2 + 1,0 x_1 + 2,0 + \text{bruit gaussien}$.

Notez que lorsqu'il y a des variables multiples, la régression polynomiale est capable de mettre en évidence des relations entre ces variables (ce que ne peut pas faire un modèle de régression linéaire simple). Ceci est rendu possible par le fait que `PolynomialFeatures` ajoute toutes les combinaisons de variables jusqu'à un certain degré. Si vous avez par exemple deux variables a et b , `PolynomialFeatures` avec `degree=3` ne va pas seulement ajouter les variables a^2 , a^3 , b^2 et b^3 , mais aussi les combinaisons ab , a^2b et ab^2 .



`PolynomialFeatures(degree=d)` transforme un tableau comportant n variables en un tableau comportant $\frac{(n+d)!}{d!n!}$ variables, où factorielle n (notée $n!$) = $1 \times 2 \times 3 \times \dots \times n$. Attention à l'explosion combinatoire du nombre de variables !

1.8 COURBES D'APPRENTISSAGE

Si vous effectuez une régression polynomiale de haut degré, il est probable que vous ajusterez beaucoup mieux les données d'entraînement qu'avec une régression linéaire simple. La figure 1.17 applique par exemple un modèle polynomial de degré 300 aux données d'entraînement précédentes, puis compare le résultat à un modèle purement linéaire et à un modèle quadratique (polynôme du second degré). Notez comme le modèle polynomial de degré 300 ondule pour s'approcher autant que possible des observations d'entraînement.

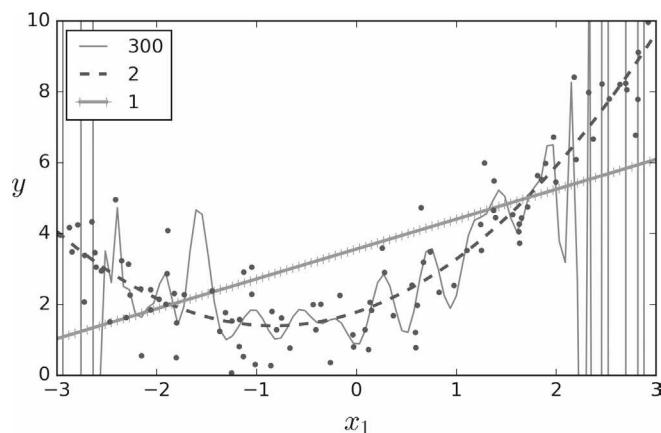


Figure 1.17 – Régression polynomiale de haut degré

Ce modèle polynomial de haut degré surajuste considérablement les données d'entraînement, alors que le modèle linéaire les sous-ajuste. Le modèle qui se généralisera le mieux dans ce cas est le modèle du second degré. C'est logique, vu que les données ont été générées à l'aide d'un polynôme du second degré. Mais en général,

vous ne saurez pas quelle fonction a générée les données, alors comment décider de la complexité à donner au modèle ? Comment pouvez-vous déterminer si votre modèle surajuste ou sous-ajuste les données ?

Une solution, comme nous l'avons vu plus haut, consiste à entraîner plusieurs fois le modèle avec des degrés polynomiaux différents, jusqu'à trouver le degré qui produit le meilleur modèle, évalué sur un jeu de données de validation.



Plutôt que de réservier des données pour le jeu de validation, on peut effectuer ce qu'on appelle une *validation croisée*: on découpe le jeu de données d'entraînement en k morceaux (ou k *folds* en anglais), et on entraîne le modèle sur tous les morceaux sauf le premier, que l'on utilise ensuite pour évaluer le modèle. Puis on réinitialise ce modèle et on l'entraîne à nouveau, mais cette fois-ci sur tous les morceaux sauf le deuxième, que l'on utilise pour évaluer le modèle, et ainsi de suite. Pour chaque combinaison d'hyperparamètres, on obtient ainsi k évaluations. On peut alors choisir la combinaison qui produit la meilleure évaluation en moyenne. Voir les classes `GridSearchCV` et `RandomizedSearchCV` de Scikit-Learn.

Un autre moyen consiste à regarder les *courbes d'apprentissage* : il s'agit de diagrammes représentant les résultats obtenus par le modèle sur le jeu d'entraînement et sur le jeu de validation en fonction de la taille du jeu d'entraînement ou de l'itération d'entraînement. Pour générer ces graphiques, entraînez le modèle plusieurs fois sur des sous-ensembles de différentes tailles du jeu d'entraînement. Le code suivant définit une fonction qui trace les courbes d'apprentissage d'un modèle pour un jeu d'entraînement donné :

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Cette fonction commence par découper le jeu de données en un jeu d'entraînement et un jeu de validation, grâce à la fonction `test_train_split()` de Scikit-Learn. Ensuite, en augmentant progressivement la taille du jeu de données d'entraînement, il entraîne un modèle sur ce jeu d'entraînement et l'évalue à la fois sur le jeu d'entraînement et sur le jeu de validation. Enfin, il utilise Matplotlib pour afficher les deux courbes: l'erreur d'entraînement et l'erreur de validation, en fonction de la taille du jeu de données d'entraînement. Examinons les courbes d'apprentissage d'un simple modèle linéaire :

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

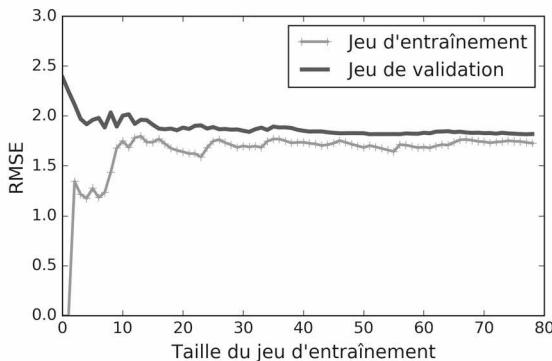


Figure 1.18 – Courbes d'apprentissage

Ce modèle qui sous-ajuste mérite quelques explications. Voyons d'abord les performances sur le jeu d'entraînement : lorsque celui-ci ne comporte qu'une ou deux observations, le modèle peut les ajuster parfaitement, c'est pourquoi la courbe commence à zéro. Mais à mesure qu'on ajoute de nouvelles observations au jeu d'entraînement, le modèle a de plus en plus de mal à les ajuster, d'une part à cause du bruit, et d'autre part parce que ce n'est pas linéaire du tout. C'est pourquoi l'erreur sur le jeu d'entraînement augmente jusqu'à atteindre un plateau : à partir de là, l'ajout de nouvelles observations au jeu d'entraînement ne modifie plus beaucoup l'erreur moyenne, ni en bien ni en mal. Voyons maintenant les performances du modèle sur les données de validation : lorsque le modèle est entraîné à partir de très peu d'observations, il est incapable de généraliser correctement, c'est pourquoi l'erreur de validation est relativement importante au départ. Puis le modèle s'améliore à mesure qu'il reçoit davantage d'exemples d'entraînement, c'est pourquoi l'erreur de validation diminue lentement. Cependant, il n'est toujours pas possible de modéliser correctement les données à l'aide d'une ligne droite, c'est pourquoi l'erreur finit en plateau, très proche de l'autre courbe.

Ces courbes d'apprentissage sont caractéristiques d'un modèle qui sous-ajuste : les deux courbes atteignent un plateau, elles sont proches et relativement hautes.



Si votre modèle sous-ajuste les données d'entraînement, ajouter d'autres exemples d'entraînement ne servira à rien. Vous devez choisir un modèle plus complexe ou trouver de meilleures variables.

Voyons maintenant les courbes d'apprentissage d'un modèle polynomial de degré 10 sur les mêmes données :

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
```

```

("lin_reg", LinearRegression(),
])
plot_learning_curves(polynomial_regression, X, y)

```

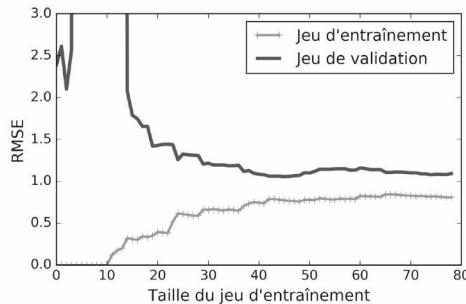


Figure 1.19 – Courbes d'apprentissage pour le modèle polynomial de degré 10

Ces courbes d'apprentissage ressemblent un peu aux précédentes, mais il y a deux différences très importantes :

1. L'erreur sur les données d'entraînement est très inférieure à celle du modèle de régression linéaire.
2. Il y a un écart entre les courbes. Cela signifie que le modèle donne des résultats nettement meilleurs sur le jeu d'entraînement que sur le jeu de validation, ce qui est la marque d'un modèle qui surajuste. Cependant, si vous augmentez le nombre d'observations de votre jeu d'entraînement, les deux courbes se rapprocheront.



Un moyen d'améliorer un modèle qui surajuste consiste à lui donner davantage d'observations d'entraînement, jusqu'à ce que l'erreur de validation se rapproche de l'erreur d'entraînement.

Le compromis entre biais et variance

Résultat théorique important en statistiques et en apprentissage automatique, l'erreur de généralisation d'un modèle peut s'exprimer comme la somme de trois erreurs très différentes :

- *Biais* : cette composante de l'erreur de généralisation est due à de mauvaises hypothèses, comme par exemple de supposer que les données sont linéaires lorsqu'elles sont quadratiques. Un modèle à haut biais a plus de chances de sous-ajuster les données d'entraînement.
- *Variance* : cette composante de l'erreur est due à la sensibilité excessive du modèle à de petites variations dans le jeu d'entraînement. Un modèle ayant beaucoup de degrés de liberté (comme par exemple un modèle polynomial de

degré élevé) aura vraisemblablement une variance élevée, et par conséquent surajuster les données d'entraînement.

- *Erreur irréductible* : elle est due au bruit présent dans les données. Le seul moyen de réduire cette composante de l'erreur consiste à nettoyer les données (c'est-à-dire à réparer les sources de données, par exemple les capteurs défectueux, à détecter et à supprimer les données aberrantes, etc.).

Accroître la complexité d'un modèle va en général accroître sa variance et réduire son biais. Inversement, réduire la complexité d'un modèle accroît son biais et réduit sa variance. C'est pourquoi cet arbitrage entre biais et variance est qualifié de compromis.

1.9 MODÈLES LINÉAIRES RÉGULARISÉS

Un bon moyen de réduire le surajustement consiste à régulariser le modèle (c'est-à-dire à lui imposer des contraintes) : moins il a de degrés de liberté, plus il lui est difficile de surajuster les données. Un moyen simple de régulariser un modèle polynomial consiste à réduire le nombre de degrés du polynôme.

Pour un modèle linéaire, la régularisation consiste en général à imposer des contraintes aux coefficients de pondération du modèle. Nous allons maintenant étudier la régression ridge, la régression lasso et elastic net qui imposent des contraintes sur ces pondérations de trois façons différentes.

1.9.1 Régression ridge

Également appelée régularisation de Tikhonov, la *régression ridge* (appelée aussi *régression de crête*) est une version régularisée de la régression linéaire : un *terme de régularisation* égal à $\alpha \sum_{i=1}^n \theta_i^2$ est ajouté à la fonction de coût. Ceci force l'algorithme d'apprentissage non seulement à ajuster les données, mais aussi à maintenir les coefficients de pondération du modèle aussi petits que possible. Notez que le terme de régularisation ne doit être ajouté à la fonction de coût que durant l'entraînement. Une fois le modèle entraîné, vous évaluerez les performances du modèle en utilisant une mesure de performance non régularisée.



Il est assez courant que la fonction de coût utilisée durant l'entraînement soit différente de la mesure de performance utilisée pour tester. En dehors de la régularisation, il existe une autre raison pour laquelle elles peuvent différer : une bonne fonction de coût pour l'entraînement doit avoir des dérivées permettant une bonne optimisation, tandis que la mesure de performance utilisée pour tester doit vérifier si l'estimation est proche de l'objectif final. Par exemple, un classificateur est souvent entraîné en minimisant une fonction de coût *log loss* (présentée plus loin), mais au final on évalue la performance du classificateur en observant son *exactitude* (le taux de classifications correctes), ou bien, dans le cas d'un classificateur binaire tel qu'un détecteur de spam, sa *précision* (le taux d'observations classées comme positives et qui le sont effectivement) ou son *rappel* (en anglais *recall*, le taux d'observations positives qui sont bien détectées).

L'hyperparamètre α contrôle la quantité de régularisation que vous voulez imposer au modèle. Si $\alpha = 0$, on a tout simplement affaire à une régression linéaire. Si α est très grand, alors tous les coefficients de pondération finiront par avoir des valeurs très proches de zéro et le résultat sera une ligne horizontale passant par la moyenne des données. L'équation suivante présente la fonction de coût d'une régression ridge¹⁶.

Équation 1.10 – Fonction de coût d'une régression ridge

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Notez que le terme constant θ_0 n'est pas régularisé (la somme commence à $i = 1$, et non 0). Si nous définissons \mathbf{w} comme le vecteur de pondération des variables (θ_1 à θ_n), alors le terme de régularisation est simplement égal à $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, où $\|\mathbf{w}\|_2$ représente la norme ℓ_2 du vecteur de pondération \mathbf{w} (voir l'encart sur les normes ℓ_k , ci-dessous).

Pour une descente de gradient, ajoutez simplement $\alpha \mathbf{w}$ au vecteur gradient de la MSE (voir équation 1.8).

Les normes ℓ_k

Il existe diverses mesures de distance ou *normes*:

- Celle qui nous est la plus familière est la *norme euclidienne*, également appelée *norme ℓ_2* . La norme ℓ_2 d'un vecteur \mathbf{v} , notée $\|\mathbf{v}\|_2$ (ou tout simplement $\|\mathbf{v}\|$) est égale à la racine carrée de la somme des carrés de ses éléments. Par exemple, la norme d'un vecteur contenant les éléments 3 et 4 est égale à la racine carrée de $3^2 + 4^2 = 25$, c'est-à-dire 5. Une ville située 3 km à l'est et 4 km au sud se situe à 5 km à vol d'oiseau.
- La *norme ℓ_1* d'un vecteur \mathbf{v} , notée $\|\mathbf{v}\|_1$, est égale à la somme de la valeur absolue des éléments de \mathbf{v} . On l'appelle parfois *norme de Manhattan* car elle mesure la distance entre deux points dans une ville où l'on ne peut se déplacer que le long de rues à angle droit. Par exemple si vous avez rendez-vous à 3 blocs vers l'est et 4 blocs vers le sud, vous devrez parcourir $4 + 3 = 7$ blocs.
- Plus généralement, la norme ℓ_k d'un vecteur \mathbf{v} contenant n éléments est définie par la formule: $\|\mathbf{v}\|_k = (\|v_0\|^k + \|v_1\|^k + \dots + \|v_n\|^k)^{\frac{1}{k}}$. La norme ℓ_0 donne simplement le nombre d'éléments non nuls du vecteur, tandis que la norme ℓ_∞ fournit la plus grande des valeurs absolues des éléments de ce vecteur. Plus l'index de la norme est élevé, plus celle-ci donne d'importance aux grandes valeurs en négligeant les petites. Les plus utilisées, de loin, sont les normes ℓ_1 et ℓ_2 .

16. Il est courant d'utiliser la notation $J(\boldsymbol{\theta})$ pour les fonctions de coût ne disposant pas d'un nom court : nous utiliserons souvent cette notation dans la suite de ce livre. Le contexte expliquera clairement de quelle fonction de coût il s'agit.



Il est important de normaliser les données (en utilisant par exemple un `StandardScaler`) avant d'effectuer une régression ridge, car celle-ci est sensible aux différences d'échelle des variables d'entrée. C'est vrai de la plupart des modèles régularisés.

La figure 1.20 présente différents modèles de régression ridge entraînés sur des données linéaires avec différentes valeurs α . À gauche, on a effectué des régressions ridge ordinaires, ce qui conduit à des prédictions linéaires. À droite, les données ont été tout d'abord étendues en utilisant `PolynomialFeatures (degree=10)`, puis normalisées en utilisant `StandardScaler`, et enfin on a appliqué aux variables résultantes un modèle ridge, correspondant donc à une régression polynomiale avec régularisation ridge. Notez comme en accroissant α on obtient des prédictions plus lisses, moins extrêmes, plus raisonnables¹⁷ : ceci correspond à une réduction de la variance du modèle, mais à un accroissement de son biais.

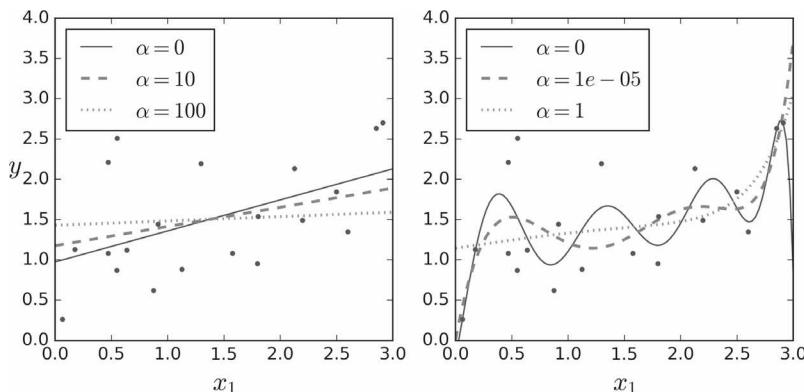


Figure 1.20 – Un modèle linéaire (à gauche) et un modèle polynomial (à droite), tous deux avec différents niveaux de régularisation ridge

Tout comme la régression linéaire, la régression ridge peut s'effectuer soit en résolvant une équation (solution analytique), soit en effectuant une descente de gradient. Les avantages et inconvénients sont les mêmes. L'équation 1.11 présente la solution analytique, où \mathbf{A} est la matrice identité¹⁸ $(n+1) \times (n+1)$, à l'exception d'une valeur 0 dans la cellule en haut à gauche, correspondant au terme constant.

Équation 1.11 – Solution analytique d'une régression ridge

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

17. D'où le nom anglais de « ridge », qui signifie crête ou arête.

18. Une matrice carrée remplie de zéros, à l'exception des 1 sur la diagonale principale (d'en haut à gauche à en bas à droite).

Voici comment effectuer une régression ridge par la méthode analytique avec Scikit-Learn (il s'agit d'une variante de la solution ci-dessus utilisant une technique de factorisation de matrice d'André-Louis Cholesky) :

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([1.55071465])
```

Et en utilisant une descente de gradient stochastique¹⁹ :

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

L'hyperparamètre `penalty` définit le type de terme de régularisation à utiliser. En spécifiant "l2", vous choisissez d'ajouter à la fonction de coût un terme de régularisation égal à la moitié du carré de la norme ℓ_2 du vecteur de pondération : c'est simplement la régression ridge.

1.9.2 Régression lasso

La régression *Least Absolute Shrinkage and Selection Operator* (appelée plus simplement *régression lasso*) est une autre version régularisée de la régression linéaire : tout comme la régression ridge, elle ajoute un terme de régularisation à la fonction de coût, mais elle utilise la norme ℓ_1 du vecteur de pondération au lieu de la moitié du carré de la norme ℓ_2 .

Équation 1.12 – Fonction de coût de la régression lasso

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

La figure 1.12 présente les mêmes résultats que la figure 1.20, mais en remplaçant les régressions ridge par des régressions lasso et en utilisant des valeurs α plus petites.

19. Vous pouvez aussi utiliser la classe `Ridge` avec `solver="sag"`. La descente de gradient moyenne stochastique (Stochastic Average GD ou SAG) est une variante de la descente de gradient stochastique SGD. Pour plus de détails, reportez-vous à la présentation de Mark Schmidt *et al.*, University of British Columbia : <https://homl.info/12>.

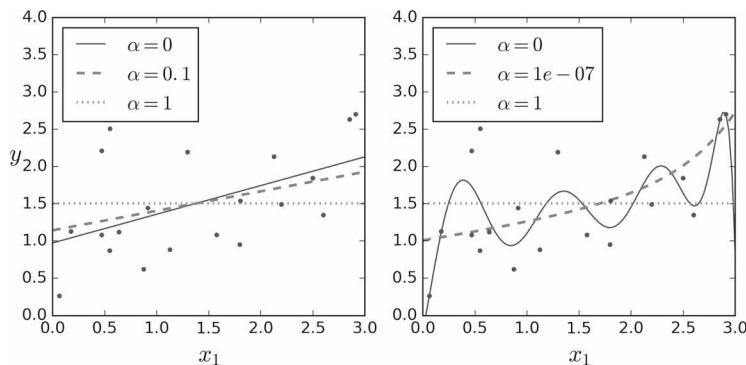


Figure 1.21 – Un modèle linéaire (à gauche) et un modèle polynomial (à droite), tous deux utilisant différents niveaux de régularisation lasso

Une caractéristique importante de la régression lasso est qu'elle tend à éliminer les poids des variables les moins importantes (elle leur donne la valeur zéro). Voyez par exemple la ligne à tirets du graphique de droite de la figure 1.21 (avec $\alpha = 10^{-7}$) qui semble quadratique, presque linéaire : tous les coefficients de pondération des variables polynomiales de haut degré sont nuls. Autrement dit, la régression lasso effectue automatiquement une sélection des variables et produit un *modèle creux* (*sparse*, en anglais), avec seulement quelques coefficients de pondération non nuls.

L'observation de la figure 1.22 vous permettra de comprendre intuitivement pourquoi : les axes représentent deux paramètres du modèle et les courbes de niveau en arrière-plan représentent différentes fonctions de perte. Sur le graphique en haut à gauche, les courbes de niveau correspondent à la perte $\ell_1(|\theta_1| + |\theta_2|)$, qui décroît linéairement lorsque vous vous rapprochez de l'un des axes. Ainsi, si vous initialisez les paramètres du modèle à $\theta_1 = 2$ et $\theta_2 = 0,5$, effectuer une descente de gradient fera décroître de la même manière les deux paramètres (comme le montre la ligne pointillée), et par conséquent θ_2 atteindra en premier la valeur 0 (étant donné qu'il était plus proche de 0 au départ). Après quoi, la descente de gradient suit la rigole jusqu'à atteindre $\theta_1 = 0$ (par bonds successifs étant donné que les gradients de ℓ_1 ne sont jamais proches de zéro, mais valent soit -1 , soit 1 pour chaque paramètre). Sur le graphique en haut à droite, les courbes de niveau représentent la fonction de coût lasso (c.-à-d. une fonction de coût MSE plus une perte ℓ_1). Les petits cercles blancs matérialisent le chemin suivi par la descente de gradient pour optimiser certains paramètres du modèle initialisés aux alentours de $\theta_1 = 0,25$ et $\theta_2 = -1$: remarquez à nouveau que le cheminement atteint rapidement $\theta_2 = 0$, puis suit la rigole et finit par osciller aux alentours de l'optimum global (représenté par le carré). Si nous augmentons α , l'optimum global se déplacerait vers la gauche le long de la ligne pointillée, alors qu'il se déplacerait vers la droite si nous diminuions α (dans cet exemple, les paramètres optimaux pour la fonction de coût MSE non régularisée sont $\theta_1 = 2$ et $\theta_2 = 0,5$).

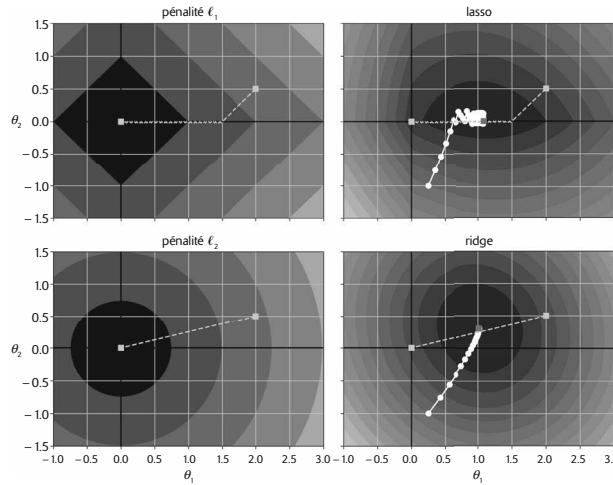


Figure 1.22 – Régularisation de régressions lasso et ridge

Les deux graphiques du bas illustrent la même chose mais cette fois pour une pénalité ℓ_2 . Sur le graphique en bas à gauche, vous pouvez remarquer que la perte ℓ_2 diminue avec la distance à l'origine, c'est pourquoi la descente de gradient progresse directement vers ce point. Sur le graphique en bas à droite, les courbes de niveau représentent la fonction de coût d'une régression ridge (c.-à-d. une fonction de coût MSE plus une perte ℓ_2). Il y a deux différences principales avec lasso. Tout d'abord, les gradients diminuent à mesure que les paramètres s'approchent de l'optimum global, ce qui fait que la descente de gradient ralentit naturellement, ce qui l'aide à converger (sans rebondir de part et d'autre). Ensuite, les paramètres optimaux (représentés par le carré) se rapprochent de plus en plus de l'origine lorsque vous augmentez α , mais ne sont jamais éliminés entièrement.



Lorsque vous utilisez une régression lasso, pour éviter que la descente de gradient ne rebondisse à la fin aux alentours de l'optimum, vous devez diminuer graduellement le taux d'apprentissage durant l'entraînement (l'algorithme continuera à osciller autour de l'optimum, mais le pas sera de plus en plus petit, et donc il convergera).

La fonction de coût de la régression lasso n'est pas différentiable en $\theta_i = 0$ (pour $i = 1, 2, \dots, n$), mais la descente de gradient fonctionne néanmoins fort bien si vous utilisez à la place un vecteur de sous-gradient \mathbf{g}^{20} lorsque $\theta_i = 0$, quel que soit i . L'équation 1.13

20. Vous pouvez considérer qu'un vecteur de sous-gradient en un point non différentiable est un vecteur intermédiaire entre les vecteurs de gradient autour de ce point. Par exemple, la fonction $f(x) = |x|$ n'est pas dérivable en 0, mais sa dérivée est -1 pour $x < 0$ et $+1$ pour $x > 0$, donc toute valeur comprise entre -1 et $+1$ est une sous-dérivée de $f(x)$ en 0.

définit un vecteur de sous-gradient que vous pouvez utiliser pour la descente de gradient dans le cas d'une fonction de coût lasso.

Équation 1.13 – Vecteur de sous-gradient pour une régression lasso

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{signe}(\theta_1) \\ \text{signe}(\theta_2) \\ \vdots \\ \text{signe}(\theta_n) \end{pmatrix} \quad \text{où } \text{signe}(\theta_i) = \begin{cases} -1 & \text{si } \theta_i < 0, \\ 0 & \text{si } \theta_i = 0, \\ +1 & \text{si } \theta_i > 0. \end{cases}$$

Voici un petit exemple Scikit-Learn utilisant la classe Lasso :

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Remarquez que vous pourriez également utiliser un SGDRegressor (penalty="l1").

1.9.3 Elastic net

La régularisation elastic net (que l'on peut traduire par « filet élastique ») est un compromis entre ridge et lasso : le terme de régularisation est un simple mélange des termes de régularisation de ces deux régressions, ce mélange est contrôlé par le ratio de mélange (ou mix ratio) r : lorsque $r = 0$, elastic net équivaut à la régression ridge, et quand $r = 1$, c'est l'équivalent d'une régression lasso.

Équation 1.14 – Fonction de coût d'elastic net

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

Alors quand devriez-vous effectuer une régression linéaire simple (c.-à-d. sans régularisation), ou au contraire utiliser une régularisation ridge, lasso ou elastic net ? Il est pratiquement toujours préférable d'avoir au moins un petit peu de régularisation, c'est pourquoi vous devriez éviter en général d'effectuer une régression linéaire simple. La régression ridge est un bon choix par défaut, mais si vous soupçonnez que seules quelques variables sont utiles, vous devriez préférer une régression lasso ou elastic net, car elles tendent à annuler les coefficients de pondération des variables inutiles, comme nous l'avons expliqué. En général, elastic net est préférée à lasso, étant donné que lasso peut se comporter de manière imprévisible lorsque le nombre de variables est supérieur au nombre d'observations du jeu d'entraînement ou lorsque plusieurs des variables sont fortement corrélées.

Voici un court exemple Scikit-Learn utilisant ElasticNet (`l1_ratio` correspond au ratio de mélange r) :

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
```

```
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

1.9.4 Arrêt précoce

Une manière très différente de régulariser les algorithmes d'apprentissage itératifs tels que la descente de gradient consiste à stopper cet apprentissage dès que l'erreur de validation atteint un minimum. C'est ce qu'on appelle l'*arrêt précoce* (*early stopping*). La figure 1.23 présente un modèle complexe (ici un modèle de régression polynomiale de haut degré) entraîné à l'aide d'une descente de gradient ordinaire : au fur et à mesure des cycles ou « époques », l'algorithme apprend et son erreur de prédiction (RMSE) sur le jeu d'apprentissage décroît avec son erreur de prédiction sur le jeu de validation. Cependant, au bout d'un moment l'erreur de validation cesse de décroître et commence même à augmenter à nouveau. Ceci indique que le modèle a commencé à surajuster les données d'entraînement. Avec l'arrêt précoce, vous interrompez simplement l'entraînement dès que l'erreur de validation atteint le minimum. C'est une technique de régularisation tellement simple et efficace que Geoffrey Hinton l'a qualifiée de « superbe repas gratuit ».

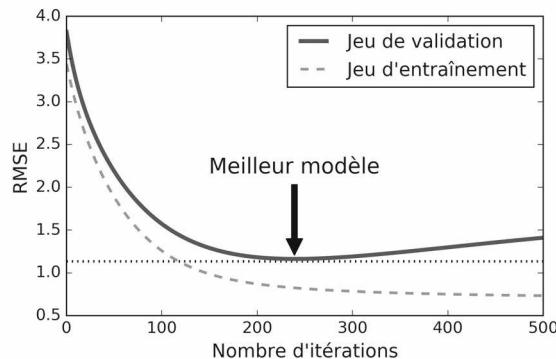


Figure 1.23 – Régularisation par arrêt précoce



Avec les descentes de gradient stochastique et par mini-lots, les courbes ne sont pas si régulières et il peut être difficile de savoir si l'on a atteint le minimum ou non. Une solution consiste à ne s'arrêter que lorsque l'erreur de validation a été supérieure au minimum pendant un certain temps (c'est-à-dire lorsque vous êtes convaincus que le modèle ne fera pas mieux), puis d'effectuer un retour arrière vers les paramètres du modèle pour lesquels l'erreur de validation était minimale.

Voici une implémentation simple de l'arrêt précoce :

```
from sklearn.base import clone

# Prépare les données
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler())
])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # Reprend au point d'arrêt
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Notez que si le `SGDRegressor` est créé avec l'option `warm_start=True`, alors sa méthode `fit()`, lorsqu'elle est appelée, reprend l'entraînement là où il a été interrompu au lieu de recommencer entièrement.

1.10 RÉGRESSION LOGISTIQUE

Certains algorithmes de régression peuvent être utilisés pour la classification (et inversement). La *régression logistique* (appelée également *régression logit*) est utilisée couramment pour estimer la probabilité qu'une observation appartienne à une classe particulière (p. ex. quelle est la probabilité que cet e-mail soit un pourriel ?). Si la probabilité estimée est supérieure à 50 %, alors le modèle prédit que l'observation appartient à cette classe (appelée la *classe positive*, d'étiquette « 1 »), sinon il prédit qu'elle appartient à l'autre classe (la *classe négative*, d'étiquette « 0 »). C'est en fait un classificateur binaire.

1.10.1 Estimation des probabilités

Comment cela fonctionne-t-il ? Tout comme un modèle de régression linéaire, un modèle de régression logistique calcule une somme pondérée des caractéristiques d'entrée (plus un terme constant), mais au lieu de fournir le résultat directement comme le fait le modèle de régression linéaire, il fournit la *logistique* du résultat :

Équation 1.15 – Probabilité estimée par le modèle de régression logistique (forme vectorielle)

$$\hat{p} = h_{\theta}(x) = \sigma(x^T \theta)$$

La fonction logistique (notée $\sigma()$) est une *fonction sigmoïde* (c'est-à-dire en forme de « S ») qui renvoie des valeurs comprises entre 0 et 1. Elle est définie par :

Équation 1.16 – Fonction logistique

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

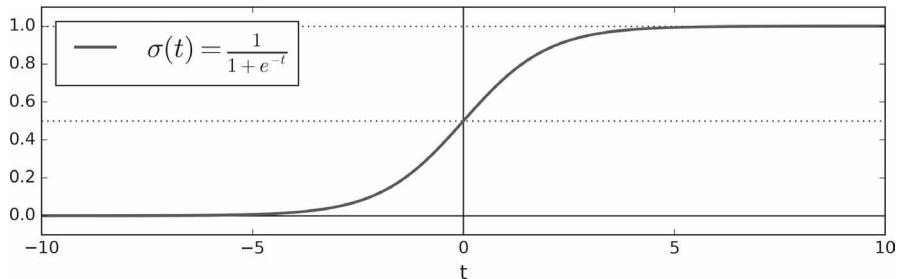


Figure 1.24 – Fonction logistique

Une fois que le modèle de régression logistique a estimé la probabilité $\hat{p} = h_{\theta}(x)$ qu'une observation x appartienne à la classe positive, il peut alors faire aisément sa prédiction \hat{y} :

Équation 1.17 – Prédiction du modèle de régression logistique

$$\hat{y} = \begin{cases} 0 & \text{si } \hat{p} < 0,5 \\ 1 & \text{si } \hat{p} \geq 0,5 \end{cases}$$

Remarquez que $\sigma(t) < 0,5$ lorsque $t < 0$, et $\sigma(t) \geq 0,5$ lorsque $t \geq 0$, c'est pourquoi un modèle de régression logistique prédit 1 si $x^T\theta$ est positif, ou 0 s'il est négatif.



Le score t est souvent appelé *logit*: ceci parce que la fonction logit, définie par $\text{logit}(p) = \log(p / (1 - p))$, est l'inverse de la fonction logistique. En fait, si vous calculez le logit de la probabilité estimée p , vous constaterez que le résultat est t . Le logit est aussi appelé *logarithme de cote* (en anglais, *log-odds*) car il représente le logarithme du rapport entre la probabilité estimée de la classe positive et la probabilité estimée de la classe négative.

1.10.2 Entraînement et fonction de coût

Nous savons maintenant comment un modèle de régression logistique estime les probabilités et effectue ses prédictions. Mais comment est-il entraîné ? L'objectif de l'entraînement consiste à définir le vecteur des paramètres θ afin que le modèle estime des probabilités élevées pour les observations positives ($y = 1$) et des probabilités basses pour les observations négatives ($y = 0$). La fonction de coût suivante traduit cette idée dans le cas d'une unique observation d'entraînement x :

Équation 1.18 – Fonction de coût pour une seule observation d’entraînement

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{si } y = 1 \\ -\log(1 - \hat{p}) & \text{si } y = 0 \end{cases}$$

Cette fonction de coût fonctionne parce que $-\log(t)$ devient très grand lorsque t s’approche de 0, par conséquent le coût sera grand si le modèle estime une probabilité proche de 0 pour une observation positive, et il sera également très grand si le modèle estime une probabilité proche de 1 pour une observation négative. Par ailleurs, $-\log(t)$ est proche de 0 lorsque t est proche de 1, et par conséquent le coût sera proche de 0 si la probabilité estimée est proche de 0 pour une observation négative ou proche de 1 pour une observation positive, ce qui est précisément ce que nous voulons.

La fonction de coût sur l’ensemble du jeu d’entraînement est le coût moyen sur l’ensemble de ses observations. Elle peut s’écrire sous forme d’une simple expression, nommée *perte logistique* (en anglais, *log loss*) :

Équation 1.19 – Fonction de coût de la régression logistique (perte logistique)

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

La mauvaise nouvelle, c’est qu’il n’existe pas de solution analytique connue pour calculer la valeur de $\boldsymbol{\theta}$ qui minimise cette fonction de coût (il n’y a pas d’équivalent de l’équation normale). La bonne nouvelle, c’est que cette fonction de coût est convexe, c’est pourquoi un algorithme de descente de gradient (comme tout autre algorithme d’optimisation) est assuré de trouver le minimum global (si le taux d’apprentissage n’est pas trop grand et si vous attendez suffisamment longtemps). La dérivée partielle de la fonction de coût par rapport au $j^{\text{ème}}$ paramètre du modèle θ_j se calcule comme suit :

Équation 1.20 – Dérivée partielle de la fonction de coût logistique

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \left(\sigma(\boldsymbol{\theta}^T \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Cette équation ressemble beaucoup à l’équation 1.7 : pour chaque observation, elle calcule l’erreur de prédiction et la multiplie par la valeur de la $j^{\text{ème}}$ variable, puis elle calcule la moyenne sur toutes les observations d’apprentissage. Une fois que vous avez le vecteur gradient contenant toutes les dérivées partielles, vous pouvez l’utiliser dans un algorithme de descente de gradient ordinaire. Et voilà, vous savez maintenant comment entraîner un modèle de régression logistique. Pour une descente de gradient stochastique, vous ne prendrez qu’une seule observation à la fois, et, pour une descente de gradient par mini-lots, vous n’utiliserez qu’un mini-lot à la fois.

1.10.3 Frontières de décision

Utilisons le jeu de données Iris pour illustrer la régression logistique : c'est un jeu de données très connu qui comporte la longueur et la largeur des sépales et des pétales de 150 fleurs d'iris de trois espèces différentes : *Iris setosa*, *Iris versicolor* et *Iris virginica* (voir la figure 1.25).

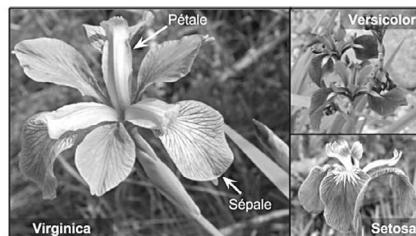


Figure 1.25 – Fleurs de trois espèces d'iris²¹

Essayons de construire un classificateur pour détecter les iris de type *virginica* en se basant uniquement sur la largeur du pétale. Tout d'abord, chargeons les données :

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:] # largeur de pétale
>>> y = (iris["target"] == 2).astype(np.int) # 1 si Iris virginica, 0 sinon
```

Maintenant, entraînons un modèle de régression logistique :

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Examinons les probabilités estimées par le modèle pour les fleurs ayant des tailles de pétales comprises entre 0 cm et 3 cm (figure 1.26)²² :

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Iris non-virginica")
# + encore un peu de code Matplotlib pour améliorer l'image
```

21. Photos reproduites à partir des pages Wikipédia correspondantes. Photo d'*Iris virginica* de Frank Mayfield (Creative Commons BY-SA 2.0), photo d'*Iris versicolor* de D. Gordon E. Robertson (Creative Commons BY-SA 3.0), photo d'*Iris setosa* dans le domaine public.

22. La fonction `reshape()` de NumPy permet d'avoir une dimension égale à -1, c'est-à-dire non spécifiée : la valeur est déduite de la longueur du tableau et des autres dimensions.

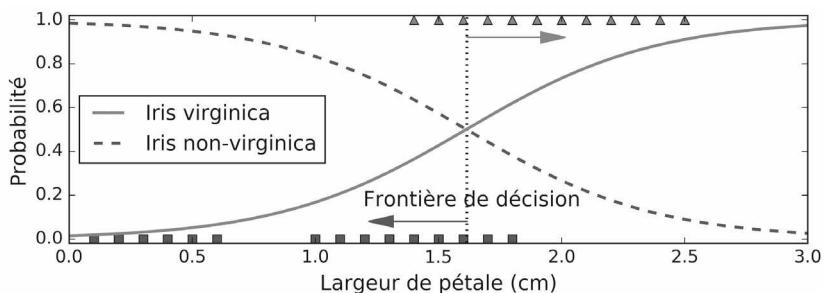


Figure 1.26 – Probabilités estimées et frontière de décision

La largeur des pétales des fleurs d'*Iris virginica* (représentées par des triangles) s'étage de 1,4 cm à 2,5 cm, tandis que les autres fleurs d'iris (représentées par des carrés) ont en général une largeur de pétales inférieure, allant de 0,1 cm à 1,8 cm. Remarquez qu'il y a un léger recouvrement. Au-dessus de 2 cm, le classificateur estime avec une grande confiance que la fleur est un *Iris virginica* (il indique une haute probabilité pour cette classe), tandis qu'en dessous de 1 cm, il estime avec une grande confiance que la fleur n'est pas un *Iris virginica* (haute probabilité pour la classe « *Iris non-virginica* »). Entre ces deux extrêmes, le classificateur n'a pas de certitude. Cependant, si vous lui demandez de prédire la classe (en utilisant la méthode `predict()` plutôt que la méthode `predict_proba()`), il renverra la classe la plus probable, et par conséquent il y a une *frontière de décision* aux alentours de 1,6 cm où les deux probabilités sont égales à 50 % : si la largeur de pétale est supérieure à 1,6 cm, le classificateur prédira que la fleur est un *Iris virginica*, sinon il prédira que ce n'en est pas un (même s'il n'est pas vraiment sûr de cela) :

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

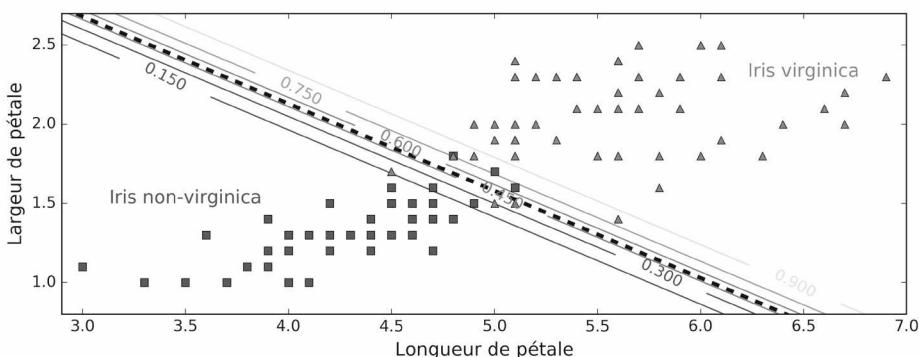


Figure 1.27 – Frontière de décision linéaire

La figure 1.27 est une autre représentation graphique du même jeu de données, obtenue cette fois-ci en croisant deux variables : la largeur des pétales et leur longueur.

Une fois entraîné, le classificateur de régression logistique peut estimer la probabilité qu'une nouvelle fleur soit un *Iris virginica* en se basant sur ces deux variables. La ligne en pointillé représente les points où le modèle estime une probabilité de 50 % : c'est la frontière de décision du modèle. Notez que cette frontière est linéaire²³. Chaque ligne parallèle matérialise les points où le modèle estime une probabilité donnée, de 15 % (en bas à gauche), jusqu'à 90 % (en haut à droite). D'après le modèle, toutes les fleurs au-dessus de la ligne en haut à droite ont plus de 90 % de chances d'être des *Iris virginica*.

Tout comme les autres modèles linéaires, les modèles de régression logistique peuvent être régularisés à l'aide de pénalités ℓ_1 ou ℓ_2 . En pratique, Scikit-Learn ajoute une pénalité ℓ_2 par défaut.



L'hyperparamètre contrôlant l'importance de la régularisation du modèle `LogisticRegression` de Scikit-Learn n'est pas `alpha` (comme pour les autres modèles linéaires), mais son inverse : `C`. Plus la valeur de `C` est élevée, moins le modèle est régularisé.

1.10.4 Régression softmax

Le modèle de régression logistique peut être généralisé de manière à prendre en compte plusieurs classes directement, sans avoir à entraîner plusieurs classificateurs binaires puis à les combiner (comme avec la stratégie OvR, décrite plus loin dans ce chapitre). C'est ce qu'on appelle la *régression softmax*, ou *régression logistique multinaire*.

Le principe en est simple : étant donné une observation \mathbf{x} , le modèle de régression softmax calcule d'abord un score $s_k(\mathbf{x})$ pour chaque classe k , puis estime la probabilité de chaque classe en appliquant aux scores la *fonction softmax* (encore appelée *exponentielle normalisée*). La formule permettant de calculer $s_k(\mathbf{x})$ devrait vous sembler familière, car c'est la même que pour calculer la prédiction en régression linéaire :

Équation 1.21 – Score softmax pour la classe k

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(k)})^T \mathbf{x}$$

Notez que chaque classe possède son propre vecteur de paramètres $\boldsymbol{\theta}^{(k)}$. Tous ces vecteurs constituent les lignes de la *matrice de paramètres* $\boldsymbol{\Theta}$.

Une fois que vous avez calculé le score de chaque classe pour l'observation \mathbf{x} , vous pouvez estimer la probabilité \hat{p}_k que cette observation appartienne à la classe k en transformant ces scores par la fonction softmax. La fonction calcule l'exponentielle de chaque score puis les normalise (en divisant par la somme de toutes les exponentielles). Les scores sont souvent appelés logits ou log-odds (bien qu'il s'agisse en fait de log-odds non normalisés).

23. C'est l'ensemble des points \mathbf{x} tels que $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, ce qui définit une ligne droite.

Équation 1.22 – Fonction softmax

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

Dans cette équation :

- K est le nombre de classes.
- $s(\mathbf{x})$ est un vecteur contenant les scores de chaque classe pour l'observation \mathbf{x} .
- $\sigma(s(\mathbf{x}))_k$ est la probabilité estimée que l'observation \mathbf{x} appartienne à la classe k compte tenu des scores de chaque classe pour cette observation.

Tout comme le classificateur de régression logistique, le classificateur de régression softmax prédit la classe ayant la plus forte probabilité estimée (c'est-à-dire simplement la classe ayant le plus haut score) :

Équation 1.23 – Prédiction du classificateur de régression softmax

$$\hat{y} = \underset{k}{\operatorname{argmax}} \sigma(s(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}} s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \left((\boldsymbol{\theta}^{(k)})^T \mathbf{x} \right)$$

L'opérateur argmax renvoie la valeur d'une variable qui maximise une fonction. Dans cette équation, il renvoie la valeur de k qui maximise la probabilité estimée $\sigma(s(\mathbf{x}))_k$.



Le classificateur de régression softmax ne prédit qu'une classe à la fois (c'est-à-dire qu'il est multi-classes, mais non multi-sorties) c'est pourquoi il ne doit être utilisé qu'avec des classes mutuellement exclusives, comme c'est le cas par exemple pour les différentes variétés d'une même plante. Vous ne pouvez pas l'utiliser pour reconnaître plusieurs personnes sur une image.

Maintenant que vous savez comment ce modèle estime les probabilités et fait des prédictions, intéressons-nous à l'entraînement. L'objectif est d'avoir un modèle qui estime une forte probabilité pour la classe ciblée (et par conséquent de faibles probabilités pour les autres classes). Minimiser la fonction de coût suivante, appelée *entropie croisée* (en anglais, *cross entropy*), devrait aboutir à ce résultat car le modèle est pénalisé lorsqu'il estime une faible probabilité pour la classe ciblée. On utilise fréquemment l'entropie croisée pour mesurer l'adéquation entre un ensemble de probabilités estimées d'appartenance à des classes et les classes ciblées.

Équation 1.24 – Fonction de coût d'entropie croisée

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

Dans cette équation, $y_k^{(i)}$ est la probabilité cible que la $i^{\text{ème}}$ observation appartient à la classe k . En général, cette probabilité cible est soit 1, soit 0, selon que l'observation appartient ou non à la classe k .

Notez que lorsqu'il n'y a que deux classes ($K = 2$), cette fonction de coût est équivalente à celle de la régression logistique (log loss, équation 1.19).

Entropie croisée

L'entropie croisée trouve son origine dans la théorie de l'information. Supposons que vous vouliez transmettre quotidiennement et de manière efficace des informations météorologiques. S'il y a 8 possibilités (ensoleillé, pluvieux, etc.), vous pouvez encoder chacune de ces options sur 3 bits (puisque $2^3 = 8$). Cependant, si vous pensez que le temps sera ensoleillé pratiquement tous les jours, il serait plus efficace de coder « ensoleillé » sur un seul bit, et les 7 autres options sur 4 bits (en commençant par un 1). L'entropie croisée mesure le nombre moyen de bits que vous transmettez pour chaque option. Si les suppositions que vous faites sur le temps sont parfaites, l'entropie croisée sera égale à l'entropie des données météorologiques elles-mêmes (à savoir leur caractère imprévisible intrinsèque). Mais si vos suppositions sont fausses (p. ex. s'il pleut souvent), l'entropie croisée sera supérieure, d'un montant supplémentaire appelé *divergence de Kullback-Leibler*.

L'entropie croisée entre deux distributions de probabilités p et q est définie par $H(p,q) = -\sum_x p(x) \log q(x)$ (du moins lorsque les distributions sont discrètes). Pour plus de détails, consultez ma vidéo sur le sujet : <http://homl.info/xentropy>.

Le vecteur gradient par rapport à $\Theta^{(k)}$ de cette fonction de coût se définit comme suit :

Équation 1.25 – Vecteur gradient de l'entropie croisée pour la classe k

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Vous pouvez maintenant calculer le vecteur gradient de chaque classe, puis utiliser une descente de gradient (ou un autre algorithme d'optimisation) pour trouver la matrice des paramètres Θ qui minimise la fonction de coût.

Utilisons la régression softmax pour répartir les fleurs d'iris en trois classes. La classe `LogisticRegression` de Scikit-Learn utilise par défaut une stratégie un contre le reste (*one-vs-the-rest*, OvR, également appelée *one-vs-all*, OvA) lorsque vous l'entraînez sur plus de deux classes, c'est-à-dire qu'elle entraîne en réalité un classificateur binaire pour chaque classe, pour la distinguer des autres classes. Elle réalise ensuite ses prédictions en choisissant pour chaque observation la classe dont le classificateur donne le score le plus élevé. Toutefois, vous pouvez donner à l'hyperparamètre `multi_class` la valeur "multinomial" pour la transformer en régression softmax. Vous devez alors spécifier également un `solver` compatible avec la régression softmax, comme "`lbfgs`" par exemple (pour plus de détails, reportez-vous à la documentation Scikit-Learn). Elle applique aussi par défaut une régularisation ℓ_2 , que vous pouvez contrôler à l'aide de l'hyperparamètre `C` :

```
X = iris["data"][:, (2, 3)] # longueur, largeur de pétales
y = iris["target"]
```

```
softmax_reg = LogisticRegression(multi_class="multinomial",
                                  solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Par conséquent, la prochaine fois que vous trouverez un iris ayant des pétales de 5 cm de long et de 2 cm de large et que vous demanderez à votre modèle de vous dire de quel type d'iris il s'agit, il vous répondra *Iris virginica* (classe 2) avec une probabilité de 94,2 % (ou *Iris versicolor* avec une probabilité de 5,8 %) :

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

La figure 1.28 présente les frontières de décision qui en résultent, matérialisées par les couleurs d'arrière-plan : vous remarquerez que les frontières de décision entre les classes prises deux à deux sont linéaires. La figure présente aussi les probabilités pour la classe *Iris versicolor*, représentées par des courbes (ainsi, la ligne étiquetée 0.450 représente la frontière des 45 % de probabilité). Notez que le modèle peut prédire une classe ayant une probabilité estimée inférieure à 50 %. Ainsi, au point d'intersection de toutes les frontières de décision, toutes les classes ont une même probabilité de 33 %.

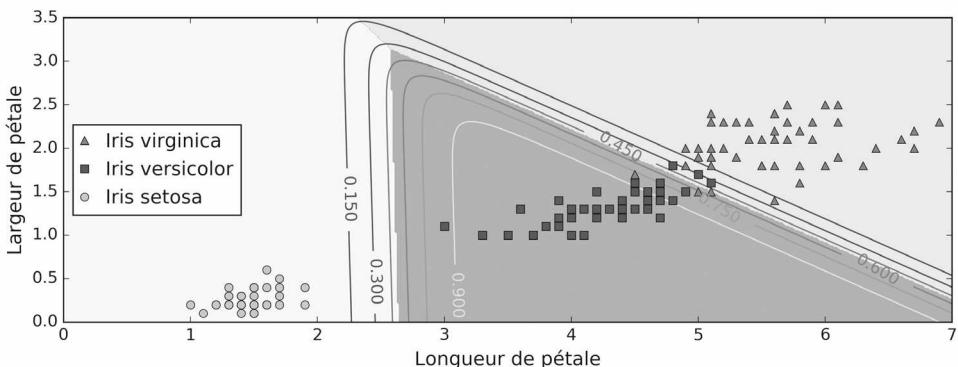


Figure 1.28 – Frontières de décision d'une régression softmax

1.11 EXERCICES

- Quel algorithme d'entraînement de régression linéaire pouvez-vous utiliser si vous avez un jeu d'entraînement comportant des millions de variables ?
- Supposons que les variables de votre jeu d'entraînement aient des échelles très différentes. Quels algorithmes peuvent en être affectés, et comment ? Comment pouvez-vous y remédier ?
- Une descente de gradient peut-elle se bloquer sur un minimum local lorsque vous entraînez un modèle de régression logistique ?

4. Tous les algorithmes de descente de gradient aboutissent-ils au même modèle si vous les laissez s'exécuter suffisamment longtemps ?
5. Supposons que vous utilisez une descente de gradient ordinaire, en représentant graphiquement l'erreur de validation à chaque cycle (ou époque) : si vous remarquez que l'erreur de validation augmente régulièrement, que se passe-t-il probablement ? Comment y remédier ?
6. Est-ce une bonne idée d'arrêter immédiatement une descente de gradient par mini-lots lorsque l'erreur de validation augmente ?
7. Parmi les algorithmes de descente de gradient que nous avons étudiés, quel est celui qui arrive le plus vite à proximité de la solution optimale ? Lequel va effectivement converger ? Comment pouvez-vous faire aussi converger les autres ?
8. Supposons que vous utilisez une régression polynomiale. Après avoir imprimé les courbes d'apprentissage, vous remarquez qu'il y a un écart important entre l'erreur d'entraînement et l'erreur de validation. Que se passe-t-il ? Quelles sont les trois manières de résoudre le problème ?
9. Supposons que vous utilisez une régression ridge. Vous remarquez que l'erreur d'entraînement et l'erreur de validation sont à peu près identiques et assez élevées : à votre avis, est-ce le biais ou la variance du modèle qui est trop élevé(e) ? Devez-vous accroître l'hyperparamètre de régularisation α ou le réduire ?
10. Qu'est-ce qui pourrait vous inciter à choisir une régression...
 - ridge plutôt qu'une simple (c.-à-d. sans régularisation) ?
 - lasso plutôt que ridge ?
 - elastic net plutôt que lasso ?
11. Supposons que vous vouliez classer des photos en extérieur/intérieur et jour/nuit. Devez-vous utiliser deux classificateurs de régression logistique ou un classificateur de régression softmax ?
12. Implémentez une descente de gradient ordinaire avec arrêt précoce pour une régression softmax (sans utiliser Scikit-Learn).

Les solutions de ces exercices sont données à l'annexe A.

2

Introduction aux réseaux de neurones artificiels avec Keras

Les oiseaux nous ont donné l'envie de voler, la bardane est à l'origine du Velcro, et bien d'autres inventions se sont inspirées de la nature. Il est donc naturel de s'inspirer du fonctionnement du cerveau pour construire une machine intelligente. Voilà la logique à l'origine des *réseaux de neurones artificiels* (RNA) : un RNA est un modèle d'apprentissage automatique inspiré des réseaux de neurones biologiques que l'on trouve dans notre cerveau. Cependant, même si les avions ont les oiseaux pour modèle, ils ne battent pas des ailes. De façon comparable, les RNA sont progressivement devenus assez différents de leurs cousins biologiques. Certains chercheurs soutiennent même qu'il faudrait éviter totalement l'analogie biologique, par exemple en disant *unité* au lieu de *neurone*, de peur que nous ne limitions notre créativité aux systèmes biologiquement plausibles²⁴.

Les RNA sont au cœur de l'apprentissage profond. Ils sont polyvalents, puissants et extensibles, ce qui les rend parfaitement adaptés aux tâches d'apprentissage automatique extrêmement complexes, comme la classification de milliards d'images (p. ex., Google Images), la reconnaissance vocale (p. ex., Apple Siri), la recommandation de vidéos auprès de centaines de millions d'utilisateurs (p. ex., YouTube) ou l'apprentissage nécessaire pour battre le champion du monde du jeu de go (AlphaGo de DeepMind).

La première partie de ce chapitre est une introduction aux réseaux de neurones artificiels, en commençant par une description rapide des toutes premières architectures de RNA. Nous présenterons ensuite les *perceptrons multicouches* (PMC), qui sont largement employés aujourd'hui (d'autres architectures seront détaillées dans

24. Nous pouvons garder le meilleur des deux mondes en restant ouverts aux sources d'inspiration naturelles, sans craindre de créer des modèles biologiques irréalistes, tant qu'ils fonctionnent bien.

les chapitres suivants). La deuxième partie expliquera comment mettre en œuvre des réseaux de neurones à l'aide de l'API Keras. Il s'agit d'une API de haut niveau, simple et très bien conçue, qui permet de construire, d'entraîner, d'évaluer et d'exécuter des réseaux de neurones. Mais ne vous y trompez pas, malgré sa simplicité, elle est suffisamment expressive et souple pour construire une large diversité d'architectures de réseaux de neurones. En réalité, elle suffira probablement à la plupart de vos utilisations. Par ailleurs, si jamais vous aviez besoin d'une plus grande souplesse encore, vous avez la possibilité d'écrire vos propres composants Keras à l'aide de son API de bas niveau (voir le chapitre 4).

Commençons par un petit retour en arrière afin de comprendre comment sont nés les réseaux de neurones artificiels !

2.1 DU BIOLOGIQUE À L'ARTIFICIEL

Les RNA existent depuis déjà un bon moment. Ils ont été décrits pour la première fois en 1943 par le neurophysiologiste Warren McCulloch et le mathématicien Walter Pitts. Dans leur article fondateur²⁵, ils ont présenté un modèle informatique simplifié du fonctionnement combiné des neurones biologiques du cerveau des animaux dans le but de résoudre des calculs complexes à l'aide de la *logique propositionnelle*. Il s'agissait de la première architecture de réseaux de neurones artificiels. Depuis lors, de nombreuses autres ont été imaginées.

Les premiers succès des RNA ont laissé croire qu'il serait rapidement possible de discuter avec des machines véritablement intelligentes. Dans les années 1960, quand il est devenu clair que cette promesse ne serait pas tenue (tout au moins pas avant un certain temps), les financements ont trouvé d'autres destinations et les RNA sont entrés dans une longue période sombre. Le début des années 1980 a vu renaître l'intérêt pour le *connexionnisme* (l'étude des réseaux de neurones), lorsque de nouvelles architectures ont été inventées et que de meilleures techniques d'apprentissage ont été développées. Mais les progrès étaient lents et, dans les années 1990, d'autres méthodes d'apprentissage automatique puissantes ont été proposées, comme les machines à vecteurs de support (SVM)²⁶. Elles semblaient offrir de meilleurs résultats et se fondaient sur des bases théoriques plus solides que les RNA. Une fois encore, l'étude des réseaux de neurones est retournée dans l'ombre.

Nous assistons à présent à un regain d'intérêt pour les RNA. Va-t-il s'évaporer comme les précédents ? Il y a quelques bonnes raisons de croire que celui-ci sera différent et que le regain d'intérêt pour les RNA aura un impact bien plus profond sur nos vies :

- Il existe des données en quantités absolument gigantesques pour entraîner les RNA, et ils sont souvent bien meilleurs que les autres techniques d'apprentissage automatique sur les problèmes larges et complexes.

25. Warren S. McCulloch et Walter Pitts, « A Logical Calculus of the Ideas Immanent in Nervous Activity », *The Bulletin of Mathematical Biology*, 5, n° 4 (1943), 115-113 : <https://homl.info/43>.

26. Voir le chapitre 5 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

- L'extraordinaire augmentation de la puissance de calcul depuis les années 1990 rend aujourd'hui possible l'entraînement de grands réseaux de neurones en un temps raisonnable. Cela est en partie dû à la loi de Moore (le nombre de composants dans les circuits intégrés a doublé tous les deux ans environ au cours des 50 dernières années), mais également à l'industrie du jeu qui a stimulé la production par millions de cartes graphiques équipées de GPU puissants. Par ailleurs, grâce aux plateformes de Cloud, tout le monde a accès à cette puissance.
- Les algorithmes d'entraînement ont également été améliorés. Pour être honnête, ils ne sont que légèrement différents de ceux des années 1990, mais ces ajustements relativement limités ont eu un impact extrêmement positif.
- Certaines limites théoriques des RNA se sont avérées plutôt bénignes dans la pratique. Par exemple, de nombreuses personnes pensaient que les algorithmes d'entraînement des RNA étaient condamnés car ils resteraient certainement bloqués dans des optima locaux, alors que ces cas se sont révélés plutôt rares en pratique (et, lorsqu'ils surviennent, ces optima locaux sont en général assez proches de l'optimum global).
- Les RNA semblent être entrés dans un cercle vertueux de financement et de progrès. Des produits incroyables fondés sur les RNA font régulièrement la une de l'actualité. Les RNA attirent ainsi de plus en plus l'attention, et donc les fonds. Cela conduit à de nouvelles avancées et encore plus de produits étonnantes.

2.1.1 Neurones biologiques

Avant d'aborder les neurones artificiels, examinons rapidement un neurone biologique (voir la figure 2.1). Il s'agit d'une cellule à l'aspect inhabituel que l'on trouve principalement dans les cerveaux des animaux. Elle est constituée d'un *corps cellulaire*, qui comprend le noyau et la plupart des éléments complexes de la cellule, ainsi que de nombreux prolongements appelés *dendrites* et un très long prolongement appelé *axone*. L'axone peut être juste un peu plus long que le corps cellulaire, mais aussi jusqu'à des dizaines de milliers de fois plus long. Près de son extrémité, il se décompose en plusieurs ramifications appelées *télodendrons*, qui se terminent par des structures minuscules appelées *synapses terminales* (ou simplement *synapses*) et reliées aux dendrites ou directement aux corps cellulaires d'autres neurones²⁷. Les neurones biologiques produisent de courtes impulsions électriques appelées *potentiels d'action* (PA, ou *signaux*), qui voyagent le long des axones et déclenchent, au niveau des synapses, la libération de signaux chimiques appelés *neurotransmetteurs*. Lorsqu'un neurone reçoit en quelques millisecondes un nombre suffisant de ces neurotransmetteurs, il déclenche ses propres impulsions électriques (en réalité cela dépend des neurotransmetteurs, car certains d'entre eux inhibent ce déclenchement).

27. En réalité, elles ne sont pas reliées, juste suffisamment proches pour échanger très rapidement des signaux chimiques.

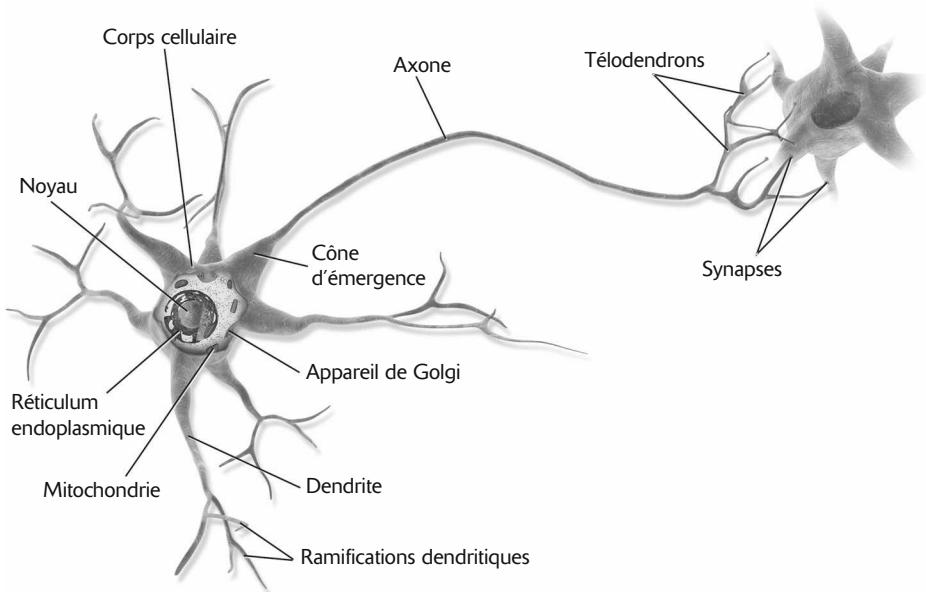


Figure 2.1 – Un neurone biologique²⁸

Chaque neurone biologique semble donc se comporter de façon relativement simple, mais ils sont organisés en un vaste réseau de milliards de neurones, chacun étant en général relié à des milliers d'autres. Des calculs extrêmement complexes peuvent être réalisés par un réseau de neurones relativement simples, de la même manière qu'une fourmilière complexe peut être construite grâce aux efforts combinés de simples fourmis. L'architecture des réseaux de neurones biologiques²⁹ (RNB) fait encore l'objet d'une recherche active, mais certaines parties du cerveau ont été cartographiées et il semble que les neurones soient souvent organisés en couches successives, notamment dans le cortex cérébral (la couche externe de notre cerveau), comme l'illustre la figure 2.2.

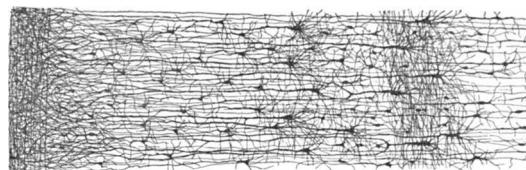


Figure 2.2 – Couches multiples dans un réseau de neurones biologiques (cortex humain)³⁰

28. Image de Bruce Blaus (Creative Commons 3.0, <https://creativecommons.org/licenses/by/3.0/>), source <https://en.wikipedia.org/wiki/Neuron>.

29. Dans le contexte de l'apprentissage automatique, l'expression *réseau de neurones* fait en général référence non pas aux RNB mais aux RNA.

30. Dessin de la stratification corticale par S. Ramon y Cajal (domaine public), source https://en.wikipedia.org/wiki/Cerebral_cortex.

2.1.2 Calculs logiques avec des neurones

McCulloch et Pitts ont proposé un modèle très simple de neurone biologique, c'est-à-dire le premier *neurone artificiel*: il présente une ou plusieurs entrées binaires (active/inactive) et une sortie binaire. Le neurone artificiel active simplement sa sortie lorsque le nombre de ses entrées actives dépasse un certain seuil. Ils ont montré que, malgré la simplicité de ce modèle, il est possible de construire un réseau de neurones artificiels qui calcule n'importe quelle proposition logique. Par exemple, nous pouvons construire des réseaux de neurones artificiels qui effectuent différents calculs logiques (voir la figure 2.3), en supposant qu'un neurone est activé lorsqu'au moins deux de ses entrées le sont.

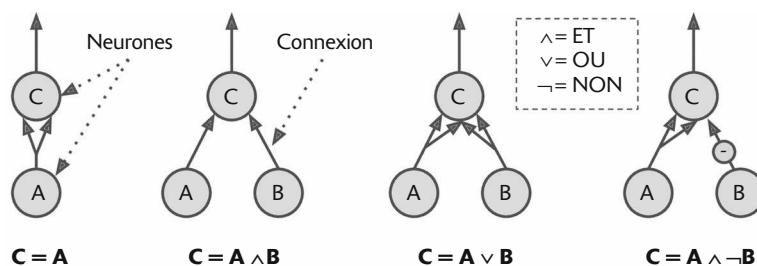


Figure 2.3 – Réseaux de neurones artificiels réalisant des calculs logiques élémentaires

Voyons ce que réalisent ces réseaux :

- Le premier réseau à gauche correspond à la fonction identité. Si le neurone A est activé, alors le neurone C l'est également (puisque il reçoit deux signaux d'entrée du neurone A). De la même façon, si le neurone A est désactivé, le neurone C l'est aussi.
- Le deuxième réseau réalise un ET logique. Le neurone C est activé uniquement lorsque les neurones A et B sont activés (un seul signal d'entrée ne suffit pas à activer le neurone C).
- Le troisième neurone effectue un OU logique. Le neurone C est activé si le neurone A ou le neurone B est activé (ou les deux).
- Enfin, si nous supposons qu'une connexion d'entrée peut inhiber l'activité du neurone (ce qui est le cas avec les neurones biologiques), alors le quatrième réseau met en œuvre une proposition logique un peu plus complexe. Le neurone C est activé uniquement si le neurone A est actif et si le neurone B est inactif. Si le neurone A est actif en permanence, nous obtenons alors un NON logique : le neurone C est actif lorsque le neurone B est inactif, et inversement.

Vous pouvez imaginer comment ces réseaux peuvent être combinés pour calculer des expressions logiques complexes (voir les exercices à la fin de ce chapitre).

2.1.3 Le perceptron

Le *perceptron*, inventé en 1957 par Frank Rosenblatt, est l'une des architectures de RNA les plus simples. Il se fonde sur un neurone artificiel légèrement différent (voir la figure 2.4), appelé *unité logique à seuil* (TLU, *Threshold Logic Unit*) ou parfois *unité linéaire à seuil* (LTU, *Linear Threshold Unit*). Les entrées et la sortie sont à présent des nombres (à la place de valeurs binaires, actif/inactif) et chaque connexion en entrée possède un poids. Le TLU calcule une somme pondérée des entrées ($z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{x}^T \mathbf{w}$), puis il applique une *fonction échelon* (*step*) à cette somme et produit le résultat : $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{x}^T \mathbf{w})$.

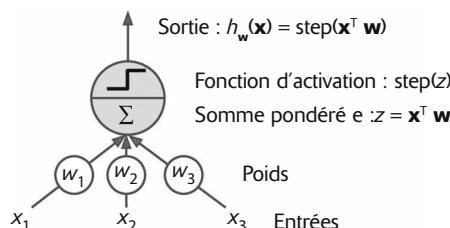


Figure 2.4 – Une unité logique à seuil: un neurone artificiel qui calcule une somme pondérée de ses entrées, puis qui applique une fonction échelon

Dans les perceptrons, la fonction échelon la plus répandue est la *fonction de Heaviside* (voir l'équation 2.1). La fonction signe est parfois utilisée à la place.

Équation 2.1 – Fonctions échelon répandues dans les perceptrons
(en supposant que le seuil soit égal à 0)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{si } z < 0 \\ 0 & \text{si } z = 0 \\ +1 & \text{si } z > 0 \end{cases}$$

Un seul TLU peut être employé pour une classification binaire linéaire simple. Il calcule une combinaison linéaire des entrées et, si le résultat dépasse un seuil, présente en sortie la classe positive, sinon la classe négative (tout comme un classificateur à régression logistique ou un classificateur SVM linéaire). Par exemple, nous pouvons utiliser un seul LTU pour classer les iris en fonction de la longueur et de la largeur des pétales (en ajoutant une caractéristique de terme constant supplémentaire $x_0 = 1$). L'entraînement d'un TLU consiste à trouver les valeurs appropriées pour les poids w_0 , w_1 et w_2 (l'algorithme d'entraînement sera présenté plus loin).

Un perceptron est constitué d'une seule couche de TLU³¹, chaque TLU étant connecté à toutes les entrées. Lorsque tous les neurones d'une couche sont connectés à chaque neurone de la couche précédente (c'est-à-dire les neurones d'entrée), la

31. Le terme *perceptron* est parfois employé pour désigner un minuscule réseau constitué d'un seul TLU.

couche est une *couche intégralement connectée*, ou *couche dense*. Les entrées du perceptron sont transmises à des neurones intermédiaires particuliers appelés *neurones d'entrée* : ils se contentent de sortir l'entrée qui leur a été fournie. L'ensemble des neurones d'entrée forment la *couche d'entrée*. Là encore, une caractéristique de biais est souvent ajoutée ($x_0 = 1$). Elle est généralement représentée à l'aide d'un neurone de type particulier appelé *neurone de terme constant*, ou *neurone de biais*, dont la sortie est toujours 1. Un perceptron doté de deux entrées et de trois sorties est représenté à la figure 2.5. Il est capable de classer des instances dans trois classes binaires différentes, ce qui en fait un classificateur multi-sorties.

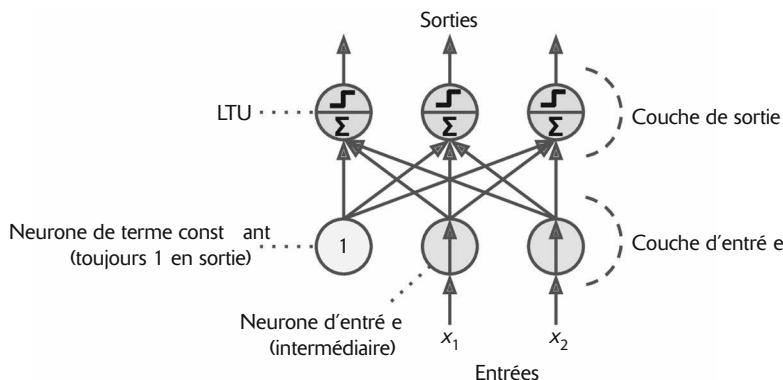


Figure 2.5 – Architecture perceptron avec deux neurones d'entrée, un neurone de terme constant et trois neurones de sortie

Grâce à l'algèbre linéaire, l'équation 2.2 permet de calculer efficacement les sorties d'une couche de neurones artificiels pour plusieurs instances à la fois.

Équation 2.2 – Calculer les sorties d'une couche intégralement connectée

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

Dans cette équation :

- \mathbf{X} représente la matrice des caractéristiques d'entrée. Elle comprend une ligne par instance et une colonne par caractéristique.
- La matrice des poids \mathbf{W} contient tous les poids des connexions, excepté ceux des neurones de terme constant. Elle comprend une ligne par neurone d'entrée et une colonne par neurone artificiel de la couche.
- Le vecteur de termes constants \mathbf{b} contient tous les poids des connexions entre le neurone de terme constant et les neurones artificiels. Il comprend un terme constant par neurone artificiel.
- La fonction ϕ est la *fonction d'activation* : lorsque les neurones artificiels sont des TLU, il s'agit d'une fonction échelon (nous verrons d'autres fonctions d'activation plus loin).

Comment entraîne-t-on un perceptron ? L'algorithme d'entraînement du perceptron proposé par Rosenblatt se fonde largement sur la *règle de Hebb*. Dans son ouvrage *The Organization of Behavior* publié en 1949 (Wiley), Donald Hebb suggérait que, si un neurone biologique déclenche souvent un autre neurone, alors la connexion entre ces deux neurones se renforce. Cette idée a été ensuite résumée par la phrase de Siegrid Löwel : « *cells that fire together, wire together* », ou « les neurones qui s'activent en même temps se lient entre eux » ; autrement dit, le poids d'une connexion entre deux neurones tend à augmenter lorsqu'ils s'activent simultanément. Cette règle est devenue plus tard la règle de Hebb (ou *apprentissage hebbien*). Les perceptrons sont entraînés à partir d'une variante de cette règle qui prend en compte l'erreur effectuée par le réseau lorsqu'il réalise une prédiction ; la règle d'apprentissage du perceptron renforce les connexions qui aident à réduire l'erreur. Plus précisément, le perceptron reçoit une instance d'entraînement à la fois et, pour chacune, effectue ses prédictions. Pour chaque neurone de sortie qui produit une prédiction erronée, il renforce les poids des connexions liées aux entrées qui auraient contribué à la prédiction juste. La règle est illustrée à l'équation 2.3.

Équation 2.3 – Règle d'apprentissage du perceptron (mise à jour du poids)

$$w_{i,j}^{(\text{étape suivante})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

Dans cette équation :

- $w_{i,j}$ correspond au poids de la connexion entre le $i^{\text{ème}}$ neurone d'entrée et le $j^{\text{ème}}$ neurone de sortie ;
- x_i est la $i^{\text{ème}}$ valeur d'entrée de l'instance d'entraînement courante ;
- \hat{y}_j est la sortie du $j^{\text{ème}}$ neurone de sortie pour l'instance d'entraînement courante ;
- y_j est la sortie souhaitée pour le $j^{\text{ème}}$ neurone de sortie pour l'instance d'entraînement courante ;
- η est le taux d'apprentissage.

Puisque la frontière de décision de chaque neurone de sortie est linéaire, les perceptrons sont incapables d'apprendre des motifs complexes (tout comme les classificateurs à régression logistique). Cependant, si les instances d'entraînement peuvent être séparées de façon linéaire, Rosenblatt a montré que l'algorithme converge forcément vers une solution³². Il s'agit du *théorème de convergence du perceptron*.

Scikit-Learn fournit une classe `Perceptron` qui implémente un réseau de TLU. Nous pouvons l'employer très facilement, par exemple sur le jeu de données Iris (présenté au chapitre 1) :

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # longueur et largeur du pétales
```

32. Notez que cette solution n'est pas unique : lorsque les points peuvent être séparés de façon linéaire, il existe une infinité d'hyperplans qui peuvent les séparer.

```

y = (iris.target == 0).astype(np.int) # Iris setosa ?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])

```

Vous l'avez peut-être remarqué, l'algorithme d'entraînement du perceptron ressemble énormément à la descente de gradient stochastique. En réalité, l'utilisation de la classe `Perceptron` de Scikit-Learn équivaut à employer un `SGDClassifier` avec les hyperparamètres suivants: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (le taux d'apprentissage) et `penalty=None` (aucune régularisation).

Contrairement aux classificateurs à régression logistique, les perceptrons ne produisent pas en sortie une probabilité de classe. Ils se contentent de prédictions basées sur un seuil figé. Voilà l'une des bonnes raisons de préférer la régression logistique aux perceptrons.

Dans leur monographie de 1969 intitulée *Perceptrons*, Marvin Minsky et Seymour Papert ont dévoilé plusieurs faiblesses importantes des perceptrons, notamment le fait qu'ils sont incapables de résoudre certains problèmes triviaux, comme le problème de classification du OU exclusif (XOR) (voir la partie gauche de la figure 2.6). Bien entendu, cela reste vrai pour n'importe quel modèle de classification linéaire, comme les classificateurs à régression logistique, mais les chercheurs en attendaient beaucoup plus des perceptrons et la déception a été si profonde que certains ont totalement écarté les réseaux de neurones au profit d'autres problèmes de plus haut niveau, comme la logique, la résolution de problèmes et les recherches.

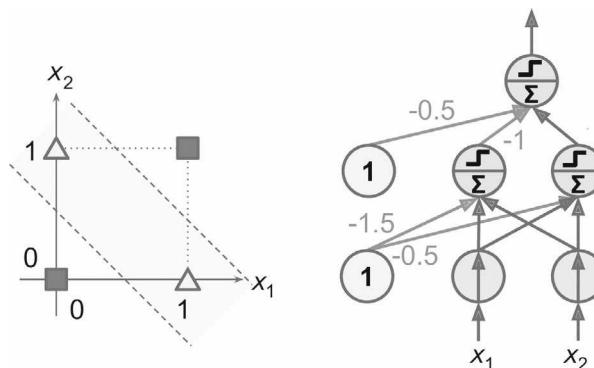


Figure 2.6 – Problème de classification OU exclusif et PMC pour le résoudre

Il est cependant possible de lever certaines limites des perceptrons en empilant plusieurs perceptrons. Le RNA résultant est appelé *perceptron multicouche* (PMC). Un PMC est capable de résoudre le problème du OU exclusif, comme nous pouvons le vérifier en calculant la sortie du PMC représenté en partie droite de la figure 2.6. Avec les entrées (0; 0) ou (1; 1), le réseau produit la sortie 0. Avec les entrées (0; 1)

ou $(1; 0)$, il génère 1. Toutes les connexions ont un poids égal à 1, à l'exception des quatre connexions dont le poids est indiqué. Essayez de vérifier que ce réseau résoud bien le problème du OU exclusif !

2.1.4 Perceptron multicouche et rétropropagation

Un PMC est constitué d'une *couche d'entrée* (de transfert uniquement), d'une ou plusieurs couches de TLU appelées *couches cachées* et d'une dernière couche de TLU appelée *couche de sortie* (voir la figure 2.7). Les couches proches de la couche d'entrée sont généralement appelées les *couches basses*, tandis que celles proches des sorties sont les *couches hautes*. Chaque couche, à l'exception de la couche de sortie, comprend un neurone de terme constant et elle est intégralement reliée à la suivante.

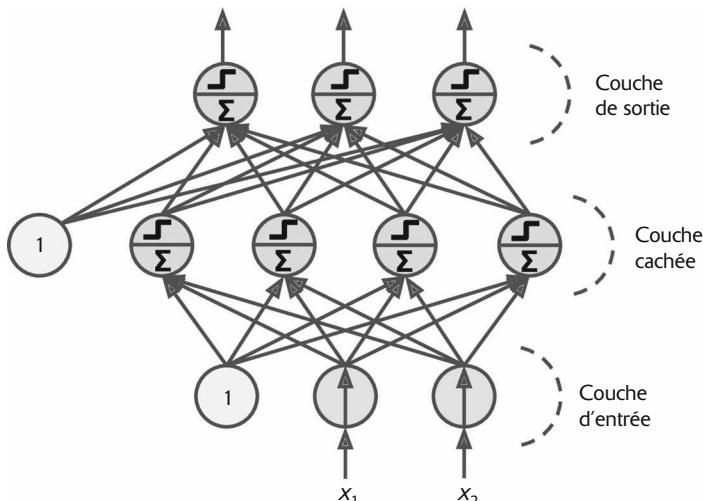


Figure 2.7 – Perceptron multicouche avec deux entrées, une couche cachée de quatre neurones et trois neurones de sortie (les neurones de terme constant sont représentés, mais ils sont habituellement implicites)



Puisque le signal va dans une seule direction (des entrées vers les sorties), cette architecture est un exemple de *réseau de neurones non bouclé* (FNN, *Feedforward Neural Network*).

Lorsqu'un RNA possède un grand nombre de couches cachées³³, on parle de *réseau de neurones profond* (RNP), ou, en anglais, *Deep Neural Network* (DNN). Le domaine

33. Dans les années 1990, un RNA possédant plus de deux couches cachées était considéré profond. Aujourd'hui, il n'est pas rare de rencontrer des RNA avec des dizaines de couches, voire des centaines. La définition de « profond » est donc relativement floue.

du Deep Learning étudie les RNP et plus généralement les modèles qui comprennent des piles de calcul profondes. Toutefois, de nombreuses personnes parlent de Deep Learning dès que des réseaux de neurones entrent en jeu (même s'ils manquent de profondeur).

Pendant de nombreuses années, les chercheurs se sont efforcés de trouver une manière d'entraîner les PMC, sans succès. En 1986, David Rumelhart, Geoffrey Hinton et Ronald Williams ont publié un article révolutionnaire³⁴ dans lequel ils introduisirent un algorithme d'entraînement à *rétropropagation*, toujours employé aujourd'hui. En bref, il s'agit d'une descente de gradient³⁵ utilisant une technique efficace de calcul automatique des gradients³⁶: en seulement deux passages dans le réseau (un avant, un en arrière), l'algorithme de rétropropagation est capable de calculer le gradient de l'erreur du réseau en lien avec chaque paramètre individuel du modèle. Autrement dit, il peut déterminer l'ajustement à appliquer à chaque poids de connexion et à chaque terme constant pour réduire l'erreur. Après avoir obtenu les gradients, il réalise simplement une étape de descente de gradient classique. L'intégralité du processus est répétée jusqu'à la convergence du réseau vers la solution.



Le calcul automatique des gradients est appelé *différentiation automatique* ou *autodiff*. Il existe différentes techniques de différentiation automatique, chacune présentant des avantages et des inconvénients. Celle employée pour la rétropropagation se nomme *différentiation automatique en mode inverse*. Elle est rapide et précise, et convient parfaitement lorsque la fonction de différentiation possède de nombreuses variables (par exemple des poids de connexions) et peu de sorties (par exemple une perte). L'annexe D décrit plus en détail la différentiation automatique.

Examinons en détail cet algorithme :

- Il traite un mini-lot à la fois (par exemple de 32 instances chacun) et passe à plusieurs reprises sur le jeu d'entraînement complet. Chaque passe est appelée *époque* (*epoch*)
- Chaque mini-lot est transmis à la couche d'entrée du réseau, qui l'envoie à la première couche cachée. L'algorithme calcule ensuite la sortie de chaque neurone dans cette couche (pour chaque instance du mini-lot). Le résultat est transmis à la couche suivante, sa sortie est déterminée et passée à la couche suivante. Le processus se répète jusqu'à la sortie de la dernière couche, la couche de sortie. Il s'agit de la *passe en avant*, comme pour réaliser les prédictions, excepté que tous les résultats intermédiaires sont conservés car ils sont nécessaires à la passe vers l'arrière.

34. David Rumelhart *et al.*, *Learning Internal Representations by Error Propagation* (Defense Technical Information Center technical report, 1985) : <https://homl.info/44>.

35. Voir le chapitre 1.

36. En réalité, cette technique a été inventée à plusieurs reprises de façon indépendante par plusieurs chercheurs dans des domaines variés, à commencer par Paul Werbos en 1974.

- Il mesure ensuite l'erreur de sortie du réseau (il utilise une fonction de perte qui compare la sortie souhaitée et la sortie réelle du réseau, et qui retourne une mesure de l'erreur).
- Puis il détermine dans quelle mesure chaque connexion de sortie a contribué à l'erreur. Il procède de façon analytique en appliquant la *règle de la chaîne* (probablement la règle essentielle des calculs), qui donne rapidité et précision à cette étape.
- L'algorithme poursuit en mesurant la portion de ces contributions à l'erreur qui revient à chaque connexion de la couche inférieure, de nouveau avec la règle de la chaîne et *bis repetita* jusqu'à la couche d'entrée. Cette passe en arrière mesure efficacement le gradient d'erreur sur tous les poids des connexions du réseau en rétropropageant le gradient d'erreur dans le réseau (d'où le nom de l'algorithme).
- L'algorithme se termine par une étape de descente de gradient de façon à ajuster tous les poids des connexions dans le réseau, en utilisant les gradients d'erreurs calculés précédemment.

Cet algorithme est si important qu'il est bon de le résumer à nouveau : pour chaque instance d'entraînement, l'algorithme de rétropropagation commence par effectuer une prédiction (passe vers l'avant), mesure l'erreur, traverse chaque couche en arrière pour mesurer la contribution à l'erreur de chaque connexion (passe vers l'arrière) et termine en ajustant légèrement les poids des connexions de manière à réduire l'erreur (étape de descente de gradient).



Il est important d'initialiser de façon aléatoire les poids des connexions pour toutes les couches cachées, sinon l'entraînement va échouer. Par exemple, si tous les poids et les termes constants sont fixés à zéro, alors tous les neurones d'une couche donnée seront identiques, la rétropropagation les affectera exactement de la même manière et ils resteront donc égaux. Autrement dit, malgré les centaines de neurones de chaque couche, notre modèle fonctionnera comme si chaque couche ne contenait qu'un seul neurone et ne sera donc pas opérationnel. À la place, si l'initialisation est aléatoire, on *brise la symétrie* et permet à la rétropropagation d'entraîner une équipe variée de neurones.

Pour que cet algorithme fonctionne correctement, les auteurs ont apporté une modification essentielle à l'architecture du PMC. Ils ont remplacé la fonction échelon par la fonction logistique, $\sigma(z) = 1 / (1 + \exp(-z))$. Ce changement est fondamental, car la fonction échelon comprend uniquement des segments plats et il n'existe donc aucun gradient à exploiter (la descente de gradient ne peut pas se déplacer sur une surface plane). En revanche, la fonction logistique possède une dérivée non nulle en tout point, ce qui permet à la descente de gradient de progresser à chaque étape. L'algorithme de rétropropagation peut être employé avec d'autres *fonctions d'activation*, à la place de la fonction logistique. En voici deux autres souvent utilisées :

- La fonction *tangente hyperbolique* $\tanh(z) = 2\sigma(2z) - 1$:
À l'instar de la fonction logistique, cette fonction d'activation a une forme de « S », est continue et dérivable, mais ses valeurs de sortie se trouvent dans

la plage -1 à 1 (à la place de 0 à 1 pour la fonction logistique), ce qui tend à rendre la sortie de chaque couche plus ou moins centrée sur zéro au début de l'entraînement. La convergence s'en trouve souvent accélérée.

- La fonction ReLU (*Rectified Linear Unit*) : $\text{ReLU}(z) = \max(0, z)$:

Elle est continue, mais non dérivable en $z = 0$ (la pente change brutalement, ce qui fait rebondir la descente de gradient de part et d'autre de ce point de rupture) et sa dérivée pour $z < 0$ est 0 . Cependant, dans la pratique, elle fonctionne très bien et a l'avantage d'être rapide à calculer. Elle est donc devenue la fonction par défaut³⁷. Plus important encore, le fait qu'elle n'ait pas de valeur de sortie maximale aide à diminuer certains problèmes au cours de la descente de gradient (nous y reviendrons au chapitre 3).

Ces fonctions d'activation répandues et leur dérivée sont représentées à la figure 2.8. Mais pourquoi avons-nous besoin d'une fonction d'activation ? Si nous enchaînons plusieurs transformations linéaires, nous obtenons une transformation linéaire. Prenons, par exemple, $f(x) = 2x + 3$ et $g(x) = 5x - 1$. L'enchaînement de ces deux fonctions linéaires donne une autre fonction linéaire : $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. Par conséquent, si nous n'avons pas une certaine non-linéarité entre les couches, quelle que soit la profondeur de la pile des couches, elle équivaut à une seule couche. Il est alors impossible de résoudre des problèmes très complexes. Inversement, un RNP suffisamment large avec des fonctions d'activation non linéaires peut, en théorie, se rapprocher de toute fonction continue.

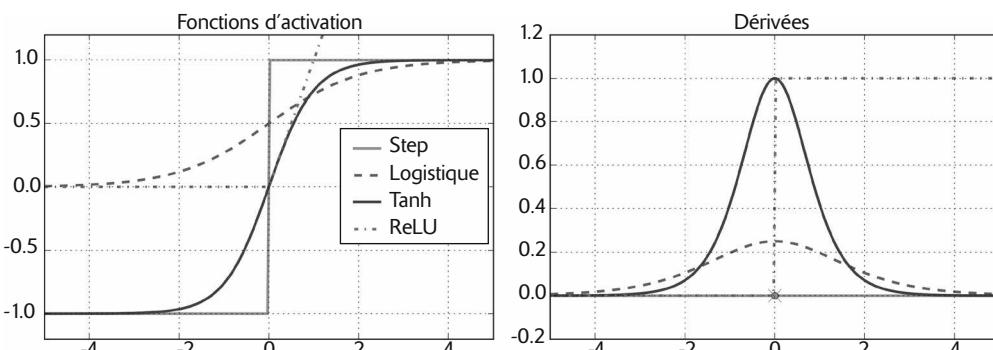


Figure 2.8 – Fonctions d'activation et leur dérivée

Vous savez à présent d'où viennent les réseaux neuronaux, quelle est leur architecture et comment leurs sorties sont calculées. Vous avez également découvert l'algorithme de rétropropagation. Mais à quoi pouvons-nous réellement employer ces réseaux ?

37. Puisque les neurones biologiques semblent mettre en œuvre une fonction d'activation de type sigmoïde (en forme de « S »), les chercheurs se sont longtemps bornés à des fonctions de ce type. Mais, en général, on constate que la fonction d'activation ReLU convient mieux aux RNA. Voilà l'un des cas où l'analogie avec la nature a pu induire en erreur.

2.1.5 PMC de régression

Tout d'abord, les PMC peuvent être utilisés pour des tâches de régression. Si nous souhaitons prédire une seule valeur (par exemple le prix d'une maison en fonction de ses caractéristiques), nous n'avons besoin que d'un seul neurone de sortie : sa sortie sera la valeur prédite. Pour une régression multivariée (c'est-à-dire prédire de multiples valeurs à la fois), nous avons besoin d'un neurone de sortie pour chaque dimension de sortie. Par exemple, pour localiser le centre d'un objet dans une image, nous devons prédire des coordonnées en 2D et avons donc besoin de deux neurones de sortie. Pour placer un cadre d'encombrement autour de l'objet, nous avons besoin de deux valeurs supplémentaires : la largeur et la hauteur de l'objet. Nous arrivons donc à quatre neurones de sortie.

En général, lors de la construction d'un PMC pour la régression, nous préférerons ne pas utiliser de fonction d'activation pour les neurones de sortie. Ils sont donc libres de produire toute plage de valeurs en sortie. Pour garantir que la sortie sera toujours positive, nous pouvons employer la fonction d'activation ReLU dans la couche de sortie. Nous pouvons également choisir la fonction d'activation *softplus*, qui est une variante lisse de la ReLU : $\text{softplus}(z) = \log(1 + \exp(z))$. Elle est proche de 0 lorsque z est négatif et proche de z lorsque z est positif. Enfin, si nous voulons garantir que les prédictions tomberont dans une certaine plage de valeurs, nous pouvons utiliser la fonction logistique ou la tangente hyperbolique, puis dimensionner les étiquettes dans la plage appropriée : 0 à 1 pour la fonction logistique, et -1 à 1 pour la tangente hyperbolique.

La fonction de perte utilisée pendant l'entraînement est en général l'erreur quadratique moyenne, mais si le jeu d'entraînement comprend un grand nombre de valeurs aberrantes, l'erreur absolue moyenne sera peut-être préférable. Il est également possible d'utiliser la fonction de perte de Huber, qui combine les deux précédentes.



La perte de Huber est quadratique lorsque l'erreur est inférieure à un seuil δ (en général 1), mais linéaire lorsque l'erreur est supérieure à δ . La partie linéaire la rend moins sensible aux valeurs aberrantes que l'erreur quadratique moyenne, tandis que la partie quadratique lui permet de converger plus rapidement et plus précisément que l'erreur absolue moyenne.

Le tableau 2.1 récapitule l'architecture type d'un PMC de régression.

Tableau 2.1 – Architecture type d'un PMC de régression

Hyperparamètre	Valeur type
Nombre de neurones d'entrée	Un par caractéristiques d'entrée (ex.: $28 \times 28 = 784$ pour MIST)
Nombre de couches cachées	Dépend du problème, mais en général entre 1 et 5

Hyperparamètre	Valeur type
Nombre de neurones par couche cachée	Dépend du problème, mais en général entre 10 et 100
Nombre de neurones de sortie	1 par dimension de prédiction
Fonction d'activation de la couche cachée	ReLU (ou SELU, voir le chapitre 3)
Fonction d'activation de la sortie	Aucune ou ReLU/softplus (pour des sorties positives) ou logistic/tanh (pour des sorties bornées)
Fonction de perte	MSE ou MAE/Huber (en cas de valeurs aberrantes)

2.1.6 PMC de classification

Les PMC peuvent également être employés pour des tâches de classification. Dans le cas d'un problème de classification binaire, nous avons besoin d'un seul neurone de sortie avec la fonction d'activation logistique : la sortie sera une valeur entre 0 et 1, que nous pouvons interpréter comme la probabilité estimée de la classe positive. La probabilité estimée de la classe négative est égale à 1 moins cette valeur.

Ils sont également capables de prendre en charge les tâches de classification binaire à étiquettes multiples³⁸. Par exemple, nous pouvons disposer d'un système de classification des courriers électroniques qui prédit si chaque message entrant est un courrier sollicité (*ham*) ou non sollicité (*spam*), tout en prédisant s'il est urgent ou non. Dans ce cas, nous avons besoin de deux neurones de sortie, tous deux avec la fonction d'activation logistique : le premier indiquera la probabilité que le courrier est non sollicité, tandis que le second indiquera la probabilité qu'il est urgent. Plus généralement, nous affectons un neurone de sortie à chaque classe positive. Notez que le total des probabilités de sortie ne doit pas nécessairement être égal à 1. Cela permet au modèle de sortir toute combinaison d'étiquettes : nous pouvons avoir du courrier sollicité non urgent, du courrier sollicité urgent, du courrier non sollicité non urgent et même du courrier non sollicité urgent (mais ce cas sera probablement une erreur).

Lorsque chaque instance ne peut appartenir qu'à une seule classe, parmi trois classes possibles ou plus (par exemple, les classes 0 à 9 pour la classification d'image de chiffres), nous avons besoin d'un neurone de sortie par classe et nous pouvons utiliser la fonction d'activation softmax pour l'intégralité de la couche de sortie (voir la figure 2.9). Cette fonction³⁹ s'assurera que toutes les probabilités estimées se trouvent entre 0 et 1 et que leur total est égal à 1 (ce qui est obligatoire si les classes sont exclusives). Il s'agit d'une classification multiclasse.

38. Voir le chapitre 3 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

39. Voir le chapitre 1.

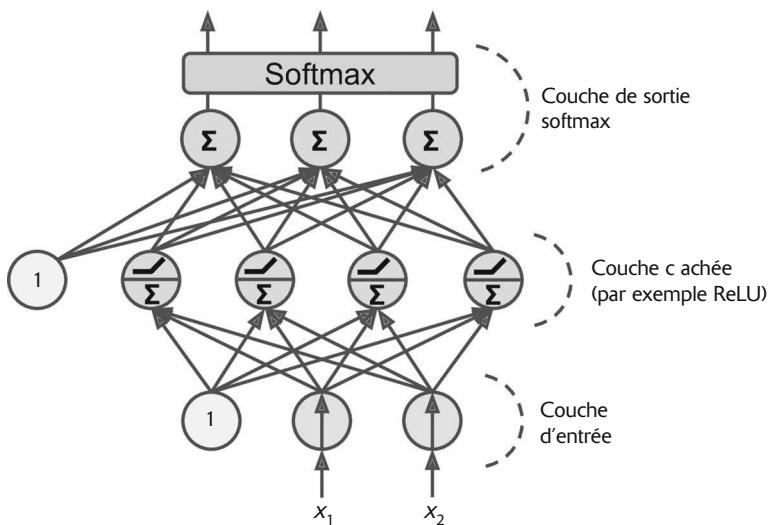


Figure 2.9 – Un PMC moderne (avec ReLU et softmax) pour la classification

Concernant la fonction de perte, puisque nous réalisons des distributions de probabilités, la perte d'entropie croisée (également appelée perte logistique, voir le chapitre 1) constitue généralement un bon choix.

Le tableau 2.2 récapitule l'architecture type d'un PMC de classification.

Tableau 2.2 – Architecture type d'un PMC de classification

Hyperparamètre	Classification binaire	Classification binaire multi-étiquette	Classification multiclasse
Couches d'entrée et couches cachées	Idem à la régression	Idem à la régression	Idem à la régression
Nombre de neurones de sortie	1	1 par étiquette	1 par classe
Fonction d'activation de la couche de sortie	Logistique	Logistique	Softmax
Fonction de perte	Entropie croisée	Entropie croisée	Entropie croisée



Avant d'aller plus loin, nous vous conseillons de faire l'exercice 1 donné à la fin de ce chapitre. Il vous permettra de jouer avec diverses architectures de réseaux de neurones et de visualiser leurs sorties avec *TensorFlow Playground*. Ainsi, vous comprendrez mieux les MPC, y compris les effets de tous les hyperparamètres (nombre de couches et de neurones, fonctions d'activation, etc.).

À présent que les concepts ont été établis, nous pouvons commencer à implémenter des MPC avec Keras !

2.2 IMPLÉMENTER DES MPC AVEC KERAS

Keras est une API de haut niveau pour le Deep Learning. Elle permet de construire, d'entraîner, d'évaluer et d'exécuter facilement toutes sortes de réseaux de neurones. Sa documentation (ou spécification) est disponible à l'adresse <https://keras.io/>. L'implémentation de référence (<https://github.com/keras-team/keras>), également nommée Keras, a été développée par François Chollet dans le cadre d'un projet de recherche⁴⁰ et publiée en tant que projet open source en mars 2015. Elle est rapidement devenue populaire, en raison de sa facilité d'utilisation, de sa souplesse et de sa belle conception. Pour effectuer les lourds calculs imposés par les réseaux de neurones, cette implémentation de référence se fonde sur un backend de calcul. Pour le moment, nous avons le choix entre trois bibliothèques open source de Deep Learning répandues : TensorFlow, Microsoft Cognitive Toolkit (CNTK) et Theano. Afin d'éviter toute confusion, nous désignerons cette implémentation de référence sous le terme *Keras multibackend*.

Depuis la fin 2016, d'autres implémentations sont apparues. Nous pouvons à présent exécuter Keras sur Apache MXNet, Core ML d'Apple, Javascript ou Typescript (pour exécuter du code Keras dans un navigateur web) et PlaidML (qui peut fonctionner sur toutes sortes de cartes graphiques, pas uniquement celles de Nvidia). TensorFlow fournit sa propre implémentation de Keras, tf.keras. Dans ce cas, le seul backend pris en charge est TensorFlow, mais elle a l'avantage d'offrir des possibilités supplémentaires très utiles (voir la figure 2.10). Par exemple, elle reconnaît l'API Data de TensorFlow, ce qui permet de charger et de prétraiter efficacement des données. Voilà pourquoi nous utilisons tf.keras dans cet ouvrage. Cependant, dans ce chapitre, nous n'utiliserons aucune des caractéristiques spécifiques à TensorFlow. Le code devrait donc être pleinement compatible avec d'autres implémentations de Keras (tout au moins dans Python), avec seulement quelques changements mineurs, comme la modification des importations.

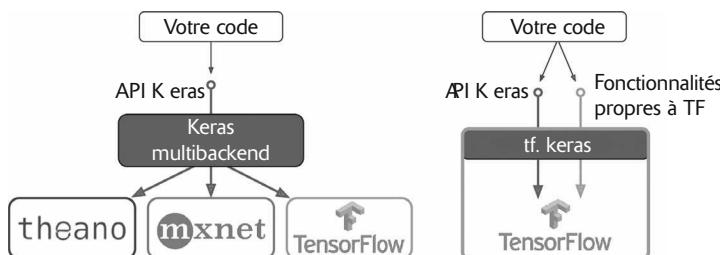


Figure 2.10 – Deux implémentations de l'API Keras : Keras multibackend (à gauche) et tf.keras (à droite)

Après Keras et TensorFlow, la bibliothèque de Deep Learning la plus populaire est PyTorch de Facebook (<https://pytorch.org/>). Elle est assez semblable à Keras (en

40. Projet ONEIROS (*Open-ended Neuro-Electronic Intelligent Robot Operating System*).

partie parce que ces deux API se sont inspirées de Scikit-Learn et de Chainer, <https://chainer.org/>) et, si vous maîtrisez Keras, vous n'aurez pas de difficultés à basculer sur PyTorch en cas de besoin. La popularité de PyTorch a considérablement augmenté en 2018, essentiellement grâce à sa simplicité et à son excellente documentation, ce qui n'était pas véritablement les points forts de TensorFlow 1.x. Toutefois, certains estiment que TensorFlow 2 est aussi simple que PyTorch, puisque Keras est devenue son API de haut niveau officielle et que le reste de l'API a également été largement simplifié et nettoyé. La documentation a également été totalement réorganisée et il est beaucoup plus facile d'y trouver ce que l'on cherche. De façon comparable, les principales faiblesses de PyTorch (par exemple une portabilité limitée et aucune analyse du graphe de calcul) ont été grandement comblées dans PyTorch 1.0. Une saine compétition est bénéfique pour tout le monde.

À présent, du code ! Puisque tf.keras vient avec TensorFlow, commençons par installer ce dernier.

2.2.1 Installer TensorFlow 2

Si vous avez bien suivi les instructions d'installation du chapitre 1, TensorFlow est déjà installé sur votre ordinateur. Pour tester votre installation, ouvrez un terminal, activez l'environnement conda tf2, démarrez un shell Python, importez tensorflow et keras, et affichez leurs versions :

```
$ conda activate tf2
$ python
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.1.0'
>>> keras.__version__
'2.2.4-tf'
```

La deuxième version est celle de l'API Keras mise en œuvre par tf.keras. Vous remarquerez qu'elle se termine par -tf, ce qui indique que tf.keras implémente non seulement l'API Keras, mais également quelques fonctionnalités supplémentaires propres à TensorFlow.

Si vous disposez d'une carte graphique (GPU) compatible avec TensorFlow et que vous avez installé son pilote, alors vous pouvez vérifier que votre GPU est bien détecté :

```
>>> tf.test.is_gpu_available()
True
```

Commençons à utiliser tf.keras, en construisant un classificateur d'images simple.

2.2.2 Construire un classificateur d'images avec l'API Sequential

Nous devons tout d'abord charger un jeu de données. Nous choisissons *Fashion MNIST*, qui est un équivalent exact de MNIST⁴¹. Leur format est identique (70 000 images en

41. Voir le chapitre 3 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

niveaux de gris de 28×28 pixels chacune, avec 10 classes), mais les images de Fashion MNIST représentent des articles de mode à la place de chiffres manuscrits. Chaque classe est donc plus variée et le problème se révèle plus compliqué. Par exemple, un modèle linéaire simple donne une précision de 92 % avec MNIST, mais seulement de 83 % avec Fashion MNIST.

Charger le jeu de données avec Keras

Keras dispose de fonctions utilitaires pour récupérer et charger des jeux de données communs, comme MNIST, Fashion MNIST et le jeu de données California Housing⁴². Chargeons Fashion MNIST :

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

Le chargement de MNIST ou de Fashion MNIST avec Keras, à la place de Scikit-Learn, présente une différence importante : chaque image est représentée non pas sous forme d'un tableau à une dimension de 784 éléments, mais sous forme d'un tableau 28×28. Par ailleurs, les intensités des pixels sont représentées par des entiers (de 0 à 255) plutôt que par des nombres à virgule flottante (de 0,0 à 255,0). Jetons un coup d'œil à la forme et au type de données du jeu d'entraînement :

```
>>> X_train_full.shape
(60000, 28, 28)
>>> X_train_full.dtype
dtype('uint8')
```

Notez que le jeu de données est déjà divisé en un jeu d'entraînement et un jeu de test, mais qu'il n'y a pas de jeu de validation. Nous allons donc en créer un. De plus, puisque nous allons entraîner le réseau de neurones avec la descente de gradient, nous devons mettre à l'échelle les caractéristiques d'entrée. Pour une question de simplicité, nous allons réduire les intensités de pixels à la plage 0-1 en les divisant par 255,0 (cela les convertit également en nombres à virgule flottante) :

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

Avec MNIST, lorsque l'étiquette est égale à 5, cela signifie que l'image représente le chiffre manuscrit 5. Facile. En revanche, avec Fashion MNIST, nous avons besoin de la liste des noms de classes pour savoir ce que nous manipulons :

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Par exemple, la première image du jeu d'entraînement représente un manteau :

```
>>> class_names[y_train[0]]
'Coat'
```

⁴². Ce jeu de données, aussi utilisé dans l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019), a été originellement présenté par R. Kelley Pace et Ronald Barry (1997), dans « Sparse Spatial Autoregressions », *Statistics and Probability Letters*, 33, n° 3, 291-297.

La figure 2.11 montre quelques éléments du jeu de données Fashion MNIST.



Figure 2.11 – Quelques exemples tirés de Fashion MNIST

Créer le modèle en utilisant l'API Sequential

Construisons à présent le réseau de neurones ! Voici un MPC de classification avec deux couches cachées :

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

Détaillons ce code :

- La première ligne crée un modèle `Sequential`. C'est le modèle Keras le plus simple pour les réseaux de neurones : il est constitué d'une seule pile de couches connectées de façon séquentielle. Il s'agit de l'API `Sequential`.
- Ensuite, nous construisons la première couche et l'ajoutons au modèle. Il s'agit d'une couche `Flatten` dont le rôle est de convertir chaque image d'entrée en un tableau à une dimension : si elle reçoit une donnée d'entrée `X`, elle calcule `X.reshape(-1, 1)`. Cette couche ne prend aucun paramètre et a pour seule fonction d'effectuer un prétraitement simple. Puisqu'elle est la première couche du modèle, nous devons préciser le `input_shape`, qui n'inclut pas la taille du lot, seulement la forme des instances. Il serait également possible d'ajouter en première couche un `keras.layers.InputLayer`, en précisant `input_shape=[28, 28]`.
- Puis, nous ajoutons une couche cachée `Dense` constituée de 300 neurones. Elle utilise la fonction d'activation ReLU. Chaque couche `Dense` gère sa propre matrice de poids, qui contient tous les poids des connexions entre les neurones et leurs entrées. Elle gère également un vecteur de termes constants (un par neurone). Lorsqu'elle reçoit des données d'entrée, elle calcule l'équation 2.2.

- Une deuxième couche cachée Dense de 100 neurones est ensuite ajoutée, elle aussi avec la fonction d'activation ReLU.
- Enfin, nous ajoutons une couche de sortie Dense avec 10 neurones (un par classe) en utilisant la fonction d'activation softmax (car les classes sont exclusives).



Spécifier `activation="relu"` équivaut à spécifier `activation=keras.activations.relu`. D'autres fonctions d'activation sont disponibles dans le package `keras.activations` et nous en utilisons plusieurs dans cet ouvrage. La liste complète est disponible à l'adresse <https://keras.io/activations/>.

Au lieu d'ajouter les couches une par une comme nous l'avons fait, nous pouvons passer une liste de couches au moment de la création du modèle Sequential :

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Utiliser les exemples de code provenant de keras.io

Les exemples de code documentés sur keras.io fonctionneront parfaitement avec tf.keras, mais il vous faudra modifier des importations. Examinons, par exemple, le code keras.io suivant :

```
from keras.layers import Dense
output_layer = Dense(10)
```

Voici comment modifier l'importation :

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Ou, si vous le préférez, utilisez simplement des chemins complets :

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

Même si cette solution est plus verbeuse, nous l'avons retenue dans cet ouvrage car elle permet de voir plus facilement les packages à employer et d'éviter toute confusion entre les classes standard et les classes personnalisées. Dans un code de production, nous préférerons l'approche précédente. Nombreux sont également ceux à utiliser `from tensorflow.keras import layers` suivi de `layers.Dense(10)`.

La méthode `summary()` du modèle affiche toutes les couches du modèle⁴³, y compris leur nom (généré automatiquement, sauf s'il est précisé au moment de la création de la couche), la forme de leur sortie (`None` signifie que la taille du lot peut être quelconque) et leur nombre de paramètres. Le résumé se termine par le nombre

43. Vous pouvez utiliser `keras.utils.plot_model()` pour générer une image du modèle.

total de paramètres, qu'ils soient entraînables ou non. Dans notre exemple, nous avons uniquement des paramètres entraînables (nous verrons des exemples de paramètres non entraînables au chapitre 3) :

```
>>> model.summary()
Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
flatten (Flatten)            (None, 784)               0
dense (Dense)                (None, 300)              235500
dense_1 (Dense)              (None, 100)              30100
dense_2 (Dense)              (None, 10)               1010
=====
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

Les couches `Dense` possèdent souvent un *grand nombre* de paramètres. Par exemple, la première couche cachée a 784×300 poids de connexions et 300 termes constants. Au total, cela fait 235 500 paramètres ! Le modèle dispose ainsi d'une grande souplesse d'ajustement aux données d'entraînement, mais le risque de surajustement est accru, en particulier lorsque les données d'entraînement sont peu nombreuses. Nous y reviendrons ultérieurement.

Nous pouvons aisément obtenir une liste des couches du modèle, pour ensuite retrouver une couche par son indice ou son nom :

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

Tous les paramètres d'une couche sont accessibles à l'aide des méthodes `get_weights()` et `set_weights()`. Dans le cas d'une couche `Dense`, cela comprend à la fois les poids des connexions et les termes constants :

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
      [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
       0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
```

```
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

La couche `Dense` initialise les poids des connexions de façon aléatoire (indispensable pour briser la symétrie, comme nous l'avons expliqué) et les termes constants à zéro, ce qui convient parfaitement. Pour employer une méthode d'initialisation différente, il suffit de fixer `kernel_initializer` (`kernel`, ou `noyau`, est un autre nom pour la matrice des poids des connexions) ou `bias_initializer` au moment de la création de la couche. Nous reviendrons sur les initialiseurs au chapitre 3, mais vous en trouverez la liste complète à l'adresse <https://keras.io/initializers/>.



La forme de la matrice des poids dépend du nombre d'entrées. C'est pourquoi il est conseillé de préciser le `input_shape` lors de la création de la première couche dans un modèle `Sequential`. Vous pouvez très bien ne pas le faire, auquel cas Keras attendra simplement de connaître la forme de l'entrée pour construire réellement le modèle. Cela se produira lorsque vous lui fournirez des données réelles (par exemple, au cours de l'entraînement) ou lorsque vous appellerez sa méthode `build()`. Tant que le modèle n'est pas véritablement construit, les couches n'auront aucun poids et certaines opérations ne seront pas possibles (comme sauvegarder le modèle ou afficher son résumé). Par conséquent, si vous connaissez la forme de l'entrée au moment de la création du modèle, il est préférable de l'indiquer.

Compiler le modèle

Après qu'un modèle a été créé, nous devons invoquer sa méthode `compile()` de manière à préciser la fonction de perte et l'optimiseur. Nous pouvons également indiquer une liste d'indicateurs supplémentaires qui seront mesurés au cours de l'entraînement et de l'évaluation :

```
model.compile(loss="sparse_categorical_crossentropy",
               optimizer="sgd",
               metrics=["accuracy"])
```



Utiliser `loss="sparse_categorical_crossentropy"` équivaut à utiliser `loss=keras.losses.sparse_categorical_crossentropy`. De même, spécifier `optimizer="sgd"` équivaut à spécifier `optimizer=keras.optimizers.SGD()`, et `metrics=["accuracy"]` est équivalent à `metrics=[keras.metrics.sparse_categorical_accuracy]` (lorsque cette fonction de perte est employée). Nous utiliserons beaucoup d'autres fonctions de perte, optimiseurs et indicateurs dans cet ouvrage. Vous en trouverez les listes complètes aux adresses <https://keras.io/losses/>, <https://keras.io/optimizers/> et <https://keras.io/metrics/>.

Ce code mérite quelques explications. Tout d'abord, nous utilisons la perte "sparse_categorical_crossentropy" car nous avons des étiquettes clair-semées (pour chaque instance, il n'existe qu'un seul indice de classe cible, de 0 à 9 dans ce cas) et les classes sont exclusives. Si, à la place, nous avions une probabilité

cible par classe pour chaque instance (comme des vecteurs *one-hot*, par exemple [0., 0., 0., 1., 0., 0., 0., 0.] pour représenter une classe 3), nous opterions pour la fonction de perte "categorical_crossentropy". Si nous réalisions une classification binaire (avec une ou plusieurs étiquettes binaires), nous choisirions alors la fonction d'activation "sigmoid" (c'est-à-dire logistique) dans la couche de sortie à la place de la fonction d'activation "softmax", ainsi que la fonction de perte "binary_crossentropy".



Pour convertir des étiquettes clairsemées (c'est-à-dire des indices de classes) en étiquettes de vecteurs *one-hot*, servez-vous de la fonction `keras.utils.to_categorical()`. La fonction `np.argmax()`, avec `axis=1`, effectue l'opération inverse.

Quant à l'optimiseur, "sgd" signifie que nous entraînerons le modèle à l'aide d'une descente de gradient stochastique simple. Autrement dit, Keras mettra en œuvre l'algorithme de rétropropagation décrit précédemment (c'est-à-dire une différentiation automatique en mode inverse plus une descente de gradient). Nous verrons des optimiseurs plus efficaces au chapitre 3 (ils améliorent la descente de gradient, non la différentiation automatique).



Lorsqu'on utilise l'optimiseur SGD, le réglage du taux d'apprentissage est important. Par conséquent, pour fixer le taux d'apprentissage, nous utiliserons généralement `optimizer=keras.optimizers.SGD(lr=???)` plutôt que `optimizer="sgd"`, qui prend par défaut `lr=0.01`.

Enfin, puisqu'il s'agit d'un classificateur, il est utile de mesurer sa précision ("accuracy") pendant un entraînement et une évaluation.

Entrainer et évaluer le modèle

Le modèle est maintenant prêt pour l'entraînement. Pour cela, nous devons simplement appeler sa méthode `fit()` :

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                      validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218      - accuracy: 0.7660
                                         - val_loss: 0.4973 - val_accuracy:
0.8366
Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840      - accuracy: 0.8327
                                         - val_loss: 0.4456 - val_accuracy:
0.8480
[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252      - accuracy: 0.9192
                                         - val_loss: 0.2999 - val_accuracy:
0.8926
```

Nous lui fournissons les caractéristiques d'entrée (`X_train`) et les classes cibles (`y_train`), ainsi que le nombre d'époques d'entraînement (dans le cas contraire, il n'y en aurait qu'une, ce qui serait clairement insuffisant pour converger vers une bonne solution). Nous passons également un jeu de validation (facultatif). Keras mesurera la perte et les indicateurs supplémentaires sur ce jeu à la fin de chaque époque, ce qui se révélera très utile pour déterminer les performances réelles du modèle. Si les performances sur le jeu d'entraînement sont bien meilleures que sur le jeu de validation, il est probable que le modèle surajuste le jeu d'entraînement (ou qu'il y ait une erreur, comme une différence entre les données du jeu d'entraînement et celles du jeu de validation).

Et voilà, le réseau de neurones est entraîné⁴⁴. Au cours de chaque époque de l'entraînement, Keras affiche le nombre d'instances traitées (ainsi qu'une barre de progression), le temps moyen d'entraînement par échantillon, ainsi que la perte et la précision (ou tout autre indicateur supplémentaire demandé) à la fois sur le jeu d'entraînement et sur celui de validation. Vous constatez que la perte dans l'entraînement a diminué, ce qui est bon signe, et que la précision de la validation a atteint 89,26 % après 30 époques, ce qui n'est pas très loin de la précision de l'entraînement. Nous pouvons donc en conclure que s'il y a surajustement, il n'est pas trop important.



Au lieu de passer un jeu de validation avec l'argument `validation_data`, vous pouvez indiquer dans `validation_split` la portion du jeu d'entraînement que Keras doit utiliser pour la validation. Par exemple, `validation_split=0.1` lui demande d'utiliser les derniers 10 % des données (avant mélange) pour la validation.

Si le jeu d'entraînement est assez inégal, avec certaines classes surreprésentées et d'autres sous-représentées, il peut être utile de définir l'argument `class_weight` lors de l'appel à la méthode `fit()` afin de donner un poids plus important aux classes sous-représentées et un poids plus faible à celle surreprésentées. Ils seront utilisés par Keras dans le calcul de la perte. Si des poids sont nécessaires pour chaque instance, nous pouvons utiliser l'argument `sample_weight` (si `class_weight` et `sample_weight` sont tous deux précisés, Keras les multiplie). Les poids par instance peuvent être utiles lorsque certaines instances ont été libellées par des experts tandis que d'autres l'ont été au travers d'une collaboration participative : les premières peuvent avoir un poids plus important. Nous pouvons également fournir des poids d'instance (mais pas de classe) pour le jeu de validation en les ajoutant en troisième élément du paramètre `validation_data`.

44. Si les données d'entraînement ou de validation n'ont pas la forme attendue, une exception est générée. Cette erreur étant probablement la plus fréquente, vous devez vous familiariser avec le message d'erreur. Il est relativement clair. Par exemple, si vous tentez d'entraîner ce modèle avec un tableau qui contient des images aplatis (`X_train.reshape(-1, 784)`), vous recevez l'exception suivante : « `ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)` ». Le message explique que `flatten_input` doit avoir trois dimensions alors que l'entrée fournie est un tableau de format (60000, 784).

La méthode `fit()` retourne un objet `History` qui contient les paramètres d'entraînement (`history.params`), la liste des époques effectuées (`history.epoch`) et, le plus important, un dictionnaire (`history.history`) donnant la perte et les indicateurs supplémentaires mesurés à la fin de chaque époque sur le jeu d'entraînement et le jeu de validation (si présent). En utilisant ce dictionnaire pour créer un DataFrame pandas et en invoquant sa méthode `plot()`, nous obtenons les courbes d'apprentissage illustrées à la figure 2.12 :

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylims(0, 1) # Régler la plage verticale sur [0-1]
plt.show()
```

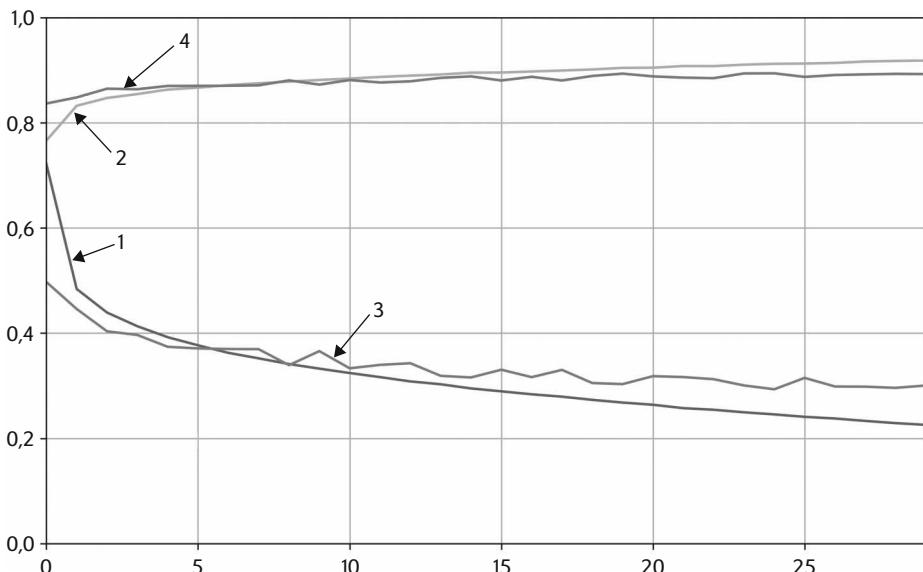


Figure 2.12 – Courbes d'apprentissage : la perte (1) et la précision (2) d'entraînement moyennes mesurées sur chaque époque, ainsi que la perte (3) et la précision (4) de validation moyennes mesurées à la fin de chaque époque

La précision d'entraînement et celle de validation augmentent toutes deux régulièrement au cours de l'entraînement, tandis que les pertes d'entraînement et de validation décroissent. Parfait ! Par ailleurs, les courbes de validation sont proches des courbes d'entraînement, ce qui signifie que le surentraînement n'est pas très important. Dans ce cas particulier, le modèle semble présenter de meilleures performances sur le jeu de validation que sur le jeu d'entraînement en début d'entraînement. Mais ce n'est pas le cas : l'erreur de validation est calculée à la fin de chaque époque, tandis que l'erreur d'entraînement est calculée à l'aide d'une moyenne glissante pendant chaque époque. La courbe d'entraînement doit donc être décalée d'une

moitié d'époque vers la gauche. Avec ce décalage, nous constatons que les courbes d'entraînement et de validation se chevauchent presque parfaitement au début de l'entraînement.



Lors de son affichage, la courbe d'entraînement doit être décalée d'une moitié d'époque vers la gauche.

Les performances du jeu d'entraînement finissent par dépasser celles du jeu de validation, comme c'est généralement le cas lorsque l'entraînement est suffisamment long. Nous pouvons voir que le modèle n'a pas encore assez convergé, car la perte de validation continue à diminuer. Il serait donc préférable de poursuivre l'entraînement. Il suffit d'invoquer une nouvelle fois la méthode `fit()`, car Keras reprend l'entraînement là où il s'était arrêté (une précision de validation proche de 89 % devrait pouvoir être atteinte).

Si vous n'êtes pas satisfait des performances de votre modèle, revenez en arrière et ajustez les hyperparamètres. Le premier à examiner est le taux d'apprentissage. Si cela ne change rien, essayez un autre optimiseur (réajustez toujours le taux d'apprentissage après chaque changement d'un hyperparamètre). Si les performances sont toujours mauvaises, essayez d'ajuster les hyperparamètres du modèle, par exemple le nombre de couches, le nombre de neurones par couche et le type des fonctions d'activation attribuées à chaque couche cachée. Vous pouvez également affiner d'autres hyperparamètres, comme la taille du lot (fixée dans la méthode `fit()` à l'aide de l'argument `batch_size`, dont la valeur par défaut est 32). Nous reviendrons sur le réglage des hyperparamètres à la fin de ce chapitre. Dès que vous êtes satisfait de la précision de validation de votre modèle, vous devez, avant de le mettre en production, l'évaluer sur le jeu de test afin d'estimer l'erreur de généralisation. Pour cela, il suffit d'utiliser la méthode `evaluate()` (elle reconnaît plusieurs autres arguments, comme `batch_size` et `sample_weight`; voir sa documentation pour plus de détails) :

```
>>> model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

Il est fréquent d'obtenir des performances légèrement moins bonnes sur le jeu de test que sur le jeu de validation⁴⁵. En effet, les hyperparamètres sont ajustés non pas sur le jeu de test mais sur le jeu de validation (cependant, dans cet exemple, puisque les hyperparamètres n'ont pas été affinés, la précision inférieure est juste due au manque de chance). Résistez à la tentation d'ajuster les hyperparamètres sur le jeu de test car votre estimation de l'erreur de généralisation sera alors trop optimiste.

Utiliser le modèle pour faire des prédictions

Nous pouvons ensuite utiliser la méthode `predict()` pour effectuer des prédictions sur de nouvelles instances. Puisque nous n'avons pas de véritables nouvelles instances, nous utilisons simplement les trois premières du jeu de test :

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.   , 0.   , 0.   , 0.   , 0.03, 0.   , 0.01, 0.   , 0.96],
       [0.   , 0.   , 0.98, 0.   , 0.02, 0.   , 0.   , 0.   , 0.   ],
       [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ]],
      dtype=float32)
```

Pour chaque instance, le modèle estime une probabilité par classe, de la classe 0 à la classe 9. Par exemple, pour la première image, il estime que la probabilité de la classe 9 (bottine) est de 96 %, que celle de la classe 5 (sandale) est de 3 %, que celle de la classe 7 (basket) est de 1 %, et que celles des autres classes sont négligeables. Autrement dit, il « croit » que la première image est une chaussure, probablement une bottine, mais éventuellement une sandale ou une basket. Si nous nous intéressons uniquement à la classe dont la probabilité estimée est la plus élevée (même si celle-ci est relativement faible), nous pouvons utiliser à la place la méthode `predict_classes()` :

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Le classificateur réalise une classification correcte des trois images (elles sont représentées à la figure 2.13) :

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

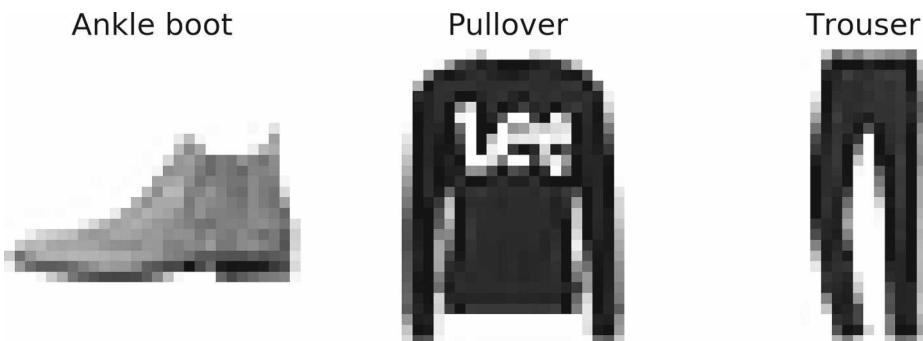


Figure 2.13 – Classification correcte d'images du jeu Fashion MNIST

Vous savez à présent utiliser l'API Sequential pour construire, entraîner, évaluer et utiliser un MPC de classification. Mais qu'en est-il de la régression ?

2.2.3 Construire un MPC de régression avec l'API Sequential

Passons au problème de logement en Californie et abordons-le en utilisant un réseau de neurones de régression. Pour une question de simplicité, nous chargerons les données avec la fonction `fetch_california_housing()` de Scikit-Learn⁴⁶. Après que les données ont été chargées, nous les répartissons en jeux d'entraînement, de validation et de test, et nous dimensionnons toutes les caractéristiques :

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

Avec l'API Sequential, la méthode de construction, d'entraînement, d'évaluation et d'utilisation d'un MPC de régression de façon à effectuer des prédictions est comparable à celle employée dans le cas de la classification. Les principales différences résident dans le fait que la couche de sortie comprend un seul neurone (puisque nous voulons prédire une seule valeur) et aucune fonction d'activation, et que la fonction de perte est l'erreur quadratique moyenne. Puisque le jeu de données contient beaucoup de bruit, nous utilisons simplement une seule couche cachée avec moins de neurones que précédemment. Cela permet d'éviter le surajustement :

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                     validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # Prétendre qu'il s'agit de nouvelles instances
y_pred = model.predict(X_new)
```

Vous le constatez, l'utilisation de l'API Sequential n'a rien de compliqué. Cependant, bien que les modèles séquentiels soient extrêmement courants, il est parfois utile de construire des réseaux de neurones à la topologie plus complexe ou possédant plusieurs entrées ou sorties. Pour cela, Keras offre l'API Functional.

46. Ce jeu de données est plus simple que California Housing, utilisé au chapitre 5, car il contient uniquement des caractéristiques numériques (la caractéristique `ocean_proximity` est absente) et il ne manque aucune valeur.

2.2.4 Construire des modèles plus complexes avec l'API Functional

Un réseau de neurones *Wide & Deep* est un exemple de réseau non séquentiel. Cette architecture a été présentée en 2016 dans un article publié par Heng-Tze Cheng et al.⁴⁷ Elle connecte toutes les entrées, ou seulement une partie, directement à la couche de sortie (voir la figure 2.14). Grâce à cette architecture, le réseau de neurones est capable d'apprendre à la fois les motifs profonds (en utilisant le chemin profond) et les règles simples (au travers du chemin court)⁴⁸. À l'opposé, un MPC classique oblige toutes les données à passer par l'intégralité des couches, et cette suite de transformations peut finir par déformer les motifs simples présents dans les données.

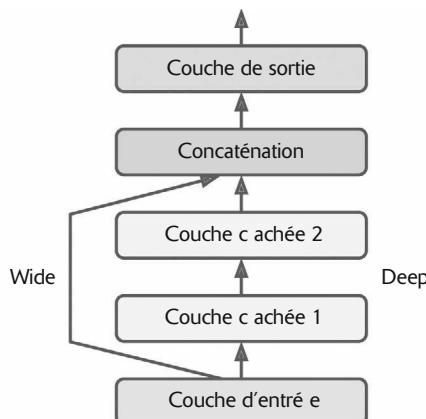


Figure 2.14 – Réseau de neurones Wide & Deep

Construisons un tel réseau pour traiter le problème de logement en Californie :

```

input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
  
```

Étudions chacune des lignes de ce code :

- Nous devons tout d'abord créer un objet `Input`⁴⁹. Il spécifie le type de l'entrée qui sera fournie au modèle, y compris sa forme (`shape`) et son type (`dtype`). En réalité, un modèle peut avoir plusieurs entrées, comme nous le verrons plus loin.
- Ensuite, nous créons une couche `Dense` avec 30 neurones grâce à la fonction d'activation ReLU. Dès qu'elle est créée, nous l'appelons comme une fonction,

47. Heng-Tze Cheng et al., « Wide & Deep Learning for Recommender Systems », *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016), 7-10 : <https://hml.info/widedeep>.

48. Le chemin court peut également servir à fournir au réseau de neurones des caractéristiques préparées manuellement.

49. Nous employons le nom `input_` afin d'éviter de faire de l'ombre à la fonction `input()` de Python.

en lui passant l'entrée. Voilà pourquoi l'API est dite fonctionnelle. Nous indiquons simplement à Keras comment il doit connecter les couches ; aucune donnée réelle n'est encore traitée.

- Puis nous créons une seconde couche cachée et l'utilisons de nouveau comme une fonction. Nous lui passons la sortie de la première couche cachée.
- Vient ensuite la création d'une couche `Concatenate`, utilisée immédiatement comme une fonction, pour concaténer l'entrée et la sortie de la seconde couche cachée. Vous pourriez préférer la fonction `keras.layers.concatenate()`, qui crée une couche `Concatenate` et l'invoque immédiatement avec les entrées données.
- La couche de sortie est alors créée avec un seul neurone et sans fonction d'activation. Nous l'appelons comme une fonction, en lui passant le résultat de la concaténation.
- Enfin, nous créons un `Model` Keras, en précisant les entrées et les sorties à utiliser.

Après que le modèle Keras a été construit, tout se passe comme précédemment : nous devons le compiler, l'entraîner, l'évaluer et l'utiliser pour des prédictions.

Mais comment pouvons-nous faire passer un sous-ensemble des caractéristiques par le chemin large et un sous-ensemble différent (avec un chevauchement éventuel) par le chemin profond (voir la figure 2.15) ? Pour cela, une solution consiste à utiliser de multiples entrées. Par exemple, supposons que nous souhaitions envoyer cinq caractéristiques sur le chemin large (0 à 4) et six caractéristiques sur le chemin profond (2 à 7) :

```
input_A = keras.layers.Input(shape=[5, name="wide_input"])
input_B = keras.layers.Input(shape=[6, name="deep_input"])
hidden1 = keras.layers.Dense(30, activation="relu") (input_B)
hidden2 = keras.layers.Dense(30, activation="relu") (hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output") (concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

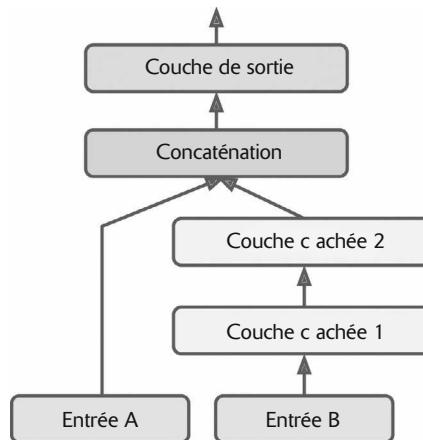


Figure 2.15 – Prise en charge d'entrées multiples

Le code est facile à comprendre par lui-même. Il est préférable de nommer au moins les couches les plus importantes, surtout lorsque le modèle devient un peu plus complexe. Notez que nous avons spécifié `inputs=[input_A, input_B]` lors de la création du modèle. Nous pouvons à présent le compiler de façon habituelle, mais, lorsque nous invoquons la méthode `fit()`, nous devons passer non pas une seule matrice d'entrée `X_train` mais un couple de matrices (`X_train_A, X_train_B`), une pour chaque entrée⁵⁰. Cela reste vrai pour `X_valid`, mais aussi pour `X_test` et `X_new` lors des appels à `evaluate()` et à `predict()`:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

Les sorties multiples pourront être utiles dans de nombreux cas :

- L'objectif peut l'imposer. Par exemple, nous pourrions souhaiter localiser et classifier le principal objet d'une image. Il s'agit à la fois d'une tâche de régression (trouver des coordonnées du centre de l'objet, ainsi que sa largeur et sa hauteur) et d'une tâche de classification.
- De manière comparable, nous pouvons avoir plusieurs tâches indépendantes basées sur les mêmes données. Nous pourrions évidemment entraîner un réseau de neurones par tâche, mais, dans de nombreux cas, les résultats obtenus sur toutes les tâches seront meilleurs en entraînant un seul réseau de neurones avec une sortie par tâche. En effet, le réseau de neurones peut apprendre des caractéristiques dans les données qui seront utiles à toutes les tâches. Par exemple, nous pouvons effectuer une *classification multitâche* sur des images de visages en utilisant une sortie pour classifier l'expression faciale de la personne (sourire, surprise, etc.) et une autre pour déterminer si elle porte des lunettes.
- La technique de la régularisation (c'est-à-dire une contrainte d'entraînement dont l'objectif est de réduire le surajustement et d'améliorer ainsi la capacité de généralisation du modèle) constitue un autre cas d'utilisation. Par exemple, nous pourrions souhaiter ajouter des sorties supplémentaires dans une architecture de réseau de neurones (voir la figure 2.16) afin que la partie sous-jacente du réseau apprenne par elle-même des choses intéressantes sans s'appuyer sur le reste du réseau.

50. Il est également possible de passer un dictionnaire qui établit la correspondance entre les noms des entrées et leurs valeurs, comme `{"wide_input": X_train_A, "deep_input": X_train_B}`. Cela sera particulièrement utile pour éviter un ordre erroné quand le nombre d'entrées est important.

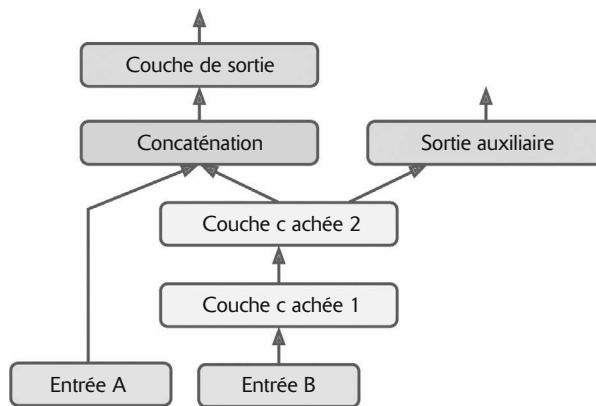


Figure 2.16 – Prise en charge des sorties multiples, dans ce cas pour ajouter une sortie supplémentaire dans un but de régularisation

L'ajout de sorties supplémentaires n'a rien de compliqué : il suffit de les connecter aux couches appropriées et de les ajouter à la liste des sorties du modèle. Par exemple, le code suivant construit le réseau représenté à la figure 2.16 :

```
[...] # Comme précédemment, jusqu'à la couche de sortie principale
output = keras.layers.Dense(1, name="main_output") (concat)
aux_output = keras.layers.Dense(1, name="aux_output") (hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Chaque sortie a besoin de sa propre fonction de perte. Lorsque nous compilons le modèle, nous devons donc lui passer une liste de fonctions de perte⁵¹ (si nous passons une seule fonction de perte, Keras suppose qu'elle concerne toutes les sorties). Par défaut, Keras calculera toutes ces pertes et les additionnera simplement afin d'obtenir la perte finale utilisée pour l'entraînement. Puisque nous accordons plus d'importance à la sortie principale qu'à la sortie auxiliaire (qui ne sert qu'à la régularisation), nous voulons donner un poids supérieur à la perte de la sortie principale. Heureusement, il est possible de fixer tous les poids des pertes lors de la compilation du modèle :

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Au moment de l'entraînement du modèle, nous devons fournir des étiquettes pour chaque sortie. Dans cet exemple, la sortie principale et la sortie auxiliaire doivent essayer de prédire la même chose. Elles doivent donc utiliser les mêmes étiquettes. En conséquence, au lieu de passer `y_train`, nous devons indiquer (`y_train, y_train`) (il en va de même pour `y_valid` et `y_test`) :

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

51. Nous pouvons également passer un dictionnaire qui fait la correspondance entre les noms des sorties et la fonction de perte correspondante. Comme pour les entrées, cette solution permettra d'éviter des erreurs d'ordre en cas de sorties multiples. Les poids de perte et les indicateurs (voir plus loin) sont également fixés à l'aide de dictionnaires.

Lors de l'évaluation du modèle, Keras retournera la perte totale, ainsi que les pertes individuelles :

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

De manière comparable, la méthode `predict()` va retourner des prédictions pour chaque sortie :

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Vous le constatez, vous pouvez construire assez facilement toute sorte d'architectures avec l'API Functional. Mais voyons une dernière façon de construire des modèles Keras.

2.2.5 Construire des modèles dynamiques avec l'API Subclassing

Les API Sequential et Functional sont toutes deux déclaratives : nous commençons par déclarer les couches à utiliser, ainsi que leurs connexions, et seulement ensuite nous alimentons le modèle en données à des fins d'entraînement ou d'inférence.

Cette approche présente plusieurs avantages : le modèle peut facilement être sauvegardé, cloné et partagé ; sa structure peut être affichée et analysée ; le framework étant capable de déduire des formes et de vérifier des types, les erreurs peuvent être déterminées précocement (c'est-à-dire avant que des données ne traversent le modèle). Elle permet également un débogage facile, puisque l'intégralité du modèle est un graphe statique de couches. En revanche, son inconvénient est justement son caractère statique. Certains modèles demandent des boucles, des formes variables, des branchements conditionnels et d'autres comportements dynamiques. Dans de tels cas, ou simplement si un style de programmation plus impératif nous convient mieux, nous pouvons nous tourner vers l'API Subclassing.

Il suffit de créer une sous-classe de la classe `Model`, de créer les couches requises dans le constructeur et de les utiliser ensuite dans la méthode `call()` pour effectuer les calculs nécessaires. Par exemple, la création d'une instance de la classe `WideAndDeepModel` suivante donne un modèle équivalent à celui construit précédemment à l'aide de l'API Functional. Nous pouvons le compiler, l'évaluer et l'utiliser pour réaliser des prédictions :

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # Gère les arguments standard (ex. : name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
```

```

    aux_output = self.aux_output(hidden2)
    return main_output, aux_output

model = WideAndDeepModel()

```

Cet exemple ressemble énormément à celui de l'API Functional, excepté qu'il est inutile de créer les entrées. Nous utilisons uniquement l'argument `input` de la méthode `call()` et séparons la création des couches⁵² dans le constructeur de leur usage dans la méthode `call()`. La grande différence est que nous pouvons faire quasiment tout ce que nous voulons dans la méthode `call()` : boucles `for`, instructions `if`, opérations TensorFlow de bas niveau – la seule limite sera votre imagination (voir le chapitre 4) ! Cette API convient donc parfaitement aux chercheurs qui souhaitent expérimenter de nouvelles idées.

Cette souplesse supplémentaire a un prix : l'architecture du modèle est cachée dans la méthode `call()`. Keras ne peut donc pas facilement l'inspecter, il ne peut pas l'enregistrer ou la cloner, et, lors de l'appel à `summary()`, nous n'obtenons qu'une liste de couches, sans information sur leurs interconnexions. Par ailleurs, Keras étant incapable de vérifier préalablement les types et les formes, les erreurs sont faciles. En conséquence, à moins d'avoir réellement besoin de cette flexibilité, il est préférable de rester avec l'API Sequential ou l'API Functional.



Les modèles Keras peuvent être utilisés comme des couches normales. Vous pouvez donc facilement les combiner pour construire des architectures complexes.

Vous savez à présent comment construire et entraîner des réseaux de neurones avec Keras. Voyons comment vous pouvez les enregistrer !

2.2.6 Enregistrer et restaurer un modèle

Avec les API Sequential et Functional, l'enregistrement d'un modèle Keras entraîné peut difficilement être plus simple :

```

model = keras.models.Sequential([...]) # ou keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")

```

Keras utilise le format HDF5 pour sauvegarder à la fois l'architecture du modèle (y compris les hyperparamètres de chaque couche) et les valeurs de tous les paramètres du modèle pour chaque couche (par exemple, les poids des connexions et les termes constants). Il enregistre également l'optimiseur (avec ses hyperparamètres et son état). Au chapitre 11, nous verrons comment enregistrer un modèle `tf.keras` en utilisant à la place le format `SavedModel` de TensorFlow.

52. Des modèles Keras disposant d'un attribut `output`, nous ne pouvons pas nommer ainsi la couche de sortie principale. Nous choisissons donc `main_output`.

En général, nous avons un script qui entraîne un modèle et l'enregistre, ainsi qu'un ou plusieurs autres scripts (ou des services web) qui le chargent et l'utilisent pour effectuer des prédictions. Le chargement du modèle est tout aussi facile :

```
model = keras.models.load_model("my_keras_model.h5")
```



Cela fonctionne avec les API Sequential et Functional, mais, malheureusement, pas lors de la création d'une sous-classe de Model. Vous pouvez employer `save_weights()` et `load_weights()` pour enregistrer et restaurer au moins les paramètres du modèle, mais vous devrez enregistrer et restaurer vous-même tout le reste.

Et si l'entraînement dure plusieurs heures ? Cette situation est relativement fréquente, en particulier lors de l'entraînement sur des jeux de données volumineux. Dans ce cas, nous devons non pas enregistrer le modèle uniquement à la fin de l'entraînement, mais réaliser des sauvegardes intermédiaires à intervalles réguliers afin d'éviter de perdre des résultats si jamais l'ordinateur tombait en panne. Pour demander à la méthode `fit()` d'effectuer ces sauvegardes intermédiaires, nous utilisons les rappels (*callbacks*).

2.2.7 Utiliser des rappels

La méthode `fit()` accepte un argument `callbacks` pour indiquer la liste des objets que Keras invoquera au cours de l'entraînement, au début et à la fin de l'entraînement, au début et à la fin de chaque époque, et même avant et après le traitement de chaque lot. Par exemple, le rappel `ModelCheckpoint` enregistre des points de restauration du modèle à intervalle régulier au cours de l'entraînement, par défaut à la fin de chaque époque :

```
[...] # Construire et compiler le modèle
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Par ailleurs, si nous utilisons un jeu de validation au cours de l'entraînement, nous pouvons indiquer `save_best_only=True` lors de la création de `ModelCheckpoint`, ce qui permet d'enregistrer le modèle uniquement lorsque ses performances sur le jeu de validation sont meilleures. Ainsi, nous n'avons pas besoin de nous préoccuper d'un entraînement trop long et du surajustement du jeu entraînement : il suffit de restaurer le dernier modèle enregistré après l'entraînement pour récupérer le meilleur modèle sur le jeu de validation. Le code suivant propose une manière simple d'implémenter un arrêt prématuré⁵³ :

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # Revenir au meilleur
# modèle
```

53. Voir le chapitre 1.

Une autre solution se fonde sur le rappel EarlyStopping. Il interrompra l'entraînement lorsqu'il ne constatera plus aucun progrès sur le jeu de validation pendant un certain nombre d'époques (fixé dans l'argument `patience`) et, en option, reviendra au meilleur modèle. Nous pouvons combiner ces deux rappels afin d'effectuer des sauvegardes intermédiaires du modèle (en cas de panne de l'ordinateur) et d'interrompre l'entraînement de façon prématuée lorsque les progrès sont absents (afin d'éviter de gaspiller du temps et des ressources) :

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                 restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

Le nombre d'époques peut être élevé car l'entraînement s'arrêtera automatiquement en cas d'absence de progrès. Dans ce cas, il est inutile de restaurer le meilleur modèle enregistré car le rappel EarlyStopping conservera les meilleurs poids et les restaurera pour vous à la fin de l'entraînement.



Le package `keras.callbacks` propose de nombreux autres rappels (voir <https://keras.io/callbacks/>).

Si nous avons besoin d'un plus grand contrôle, nous pouvons aisément écrire nos propres rappels. Par exemple, le rappel personnalisé suivant affiche le rapport entre la perte de validation et la perte d'entraînement au cours de l'entraînement (par exemple, pour détecter un surajustement) :

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

Vous l'aurez compris, nous pouvons implémenter `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()` et `on_batch_end()`. En cas de besoin, les rappels peuvent également être employés au cours d'une évaluation et des prédictions, par exemple pour le débogage. Dans le cas de l'évaluation, nous pouvons implémenter `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()` ou `on_test_batch_end()` (appelées par `evaluate()`), tandis que, dans le cas des prédictions, nous pouvons mettre en œuvre `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()` ou `on_predict_batch_end()` (appelées par `predict()`).

Examinons à présent un autre instrument que vous devez absolument ajouter à votre boîte à outils si vous utilisez `tf.keras:TensorBoard`.

2.2.8 Utiliser TensorBoard pour la visualisation

TensorBoard est un excellent outil interactif de visualisation que nous pouvons employer pour examiner les courbes d'apprentissage pendant l'entraînement,

comparer les courbes d'apprentissage de plusieurs exécutions, visualiser le graphe de calcul, analyser des statistiques d'entraînement, afficher des images générées par notre modèle, visualiser des données multidimensionnelles complexes projetées en 3D et regroupées automatiquement pour nous, etc. Puisque cet outil est installé en même temps que TensorFlow, vous en disposez déjà.

Pour l'utiliser, nous devons modifier notre programme afin qu'il place les données à visualiser dans des fichiers de journalisation binaires spécifiques appelés *fichier d'événements*. Chaque enregistrement de données est appelé *résumé* (*summary*). Le serveur TensorBoard surveille le répertoire de journalisation et récupère automatiquement les modifications afin de mettre à jour les visualisations: nous pouvons ainsi visualiser des données dynamiques (avec un court délai), comme les courbes d'apprentissage pendant l'entraînement. En général, nous indiquons au serveur TensorBoard un répertoire de journalisation racine et configurons notre programme de sorte qu'il écrive dans un sous-répertoire différent à chaque exécution. De cette manière, la même instance de serveur TensorBoard nous permet de visualiser et de comparer des données issues de plusieurs exécutions du programme, sans que tout se mélange.

Commençons par définir le répertoire de journalisation racine dans lequel seront placés les journaux TensorBoard. Nous définissons également une petite fonction qui générera un chemin de sous-répertoire en fonction de la date et de l'heure courantes afin que le nom soit différent à chaque exécution. En ajoutant des informations supplémentaires dans le nom du répertoire de journalisation, comme des valeurs d'hyperparamètres testés, il sera plus facile de savoir ce que nous visualisons dans TensorBoard:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # Exemple : './my_logs/run_2019_06_07-15_15_22'
```

Point très intéressant, Keras fournit un rappel `TensorBoard()`:

```
[...] # Construire et compiler le modèle
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                     validation_data=(X_valid, y_valid),
                     callbacks=[tensorboard_cb])
```

Ce n'est pas plus compliqué que cela! Le rappel `TensorBoard()` se charge de la création du répertoire de journalisation (y compris les répertoires parents, si nécessaire) et, pendant l'entraînement, il crée des fichiers d'événements en y écrivant les résumés. Après une deuxième exécution du programme (éventuellement en modifiant la valeur d'un hyperparamètre), nous obtenons une structure de répertoires comparable à la suivante:

```

my_logs/
└── run_2019_06_07-15_15_22
    ├── train
    │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
    │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
    │   └── plugins/profile/2019-06-07_15-15-32
        └── local.trace
    └── validation
        └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
└── [...]

```

Il existe un répertoire pour chaque exécution, chacun contenant un sous-répertoire pour les journaux d'entraînement et un autre pour les journaux de validation. Ces deux sous-répertoires contiennent des fichiers d'événements, mais les journaux d'entraînement incluent également des informations de profilage. Cela permet à TensorBoard d'afficher précisément le temps passé par le modèle sur chacune de ses parties, sur tous les périphériques. Il sera ainsi plus facile de localiser les goulots d'étranglement des performances.

Nous devons ensuite démarrer le serveur TensorBoard. Pour cela, il suffit d'exécuter une commande dans une fenêtre de terminal. Si vous avez installé TensorFlow dans un environnement isolé avec `virtualenv`, vous devez l'activer. Ensuite, exéutez la commande suivante à la racine du projet (ou depuis tout autre endroit en indiquant le répertoire de journalisation appropriée) :

```

$ tensorboard --logdir=~/my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)

```

Si votre shell ne trouve pas le script `tensorboard`, mettez à jour la variable d'environnement `PATH` de sorte qu'elle contienne le répertoire dans lequel le script a été installé (ou bien, remplacez simplement `tensorboard` dans la ligne de commande par `python -m tensorboard.main`). Après que le serveur a démarré, vous pouvez ouvrir un navigateur web sur l'adresse `http://localhost:6006`.

Il est également possible d'utiliser TensorBoard directement depuis Jupyter en lançant les commandes suivantes. La première ligne charge l'extension TensorBoard, la seconde démarre un serveur TensorBoard sur le port 6006 (sauf s'il est déjà actif) et s'y connecte :

```

%load_ext tensorboard
%tensorboard --logdir=~/my_logs --port=6006

```

Quelle que soit la méthode employée, l'interface web de TensorBoard doit s'afficher. Cliquez sur l'onglet SCALARS pour visualiser les courbes d'apprentissage (voir la figure 2.17). Dans l'angle inférieur gauche, sélectionnez les journaux à visualiser (par exemple les journaux d'entraînement de la première et de la seconde exécution), puis cliquez sur `epoch_loss`. Vous remarquerez que la perte d'apprentissage a diminué pendant les deux exécutions mais que la baisse a été beaucoup plus rapide dans la seconde. Évidemment, car nous avons utilisé un taux d'apprentissage de 0,05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) à la place de 0,001.

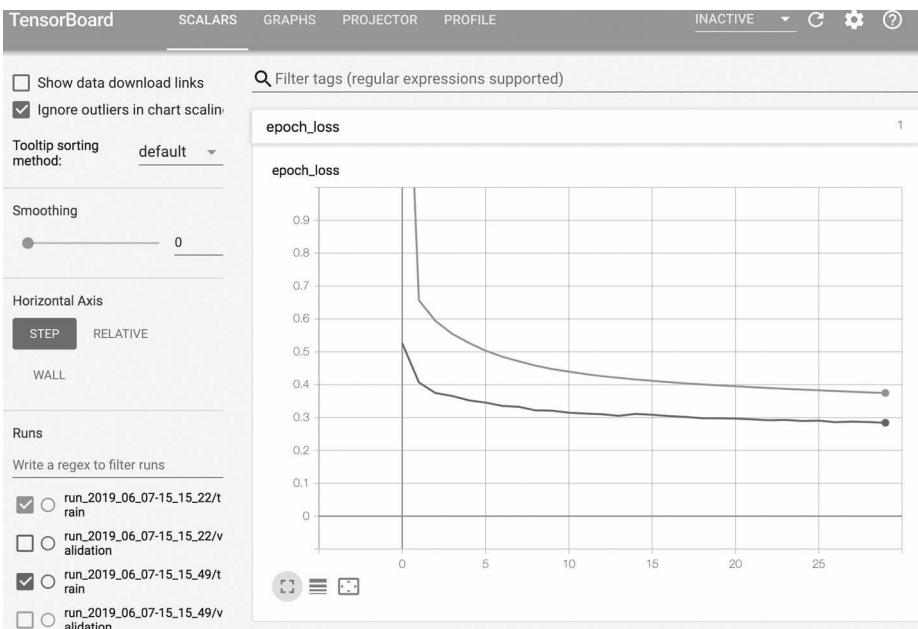


Figure 2.17 – Visualisation des courbes d'apprentissage avec TensorBoard

Vous pouvez également visualiser l'intégralité du graphe, les poids appris (avec une projection en 3D) et les informations de profilage. Le rappel `TensorBoard()` dispose également d'options pour journaliser des données supplémentaires, comme les plongements (voir le chapitre 5).

Par ailleurs, TensorFlow offre une API de bas niveau dans le package `tf.summary`. Le code suivant crée un `SummaryWriter` à l'aide de la fonction `create_file_writer()` et utilise cet écrivain comme contexte pour la journalisation des valeurs scalaires, des histogrammes, des images, de l'audio et du texte, autant d'éléments qui peuvent être visualisés avec TensorBoard (faites l'essai !) :

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100 # des données
                                                # aléatoires
        tf.summary.histogram("my_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3) # images RVB 32x32 aléatoires
        tf.summary.image("my_images", images * step / 1000, step=step)
        texts = ["The step is " + str(step), "Its square is " + str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

Cet outil de visualisation est en fait indispensable, même en dehors de TensorFlow ou du Deep Learning.

Faisons le point sur ce que vous avez appris jusqu'ici dans ce chapitre. Nous avons vu d'où viennent les réseaux de neurones, ce qu'est un MPC et comment l'utiliser pour la classification et la régression, comment utiliser l'API Sequential de tf.keras pour construire des MPC et comment employer l'API Functional ou Subclassing pour construire des modèles à l'architecture encore plus complexe. Nous avons expliqué comment enregistrer et restaurer un modèle, utiliser les rappels pour les sauvegardes intermédiaires, l'arrêt prématûr, et d'autres fonctions. Enfin, nous avons présenté la visualisation avec TensorBoard. Avec ces connaissances, vous pouvez déjà utiliser les réseaux de neurones pour attaquer de nombreux problèmes ! Mais vous vous demandez peut-être comment choisir le nombre de couches cachées, le nombre de neurones dans le réseau et tous les autres hyperparamètres. C'est ce que nous allons voir à présent.

2.3 RÉGLER PRÉCISEMENT LES HYPERPARAMÈTRES D'UN RÉSEAU DE NEURONES

La souplesse des réseaux de neurones constitue également l'un de leurs principaux inconvénients : de nombreux hyperparamètres doivent être ajustés. Il est non seulement possible d'utiliser n'importe quelle architecture de réseau imaginable mais, même dans un simple PMC, nous pouvons modifier le nombre de couches, le nombre de neurones par couche, le type de fonction d'activation employée par chaque couche, la logique d'initialisation des poids, etc. Dans ce cas, comment pouvons-nous connaître la combinaison d'hyperparamètres la mieux adaptée à une tâche ?

Une solution consiste simplement à tester de nombreuses combinaisons d'hyperparamètres et de voir celle qui donne les meilleurs résultats sur le jeu de validation (ou utiliser une validation croisée en K passes [*K-fold cross-validation*]). Par exemple, nous pouvons employer GridSearchCV ou RandomizedSearchCV de manière à explorer l'espace des hyperparamètres⁵⁴. Pour cela, nous devons placer nos modèles Keras dans des objets qui imitent des régresseurs Scikit-Learn normaux. La première étape est de créer une fonction qui construira et compilera un modèle Keras avec un ensemble donné d'hyperparamètres :

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3,
                input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

54. Voir le chapitre 2 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

Cette fonction crée un simple modèle Sequential pour une régression univariée (un seul neurone de sortie), avec la forme d'entrée ainsi que le nombre de couches cachées et de neurones indiqués. Elle le compile en utilisant un optimiseur SGD configuré avec le taux d'apprentissage fixé. La bonne pratique est de donner des valeurs par défaut raisonnables à autant d'hyperparamètres que possible, comme le fait Scikit-Learn.

Ensuite, nous créons un KerasRegressor à partir de cette fonction build_model() :

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

L'objet KerasRegressor constitue une fine enveloppe autour du modèle Keras construit en utilisant build_model(). Puisque nous n'avons spécifié aucun hyperparamètre au moment de la création, il utilisera les hyperparamètres par défaut définis dans build_model(). Nous pouvons alors utiliser cet objet comme un régresseur Scikit-Learn normal : nous pouvons invoquer sa méthode fit() pour l'entraîner, puis appeler sa méthode score() pour l'évaluer, et utiliser sa méthode predict() pour effectuer des prédictions :

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Tout paramètre supplémentaire passé à la méthode fit() sera transmis au modèle Keras sous-jacent. Le score sera également à l'opposé de la MSE car Scikit-Learn veut non pas des pertes mais des scores (autrement dit, plus c'est élevé, mieux c'est).

Nous voulons non pas entraîner et évaluer un seul modèle comme cela, mais entraîner des centaines de variantes et voir celle qui donne les meilleurs résultats sur le jeu de validation. Puisqu'il existe de nombreux hyperparamètres, il est préférable d'adopter une recherche aléatoire plutôt qu'une recherche par quadrillage⁵⁵. Appliquons cela au nombre de couches cachées, au nombre de neurones et au taux d'apprentissage :

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distrib = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distrib, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

55. Voir le chapitre 2 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

La procédure est identique à celle déroulée au chapitre 2 de l'ouvrage *Machine Learning avec Scikit-Learn*⁵⁶, à l'exception des paramètres supplémentaires passés à la méthode `fit()` et retransmis aux modèles Keras sous-jacents. Puisque `RandomizedSearchCV` utilise la validation croisée à K passes, il ne se sert pas de `X_valid` ni de `y_valid`, qui ne sont là que pour l'arrêt prématué.

L'exploration peut durer plusieurs heures, en fonction du matériel, de la taille du jeu de données, de la complexité du modèle et de la valeur de `n_iter` et de `cv`. À son terme, nous pouvons accéder aux meilleurs paramètres, au meilleur score et au modèle Keras entraîné :

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

Vous pouvez enregistrer ce modèle, l'évaluer sur le jeu de test et, si ses performances vous conviennent, le déployer en production. L'utilisation d'une recherche aléatoire n'est pas trop difficile et cette approche fonctionne bien pour de nombreux problèmes relativement simples. En revanche, lorsque l'entraînement est lent (par exemple pour des problèmes plus complexes avec des jeux de données volumineux), elle n'explorera qu'une minuscule portion de l'espace des hyperparamètres. Vous pouvez pallier ce problème en partie en assistant manuellement le processus de recherche : commencez par exécuter une recherche aléatoire rapide en choisissant des plages larges pour les valeurs des hyperparamètres, puis effectuez une autre recherche en prenant des plages plus réduites centrées sur les meilleures trouvées au cours de la première exécution, et ainsi de suite. Avec un peu de chance, cette méthode permettra d'aller vers un bon ensemble d'hyperparamètres, mais elle demande beaucoup de votre temps.

Heureusement, il existe de nombreuses techniques pour explorer un espace de recherche bien plus efficacement que de manière aléatoire. L'idée de base est simple : lorsqu'une région de l'espace se révèle intéressante, elle doit être explorée plus en profondeur. De telles techniques prennent en charge le processus de focalisation à notre place et conduisent à de meilleures solutions en beaucoup moins de temps. Voici quelques bibliothèques Python qui permettent d'optimiser les hyperparamètres :

- Hyperopt (<https://github.com/hyperopt/hyperopt>)

Une bibliothèque Python populaire pour optimiser toutes sortes d'espaces de recherche complexes (y compris les valeurs réelles, comme le taux d'apprentissage, et les valeurs discrètes, comme le nombre de couches).

- Hyperas (<https://github.com/maxpumperla/hyperas>), kopt (<https://github.com/Avsecz/kopt>) ou Talos (<https://github.com/autonomio/talos>)

Des bibliothèques utiles pour l'optimisation des hyperparamètres des modèles Keras (les deux premières se fondent sur Hyperopt).

- Keras Tuner (<https://homl.info/kerastuner>)
Une bibliothèque Google simple d'emploi pour l'optimisation des hyperparamètres des modèles Keras, avec un service hébergé pour la visualisation et l'analyse.
- Scikit-Optimize (skopt) (<https://scikit-optimize.github.io/>)
Une bibliothèque d'optimisation générale. La classe BayesSearchCV effectue une optimisation bayésienne en utilisant une interface comparable à celle de GridSearchCV.
- Spearmint (<https://github.com/JasperSnoek/spearmint>)
Une bibliothèque d'optimisation bayésienne.
- Hyperband (<https://github.com/zygmuntz/hyperband>)
Une bibliothèque rapide d'ajustement des hyperparamètres qui se fonde sur l'article Hyperband⁵⁷ de Lisha Li *et al.* paru en 2018.
- Sklearn-Deap (<https://github.com/rsteca/sklearn-deap>)
Une bibliothèque d'optimisation des hyperparamètres qui se fonde sur des algorithmes évolutionnistes, avec une interface à la GridSearchCV.

De nombreuses sociétés proposent également des services d'optimisation des hyperparamètres. Nous examinerons le service de réglage des hyperparamètres de Google Cloud AI Platform (<https://homl.info/googletuning>) au chapitre 11, mais Arimo (<https://arimo.com/>), SigOpt (<https://sigopt.com/>) ainsi que Oscar (<http://oscar.calldesk.ai/>) de CallDesk font partie des options possibles.

Le réglage des hyperparamètres est un domaine de recherche toujours actif et les algorithmes évolutionnistes font un retour. Par exemple, consultez l'excellent article de DeepMind publié en 2017 dans lequel les auteurs optimisent conjointement une population de modèles et leurs hyperparamètres⁵⁸. Google a également utilisé une approche évolutionniste, non seulement pour la recherche des hyperparamètres, mais également pour trouver l'architecture de réseau de neurones la plus adaptée au problème. Leur suite AutoML est disponible sous forme de service cloud (<https://cloud.google.com/automl/>). Les jours de la construction manuelle des réseaux de neurones sont peut-être comptés. À ce sujet, consultez le billet publié par Google (<https://homl.info/automlpost>). Des algorithmes évolutionnistes ont été employés avec succès pour entraîner des réseaux de neurones individuels, remplaçant l'omniprésente descente de gradient ! À titre d'exemple, consultez le billet publié en 2017 par Uber dans lequel les auteurs introduisent leur technique de Deep Neuroevolution (<https://homl.info/neuroevol>).

Malgré tous ces progrès formidables et tous ces outils et services, il est encore bon d'avoir une idée de ce que sont les valeurs raisonnables de chaque hyperparamètre afin que vous puissiez construire rapidement un prototype et limiter l'espace de recherche. Les sections suivantes font quelques recommandations pour le choix

57. Lisha Li *et al.*, « Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization », *Journal of Machine Learning Research*, 18 (2018), 1-52 : <https://homl.info/hyperband>.

58. Max Jaderberg *et al.*, « Population Based Training of Neural Networks » (2017) : <https://homl.info/pbt>.

du nombre de couches cachées et de neurones dans un MPC, et pour la sélection des bonnes valeurs de certains hyperparamètres principaux.

2.3.1 Nombre de couches cachées

Pour bon nombre de problèmes, nous pouvons commencer avec une seule couche cachée et obtenir des résultats raisonnables. Un PMC doté d'une seule couche cachée peut théoriquement modéliser les fonctions même les plus complexes pour peu qu'il possède suffisamment de neurones. Mais, pour des problèmes complexes, les réseaux profonds ont une *efficacité paramétrique* beaucoup plus élevée que les réseaux peu profonds. Ils peuvent modéliser des fonctions complexes avec un nombre de neurones exponentiellement plus faible que les réseaux superficiels, ce qui leur permet d'atteindre de meilleures performances avec la même quantité de données d'entraînement.

Afin de comprendre pourquoi, faisons une analogie. Supposons qu'on vous demande de dessiner une forêt à l'aide d'un logiciel de dessin, mais qu'il vous soit interdit d'utiliser le copier-coller. Cela prendrait énormément de temps : il faudrait alors dessiner chaque arbre individuellement, branche par branche, feuille par feuille. Si, à la place, vous pouvez dessiner une feuille, la copier-coller pour dessiner une branche, puis copier-coller cette branche pour créer un arbre, et finalement copier-coller cet arbre pour dessiner la forêt, vous aurez terminé en un rien de temps. Les données du monde réel ont souvent une telle structure hiérarchique et les RNP en tirent automatiquement profit. Les couches cachées inférieures modélisent des structures de bas niveau (par exemple, des traits aux formes et aux orientations variées). Les couches cachées intermédiaires combinent ces structures de bas niveau pour modéliser des structures de niveau intermédiaire (par exemple, des carrés et des cercles). Les couches cachées supérieures et la couche de sortie associent ces structures intermédiaires pour modéliser des structures de haut niveau (par exemple, des visages).

Cette architecture hiérarchique accélère non seulement la convergence des RNP vers une bonne solution, mais elle améliore également leur capacité à accepter de nouveaux jeux de données. Par exemple, si vous avez déjà entraîné un modèle afin de reconnaître les visages sur des photos et si vous souhaitez à présent entraîner un nouveau réseau de neurones pour reconnaître des coiffures, vous pouvez démarrer l'entraînement avec les couches inférieures du premier réseau. Au lieu d'initialiser aléatoirement les poids et les termes constants des quelques premières couches du nouveau réseau de neurones, vous pouvez les fixer aux valeurs des poids et des termes constants des couches basses du premier. De cette manière, le réseau n'aura pas besoin de réapprendre toutes les structures de bas niveau que l'on retrouve dans la plupart des images. Il devra uniquement apprendre celles de plus haut niveau (par exemple, les coupes de cheveux). C'est ce que l'on appelle le *transfert d'apprentissage*.

En résumé, pour de nombreux problèmes, vous pouvez démarrer avec juste une ou deux couches cachées, pour de bons résultats. Par exemple, vous pouvez facilement atteindre une précision supérieure à 97 % sur le jeu de données MNIST en utilisant une seule couche cachée et quelques centaines de neurones, mais une précision de plus de 98 % avec deux couches cachées et le même nombre total de neurones, pour

un temps d'entraînement quasi identique. Lorsque les problèmes sont plus complexes, vous pouvez augmenter progressivement le nombre de couches cachées, jusqu'à ce que vous arriviez au surajustement du jeu d'entraînement. Les tâches très complexes, comme la classification de grandes images ou la reconnaissance vocale, nécessitent en général des réseaux constitués de dizaines de couches (ou même des centaines, mais non intégralement connectées, comme nous le verrons au chapitre 6) et d'énormes quantités de données d'entraînement. Cependant, ces réseaux ont rarement besoin d'être entraînés à partir de zéro. Il est beaucoup plus fréquent de réutiliser des parties d'un réseau préentraîné qui effectue une tâche comparable. L'entraînement en devient beaucoup plus rapide et nécessite moins de données (nous y reviendrons au chapitre 3).

2.3.2 Nombre de neurones par couche cachée

Le nombre de neurones dans les couches d'entrée et de sortie est évidemment déterminé par le type des entrées et des sorties nécessaires à la tâche. Par exemple, la tâche MNIST exige $28 \times 28 = 784$ neurones d'entrée et 10 neurones de sortie.

Une pratique courante consiste à dimensionner les couches cachées de façon à former un entonnoir, avec un nombre de neurones toujours plus faible à chaque couche. La logique est que de nombreuses caractéristiques de bas niveau peuvent se fondre dans un nombre de caractéristiques de haut niveau moindre. Par exemple, un réseau de neurones type pour MNIST peut comprendre trois couches cachées, la première avec 300 neurones, la deuxième, 200, et la troisième, 100. Cependant, cette pratique a été largement abandonnée car il semble qu'utiliser le même nombre de neurones dans toutes les couches cachées donne d'aussi bonnes, voire meilleures, performances dans la plupart des cas. Par ailleurs, cela fait un seul hyperparamètre à ajuster au lieu d'un par couche. Cela dit, en fonction du jeu de données, il peut être parfois utile d'avoir une première couche cachée plus importante que les autres.

Comme pour le nombre de couches, vous pouvez augmenter progressivement le nombre de neurones, jusqu'au surajustement du réseau. Mais, en pratique, il sera souvent plus simple et plus efficace de choisir un modèle avec plus de couches et de neurones que nécessaire, puis d'utiliser l'arrêt prématûre et d'autres techniques de régularisation de façon à éviter le surajustement. Vincent Vanhoucke, ingénieur chez Google, a surnommé cette approche « pantalon à taille élastique » : au lieu de perdre du temps à rechercher un pantalon qui correspond parfaitement à votre taille, il suffit d'utiliser un pantalon élastique qui s'adaptera à la bonne taille. Grâce à elle, nous évitons des couches d'étranglement qui risquent de ruiner votre modèle. En revanche, si une couche comprend trop peu de neurones, elle n'aura pas assez de puissance de représentation pour conserver toutes les informations utiles provenant des entrées (par exemple, une couche avec deux neurones ne peut produire que des données 2D et, si elle doit traiter des données 3D, certaines informations seront perdues). Quelles que soient la taille et la puissance du reste du réseau, cette information ne pourra pas être retrouvée.



En général, il sera plus intéressant d'augmenter le nombre de couches que le nombre de neurones par couche.

2.3.3 Taux d'apprentissage, taille de lot et autres hyperparamètres

Les nombres de couches cachées et de neurones ne sont pas les seuls hyperparamètres que vous pouvez ajuster dans un MPC. En voici quelques autres parmi les plus importants, avec des conseils pour le choix de leur valeur :

- *Taux d'apprentissage*

Le taux d'apprentissage est probablement l'hyperparamètre le plus important. En général, sa valeur optimale est environ la moitié du taux d'apprentissage maximal (c'est-à-dire, le taux d'apprentissage au-dessus duquel l'algorithme d'entraînement diverge⁵⁹). Une bonne manière de déterminer un taux d'apprentissage approprié consiste à entraîner le modèle sur quelques centaines d'itérations, en commençant avec un taux d'apprentissage très bas (par exemple 10^{-5}) et de l'augmenter progressivement jusqu'à une valeur très grande (par exemple 10). Pour cela, on multiplie le taux d'apprentissage par un facteur constant à chaque itération (par exemple $\exp(\log(10^6)/500)$) de façon à aller de 10^{-5} à 10 en 500 itérations). Si la perte est une fonction du taux d'apprentissage (en utilisant une échelle logarithmique pour le taux d'apprentissage), elle doit commencer par baisser. Après un certain temps, le taux d'apprentissage sera beaucoup trop grand et la perte reviendra : le taux d'apprentissage optimal se situera un peu avant le point à partir duquel la perte a commencé à grimper (typiquement environ 10 fois plus faible qu'au point de revirement). Vous pouvez ensuite réinitialiser le modèle et l'entraîner de façon normale en utilisant ce taux d'apprentissage pertinent. Nous reviendrons sur d'autres techniques de taux apprentissage au chapitre 3.

- *Optimiseur*

Le choix d'un optimiseur plus performant que la pure et simple descente de gradient par mini-lots (avec l'ajustement de ses hyperparamètres) est également très important. Nous verrons plusieurs optimiseurs élaborés au chapitre 3.

- *Taille des lots*

La taille des lots peut également avoir un impact significatif sur les performances du modèle et le temps d'entraînement. Une taille de lot importante a l'avantage d'exploiter pleinement les accélérateurs matériels, comme les GPU (voir le chapitre 11), ce qui permet à l'algorithme d'entraînement de voir plus d'instances chaque seconde. C'est pourquoi de nombreux chercheurs et professionnels conseillent de choisir une taille de lot aussi grande que possible adaptée à la mémoire RAM du GPU. Il y a cependant un problème. Dans la pratique, les tailles de lots élevées conduisent souvent à des instabilités de l'entraînement, en particulier au début, et la généralisation du modèle résultant risque d'être

moins facile que celle d'un modèle entraîné avec une taille de lot plus faible. En avril 2018, Yann LeCun a même tweeté « Friends don't let friends use mini-batches larger than 32 » (les vrais amis ne laissent pas leurs amis utiliser des mini-lots de taille supérieure à 32), citant un article⁶⁰ publié en 2018 par Dominic Masters et Carlo Luschi dans lequel les auteurs concluent que les petits lots (de 2 à 32) sont préférables car ils produisent de meilleurs modèles pour un temps d'entraînement inférieur. D'autres articles vont cependant à l'opposé : en 2017, Elad Hoffer *et al.*⁶¹, ainsi que Priya Goyal *et al.*⁶², ont montré qu'il était possible de choisir de très grandes tailles de lots (jusqu'à 8192) en utilisant diverses techniques, comme l'échauffement du taux d'apprentissage (c'est-à-dire démarrer l'entraînement avec un taux d'apprentissage faible, puis l'augmenter progressivement, comme nous le verrons au chapitre 3). Cela permet d'avoir un temps d'entraînement très court, sans impact sur la généralisation. Une stratégie consiste donc à essayer une taille de lot importante, en utilisant une phase d'échauffement du taux d'apprentissage, et, si l'entraînement est instable ou les performances finales décevantes, à basculer sur une taille de lot réduite.

- *Fonction d'activation*

Nous avons expliqué comment choisir la fonction d'activation précédemment dans ce chapitre : la fonction ReLU sera en général un bon choix par défaut pour toutes les couches cachées. Pour la couche de sortie, cela dépend réellement de la tâche.

- *Nombre d'itérations*

Dans la plupart des cas, le nombre d'itérations d'entraînement n'a pas besoin d'être ajusté : il suffit d'utiliser, à la place, de l'arrêt prématué.



Le taux d'apprentissage optimal dépend des autres paramètres, en particulier de la taille du lot. Par conséquent, si vous modifiez n'importe quel hyperparamètre, n'oubliez pas de revoir le taux d'apprentissage.

Leslie Smith a publié en 2018 un excellent article⁶³ qui regorge de bonnes pratiques concernant le réglage précis des hyperparamètres d'un réseau de neurones. N'hésitez pas à le consulter.

Voilà qui conclut notre introduction aux réseaux de neurones artificiels et à leur mise en œuvre avec Keras. Dans les prochains chapitres, nous présenterons des

60. Dominic Masters et Carlo Luschi, « Revisiting Small Batch Training for Deep Neural Networks » (2018) : <https://homl.info/smallbatch>.

61. Elad Hoffer *et al.*, « Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks », *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), 1729-1739 : <https://homl.info/largebatch>.

62. Priya Goyal *et al.*, *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour* (2017) : <https://homl.info/largebatch2>.

63. Leslie N. Smith, « A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay » (2018) : <https://homl.info/1cycle>.

techniques d'entraînement de réseaux très profonds. Nous verrons également comment personnaliser les modèles avec l'API de bas niveau de TensorFlow et comment charger et prétraiter efficacement des données avec l'API Data. Nous détaillerons d'autres architectures de réseaux de neurones répandues: réseaux de neurones convulsifs pour le traitement des images, réseaux de neurones récurrents pour les données séquentielles, autoencodeurs pour l'apprentissage des représentations et réseaux antagonistes génératifs pour la modélisation et la génération de données⁶⁴.

2.4 EXERCICES

1. TensorFlow Playground (<https://playground.tensorflow.org/>) est un simulateur de réseaux de neurones très intéressant développé par l'équipe de TensorFlow. Dans cet exercice, vous entraînerez plusieurs classificateurs binaires en seulement quelques clics et ajusterez l'architecture du modèle et ses hyperparamètres afin de vous familiariser avec le fonctionnement des réseaux de neurones et le rôle de leurs hyperparamètres. Prenez le temps d'explorer les aspects suivants:
 - a. Motifs appris par un réseau de neurones. Entraînez le réseau de neurones par défaut en cliquant sur le bouton d'exécution (Run, en haut à gauche). Notez la rapidité avec laquelle il trouve une bonne solution au problème de classification. Les neurones de la première couche cachée ont appris des motifs simples, tandis que ceux de la seconde ont appris à combiner ces motifs simples en motifs complexes. En général, plus le nombre de couches est élevé, plus les motifs peuvent être complexes.
 - b. Fonctions d'activation. Remplacez la fonction d'activation Tanh par la fonction ReLU et entraînez de nouveau le réseau. Notez que la découverte de la solution est encore plus rapide mais que, cette fois-ci, les frontières sont linéaires. Cela provient de la forme de la fonction ReLU.
 - c. Risques de minima locaux. Modifiez l'architecture du réseau de sorte qu'elle comprenne une couche cachée avec trois neurones. Entraînez le réseau à plusieurs reprises (pour réinitialiser les poids du réseau, cliquez sur le bouton de réinitialisation Reset situé à gauche du bouton de lecture Play). Notez que le temps d'entraînement varie énormément et que le processus peut même rester bloqué dans un minimum local.
 - d. Comportement lorsqu'un réseau de neurones est trop petit. Supprimez à présent un neurone pour n'en conserver que deux. Vous remarquerez que le réseau est alors incapable de trouver une bonne solution, même si vous l'exécutez à plusieurs reprises. Le modèle

64. Quelques autres architectures de RNA sont présentées dans l'annexe C.

- possède trop peu de paramètres et sous-ajuste systématiquement le jeu d'entraînement.
- e. Comportement lorsqu'un réseau de neurones est suffisamment large. Fixez le nombre de neurones à huit et entraînez le modèle plusieurs fois. Notez qu'il est à présent invariablement rapide et ne reste jamais bloqué. Cela révèle une découverte importante dans la théorie des réseaux de neurones : les réseaux de neurones vastes ne restent quasiment jamais bloqués dans des minima locaux et, même lorsqu'ils le sont, ces optima locaux sont presque aussi bons que l'optimum global. Toutefois, les réseaux peuvent rester coincés sur de longs plateaux pendant un long moment.
 - f. Risque de disparition des gradients dans des réseaux profonds. Sélectionnez à présent le jeu de données Spiral (celui dans l'angle inférieur droit sous «DATA») et modifiez l'architecture du réseau en lui donnant quatre couches cachées de huit neurones chacune. L'entraînement prend alors beaucoup plus de temps et reste souvent bloqué sur des plateaux pendant de longues périodes. Notez également que les neurones des couches supérieures (celles sur la droite) tendent à évoluer plus rapidement que ceux des couches inférieures (celles sur la gauche). Ce problème de «disparition des gradients» peut être réduit avec une meilleure initialisation des poids et d'autres techniques, de meilleurs optimiseurs (comme AdaGrad ou Adam) ou la normalisation par lots (voir le chapitre 3).
 - g. Allez plus loin. Prenez au moins une heure pour jouer avec les autres paramètres et constater leurs actions. Vous développerez ainsi une compréhension intuitive des réseaux de neurones.
2. Avec les neurones artificiels d'origine (comme ceux de la figure 2.3), dessinez un RNA qui calcule $A \oplus B$ (où \oplus représente l'opération de OU exclusif, XOR). Un indice : $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
 3. Pourquoi est-il, en général, préférable d'utiliser un classificateur à régression logistique plutôt qu'un perceptron classique (c'est-à-dire une seule couche d'unités logiques à seuils entraînées à l'aide de l'algorithme d'entraînement du perceptron)? Comment pouvez-vous modifier un perceptron pour qu'il soit équivalent à un classificateur à régression logistique?
 4. Pourquoi la fonction d'activation logistique était-elle un élément indispensable dans l'entraînement des premiers PMC?
 5. Nommez trois fonctions d'activation répandues. Savez-vous les tracer?
 6. Supposons que vous disposiez d'un PMC constitué d'une couche d'entrée avec dix neurones intermédiaires, suivie d'une couche cachée de cinquante neurones artificiels, et d'une couche de sortie avec trois neurones artificiels. Tous les neurones artificiels utilisent la fonction d'activation ReLU.

- a. Quelle est la forme de la matrice d'entrée \mathbf{X} ?
 - b. Quelle est la forme du vecteur des poids \mathbf{W}_h de la couche cachée et celle de son vecteur de termes constants \mathbf{b}_h ?
 - c. Quelle est la forme du vecteur des poids \mathbf{W}_o de la couche de sortie et celle de son vecteur de termes constants \mathbf{b}_o ?
 - d. Quelle est la forme de la matrice de sortie \mathbf{Y} du réseau?
 - e. Écrivez l'équation qui calcule la matrice de sortie \mathbf{Y} du réseau en fonction de \mathbf{X} , \mathbf{W}_h , \mathbf{b}_h , \mathbf{W}_o et \mathbf{b}_o .
7. Combien de neurones faut-il dans la couche de sortie pour classer des courriers électroniques dans les catégories *spam* ou *ham*? Quelle fonction d'activation devez-vous choisir dans la couche de sortie? Si, à la place, vous voulez traiter le jeu MNIST, combien de neurones devez-vous placer dans la couche de sortie, avec quelle fonction d'activation? Reprenez ces mêmes questions pour un réseau qui doit prédire le prix des maisons dans le jeu de données California Housing, que vous pouvez charger à l'aide de la fonction `sklearn.datasets.fetch_california_housing()`.
8. Qu'est-ce que la rétropropagation et comment opère-t-elle? Quelle est la différence entre la rétropropagation et la différentiation automatique en mode inverse?
9. Quels sont les hyperparamètres ajustables dans un PMC? Si le PMC surajuste les données d'entraînement, comment modifier les hyperparamètres pour résoudre ce problème?
10. Entraînez un PMC profond sur le jeu de données MNIST (vous pouvez le charger à l'aide de la fonction `keras.datasets.mnist.load_data()`) et voyez si vous pouvez obtenir une précision supérieure à 98 %. Essayez de rechercher le taux d'apprentissage optimal à l'aide de la méthode décrite dans ce chapitre (c'est-à-dire en augmentant le taux d'apprentissage de façon exponentielle, en affichant la perte et en déterminant le point où la perte explose). Essayez d'ajouter toutes les fioritures d'implémentation (sauvegarde de points de restauration, utilisation de l'arrêt prématûr et tracé des courbes d'apprentissage avec TensorBoard).

Les solutions de ces exercices sont données à l'annexe A.

3

Entraînement de réseaux de neurones profonds

Au chapitre 2, nous avons introduit les réseaux de neurones artificiels et entraîné nos premiers réseaux de neurones profonds. Il s'agissait de réseaux assez peu profonds, avec seulement quelques couches cachées. Comment devons-nous procéder dans le cas d'un problème très complexe, comme la détection de centaines de types d'objets dans des images en haute résolution ? Nous aurons alors probablement à entraîner un RNP beaucoup plus profond, avec peut-être des dizaines de couches, chacune contenant des centaines de neurones, reliés par des centaines de milliers de connexions. Ce n'est plus du tout une promenade de santé. Voici quelques-uns des problèmes que nous pourrions rencontrer :

- Nous pourrions être confrontés au problème difficile de *disparition des gradients*, ou au problème connexe d'*explosion des gradients*. Ils correspondent à des gradients qui deviennent de plus en plus petits ou de plus en plus grands pendant l'entraînement, au cours de la passe en arrière sur le RNP. Tous deux compliquent énormément l'entraînement des couches inférieures.
- Nous pourrions manquer de données d'entraînement pour un réseau aussi vaste, ou leur étiquetage pourrait être trop coûteux.
- L'entraînement pourrait être extrêmement lent.
- Un modèle comprenant des millions de paramètres risquera fort de conduire au surajustement du jeu d'entraînement, en particulier si nous n'avons pas assez d'instances d'entraînement ou si elles contiennent beaucoup de bruit.

Dans ce chapitre, nous allons examiner chacun de ces problèmes et proposer des techniques pour les résoudre. Nous commencerons par explorer le problème de disparition et d'explosion des gradients et présenterons quelques-unes des solutions les plus répandues. Nous aborderons ensuite le transfert d'apprentissage et le préentraînement non supervisé, qui peuvent nous aider à traiter des problèmes complexes

même lorsque les données étiquetées sont peu nombreuses. Puis nous examinerons différents optimiseurs qui permettent d'accélérer énormément l'entraînement des grands modèles. Enfin, nous décrirons quelques techniques de régularisation adaptées aux vastes réseaux de neurones.

Armés de ces outils, nous serons en mesure d'entraîner des réseaux très profonds : bienvenue dans le monde du Deep Learning !

3.1 PROBLÈMES DE DISPARITION ET D'EXPLOSION DES GRADIENTS

Nous l'avons expliqué au chapitre 2, l'algorithme de rétropropagation opère de la couche de sortie vers la couche d'entrée, en propageant au fur et à mesure le gradient d'erreur. Lorsque l'algorithme a déterminé le gradient de la fonction de coût par rapport à chaque paramètre du réseau, il utilise les gradients obtenus pour actualiser chaque paramètre au cours d'une étape de descente du gradient.

Malheureusement, alors que l'algorithme progresse vers les couches inférieures, les gradients deviennent souvent de plus en plus petits. Par conséquent, la mise à jour par descente de gradient modifie très peu les poids des connexions des couches inférieures et l'entraînement ne converge jamais vers une bonne solution. Tel est le problème de *disparition des gradients* (*vanishing gradients*). Dans certains cas, l'opposé peut se produire. Les gradients deviennent de plus en plus gros et de nombreuses couches reçoivent des poids extrêmement importants, ce qui fait diverger l'algorithme. Il s'agit du problème d'*explosion des gradients* (*exploding gradients*), qui se rencontre principalement dans les réseaux de neurones récurrents (voir le chapitre 7). Plus généralement, les réseaux de neurones profonds souffrent de l'instabilité des gradients : différentes couches peuvent apprendre à des vitesses très différentes.

Ce comportement malheureux a été observé empiriquement depuis un moment et il est l'une des raisons de l'abandon des réseaux de neurones profonds au début des années 2000. L'origine de cette instabilité des gradients pendant l'entraînement n'était pas claire, mais un éclairage est venu dans un article⁶⁵ publié en 2010 par Xavier Glorot et Yoshua Bengio. Les auteurs ont identifié quelques suspects, notamment l'association de la fonction d'activation logistique sigmoïde répandue à l'époque et de la technique d'initialisation des poids également la plus répandue à ce moment-là, à savoir une loi de distribution normale de moyenne zéro et d'écart-type 1. En résumé, ils ont montré que, avec cette fonction d'activation et cette technique d'initialisation, la variance des sorties de chaque couche est largement supérieure à celle de ses entrées. Lors de la progression dans le réseau, la variance ne cesse d'augmenter après chaque couche, jusqu'à la saturation de la fonction d'activation dans les couches supérieures. Ce comportement est aggravé par le fait que la moyenne de la fonction logistique est non pas de 0 mais de 0,5 (la fonction tangente hyperbolique a une

65. Xavier Glorot et Yoshua Bengio, « Understanding the Difficulty of Training Deep Feedforward Neural Networks », *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010), 249-256 : <https://hml.info/47>.

moyenne égale à 0 et se comporte légèrement mieux que la fonction logistique dans les réseaux profonds).

Si vous examinez la fonction d'activation logistique (voir la figure 3.1), vous constatez que, lorsque les entrées deviennent grandes (en négatif ou en positif), la fonction sature en 0 ou en 1, avec une dérivée extrêmement proche de 0. Lorsque la rétropropagation intervient, elle n'a pratiquement aucun gradient à transmettre en arrière dans le réseau. En outre, le faible gradient existant est de plus en plus dilué pendant la rétropropagation, à chaque couche traversée depuis les couches supérieures. Il ne reste donc quasi plus rien aux couches inférieures.

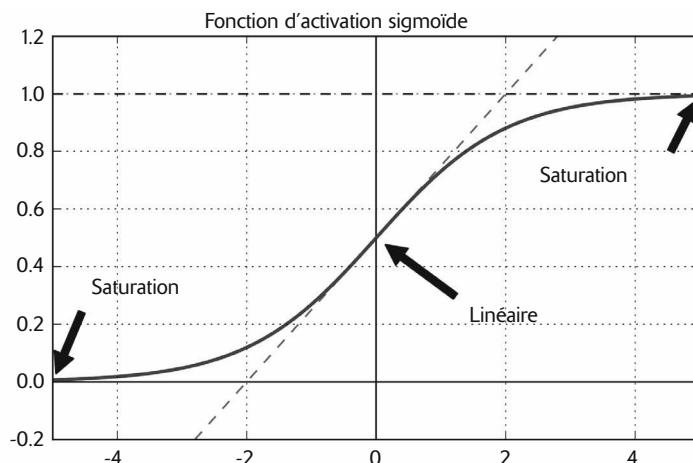


Figure 3.1 – Saturation de la fonction d'activation logistique

3.1.1 Initialisations de Glorot et de He

Dans leur article, Glorot et Bengio proposent une manière d'atténuer énormément ce problème d'instabilité des gradients. Ils soulignent que le signal doit se propager correctement dans les deux directions : vers l'avant au moment des prédictions, et vers l'arrière lors de la rétropropagation des gradients. Il ne faut pas que le signal disparaîtse, ni qu'il explose et sature. Pour que tout se passe proprement, les auteurs soutiennent que la variance des sorties de chaque couche doit être égale à la variance de ses entrées⁶⁶ et les gradients doivent également avoir une même variance avant et après le passage au travers d'une couche en sens inverse (les détails mathématiques se trouvent dans l'article). Il est impossible de garantir ces deux points, sauf si la couche possède un nombre

66. Voici une analogie. Si l'amplificateur d'un microphone est réglé près de zéro, le public n'entendra pas votre voix. S'il est réglé trop près du maximum, votre voix sera saturée et le public ne comprendra pas ce que vous direz. Imaginons à présent une série de tels amplificateurs. Ils doivent tous être réglés correctement pour que votre voix arrive parfaitement claire et audible à l'extrémité de la chaîne. Elle doit sortir de chaque amplificateur avec la même amplitude qu'en entrée.

égal de connexions d'entrée et de neurones (ces nombres sont appelés *fan-in* et *fan-out* de la couche). Ils ont cependant proposé un bon compromis, dont la validité a été montrée en pratique : les poids des connexions doivent être initialisés de façon aléatoire, comme dans l'équation 3.1, où $fan_{\text{moyen}} = \frac{(fan_{\text{entrée}} + fan_{\text{sortie}})}{2}$. Cette stratégie d'initialisation est appelée *initialisation de Xavier* ou *initialisation de Glorot*, d'après le nom du premier auteur de l'article.

Équation 3.1 – Initialisation de Glorot (si la fonction d'activation logistique est employée)

Distribution normale avec une moyenne de 0 et un écart-type $\sigma^2 = \frac{1}{fan_{\text{moyen}}}$

Ou une distribution uniforme entre $-r$ et $+r$, avec $r = \sqrt{\frac{3}{fan_{\text{moyen}}}}$

Si vous remplacez fan_{moyen} par $fan_{\text{entrée}}$ dans l'équation 3.1, vous obtenez la stratégie d'initialisation proposée dans les années 1990 par Yann LeCun, qu'il a nommée *initialisation de LeCun*. Genevieve Orr et Klaus-Robert Müller l'ont même conseillée dans leur ouvrage *Neural Networks: Tricks of the Trade* publié en 1998 (Springer). L'initialisation de LeCun est équivalente à l'initialisation de Glorot lorsque $fan_{\text{entrée}} = fan_{\text{sortie}}$. Il a fallu aux chercheurs plus d'une dizaine d'années pour réaliser l'importance de cette astuce. Grâce à l'initialisation de Glorot, l'entraînement est considérablement accéléré et elle représente l'une des astuces qui ont mené au succès du Deep Learning.

Certains articles récents⁶⁷ ont proposé des stratégies comparables pour d'autres fonctions d'activation. Elles diffèrent uniquement par l'échelle de la variance et par l'utilisation ou non de fan_{moyen} ou de $fan_{\text{entrée}}$ (voir le tableau 3.1) ; pour la distribution uniforme, on calcule simplement $r = \sqrt{3\sigma^2}$. La stratégie d'initialisation (<https://homl.info/48>) pour la fonction d'activation ReLU (et ses variantes, y compris la fonction d'activation ELU décrite ci-après) est parfois appelée *initialisation de He* (du nom du premier auteur de l'article). La fonction d'activation SELU sera expliquée plus loin dans ce chapitre. Elle doit être employée avec l'initialisation de LeCun (de préférence avec une fonction de distribution normale).

Tableau 3.1 – Paramètres d'initialisation pour chaque type de fonction d'activation

Initialisation	Fonctions d'activation	σ^2 (Normal)
Glorot	Aucun, tanh, logistique, softmax	$1/fan_{\text{moyen}}$
He	ReLU et ses variantes	$2/fan_{\text{entrée}}$
LeCun	SEL	$1/fan_{\text{entrée}}$

67. Par exemple, Kaiming He *et al.*, « Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification », *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015), 1026-1034.

Keras choisit par défaut l'initialisation de Glorot avec une distribution uniforme. Lors de la création d'une couche, nous pouvons opter pour l'initialisation de He en précisant `kernel_initializer="he_uniform"` ou `kernel_initializer="he_normal"`:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Si vous souhaitez une initialisation de He avec une distribution uniforme fondée sur fan_{moyen} plutôt que sur $fan_{entrée}$, vous pouvez employer l'initialiseur `VarianceScaling`:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

3.1.2 Fonctions d'activation non saturantes

L'un des enseignements apportés par l'article de Glorot et Bengio publié en 2010 a été que l'instabilité des gradients était en partie due à un mauvais choix de la fonction d'activation. Jusqu'alors, la plupart des gens supposaient que si Mère Nature avait choisi des fonctions d'activation à peu près sigmoïdes pour les neurones biologiques, c'est qu'elles devaient constituer un excellent choix. Cependant, d'autres fonctions d'activation affichent un meilleur comportement dans les réseaux de neurones profonds, notamment la fonction ReLU, principalement parce qu'elle ne sature pas pour les valeurs positives (et aussi parce qu'elle est plutôt rapide à calculer).

Malheureusement, la fonction d'activation ReLU n'est pas parfaite. Elle souffre d'un problème de *mort des ReLU* (*dying ReLU*): au cours de l'entraînement, certains neurones « meurent », c'est-à-dire arrêtent de produire autre chose que 0. Dans certains cas, il arrive que la moitié des neurones du réseau soient morts, en particulier si le taux d'apprentissage est grand. Un neurone meurt lorsque ses poids sont ajustés de sorte que la somme pondérée de ses entrées est négative pour toutes les instances du jeu d'entraînement. Lorsque cela arrive, il produit uniquement des zéros en sortie et la descente de gradient ne l'affecte plus, car le gradient de la fonction ReLU vaut zéro lorsque son entrée est négative⁶⁸.

Pour résoudre ce problème, on peut employer une variante de la fonction ReLU, par exemple *leaky ReLU* (*leaky* signifie « qui fuit »). Cette fonction se définit comme $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (voir la figure 3.2). L'hyperparamètre α définit le niveau de « fuite » de la fonction : il s'agit de la pente de la fonction pour $z < 0$. En général, il est fixé à 0,01. Grâce à cette petite pente, les « leaky ReLU » ne meurent jamais. Ils peuvent entrer dans un long coma, mais ils ont toujours une chance de se réveiller à un moment ou à un autre. Un article⁶⁹ de 2015 compare plusieurs variantes de la fonction d'activation ReLU et conclut notamment que la variante « leaky » est toujours plus performante que la version stricte. En réalité, en fixant

68. Sauf s'il fait partie de la première couche cachée, un neurone mort peut parfois ressusciter : la descente de gradient peut ajuster des neurones dans les couches inférieures de façon telle que la somme pondérée des entrées du neurone mort redevient positive.

69. Bing Xu *et al.*, « Empirical Evaluation of Rectified Activations in Convolutional Network » (2015) : <https://homl.info/49>.

α à 0,2 (fuite importante), il semble que les performances soient toujours meilleures qu'avec $\alpha = 0,01$ (fuite légère). Les auteurs ont également évalué la fonction RReLU (*Randomized leaky ReLU*), dans laquelle α est, pendant l'entraînement, choisi aléatoirement dans une plage donnée et, pendant les tests, fixé à une valeur moyenne. Elle donne de bons résultats et semble agir comme un régulariseur (en réduisant le risque de surajustement du jeu d'entraînement).

Enfin, ils ont également évalué la fonction PReLU (*Parametric leaky ReLU*), qui autorise l'apprentissage de α pendant l'entraînement (il n'est plus un hyperparamètre mais un paramètre du modèle qui, au même titre que les autres, peut être modifié par la rétropropagation). Elle surpassé largement la fonction ReLU sur les vastes jeux de données d'images, mais, avec les jeux de données plus petits, elle fait courir un risque de surajustement.

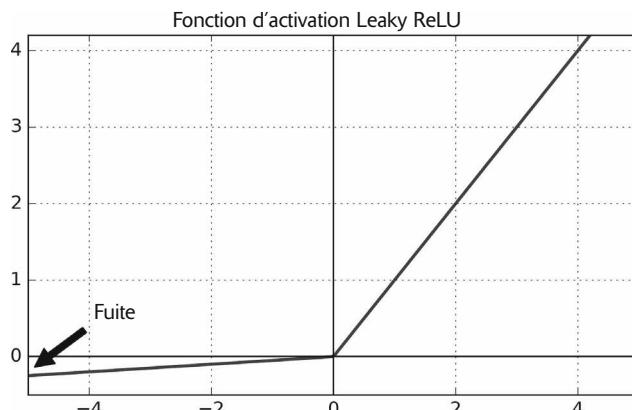


Figure 3.2 – Fonction leaky ReLU: comme ReLU, mais avec une petite pente pour les valeurs négatives

Par ailleurs, un article⁷⁰ de 2015 publié par Djork-Arné Clevert *et al.* propose une nouvelle fonction d'activation appelée ELU (*Exponential Linear Unit*), qui s'est montrée bien plus performante que toutes les variantes de ReLU dans leurs expérimentations. Le temps d'entraînement a diminué et le réseau de neurones s'est mieux comporté sur le jeu de test. Elle est représentée à la figure 3.3, et l'équation 3.2 en donne la définition.

70. Djork-Arné Clevert *et al.*, «Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)», *Proceedings of the International Conference on Learning Representations* (2016) : <https://homl.info/50>.

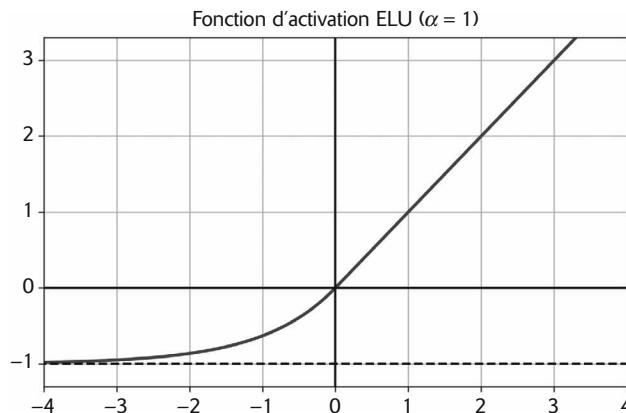


Figure 3.3 – Fonction d'activation ELU

Équation 3.2 – Fonction d'activation ELU

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{si } z < 0 \\ z & \text{si } z \geq 0 \end{cases}$$

La fonction d'activation ELU ressemble énormément à la fonction ReLU, mais avec quelques différences majeures :

- Elle prend en charge les valeurs négatives lorsque $z < 0$, ce qui permet au neurone d'avoir une sortie moyenne plus proche de zéro et aide à atténuer le problème de disparition des gradients. L'hyperparamètre α définit la valeur vers laquelle la fonction ELU tend lorsque z est un grand nombre négatif. Il est généralement fixé à 1, mais nous pouvons l'ajuster au besoin, comme n'importe quel autre hyperparamètre.
- Elle présente un gradient différent de zéro pour $z < 0$, ce qui évite le problème de mort des neurones.
- Si α est égal à 1, la fonction est continue en tout point, y compris autour de $z = 0$, ce qui permet d'accélérer la descente de gradient, car il ne rebondit pas autant à gauche et à droite de $z = 0$.

Elle a pour principal inconvénient d'être plus lente à calculer que la fonction ReLU et ses variantes (en raison de l'utilisation de la fonction exponentielle), mais, au cours de l'entraînement, cette lenteur est compensée par une vitesse de convergence plus élevée. En revanche, lors des tests, un réseau ELU sera plus lent qu'un réseau ReLU.

Pour finir, un article⁷¹ publié en 2017 par Günter Klambauer *et al.* a introduit la fonction d'activation SELU (Scaled ELU), qui, comme son nom le suggère,

71. Günter Klambauer *et al.*, « Self-Normalizing Neural Networks », *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), 972-981 : <https://homl.info/selu>.

est la fonction ELU redimensionnée. Les auteurs ont montré que si nous construisons un réseau de neurones constitué exclusivement d'une pile de couches denses et que si toutes les couches cachées utilisent la fonction d'activation SELU, alors le réseau pourra *s'autonormaliser*: pendant l'entraînement, la sortie de chaque couche aura tendance à conserver une moyenne égale à 0 et un écart-type standard égal à 1, ce qui résout les problèmes de disparition et d'explosion des gradients. Par conséquent, la fonction d'activation SELU surpassé souvent les autres fonctions d'activation dans de tels réseaux de neurones (en particulier s'ils sont profonds). Cependant, pour que l'autonormalisation se produise, quelques conditions doivent être respectées (pour la justification mathématique, voir l'article) :

- Les caractéristiques d'entrée doivent être normalisées (moyenne égale à 0 et écart-type égal à 1).
- Les poids de chaque couche cachée doivent être initialisés à l'aide de l'initialisation normale de LeCun. Dans Keras, cela signifie indiquer `kernel_initializer="lecun_normal"`.
- L'architecture du réseau doit être séquentielle. Malheureusement, si nous utilisons SELU dans des architectures non séquentielles, comme les réseaux récurrents (voir le chapitre 7) ou les réseaux avec des connexions de saut (c'est-à-dire des connexions qui sautent des couches, comme dans les réseaux Wide & Deep), l'autonormalisation n'est pas garantie et SELU ne surpassera pas nécessairement d'autres fonctions d'activation.
- L'article précise que l'autonormalisation n'est garantie que si toutes les couches sont denses, mais des chercheurs ont remarqué que la fonction SELU permet d'améliorer les performances dans les réseaux de neurones convolutifs (voir le chapitre 6).



Quelle fonction d'activation doit-on donc utiliser dans les couches cachées des réseaux de neurones ? En principe SELU > ELU > leaky ReLU et ses variantes > ReLU > tanh > logistique. Si l'architecture du réseau empêche son autonormalisation, alors ELU peut surpasser SELU (car SELU n'est pas continue en $z = 0$). Si les performances d'exécution importent, il peut être préférable d'adopter leaky ReLU. Si vous ne souhaitez pas vous embêter avec le réglage d'un hyperparamètre supplémentaire, utilisez simplement les valeurs par défaut choisies par Keras pour α (par exemple, 0,03 pour leaky ReLU). Si vous avez du temps libre et de la puissance de calcul, vous pouvez exploiter la validation croisée pour évaluer d'autres hyperparamètres et fonctions d'activation, en particulier RReLU en cas de surajustement du réseau et PReLU en cas de jeu d'entraînement immense. Cela dit, puisque ReLU est, de loin, la fonction d'activation la plus employée, de nombreux accélérateurs matériels et bibliothèques disposent d'optimisations spécifiques à cette fonction. Par conséquent, si la rapidité est votre priorité, ReLU peut parfaitement rester le meilleur choix.

Pour choisir la fonction d'activation leaky ReLU, créez une couche LeakyReLU et ajoutez-la à votre modèle juste après la couche à laquelle vous souhaitez l'appliquer :

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

Pour PReLU, remplacez `LeakyRelu(alpha=0.2)` par `PReLU()`. Il n'existe actuellement aucune mise en œuvre officielle de RReLU dans Keras, mais vous pouvez aisément implémenter la vôtre (pour apprendre à le faire, consultez les exercices proposés à la fin du chapitre 4).

Pour la fonction d'activation SELU, indiquez `activation="selu"` et `kernel_initializer="lecun_normal"` au moment de la création d'une couche :

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

3.1.3 Normalisation par lots

Bien que l'initialisation de He associée à ELU (ou toute variante de ReLU) puisse réduire énormément les risques de disparition et d'explosion des gradients au début de l'entraînement, elle ne garantit pas leur retour pendant l'entraînement.

Dans un article⁷² de 2015, Sergey Ioffe et Christian Szegedy ont proposé une technique appelée *normalisation par lots* (BN, *Batch Normalization*) pour traiter ces problèmes. Leur technique consiste à ajouter une opération dans le modèle, juste avant ou après la fonction d'activation de chaque couche cachée. Elle se contente de centrer sur zéro et de normaliser chaque entrée, puis de mettre à l'échelle et de décaler le résultat en utilisant deux nouveaux paramètres par couche : l'un pour la mise à l'échelle, l'autre pour le décalage. Autrement dit, cette opération permet au modèle d'apprendre l'échelle et la moyenne optimales des données. En général, si nous placions une couche BN en tant que toute première couche du réseau de neurones, il n'est plus utile de standardiser le jeu d'entraînement (par exemple, en utilisant `StandardScaler`), car la couche BN s'en charge (mais de façon approximative, car elle examine uniquement un lot à la fois et peut également redimensionner et décaler chaque caractéristique d'entrée).

Pour pouvoir centrer sur zéro et normaliser les entrées, l'algorithme a besoin d'évaluer la moyenne et l'écart-type de chaque entrée. Il procède en déterminant ces valeurs sur le mini-lot courant (d'où le nom de normalisation par lots). L'intégralité de l'opération est résumée dans l'équation 3.3.

72. Sergey Ioffe et Christian Szegedy, « Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift », *Proceedings of the 32nd International Conference on Machine Learning* (2015), 448-456 : <https://hml.info/51>.

Équation 3.3 – Algorithme de normalisation par lots

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

Dans cet algorithme :

- μ_B est le vecteur des moyennes des entrées, évaluées sur l'intégralité du mini-lot B (il contient une moyenne par entrée).
- σ_B^2 est le vecteur des écarts-types des entrées, également évalués sur l'intégralité du mini-lot (il contient un écart-type par entrée).
- m_B est le nombre d'instances dans le mini-lot.
- $\hat{\mathbf{x}}^{(i)}$ est le vecteur des entrées centrées sur zéro et normalisées pour l'instance i .
- γ est le vecteur de paramètres de mise à l'échelle de sortie pour la couche (il contient un paramètre d'échelle par entrée).
- \otimes représente la multiplication élément par élément (chaque entrée est multipliée par son paramètre de mise à l'échelle de sortie correspondant).
- β est le vecteur de paramètres de décalage de sortie pour la couche (il contient un paramètre de décalage par entrée). Chaque entrée est décalée de son paramètre de décalage correspondant.
- ϵ est un nombre minuscule pour éviter toute division par zéro (en général 10^{-5}). Il s'agit d'un *terme de lissage*.
- $\mathbf{z}^{(i)}$ est la sortie de l'opération BN : la version mise à l'échelle et décalée des entrées.

Pendant l'entraînement, BN standardise ses entrées, puis les redimensionne et les décale. Très bien, mais qu'en est-il pendant les tests ? Ce n'est pas aussi simple. Nous devons effectuer des prédictions non pas pour des lots d'instances mais pour des instances individuelles : nous n'avons alors aucun moyen de calculer la moyenne et l'écart-type de chaque entrée. Par ailleurs, même si nous avons un lot d'instances, il peut être trop petit ou les instances peuvent ne pas être indépendantes et distribuées de façon identique. Le calcul statistique sur le lot d'instances n'est donc pas fiable. Une solution pourrait être d'attendre la fin de l'entraînement, puis de passer l'intégralité du jeu d'entraînement au travers du réseau de neurones et calculer la moyenne et l'écart-type pour chaque entrée de la couche BN.

Après l'entraînement, afin de réaliser des prédictions, on pourrait utiliser ces moyennes et écart-types « finaux » des entrées, au lieu des moyennes et écarts-types

des entrées calculés individuellement pour chaque lot (comme on le fait lors de l'entraînement). Toutefois, la plupart des implémentations de la normalisation par lots réalisent une estimation de ces statistiques finales au cours de l'entraînement en utilisant une moyenne glissante des moyennes et des écarts-types d'entrée de la couche. C'est ce que fait automatiquement Keras lorsque nous utilisons la couche `BatchNormalization`. En bref, quatre vecteurs de paramètres sont appris dans chaque couche normalisée par lots : γ (le vecteur de mise à l'échelle de sortie) et β (le vecteur de décalages de sortie) sont déterminés au travers d'une rétropropagation classique, et μ (le vecteur final des moyennes des entrées) et σ (le vecteur final des écarts-types des entrées) sont déterminés à l'aide d'une moyenne glissante exponentielle. Notez que μ et σ sont estimés pendant l'entraînement, mais qu'ils sont utilisés uniquement après (pour remplacer les moyennes et écarts-types des entrées du lot dans l'équation 3.3).

Ioffe et Szegedy ont démontré que la normalisation par lots améliore considérablement tous les réseaux de neurones profonds qu'ils ont testés, conduisant à une énorme amélioration de la tâche de classification ImageNet (ImageNet est une volumineuse base de données d'images classifiées en de nombreuses classes et couramment employée pour évaluer les systèmes de vision par ordinateur). Le problème de disparition des gradients est fortement réduit, à tel point qu'ils ont pu utiliser avec succès des fonctions d'activation saturantes, comme la tangente hyperbolique et la fonction logistique. Les réseaux étaient également beaucoup moins sensibles à la méthode d'initialisation des poids.

Ils ont également été en mesure d'utiliser des taux d'apprentissage beaucoup plus élevés, accélérant ainsi le processus d'apprentissage. Ils ont notamment remarqué que, « appliquée à un modèle de pointe pour la classification des images, la normalisation par lots permet d'arriver à la même précision, mais avec quatorze fois moins d'étapes d'entraînement, et bat le modèle d'origine d'une bonne longueur. [...] En utilisant un ensemble de réseaux normalisés par lots, nous améliorons le meilleur résultat publié sur la classification ImageNet : en atteignant un taux d'erreur de validation top-5 de 4,9 % (et un taux d'erreur de test de 4,8 %), dépassant la précision des évaluateurs humains ». Enfin, cerise sur le gâteau, la normalisation par lots agit également comme un régulariseur, diminuant le besoin de recourir à d'autres techniques de régularisation (comme celle du *dropout* décrite plus loin).

La normalisation par lots ajoute néanmoins une certaine complexité au modèle (même si elle permet d'éviter la normalisation des données d'entrée, comme nous l'avons expliqué précédemment). Par ailleurs, elle implique également un coût à l'exécution : le réseau de neurones fait ses prédictions plus lentement en raison des calculs supplémentaires réalisés dans chaque couche. Heureusement, il est souvent possible de fusionner la couche BN à la couche précédente, après l'entraînement, évitant ainsi le coût à l'exécution. Pour cela, il suffit d'actualiser les poids et les termes constants de la couche précédente afin qu'elle produise directement des sorties ayant l'échelle et le décalage appropriés. Par exemple, si la couche précédente calcule $XW + b$, alors la couche BN calculera $\gamma \otimes (XW + b - \mu) / \sigma + \beta$ (en ignorant le terme de lissage ϵ dans le dénominateur). Si nous définissons $W' = \gamma \otimes W / \sigma$

et $\mathbf{b}' = \gamma \otimes (\mathbf{b} - \boldsymbol{\mu}) / \sigma + \boldsymbol{\beta}$, l'équation se simplifie en $\mathbf{XW}' + \mathbf{b}'$. Par conséquent, si nous remplaçons les poids et les termes constants de la couche précédente (\mathbf{W} et \mathbf{b}) par les poids et les termes constants actualisés (\mathbf{W}' et \mathbf{b}'), nous pouvons nous débarrasser de la couche BN (l'optimiseur de TFLite le fait automatiquement ; voir le chapitre 11).



Il est possible que l'entraînement soit relativement lent car, lorsque la normalisation par lots est mise en œuvre, chaque époque prend plus de temps. Ce comportement est généralement contrebalancé par le fait que la convergence est beaucoup plus rapide avec la normalisation par lots et qu'il faut moins d'époques pour arriver aux mêmes performances. Globalement, le temps écoulé (le temps mesuré par l'horloge accrochée au mur) sera en général plus court.

Implémenter la normalisation par lots avec Keras

Comme pour la plupart des opérations avec Keras, la mise en œuvre de la normalisation par lots est simple et intuitive. Il suffit d'ajouter une couche `BatchNormalization` avant ou après la fonction d'activation de chaque couche cachée et d'ajouter, éventuellement, une couche BN en première couche du modèle. Par exemple, le modèle suivant applique la normalisation par lots après chaque couche cachée et comme première couche du modèle (après avoir aplati les images d'entrée) :

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Et voilà ! Dans ce minuscule exemple comprenant uniquement deux couches cachées, il est peu probable que la normalisation par lots ait un impact notable. Mais, pour des réseaux plus profonds, elle fait une énorme différence.

Affichons le résumé du modèle :

```
>>> model.summary()
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
batch_normalization_v2 (BatchNormalization)	(None, 784)	3136
dense_50 (Dense)	(None, 300)	235500
batch_normalization_v2_1 (BatchNormalization)	(None, 300)	1200
dense_51 (Dense)	(None, 100)	30100

```

batch_normalization_v2_2 (BatchNormalisation)        400
dense_52 (Dense)                                 (None, 10)      1010
=====
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368

```

Chaque couche BN ajoute quatre paramètres par entrée : γ , β , μ et σ (par exemple, la première couche BN ajoute 3 136 paramètres, c'est-à-dire 4×784). Les deux derniers paramètres, μ et σ , sont les moyennes glissantes ; puisqu'ils ne sont pas affectés par la rétropropagation, Keras les qualifie de « non entraînables »⁷³ (si nous comptons le nombre total de paramètres, $3\,136 + 1\,200 + 400$, et divisons le résultat par 2, nous obtenons 2 368, c'est-à-dire le nombre total de paramètres non entraînables dans ce modèle).

Examinons les paramètres de la première couche BN. Deux sont entraînables (par la rétropropagation) et deux ne le sont pas :

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Lorsque nous créons une couche BN dans Keras, nous créons également deux opérations qui seront effectuées par Keras au cours de l'entraînement à chaque itération. Elles vont actualiser les moyennes glissantes. Puisque nous utilisons le backend TensorFlow, il s'agit d'opérations TensorFlow (nous étudierons les opérations TensorFlow au chapitre 4) :

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

Les auteurs de l'article sur la normalisation par lots préconisent l'ajout des couches BN avant les fonctions d'activation plutôt qu'après (comme nous venons de le faire). Ce choix est sujet à débat, car la meilleure option semble dépendre de la tâche : vous pouvez faire vos propres tests pour voir celle qui convient à votre jeu de données. Pour ajouter les couches BN avant les fonctions d'activation, nous devons retirer celles-ci des couches cachées et les ajouter en tant que couches séparées après les couches BN. Par ailleurs, puisqu'une couche de normalisation par lots comprend un paramètre de décalage par entrée, nous pouvons retirer le terme constant de la couche précédente (lors de sa création, il suffit d'indiquer `use_bias=False`) :

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
```

73. Cependant, puisqu'ils sont déterminés au cours de l'entraînement en fonction des données d'entraînement, ils sont entraînables. Dans Keras, « non entraînable » signifie en réalité « non touché par la propagation ».

```

    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
)
)

```

La classe `BatchNormalization` possède quelques hyperparamètres que nous pouvons régler. Les valeurs par défaut sont généralement appropriées, mais nous devrons parfois ajuster `momentum`. La couche `BatchNormalization` utilise cet hyperparamètre lorsqu'elle actualise les moyennes glissantes exponentielles; étant donné une nouvelle valeur v (c'est-à-dire un nouveau vecteur de moyennes ou d'écart-types d'entrée calculés sur le lot courant), la couche actualise la moyenne glissante \hat{v} en utilisant l'équation suivante :

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Une bonne valeur de `momentum` est en général proche de 1; par exemple, 0,9, 0,99 ou 0,999 (augmenter le nombre de 9 pour les jeux de données plus volumineux et les mini-lots plus petits).

`axis` est un autre hyperparamètre important: il détermine l'acte qui doit être normalisé. Sa valeur par défaut est -1 , ce qui signifie que la normalisation se fera sur le dernier axe (en utilisant les moyennes et les écart-types calculés sur les *autres* axes). Lorsque le lot d'entrées est en deux dimensions (autrement dit, quand la forme du lot est `[taille du lot, caractéristiques]`), chaque caractéristique d'entrée sera normalisée en fonction de la moyenne et de l'écart-type calculés sur toutes les instances du lot.

Par exemple, la première couche BN dans le code précédent normalisera indépendamment (et redimensionnera et décalera) chacune des 784 caractéristiques d'entrée. Si nous déplaçons la première couche BN avant la couche `Flatten`, les lots d'entrées seront en trois dimensions et de forme `[taille du lot, hauteur, largeur]`; par conséquent, la couche BN calculera 28 moyennes et 28 écart-types (1 par colonne de pixels, calculé sur toutes les instances du lot et sur toutes les lignes de la colonne), et normalisera tous les pixels d'une colonne donnée en utilisant les mêmes moyenne et écart-type. Il y aura aussi 28 paramètres d'échelle et 28 paramètres de décalage. Si, à la place, nous voulons traiter indépendamment chacun des 784 pixels, nous devons préciser `axis=[1, 2]`.

La couche BN n'effectue pas le même calcul pendant l'entraînement et après : elle utilise des statistiques de lots au cours de l'entraînement et des statistiques « finales » après l'entraînement (autrement dit, les valeurs finales des moyennes glissantes). Examinons un petit bout du code source de cette classe pour comprendre comment cette opération est réalisée :

```

class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]

```

C'est dans la méthode `call()` que se font les calculs. Elle comprend un argument supplémentaire, `training`, dont la valeur par défaut est `None`, mais la méthode `fit()` lui donne la valeur `1` pendant l'entraînement. Si vous devez écrire une couche personnalisée dont le comportement doit être différent pendant l'entraînement et pendant les tests, ajoutez un argument `training` à la méthode `call()` et utilisez-le pour déterminer les calculs à effectuer⁷⁴ (nous verrons les couches personnalisées au chapitre 4).

La couche `BatchNormalization` est devenue l'une des plus utilisées dans les réseaux de neurones profonds, à tel point qu'elle est souvent omise dans les schémas, car on suppose que la normalisation par lots est ajoutée après chaque couche. Cependant, un article⁷⁵ récent publié par Hongyi Zhang *et al.* pourrait remettre en question cette hypothèse: en utilisant une nouvelle technique d'initialisation des poids `fixed-update` (*Fixup*), les auteurs ont réussi à entraîner un réseau de neurones très profond (10000 couches !) dépourvu de normalisation par lots, tout en obtenant d'excellentes performances sur des tâches complexes de classification d'images. Toutefois, puisqu'il s'agit de résultats récents, il peut être préférable d'attendre des travaux complémentaires qui confirment cette découverte avant d'écartier la normalisation par lots.

3.1.4 Écrêtage de gradient

Pour minimiser le problème de l'explosion des gradients, une autre technique répandue consiste à simplement les écrêter pendant la rétropropagation afin qu'ils ne dépassent jamais un certain seuil. On l'appelle *écrêtage de gradient*⁷⁶ (*gradient clipping*). Elle est le plus souvent utilisée dans les réseaux de neurones récurrents, où l'emploi de la normalisation par lots est difficile (voir au chapitre 7). Pour les autres types de réseaux, la normalisation par lots suffit généralement.

Dans Keras, la mise en place de l'écrêtage de gradient se limite à préciser les arguments `clipvalue` ou `clipnorm` au moment de la création d'un optimiseur:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Cet optimiseur va écrêter chaque composant du vecteur de gradient à une valeur entre $-1,0$ et $1,0$. Cela signifie que toutes les dérivées partielles de la perte (en ce qui concerne chaque paramètre entraînable) seront écrêtées entre $-1,0$ et $1,0$. Le seuil est un hyperparamètre que nous pouvons ajuster. Notez qu'il peut modifier l'orientation du vecteur de gradient.

Par exemple, si le vecteur de gradient d'origine est $[0,9; 100,0]$, il est essentiellement orienté dans le sens du second axe. Mais, après l'avoir écrêté par valeur, nous obtenons $[0,9; 1,0]$, qui correspond approximativement à la diagonale entre

74. L'API de Keras définit également une fonction `keras.backend.learning_phase()`, qui doit retourner `1` pendant l'entraînement et `0` en dehors de l'entraînement.

75. Hongyi Zhang *et al.*, « Fixup Initialization: Residual Learning Without Normalization »(2019): <https://hml.info/fixedup>.

76. Razvan Pascanu *et al.*, « On the Difficulty of Training Recurrent Neural Networks », *Proceedings of the 30th International Conference on Machine Learning* (2013), 1310-1318: <https://hml.info/52>.

les deux axes. Dans la pratique, cette approche fonctionne bien. Si nous voulons être certains que l'écrêtage de gradient ne modifie pas la direction du vecteur de gradient, il doit se faire par norme en indiquant `clipnorm` à la place de `clipvalue`. Dans ce cas, l'écrêtage du gradient se fera si sa norme ℓ_2 est supérieure au seuil fixé. Par exemple, si nous indiquons `clipnorm=1.0`, le vecteur $[0,9; 100,0]$ sera alors écrêté à $[0,00899964; 0,9999595]$, conservant son orientation mais éliminant quasi-maintenant le premier élément.

Si nous observons une explosion des gradients au cours de l'entraînement (TensorBoard permet de suivre la taille des gradients), nous pouvons essayer un écrêtage par valeur et un écrêtage par norme, avec des seuils différents, et déterminer la meilleure option pour le jeu de validation.

3.2 RÉUTILISER DES COUCHES PRÉENTRAÎNÉES

En général, il est déconseillé d'entraîner un très grand RNP à partir de zéro. Il est préférable de toujours essayer de trouver un réseau de neurones existant qui accomplit une tâche comparable à celle visée (au chapitre 6, nous verrons comment les trouver), puis d'en réutiliser les couches les plus basses. Ce principe de *transfert d'apprentissage* (*transfer learning*) va non seulement accélérer considérablement l'entraînement, mais il permettra d'obtenir de bonnes performances avec des jeux de données d'entraînement assez petits.

Par exemple, supposons que nous ayons accès à un RNP qui a été entraîné pour classer des images en 100 catégories différentes (animaux, plantes, véhicules et tous les objets du quotidien). Nous souhaitons à présent entraîner un RNP pour classer des types de véhicules particuliers. Ces deux tâches sont très similaires et nous devons essayer de réutiliser des éléments du premier réseau (voir la figure 3.4).

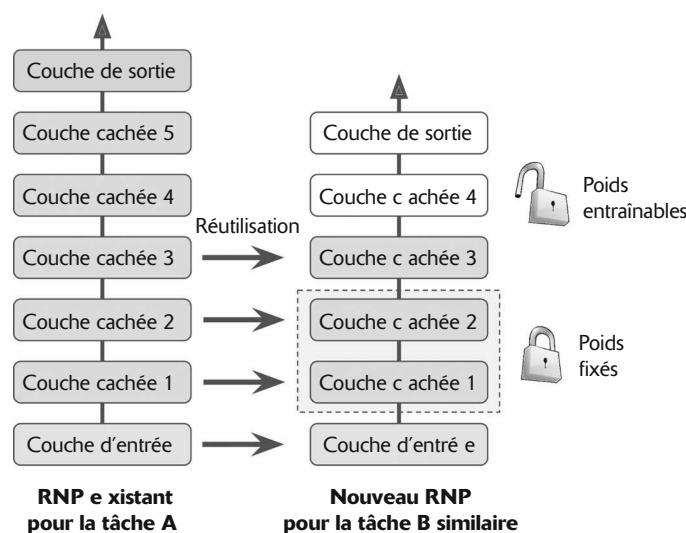


Figure 3.4 – Réutilisation de couches préentraînées



Si les images d'entrée de la nouvelle tâche n'ont pas la même taille que celles utilisées dans la tâche d'origine, il faudra rajouter une étape de pré-traitement pour les redimensionner à la taille attendue par le modèle d'origine. De façon plus générale, le transfert d'apprentissage ne peut fonctionner que si les entrées ont des caractéristiques de bas niveau comparables.

La couche de sortie du modèle d'origine doit généralement être remplacée, car il est fort probable qu'elle ne soit pas utile à la nouvelle tâche et il est même possible qu'elle ne dispose pas du nombre de sorties approprié.

De manière comparable, les couches cachées supérieures du modèle d'origine risquent d'être moins utiles que les couches inférieures, car les caractéristiques de haut niveau intéressantes pour la nouvelle tâche peuvent être très différentes de celles utiles à la tâche d'origine. Il faudra déterminer le bon nombre de couches à réutiliser.



Plus les tâches sont similaires, plus le nombre de couches réutilisables pourra être élevé (en commençant par les couches inférieures). Pour des tâches très proches, essayez de conserver toutes les couches cachées et remplacez uniquement la couche de sortie.

Nous commençons par figer toutes les couches réutilisées (autrement dit, leurs poids sont rendus non entraînables afin que la descente de gradient ne les modifie pas), puis nous entraînons le modèle et examinons ses performances. Nous essayons ensuite de libérer une ou deux des couches cachées supérieures pour que la rétro-propagation les ajuste et voyons si les performances s'améliorent. Plus la quantité de données d'entraînement est importante, plus le nombre de couches que nous pouvons libérer augmente. Il est également utile de diminuer le taux d'apprentissage lorsque des couches réutilisées sont libérées : cela évite de détruire leurs poids qui ont été ajustés finement.

Si les performances sont toujours mauvaises et si les données d'entraînement sont limitées, nous pouvons retirer la (voire les) couche(s) supérieure(s) et figer de nouveau les couches cachées restantes. Nous répétons la procédure jusqu'à trouver le bon nombre de couches à réutiliser. Si les données d'entraînement sont nombreuses, nous pouvons remplacer les couches cachées supérieures au lieu de les supprimer, et même ajouter d'autres couches cachées.

3.2.1 Transfert d'apprentissage avec Keras

Prenons un exemple. Supposons que le jeu de données Fashion MNIST ne contienne que huit classes : par exemple, toutes les classes à l'exception des sandales et des chemises. Quelqu'un a déjà construit et entraîné un modèle Keras sur ce jeu et a obtenu des performances raisonnables (précision supérieure à 90 %). Appelons ce modèle A. Nous voulons à présent travailler sur une nouvelle tâche : nous avons des images de sandales et de chemises, et nous voulons entraîner un classificateur binaire (positif = chemise, négatif = sandale). Nous disposons d'un jeu de données relativement petit, avec seulement 200 images étiquetées. Lorsque nous entraînons un nouveau

modèle pour cette tâche (appelons-le modèle B) avec la même architecture que le modèle A, nous constatons des performances plutôt bonnes (précision de 97,2 %). Toutefois, puisque la tâche est plus facile (il n'y a que deux classes), nous espérons mieux. Pendant que nous prenons notre café du matin, nous réalisons que la tâche B est assez comparable à la tâche A. Peut-être que le transfert d'apprentissage pourrait nous aider. Voyons cela !

Tout d'abord, nous devons charger un modèle A et créer un nouveau modèle à partir des couches du premier. Réutilisons toutes les couches à l'exception de la couche de sortie :

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

model_A et model_B_on_A partagent à présent certaines couches. L'entraînement de model_B_on_A va également affecter model_A. Si nous préférons l'éviter, nous devons cloner model_A avant de réutiliser ses couches. Pour cela, l'opération consiste à cloner l'architecture du modèle A avec clone_model(), puis à copier ses poids (car clone_model() ne clone pas les poids) :

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Nous pouvons à présent entraîner model_B_on_A pour la tâche B, mais, puisque la nouvelle couche de sortie a été initialisée de façon aléatoire, elle va effectuer des erreurs importantes, tout au moins au cours des quelques premières époques. Nous aurons par conséquent de grands gradients d'erreur qui risquent de saboter les poids réutilisés. Pour éviter cela, une solution consiste à geler les couches réutilisées au cours des quelques premières époques, en donnant à la nouvelle couche du temps pour apprendre des poids raisonnables. Nous fixons donc l'attribut trainable de chaque couche à False et compilons le modèle :

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])
```



Après avoir figé ou libéré des couches, le modèle doit toujours être compilé.

Nous pouvons à présent entraîner le modèle sur quelques époques, puis libérer les couches réutilisées (ce qui implique une nouvelle compilation du modèle) et poursuivre l'entraînement afin d'ajuster précisément les couches réutilisées à la tâche B. Après la libération des couches réutilisées, il est généralement conseillé d'abaisser le taux d'apprentissage, de nouveau pour éviter d'endommager les poids réutilisés :

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                            validation_data=(X_valid_B, y_valid_B))
```

```

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # Le taux d'apprentissage par défaut
                                              # est 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                            validation_data=(X_valid_B, y_valid_B))

```

Quel est le résultat final ? Sur ce modèle, la précision de test est de 99,25 %. Le transfert d'apprentissage a donc fait baisser le taux d'erreur de 2,8 % à presque 0,7 % ! Cela représente un facteur de quatre !

```

>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]

```

Êtes-vous convaincu ? Vous ne devriez pas, car nous avons triché ! Nous avons testé de nombreuses configurations jusqu'à trouver celle qui conduise à une forte amélioration. Si vous changez les classes ou le germe aléatoire, vous constaterez généralement que l'amélioration baisse, voire disparaît ou s'inverse. En réalité, nous avons « torturé les données jusqu'à ce qu'elles avouent ». Lorsqu'un article semble trop positif, vous devez devenir soupçonneux : il est possible que la nouvelle super technique apporte en réalité peu d'aide (elle peut même dégrader les performances), mais les auteurs ont essayé de nombreuses variantes et rapporté uniquement les meilleurs résultats (peut-être dus à une chance inouïe), sans mentionner le nombre des échecs. La plupart du temps, c'est sans intention malveillante, mais cela explique en partie pourquoi de nombreux résultats scientifiques ne peuvent jamais être reproduits.

Pourquoi avons-nous triché ? En réalité, le transfert d'apprentissage ne fonctionne pas très bien avec les petits réseaux denses, probablement parce que les petits réseaux apprennent peu de motifs et que les réseaux denses apprennent des motifs très spécifiques sans doute peu utiles dans d'autres tâches. Le transfert d'apprentissage est mieux adapté aux réseaux de neurones convolutifs profonds, qui ont tendance à apprendre des détecteurs de caractéristiques beaucoup plus généraux (en particulier dans les couches basses). Nous reviendrons sur le transfert d'apprentissage au chapitre 6, en utilisant les techniques que nous venons de présenter (cette fois-ci sans tricher).

3.2.2 Préentraînement non supervisé

Supposons que nous souhaitions nous attaquer à une tâche complexe pour laquelle nous n'avons que peu de données d'entraînement étiquetées et que nous ne trouvions aucun modèle déjà entraîné pour une tâche comparable. Tout espoir n'est pas perdu ! Nous devons évidemment commencer par essayer de récolter d'autres données d'entraînement étiquetées, mais, si cela est trop difficile, nous sommes peut-être en mesure d'effectuer un *préentraînement non supervisé* (voir la figure 3.5). Il est souvent peu coûteux de réunir des exemples d'entraînement non étiquetés, mais assez onéreux de les étiqueter. Si nous disposons d'un grand nombre de données d'entraînement non étiquetées, il s'agit d'essayer d'entraîner un modèle non supervisé, comme un autoencodeur ou un réseau antagoniste génératif (GAN, *generative adversarial network* ; voir le chapitre 9). Nous pouvons ensuite réutiliser les couches inférieures de l'autoencodeur

ou celles du discriminateur du GAN, ajouter la couche de sortie qui correspond à notre tâche et ajuster plus précisément le réseau en utilisant un apprentissage supervisé (c'est-à-dire avec les exemples d'entraînement étiquetés).

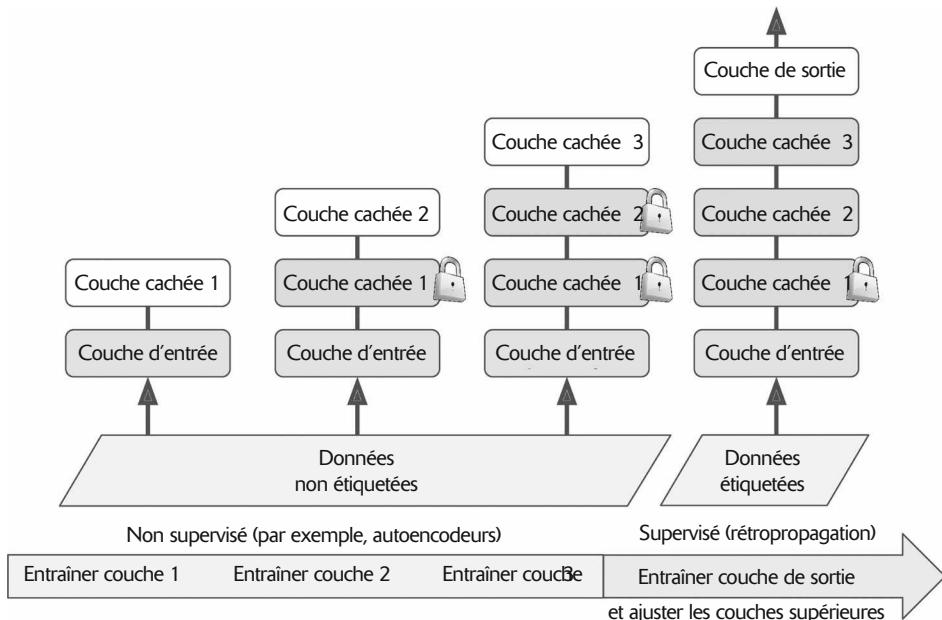


Figure 3.5 – Dans un entraînement non supervisé, un modèle est entraîné sur les données non étiquetées (ou sur toutes les données) en utilisant une technique d'apprentissage non supervisé, puis il est ajusté plus finement à la tâche finale sur les données étiquetées en utilisant une technique d'apprentissage supervisé; la partie non supervisée peut entraîner une couche à la fois, comme illustré ici, ou directement l'intégralité du modèle.

Il s'agit de la technique que Geoffrey Hinton et son équipe ont employée avec succès en 2006 et qui a conduit à la renaissance des réseaux de neurones et au succès du Deep Learning. Jusqu'en 2010, le préentraînement non supervisé, en général avec des machines de Boltzmann restreintes (RBM, *Restricted Boltzmann Machine*, voir l'annexe C), constituait la norme pour les réseaux profonds. Ce n'est qu'après avoir réduit le problème de disparition des gradients que l'entraînement uniquement supervisé des RNP est devenu plus fréquent. Le préentraînement non supervisé (aujourd'hui plutôt avec des autoencodeurs ou des GAN qu'avec des RBM) reste une bonne approche lorsque la tâche à résoudre est complexe, qu'il n'existe aucun modèle comparable à réutiliser, et que les données d'entraînement étiquetées sont peu nombreuses au contraire des données d'entraînement non étiquetées.

Aux premiers jours du Deep Learning, il était difficile d'entraîner des modèles profonds. On employait donc une technique de *préentraînement glouton par couche* (*greedy layer-wise pretraining*), illustrée à la figure 3.5. Un premier modèle non supervisé était entraîné avec une seule couche, en général une machine de Boltzmann

restreinte. Cette couche était ensuite figée et une autre couche était ajoutée par-dessus. Le modèle était de nouveau entraîné (seule la nouvelle couche était donc concernée), puis la nouvelle couche était figée et une autre couche était ajoutée par-dessus. Le modèle était à nouveau entraîné, et ainsi de suite. Aujourd’hui, les choses sont beaucoup plus simples : en général, l’intégralité du modèle non supervisé est entraînée en une fois (autrement dit, on commence directement à l’étape trois de la figure 3.5) et des autoencodeurs ou des GAN sont employés à la place des RBM.

3.2.3 Préentraînement à partir d’une tâche secondaire

Si la quantité de données d’entraînement étiquetées est réduite, une dernière option consiste à entraîner un premier réseau de neurones sur une tâche secondaire pour laquelle nous pouvons aisément obtenir ou générer des données d’entraînement étiquetées, puis à réutiliser les couches inférieures de ce réseau pour notre tâche réelle. Les couches inférieures du premier réseau de neurones effectueront l’apprentissage de détecteurs de caractéristiques que nous pourrons probablement réutiliser dans le second réseau de neurones.

Supposons, par exemple, que nous voulions construire un système de reconnaissance de visages, mais que nous n’ayons que quelques photos de chaque individu. Il est clair que cela ne suffira pas à entraîner un bon classificateur. Réunir des centaines d’images de chaque personne n’est pas envisageable. Cependant, nous pouvons trouver sur Internet un grand nombre d’images de personnes quelconques et entraîner un premier réseau de neurones pour qu’il détecte si la même personne se trouve sur deux images différentes. Un tel réseau possédera de bons détecteurs de caractéristiques faciales et nous pourrons réutiliser ses couches inférieures pour entraîner un bon classificateur de visages à partir d’un faible nombre de données d’entraînement.

Pour les applications de *traitement automatique du langage naturel* (TALN), nous pouvons télécharger un corpus de millions de documents textuels et nous en servir pour générer automatiquement des données étiquetées. Par exemple, nous pouvons masquer automatiquement certains mots et entraîner un modèle afin qu’il prédise les mots manquants (par exemple, il doit être capable de déterminer que le mot manquant dans la phrase « Que vois-__ ? » est probablement « tu » ou « je »). Si nous pouvons entraîner un modèle qui donne de bonnes performances sur cette tâche, il en connaîtra déjà beaucoup sur la langue et nous pourrons certainement le réutiliser dans notre tâche et l’ajuster pour nos données étiquetées (nous reviendrons sur les tâches de préentraînement au chapitre 7).



L’entraînement *autosupervisé* (*self-supervised learning*) consiste à générer automatiquement les étiquettes à partir des données elles-mêmes, puis à entraîner un modèle sur le jeu de données étiquetées résultant en utilisant des techniques d’apprentissage supervisé. Puisque cette approche ne nécessite pas d’intervention humaine pour l’étiquetage, il est préférable de la considérer comme une forme d’apprentissage non supervisé.

3.3 OPTIMISEURS PLUS RAPIDES

Lorsque le réseau de neurones profond est très grand, l’entraînement peut se révéler péniblement lent. Jusqu’à présent, nous avons vu quatre manières d’accélérer l’entraînement (et d’atteindre une meilleure solution) : appliquer une bonne stratégie d’initialisation des poids des connexions, utiliser une bonne fonction d’activation, utiliser la normalisation par lots, et réutiliser des parties d’un réseau préentraîné (éventuellement construit à partir d’une tâche secondaire ou en utilisant un apprentissage non supervisé). Nous pouvons également accélérer énormément l’entraînement en utilisant un optimiseur plus rapide que l’optimiseur de descente de gradient ordinaire. Dans cette section, nous allons présenter les solutions les plus répandues : optimisation avec inertie, gradient accéléré de Nesterov, AdaGrad, RMSProp et optimisation Adam et Nadam.

3.3.1 Optimisation avec inertie

Imaginons une boule de bowling qui roule sur une surface lisse en légère pente : elle démarre lentement, mais prend rapidement de la vitesse jusqu’à atteindre une vitesse maximale (s’il y a des frottements ou une résistance de l’air). Voilà l’idée simple derrière l’*optimisation avec inertie* (*Momentum Optimization*) proposée par Boris Polyak en 1964⁷⁷. À l’opposé, la descente de gradient classique ferait de petits pas réguliers vers le bas de la pente et mettrait donc plus de temps pour arriver à son extrémité.

Rappelons que la descente de gradient met simplement à jour les poids θ en soustrayant directement le gradient de la fonction de coût $J(\theta)$ par rapport aux poids (noté $\nabla_\theta J(\theta)$), multiplié par le taux d’apprentissage η . L’équation est $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$. Elle ne tient pas compte des gradients antérieurs. Si le gradient local est minuscule, elle avance doucement.

L’optimisation avec inertie s’intéresse énormément aux gradients antérieurs : à chaque itération, elle soustrait le gradient local au *vecteur d’inertie* m (multiplié par le taux d’apprentissage η) et actualise les poids θ en additionnant simplement ce vecteur d’inertie (voir l’équation 3.4). Autrement dit, le gradient est utilisé non pas comme un facteur de vitesse mais d’accélération. Pour simuler une forme de frottement et éviter que la vitesse ne s’emballe, l’algorithme introduit un nouvel hyperparamètre β , appelé simplement *inertie* (*momentum* en anglais), dont la valeur doit être comprise entre 0 (frottement élevé) et 1 (aucun frottement). Une valeur fréquemment utilisée est 0,9.

Équation 3.4 – Algorithme de l’inertie

1. $m \leftarrow \beta m - \eta \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta + m$

77. Boris T. Polyak, «Some Methods of Speeding Up the Convergence of Iteration Methods», USSR Computational Mathematics and Mathematical Physics, 4, n° 5 (1964), 1-17 : <https://homl.info/54>.

Vous pouvez facilement vérifier que si le gradient reste constant, la vitesse finale (c'est-à-dire la taille maximale des mises à jour des poids) est égale à ce gradient multiplié par le taux d'apprentissage η multiplié par $1/(1 - \beta)$ (sans tenir compte du signe). Par exemple, si $\beta = 0,9$, alors la vitesse finale est égale à 10 fois le gradient fois le taux d'apprentissage. L'optimisation avec inertie permet ainsi d'aller jusqu'à dix fois plus rapidement que la descente de gradient ! Elle peut donc sortir des zones de faux plat plus rapidement que la descente de gradient. En particulier, lorsque les entrées ont des échelles très différentes, la fonction de coût va ressembler à un bol allongé⁷⁸: la descente de gradient arrive en bas de la pente abrupte assez rapidement, mais il lui faut ensuite beaucoup de temps pour descendre la vallée. En revanche, l'optimisation avec inertie va avancer de plus en plus rapidement vers le bas de la vallée, jusqu'à l'atteindre (l'optimum). Dans les réseaux de neurones profonds qui ne mettent pas en œuvre la normalisation par lots, les couches supérieures finissent souvent par recevoir des entrées aux échelles très différentes. Dans ce cas, l'optimisation avec inertie est d'une aide précieuse. Elle permet également de sortir des optima locaux.



En raison de l'inertie, l'optimiseur peut parfois aller un peu trop loin, puis revenir, puis aller de nouveau trop loin, en oscillant ainsi à plusieurs reprises, avant de se stabiliser sur le minimum. Voilà notamment pourquoi il est bon d'avoir un peu de frottement dans le système: il évite ces oscillations et accélère ainsi la convergence.

Implémenter l'optimisation avec inertie dans Keras ne pose aucune difficulté: il suffit d'utiliser l'optimiseur SGD et de fixer son hyperparamètre momentum, puis d'attendre tranquillement !

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

L'optimisation avec inertie a pour inconvénient d'ajouter encore un autre hyperparamètre à ajuster. Cependant, la valeur 0,9 fonctionne plutôt bien dans la pratique et permet presque toujours d'aller plus vite que la descente de gradient.

3.3.2 Gradient accéléré de Nesterov

En 1983, Yurii Nesterov a proposé une petite variante de l'optimisation avec inertie⁷⁹, qui se révèle toujours plus rapide que la version d'origine. La méthode du *gradient accéléré de Nesterov* (NAG, Nesterov Accelerated Gradient), également appelée *optimisation avec inertie de Nesterov*, mesure le gradient de la fonction de coût non pas à la position locale θ mais légèrement en avant dans le sens de l'inertie $\theta + \beta m$ (voir l'équation 3.5).

Équation 3.5 – Algorithme de l'inertie de Nesterov

1. $m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m)$
2. $\theta \leftarrow \theta + m$

78. Voir la figure 1.10.

79. Yurii Nesterov, «A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$ », *Doklady AN USSR*, 269 (1983), 543-547: <https://homl.info/55>.

Ce petit ajustement fonctionne car, en général, le vecteur d'inertie pointe dans la bonne direction (c'est-à-dire vers l'optimum). Il sera donc légèrement plus précis d'utiliser le gradient mesuré un peu plus en avant dans cette direction que d'utiliser celui en position d'origine, comme le montre la figure 3.6 (où ∇_1 représente le gradient de la fonction de coût mesuré au point de départ θ , et ∇_2 le gradient au point $\theta + \beta m$).

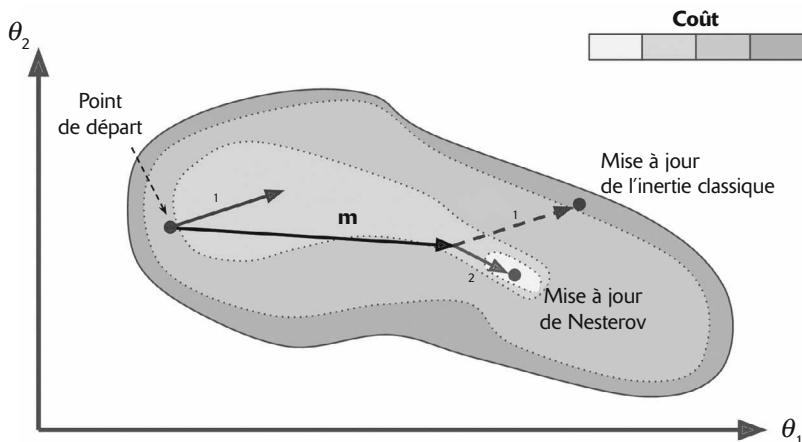


Figure 3.6 – Optimisations avec inertie classique et de Nesterov: la première applique les gradients calculés avant l'étape d'inertie, tandis que la seconde applique les gradients calculés après

Vous le constatez, la mise à jour de Nesterov arrive un peu plus près de l'optimum. Après un certain temps, ces petites améliorations se cumulent et NAG finit par être beaucoup plus rapide que l'optimisation avec inertie ordinaire. Par ailleurs, lorsque l'inertie pousse les poids au travers d'une vallée, ∇_1 continue à pousser au travers de la vallée, tandis que ∇_2 pousse en arrière vers le bas de la vallée. Cela permet de réduire les oscillations et donc de converger plus rapidement.

NAG est généralement plus rapide que l'optimisation avec inertie classique. Pour l'utiliser, il suffit de préciser `nesterov=True` lors de la création de l'optimiseur SGD:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

3.3.3 AdaGrad

Considérons à nouveau le problème du bol allongé: la descente de gradient commence par aller rapidement vers le bas de la pente la plus abrupte, qui ne pointe pas directement vers l'optimum global, puis elle va lentement vers le bas de la vallée. Il serait préférable que l'algorithme revoie son orientation plus tôt pour se diriger un peu plus vers l'optimum global. L'algorithme *AdaGrad*⁸⁰ met cette correction

80. John Duchi *et al.*, « Adaptive Subgradient Methods for Online Learning and Stochastic Optimization », *Journal of Machine Learning Research*, 12 (2011), 2121-2159 : <https://hml.info/56>.

en place en diminuant le vecteur de gradient le long des dimensions les plus raides (voir l'équation 3.6) :

Équation 3.6 – Algorithme AdaGrad

$$1. \quad s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon}$$

La première étape accumule les carrés des gradients dans le vecteur s (rappelons que le symbole \otimes représente la multiplication terme à terme). Cette forme vectorisée équivaut au calcul de $s_i \leftarrow s_i + (\partial J(\theta)/\partial \theta_i)^2$ pour chaque élément s_i du vecteur s . Autrement dit, chaque s_i accumule les carrés de la dérivée partielle de la fonction de coût en rapport avec le paramètre θ_i . Si la fonction de coût présente une pente abrupte le long de la $i^{\text{ème}}$ dimension, alors s_i va augmenter à chaque itération.

La seconde étape est quasi identique à la descente de gradient, mais avec une différence importante : le vecteur de gradient est diminué du facteur $\sqrt{s + \varepsilon}$. Le symbole \oslash représente la division terme à terme et ε est un terme de lissage qui permet d'éviter la division par zéro (sa valeur est généralement fixée à 10^{-10}). Cette forme vectorisée équivaut au calcul simultané de $\theta_i \leftarrow \theta_i - \eta \partial J(\theta)/\partial \theta_i / \sqrt{s_i + \varepsilon}$ pour tous les paramètres θ_i .

En résumé, cet algorithme abaisse progressivement le taux d'apprentissage, mais il le fait plus rapidement sur les dimensions présentant une pente abrupte que sur celles dont la pente est plus douce. Nous avons donc un *taux d'apprentissage adaptatif*. Cela permet de diriger plus directement les mises à jour résultantes vers l'optimum global (voir la figure 3.7). Par ailleurs, l'algorithme exige un ajustement moindre de l'hyperparamètre η pour le taux d'apprentissage.

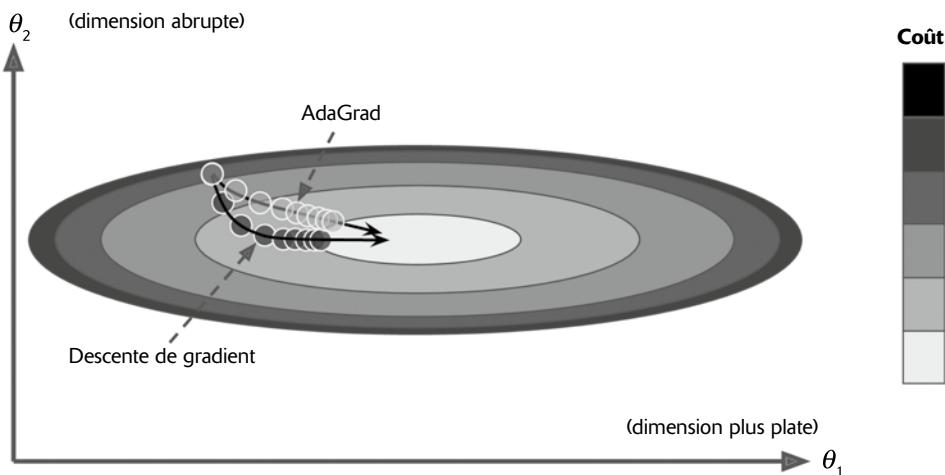


Figure 3.7 – AdaGrad contre descente de gradient : la première méthode peut corriger plus tôt sa direction pour pointer vers l'optimum

AdaGrad affiche un bon comportement pour les problèmes quadratiques simples, mais il s'arrête souvent trop tôt lors de l'entraînement des réseaux de neurones. Le taux d'apprentissage est tellement diminué que l'algorithme finit par s'arrêter totalement avant d'atteindre l'optimum global. Par conséquent, même si Keras propose l'optimiseur Adagrad, il est préférable de ne pas l'employer pour entraîner des réseaux de neurones profonds (cet optimiseur peut rester toutefois efficace pour des tâches plus simples comme la régression linéaire). Il n'en reste pas moins que comprendre AdaGrad peut aider à saisir les autres optimiseurs de taux d'apprentissage.

3.3.4 RMSProp

L'inconvénient d'AdaGrad est de ralentir un peu trop rapidement et de finir par ne jamais converger vers l'optimum global. L'algorithme RMSProp⁸¹ corrige ce problème en cumulant uniquement les gradients issus des itérations les plus récentes (plutôt que tous les gradients depuis le début l'entraînement). Pour cela, il utilise une moyenne mobile exponentielle au cours de la première étape (voir l'équation 3.7).

Équation 3.7 – Algorithme RMSProp

1. $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

Le taux de décroissance β est en général fixé à 0,9. Il s'agit encore d'un nouvel hyperparamètre, mais cette valeur par défaut convient souvent et nous avons rarement besoin de l'ajuster.

Sans surprise, Keras dispose d'un optimiseur RMSprop :

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Notez que l'argument `rho` correspond à β dans l'équation 3.7. Excepté sur les problèmes très simples, cet optimiseur affiche des performances quasi toujours meilleures qu'AdaGrad. D'ailleurs, il s'agissait de l'algorithme d'optimisation préféré de nombreux chercheurs jusqu'à l'arrivée de l'optimisation Adam.

3.3.5 Optimisation Adam et Nadam

Adam⁸², pour *Adaptive Moment Estimation*, réunit les idées de l'optimisation avec inertie et de RMSProp. Il maintient, à l'instar de la première, une moyenne mobile exponentielle des gradients antérieurs et, à l'instar de la seconde, une moyenne mobile exponentielle des carrés des gradients passés (voir l'équation 3.8)⁸³.

81. Geoffrey Hinton et Tijmen Tieleman ont créé cet algorithme en 2012, et Geoffrey Hinton l'a présenté lors de son cours sur les réseaux de neurones (les diapositives sont disponibles à l'adresse <https://homl.info/57>, la vidéo à l'adresse <https://homl.info/58>). Puisque les auteurs n'ont jamais rédigé d'article pour le décrire, les chercheurs le citent souvent sous la référence « diapositive 29 du cours 6 ».

82. Diederik P. Kingma et Jimmy Ba, « Adam: A Method for Stochastic Optimization » (2014) : <https://homl.info/59>.

83. Il s'agit d'estimations de la moyenne et de la variance (non centrée) des gradients. La moyenne est souvent appelée *premier moment*, tandis que la variance est appelée *second moment*, d'où le nom donné à l'algorithme.

Équation 3.8 – Algorithme Adam

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \nabla_{\theta} J(\theta)$
3. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}}} + \epsilon$

Dans cette équation, t représente le numéro de l'itération (en commençant à 1).

Si l'on examine uniquement les étapes 1, 2 et 5, on constate une forte similitude avec l'optimisation avec inertie et RMSProp. La seule différence est que l'étape 1 calcule une moyenne mobile exponentielle à la place d'une somme à décroissance exponentielle, mais elles sont en réalité équivalentes, à un facteur constant près (la moyenne mobile exponentielle est simplement égale à $1 - \beta_1$ multiplié par la somme à décroissance exponentielle). Les étapes 3 et 4 représentent en quelque sorte un détail technique. Puisque \mathbf{m} et \mathbf{s} sont initialisés à zéro, elles iront vers 0 au début de l'entraînement pour aider à dynamiser \mathbf{m} et \mathbf{s} à ce moment-là.

L'hyperparamètre de décroissance de l'inertie β_1 est en général initialisé à 0,9, tandis que l'hyperparamètre de décroissance de la mise à l'échelle β_2 est souvent initialisé à 0,999. Comme précédemment, le terme de lissage ϵ a habituellement une valeur initiale minuscule, par exemple 10^{-7} . Il s'agit des valeurs par défaut utilisées par la classe Adam (pour être précis, epsilon vaut par défaut None, ce qui indique à Keras d'utiliser `keras.backend.epsilon()`, qui retourne par défaut 10^{-7} ; vous pouvez la changer en utilisant `keras.backend.set_epsilon()`). Voici comment créer un optimiseur Adam avec Keras :

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Puisque Adam est un algorithme à taux d'apprentissage adaptatif, comme AdaGrad et RMSProp, le réglage de l'hyperparamètre η pour le taux d'apprentissage est moins important. Nous pouvons souvent conserver la valeur par défaut $\eta = 0,001$, ce qui rend cet algorithme encore plus facile à employer que la descente de gradient.



Si vous commencez à vous sentir submergé par toutes ces différentes techniques et vous demandez comment choisir celles qui conviennent à votre tâche, pas de panique : des conseils pratiques sont fournis à la fin de ce chapitre.

Pour finir, mentionnons deux variantes d'Adam :

- *AdaMax*

À l'étape 2 de l'équation 3.8, Adam cumule dans \mathbf{s} les carrés des gradients (avec un poids supérieur pour les gradients les plus récents). À l'étape 5, si nous ignorons ϵ et les étapes 3 et 4 (qui sont de toute manière des détails techniques),

Adam réduit la mise à jour du paramètre par la racine carrée de s . En bref, Adam réduit la mise à jour du paramètre par la norme ℓ_2 des gradients affaiblis temporellement (rappelons que la norme ℓ_2 est la racine carrée de la somme des carrés). AdaMax, décrit avec Adam dans le même article, remplace la norme ℓ_2 par la norme ℓ_∞ (une autre façon de dire le maximum). Plus précisément, il remplace l'étape 2 de l'équation 3.8 par $s \leftarrow \max(\beta_2 s, \nabla_{\theta} J(\theta))$, supprime l'étape 4 et, dans l'étape 5, réduit la mise à jour du gradient d'un facteur s , qui est simplement le gradient affaibli temporellement maximum. En pratique, cette modification peut rendre AdaMax plus stable que Adam, mais cela dépend du jeu de données, et en général Adam affiche de meilleures performances. Il ne s'agit donc que d'un autre optimiseur que vous pouvez essayer si vous rencontrez des problèmes avec Adam sur une tâche.

- *Nadam*

L'optimisation Nadam correspond à l'optimisation Adam complétée de l'astuce de Nesterov. Elle convergera donc souvent plus rapidement que Adam. Dans son rapport⁸⁴ qui présente cette technique, Timothy Dozat compare de nombreux optimiseurs sur diverses tâches et conclut que Nadam surpassé en général Adam mais qu'il est parfois dépassé par RMSProp.



Les méthodes d'optimisation adaptative (y compris RMSProp, Adam et Nadam) sont souvent intéressantes, en convergeant rapidement vers une bonne solution. Cependant, l'article⁸⁵ publié en 2017 par Ashia C. Wilson *et al.* a montré qu'elles peuvent mener à des solutions dont la généralisation est mauvaise sur certains jeux de données. En conséquence, lorsque les performances de votre modèle vous déçoivent, essayez d'utiliser à la place la version de base du gradient accéléré de Nesterov : il est possible que votre jeu de données soit simplement allergique au gradient adaptatif. N'hésitez pas à consulter également les dernières recherches, car le domaine évolue très rapidement.

Toutes les techniques d'optimisation décrites précédemment se fondent exclusivement sur les *dérivées partielles de premier ordre* (*jacobiens*). Dans la littérature sur l'optimisation, on trouve d'excellents algorithmes basés sur les *dérivées partielles de second ordre* (*hessiens*, qui sont les dérivés partielles des jacobiens). Malheureusement, ces algorithmes sont très difficiles à appliquer aux réseaux de neurones profonds, car ils ont n^2 hessiens par sortie (où n correspond au nombre de paramètres), à opposer aux seuls n jacobiens par sortie. Puisque les RNP présentent en général des dizaines de milliers de paramètres, la mémoire disponible ne permet pas d'accueillir les algorithmes d'optimisation de second ordre et, même si c'était le cas, le calcul des hessiens serait beaucoup trop long.

84. Timothy Dozat, « Incorporating Nesterov Momentum into Adam » (2016) : <https://homl.info/nadam>.

85. Ashia C. Wilson *et al.*, « The Marginal Value of Adaptive Gradient Methods in Machine Learning », *Advances in Neural Information Processing Systems*, 30 (2017), 4148-4158 : <https://homl.info/60>.

Entraîner des modèles creux

Tous les algorithmes d'optimisation donnés précédemment produisent des modèles denses. Autrement dit, la plupart des paramètres seront différents de zéro. Si vous avez besoin d'un modèle extrêmement rapide au moment de l'exécution ou si vous voulez qu'il occupe moins de place en mémoire, vous préférerez à la place arriver à un modèle creux.

Pour cela, une solution triviale consiste à entraîner le modèle de façon habituelle, puis à retirer les poids les plus faibles (les fixer à zéro). Mais, en général, vous n'obtiendrez pas un modèle très creux et cela peut dégrader les performances du modèle.

Une meilleure solution consiste à appliquer une régularisation ℓ_1 forte pendant l'entraînement (nous verrons comment plus loin dans ce chapitre), car elle incite l'optimiseur à fixer à zéro autant de poids que possible⁸⁶.

Si ces techniques ne suffisent pas, tournez-vous vers TF-MOT⁸⁷ (*TensorFlow Model Optimization Toolkit*), qui propose une API d'élagage capable de supprimer de façon itérative des connexions pendant l'entraînement en fonction de leur importance.

Le tableau 3.2 compare tous les optimiseurs décrits jusqu'à présent.

Tableau 3.2 – Comparaison des optimiseurs
(* signifie mauvais, ** signifie moyen, et *** signifie bon)

Classe	Vitesse de convergence	Qualité de convergence
SGD	*	***
SGD (momentum=...)	**	***
SGD (momentum=..., nesterov=True)	**	***
Adagrad	***	* (s'arrête trop tôt)
RMSprop	***	** ou ***
Adam	***	** ou ***
Nadam	***	** ou ***
AdaMax	***	** ou ***

3.3.6 Planifier le taux d'apprentissage

Il est très important de trouver le bon taux d'apprentissage. Si l'on choisit une valeur fortement trop élevée, l'entraînement risque de diverger⁸⁸. Si l'on choisit une valeur trop faible, l'entraînement finira par converger vers l'optimum, mais il lui faudra beaucoup de temps. S'il est fixé un peu trop haut, la progression sera initialement très rapide, mais l'algorithme finira par danser autour de l'optimum, sans jamais s'y arrêter.

86. Voir la régression lasso au § 1.9.2.

87. <https://homl.info/tfmot>

88. Voir la descente de gradient au § 1.6.

vraiment. Si le temps de calcul est compté, il faudra peut-être interrompre l'entraînement avant que l'algorithme n'ait convergé correctement, ce qui résultera en une solution non optimale (voir la figure 3.8).

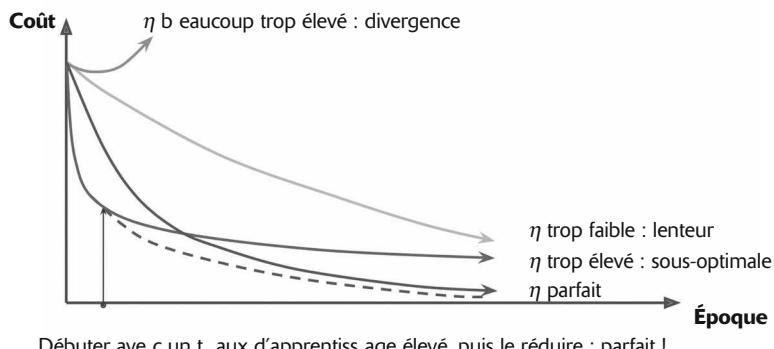


Figure 3.8 – Courbes d'apprentissage pour différents taux d'apprentissage η

Nous l'avons expliqué au chapitre 2, pour trouver un taux d'apprentissage convenable, vous pouvez entraîner un réseau pendant quelques centaines d'itérations, en augmentant exponentiellement le taux d'apprentissage depuis une valeur très faible vers une valeur très grande, puis en comparant les courbes d'apprentissage et en retenant un taux d'apprentissage légèrement inférieur à celui qui correspond à la courbe d'apprentissage qui a commencé à remonter. Vous pouvez ensuite réinitialiser votre modèle et l'entraîner avec ce taux d'apprentissage.

Mais il est possible de trouver mieux qu'un taux d'apprentissage constant. En partant d'un taux d'apprentissage élevé et en le diminuant lorsqu'il ne permet plus une progression rapide, vous pouvez arriver à une bonne solution plus rapidement qu'avec le taux d'apprentissage constant optimal. Il existe de nombreuses stratégies pour réduire le taux d'apprentissage pendant l'entraînement. Il peut également être intéressant de commencer avec un taux d'apprentissage bas, de l'augmenter, puis de le baisser à nouveau. Ces stratégies se nomment *échéanciers d'apprentissage* (*learning schedule*). Voici les plus répandues :

- *Planification par puissance*

Le taux d'apprentissage est une fonction du nombre d'itérations : $\eta(t) = \eta_0 / (1 + t/s)^c$. Le taux d'apprentissage initial η_0 , la puissance c (généralement fixée à 1) et les étapes s sont des hyperparamètres. Le taux d'apprentissage baisse à chaque étape. Après s étapes, il atteint $\eta_0/2$. Après s étapes supplémentaires, il arrive à $\eta_0/3$, puis il baisse jusqu'à $\eta_0/4$, puis $\eta_0/5$, et ainsi de suite. Cette planification commence par diminuer rapidement, puis de plus en plus lentement. Bien entendu, la planification par puissance demande d'ajuster η_0 et s (et, éventuellement, c).

- *Planification par exponentielle*

Le taux d'apprentissage est fixé à $\eta(t) = \eta_0 0,1^{t/s}$. Le taux d'apprentissage va diminuer progressivement d'un facteur 10 toutes les s étapes. Alors que la planification par puissance baisse le taux d'apprentissage de plus en plus lentement, la planification par exponentielle le réduit constamment d'un facteur 10 toutes les s étapes.

- *Taux d'apprentissage constants par morceaux*

On utilise un taux d'apprentissage constant pendant un certain nombre d'époques (par exemple, $\eta_0 = 0,1$ pendant 5 époques), puis un taux d'apprentissage plus faible pendant un certain autre nombre d'époques (par exemple, $\eta_0 = 0,001$ pendant 50 époques), et ainsi de suite. Si cette solution peut convenir, elle impose quelques tâtonnements pour déterminer les taux d'apprentissage appropriés et les durées pendant lesquelles les utiliser.

- *Planification à la performance*

On mesure l'erreur de validation toutes les N étapes (comme pour l'arrêt prématué) et on réduit le taux d'apprentissage d'un facteur λ lorsque l'erreur ne diminue plus.

- *Planification 1cycle*

Contrairement aux autres approches, *1cycle* (décrise dans un article⁸⁹ publié en 2018 par Leslie Smith) commence par augmenter le taux d'apprentissage initial η_0 , le faisant croître linéairement pour atteindre η_1 au milieu de l'entraînement. Puis le taux d'apprentissage est abaissé linéairement jusqu'à η_0 pendant la seconde moitié de l'entraînement, en terminant les quelques dernières époques par une réduction du taux de plusieurs ordres de grandeur (toujours linéairement). Le taux d'apprentissage maximal η_1 est déterminé en utilisant l'approche utilisée pour trouver le taux d'apprentissage optimal, tandis que le taux d'apprentissage initial η_0 est choisi environ 10 fois plus faible. Lorsqu'une inertie est employée, elle commence avec une valeur élevée (par exemple, 0,95), puis nous la diminuons jusqu'à une valeur plus faible pendant la première moitié de l'entraînement (par exemple, linéairement jusqu'à 0,85), pour la remonter jusqu'à la valeur maximale (par exemple, 0,95) au cours de la seconde moitié de l'entraînement, en terminant les quelques dernières époques avec cette valeur.

Smith a mené de nombreuses expériences qui montrent que cette approche permet souvent d'accélérer considérablement l'entraînement et d'atteindre de meilleures performances. Par exemple, sur le très populaire jeu de données d'images CIFAR10, elle a permis d'atteindre une précision de validation de 91,9 % en seulement 100 époques, à comparer à la précision de 90,3 % en 800 époques obtenue avec une approche standard (pour la même architecture du réseau de neurones).

89. Leslie N. Smith, « A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay » (2018) : <https://homl.info/1cycle>.

Un article⁹⁰ publié en 2013 par A. Senior *et al.* compare les performances de quelques techniques de planification parmi les plus répandues lors de l’entraînement de réseaux de neurones profonds destinés à la reconnaissance vocale, en utilisant l’optimisation avec inertie. Les auteurs concluent que, dans ce contexte, la planification à la performance et la planification par exponentielle se comportent bien, mais ils préfèrent cette dernière car elle est plus facile à régler et converge un peu plus rapidement vers la solution optimale (ils mentionnent également sa plus grande simplicité d’implémentation que la planification à la performance, mais, dans Keras, les deux méthodes sont simples). Cela dit, l’approche 1cycle semble offrir de meilleures performances encore.

L’implémentation d’une planification par puissance dans Keras reste l’option la plus simple : il suffit de fixer l’hyperparamètre `decay` au moment de la création d’un optimiseur :

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

`decay` est l’inverse de s (le nombre d’étapes après lesquelles le taux d’apprentissage est diminué d’une unité supplémentaire) et Keras suppose que c vaut 1.

La planification par exponentielle et les taux d’apprentissage constants par morceaux sont également assez simples. Vous devez tout d’abord définir une fonction qui prend l’époque en cours et retourne le taux d’apprentissage. Implémentons, par exemple, la planification par exponentielle :

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

Si vous préférez ne pas fixer en dur η_0 et s , vous pouvez créer une fonction qui retourne une fonction configurée :

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Créez ensuite un rappel `LearningRateScheduler`, en lui indiquant la fonction de planification, et passez ce rappel à la méthode `fit()` :

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Au début de chaque époque, `LearningRateScheduler` mettra à jour l’attribut `learning_rate` de l’optimiseur. En général, actualiser le taux d’apprentissage une fois par époque est suffisant, mais, si vous préférez le mettre à jour plus fréquemment, par exemple à chaque étape, vous pouvez écrire votre propre rappel (un exemple est donné dans la section « Exponential Scheduling » du notebook⁹¹).

90. Andrew Senior *et al.*, « An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition », *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013), 6724-6728 : <https://homl.info/63>.

91. Voir « 11_training_deep_neural_networks.ipynb » sur <https://github.com/ageron/handson-ml2>.

Cette actualisation à chaque étape peut être intéressante si chaque époque comprend de nombreuses étapes. Une autre solution consiste à utiliser l'approche `keras.optimizers.schedules` décrite plus loin.

La fonction de planification peut accepter en second argument le taux d'apprentissage courant. Par exemple, la fonction de planification suivante multiplie le taux d'apprentissage précédent par $0,1^{1/20}$, qui donne la même décroissance exponentielle (excepté que cette décroissance débute à présent non pas à l'époque 1 mais à l'époque 0) :

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

Puisque cette implémentation se fonde sur le taux d'apprentissage initial de l'optimiseur (au contraire de la version précédente), il est important de le fixer de façon appropriée.

L'enregistrement d'un modèle sauvegarde également l'optimiseur et son taux d'apprentissage. Autrement dit, avec cette nouvelle fonction de planification, vous pouvez simplement charger un modèle entraîné et poursuivre l'entraînement là où il s'était arrêté. En revanche, les choses se compliquent lorsque la fonction de planification exploite l'argument `epoch` : l'époque n'est pas enregistrée et elle est réinitialisée à zéro chaque fois que vous appelez la méthode `fit()`. Si vous souhaitez poursuivre l'entraînement d'un modèle à partir de son point d'arrêt, vous risquez alors d'obtenir un taux d'apprentissage très élevé, qui risque d'endommager les poids du modèle. Une solution consiste à préciser manuellement l'argument `initial_epoch` de la méthode `fit()` afin que `epoch` commence avec la bonne valeur.

Pour les taux d'apprentissage constants par morceaux, vous pouvez employer une fonction de planification semblable à la suivante (comme précédemment, vous pouvez définir une fonction plus générale si nécessaire ; un exemple est donné dans la section « Piecewise Constant Scheduling » du notebook⁹²), puis créer un rappel `LearningRateScheduler` avec cette fonction et le passer à la méthode `fit()`, exactement comme nous l'avons fait pour la planification par exponentiel :

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

Dans le cas de la planification à la performance, utilisez le rappel `ReduceLROnPlateau`. Par exemple, si vous passez le rappel suivant à la méthode `fit()`, le taux d'apprentissage sera multiplié par 0,5 dès que la meilleure perte de validation ne s'améliore pas pendant cinq époques consécutives (d'autres options sont disponibles et détaillées dans la documentation) :

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

92. Voir « 11_training_deep_neural_networks.ipynb » sur <https://github.com/ageron/handson-ml2>.

Enfin, tf.keras propose une autre solution d'implémentation d'un échéancier de taux d'apprentissage. Commencez par définir le taux d'apprentissage avec l'une des méthodes de planification disponibles dans `keras.optimizers.schedules`, puis passez ce taux d'apprentissage à n'importe quel optimiseur. Dans ce cas, le taux d'apprentissage est actualisé non pas à chaque époque mais à chaque étape. Par exemple, voici comment implémenter la planification par exponentielle définie précédemment dans la fonction `exponential_decay_fn()`:

```
s = 20 * len(X_train) // 32 # nombre d'étapes dans 20 époques
# (taille du lot = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

Beau et simple, sans compter que l'enregistrement du modèle sauvegarde également le taux d'apprentissage et son échéancier (y compris son état). Toutefois, cette solution ne fait pas partie de l'API Keras ; elle est propre à tf.keras.

L'implémentation de l'approche 1cycle ne pose pas de difficultés particulières : il suffit de créer un rappel personnalisé qui modifie le taux d'apprentissage à chaque itération (pour actualiser le taux d'apprentissage de l'optimiseur, fixez `self.model.optimizer.lr`). Un exemple est donné dans la section « 1Cycle scheduling » du notebook⁹³.

En résumé, la décroissance exponentielle, la planification à la performance et 1cycle permettent d'accélérer considérablement la convergence. N'hésitez pas à les essayer !

3.4 ÉVITER LE SURAJUSTEMENT GRÂCE À LA RÉGULARISATION

« Avec quatre paramètres je peux ajuster un éléphant, avec cinq, je peux lui faire gigoter la trompe. »

John von Neumann, cité par Enrico Fermi dans *Nature* 427

Avec des milliers de paramètres, on peut ajuster l'intégralité du zoo. Les réseaux de neurones profonds possèdent en général des dizaines de milliers de paramètres, voire des millions. Avec une telle quantité, un réseau dispose d'une liberté incroyable et peut s'adapter à une grande diversité de jeux de données complexes. Mais cette grande souplesse signifie également qu'il est sujet au surajustement du jeu d'entraînement. Nous avons donc besoin d'une régularisation.

Au chapitre 2, nous avons déjà implémenté l'une des meilleures techniques de régularisation : l'arrêt prématûre. Par ailleurs, même si la normalisation par lots a été conçue pour résoudre les problèmes d'instabilité des gradients, elle constitue un régulariseur plutôt bon. Dans cette section, nous présentons quelques-unes des techniques de régularisation les plus répandues dans les réseaux de neurones : régularisation ℓ_1 et ℓ_2 , dropout et régularisation max-norm.

93. Voir « 11_training_deep_neural_networks.ipynb » sur <https://github.com/ageron/handson-ml2>.

3.4.1 Régularisation ℓ_1 et ℓ_2

Vous pouvez employer la régularisation ℓ_2 pour contraindre les poids des connexions d'un réseau de neurones et/ou la régularisation ℓ_1 si vous voulez un modèle creux (avec de nombreux poids égaux à zéro)⁹⁴. Voici comment appliquer la régularisation ℓ_2 au poids des connexions d'une couche Keras, en utilisant un facteur de régularisation égal à 0,01 :

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

La fonction `l2()` retourne un régulariseur qui sera invoqué à chaque étape de l'entraînement de manière à calculer la perte de régularisation. Elle est ensuite ajoutée à la perte finale. Vous pouvez simplement utiliser `keras.regularizers.l1()` pour une régularisation ℓ_1 ; pour appliquer les régularisations ℓ_1 et ℓ_2 , il suffit d'utiliser `keras.regularizers.l1_l2()` (en précisant les deux facteurs de régularisation).

Puisque, en général, vous souhaitez appliquer le même régulariseur à toutes les couches du réseau, et utiliser la même fonction d'activation et la même stratégie d'initialisation dans toutes les couches cachées, vous allez répéter les mêmes arguments. Le code va devenir laid et sera sujet aux erreurs. Pour l'éviter, vous pouvez essayer de remanier le code afin d'utiliser des boucles. Mais une autre solution consiste à utiliser la fonction Python `functools.partial()` qui permet de créer une fine enveloppe autour d'un objet exécutable, avec des valeurs d'arguments par défaut :

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

3.4.2 Dropout

Dans le contexte des réseaux de neurones profonds, le dropout est l'une des techniques de régularisation les plus répandues. Elle a été proposée⁹⁵ par Geoffrey Hinton

94. Concernant les régularisations ℓ_2 et ℓ_1 , voir respectivement le § 1.9.1 et le § 1.9.2.

95. Geoffrey E. Hinton *et al.*, « Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors » (2012) : <https://hml.info/64>.

en 2012 et détaillée ensuite dans un article⁹⁶ de Nitish Srivastava *et al.* Sa grande efficacité a été démontrée : même les réseaux de neurones les plus modernes voient leur précision améliorée de 1 à 2 % par simple ajout du dropout. Cela peut sembler peu, mais une amélioration de 2 % pour un modèle dont la précision est déjà de 95 % signifie une baisse du taux d'erreur d'environ 40 % (passant d'une erreur de 5 % à environ 3 %).

L'algorithme est relativement simple. À chaque étape d'entraînement, chaque neurone (y compris les neurones d'entrée, mais pas les neurones de sortie) a une probabilité p d'être temporairement « éteint ». Autrement dit, il sera totalement ignoré au cours de cette étape d'entraînement, mais il pourra être actif lors de la suivante (voir la figure 3.9). L'hyperparamètre p est appelé *taux d'extinction* (*dropout rate*) et se situe généralement entre 10 et 50 % : plus proche de 20 à 30 % dans les réseaux de neurones récurrents (voir le chapitre 7) et plus proche de 40 à 50 % dans les réseaux de neurones convolutifs (voir le chapitre 6). Après un entraînement, les neurones ne sont plus jamais éteints. C'est tout (à l'exception d'un détail technique que nous verrons bientôt).

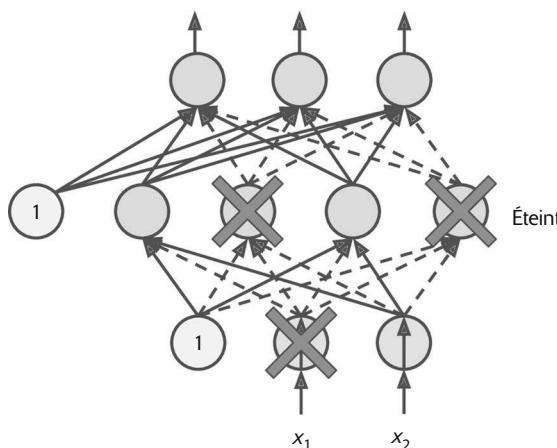


Figure 3.9 – Dans la régularisation par dropout, certains neurones choisis aléatoirement au sein d'une ou plusieurs couches (excepté la couche de sortie) sont éteints à chaque étape d'entraînement ; ils produisent 0 en sortie pendant cette itération (représentée par les flèches en pointillés).

Au premier abord, il peut sembler surprenant que cette technique destructrice fonctionne. Est-ce qu'une entreprise afficherait de meilleures performances si l'on demandait à chacun de ses employés de jouer à pile ou face chaque jour pour savoir s'il devait travailler ? Qui sait, ce serait peut-être le cas ! L'entreprise serait évidemment obligée d'adapter son organisation. Elle ne pourrait pas compter sur une seule

96. Nitish Srivastava *et al.*, « Dropout: A Simple Way to Prevent Neural Networks from Overfitting », *Journal of Machine Learning Research*, 15 (2014), 1929-1958 : <https://hml.info/65>.

personne pour remplir la machine à café ou réaliser d'autres tâches critiques. Toutes les expertises devraient être réparties sur plusieurs personnes. Les employés devraient apprendre à collaborer non pas avec une poignée de collègues, mais avec de nombreux autres. L'entreprise serait beaucoup plus résistante. Si une personne venait à la quitter, les conséquences seraient minimes.

On ne sait pas si cette idée peut réellement s'appliquer aux entreprises, mais il est certain qu'elle fonctionne parfaitement avec les réseaux de neurones. Les neurones entraînés sous dropout n'ont pas la possibilité de s'adapter de concert avec les neurones voisins : ils doivent avoir chacun une plus grande utilité propre. Par ailleurs, ils ne peuvent pas s'appuyer trop fortement sur quelques neurones d'entrée, mais doivent prêter attention à tous leurs neurones d'entrée. Ils finissent par devenir moins sensibles aux légers changements en entrée. Nous obtenons à terme un réseau plus robuste, avec une plus grande capacité de généralisation.

Pour bien comprendre la puissance du dropout, il faut réaliser que chaque étape d'entraînement génère un réseau de neurones unique. Puisque chaque neurone peut être présent ou absent, il existe un total de 2^N réseaux possibles (où N est le nombre total de neurones éteignables). Ce nombre est tellement énorme qu'il est quasi impossible que le même réseau de neurones soit produit à deux reprises. Après 10 000 étapes d'entraînement, nous avons en réalité entraîné 10 000 réseaux de neurones différents (chacun avec une seule instance d'entraînement). Ces réseaux de neurones ne sont évidemment pas indépendants, car ils partagent beaucoup de leurs poids, mais ils n'en restent pas moins tous différents. Le réseau de neurones résultant peut être vu comme un ensemble moyen de tous ces réseaux de neurones plus petits.



En pratique, vous pouvez appliquer le dropout uniquement aux neurones des une à trois couches supérieures (à l'exception de la couche de sortie).

Il reste un petit détail technique qui a son importance. Supposons que $p = 50\%$, alors, au cours des tests, un neurone sera connecté à deux fois plus de neurones d'entrée qu'il ne l'a été (en moyenne) au cours de l'entraînement. Pour compenser cela, il faut multiplier les poids des connexions d'entrée de chaque neurone par 0,5 après l'entraînement. Dans le cas contraire, chaque neurone recevra un signal d'entrée total environ deux fois plus grand qu'au cours de l'entraînement du réseau. Il est donc peu probable qu'il se comporte correctement. Plus généralement, après l'entraînement, il faut multiplier les poids des connexions d'entrée par la probabilité de conservation (*keep probability*) $1 - p$. Une autre solution consiste à diviser la sortie de chaque neurone par la probabilité de conservation, pendant l'entraînement (ces deux approches ne sont pas totalement équivalentes, mais elles fonctionnent aussi bien l'une que l'autre).

Pour implémenter la régularisation dropout avec Keras, nous pouvons mettre en place la couche `keras.layers.Dropout`. Pendant l'entraînement, elle éteint aléatoirement certaines entrées (en les fixant à 0) et divise les entrées restantes par la probabilité de conservation. Après l'entraînement, elle ne fait plus rien ;

elle se contente de passer les entrées à la couche suivante. Le code suivant applique la régularisation dropout avant chaque couche Dense, en utilisant un taux d'extinction de 0,2 :

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```



Puisque le dropout est actif uniquement pendant l'entraînement, comparer la perte d'entraînement et la perte de validation peut induire en erreur. En particulier, un modèle peut surajuster le jeu d'entraînement tout en affichant des pertes d'entraînement et de validation comparables. Faites attention à évaluer la perte d'entraînement sans la régularisation dropout (par exemple, après l'entraînement).

Si vous observez un surajustement du modèle, vous pouvez augmenter le taux d'extinction. À l'inverse, vous devez essayer de diminuer le taux d'extinction si le modèle sous-ajuste le jeu d'entraînement. Il peut également être intéressant d'augmenter le taux d'extinction lorsque les couches sont grandes, pour le réduire avec les petites. Par ailleurs, de nombreuses architectures modernes utilisent le dropout uniquement après la dernière couche cachée. Vous pouvez tester cette approche si un dropout intégral est trop fort.

La régularisation dropout a tendance à ralentir la convergence de façon importante, mais, avec les réglages adéquats, elle produit habituellement un meilleur modèle. La qualité du résultat compense généralement le temps et le travail supplémentaires.



Pour régulariser un réseau autonormalisant fondé sur la fonction d'activation SELU (comme décrit précédemment), utilisez *alpha dropout*: cette variante du dropout conserve la moyenne et l'écart-type de ses entrées (elle a été décrite dans le même article que SELU, car le dropout normal empêche l'autonormalisation).

3.4.3 Monte Carlo (MC) Dropout

En 2016, Yarin Gal et Zoubin Ghahramani ont publié un article⁹⁷ dans lequel ils ajoutent quelques bonnes raisons supplémentaires d'employer le dropout :

- Premièrement, l'article établit une profonde connexion entre les réseaux dropout (c'est-à-dire les réseaux de neurones qui comprennent une couche

⁹⁷. Yarin Gal et Zoubin Ghahramani, «Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning», *Proceedings of the 33rd International Conference on Machine Learning* (2016), 1050-1059 : <https://homl.info/mcdropout>.

Dropout avant chaque couche de poids) et l'inférence bayésienne approchée⁹⁸. Le dropout a ainsi reçu une solide justification mathématique.

- Deuxièmement, les auteurs ont introduit une technique puissante appelée MC Dropout, qui permet d'améliorer les performances de tout modèle dropout entraîné sans avoir à le réentraîner ni même à le modifier, qui fournit une bien meilleure mesure de l'incertitude du modèle et qui est également simple à mettre en œuvre.

Si vous trouvez que cela ressemble à un avertissement à la prudence, examinez le code suivant. Cette implémentation complète de MC Dropout améliore le modèle dropout entraîné précédemment sans l'entraîner de nouveau :

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Nous avons effectué seulement 100 prédictions sur le jeu de test, en fixant `training=True` pour être certain que la couche Dropout soit active et empile les prédictions. Le dropout étant actif, toutes les prédictions seront différentes. Rappelons que `predict()` retourne une matrice avec une ligne par instance et une colonne par classe. Puisque le jeu de test comprend 10000 instances et 10 classes, la forme de cette matrice est [10000, 10]. Nous empilons 100 matrices de cette forme, et `y_probas` est donc un tableau [100, 10000, 10]. Lorsque nous effectuons une moyenne sur la première dimension (`axis=0`), nous obtenons `y_proba`, un tableau de forme [10000, 10], comme ce que nous obtiendrions avec une seule prédition. C'est tout ! La moyenne sur de multiples prédictions avec le dropout actif nous donne une estimation de Monte Carlo généralement plus fiable que le résultat d'une seule prédition avec le dropout inactif. Voyons, par exemple, la prédition du modèle sur la première instance du jeu de test Fashion MNIST en désactivant le dropout :

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]], 
      dtype=float32)
```

Le modèle semble quasi certain que cette image correspond à la classe 9 (bottine). Pouvons-nous lui faire confiance ? Y a-t-il vraiment si peu de place pour le doute ? Comparons cette prédition avec celles effectuées lorsque le dropout est actif :

```
>>> np.round(y_probas[:, :1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68]],
      [[0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]],
      [[0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]],
      [...]]
```

Le résultat n'est plus du tout le même. Apparemment, lorsque nous activons le dropout, le modèle n'est plus aussi sûr de lui. Il semble avoir une préférence pour la classe 9, mais il hésite parfois entre les classes 5 (sandale) et 7 (basket) ; ce qui reste

⁹⁸. Plus précisément, ils ont notamment montré que l'entraînement d'un réseau dropout est mathématiquement équivalent à une inférence bayésienne approchée dans un type spécifique de modèle probabiliste appelé *processus gaussien profond* (*Deep Gaussian Process*).

sensé car il s'agit de chaussures. Si nous effectuons une moyenne sur la première dimension, nous obtenons les prédictions MC Dropout suivantes :

```
>>> np.round(y_proba[:1], 2)
array([0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]),
      dtype=float32)
```

Le modèle pense toujours que l'image correspond à la classe 9, mais avec une confiance de seulement 62 %, ce qui semble beaucoup plus raisonnable que les 99 % précédents. De plus, il est intéressant de connaître exactement les autres classes qu'il pense probables. Vous pouvez aussi jeter un œil à l'écart-type des estimations de probabilité (<https://xkcd.com/2110>) :

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]),
      dtype=float32)
```

Apparemment, la variance dans les estimations de probabilités est importante : si vous deviez construire un système sensible (par exemple, un système médical ou financier), vous prendriez probablement avec extrême prudence une prédition aussi incertaine. Vous ne la considéreriez certainement pas comme une prédition sûre à 99 %. Par ailleurs, la précision du modèle a légèrement augmenté, passant de 86,8 à 86,9 :

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```



Le nombre d'échantillons utilisés dans Monte Carlo (100 dans cet exemple) est un hyperparamètre que vous pouvez ajuster. Plus il est élevé, plus les prédictions et leurs estimations d'incertitude seront précises. Toutefois, si vous en doutiez, le temps d'inférence sera également doublé. Par ailleurs, au-delà d'un certain nombre d'échantillons, vous remarquerez peu d'amélioration. Votre travail est donc de trouver le bon compromis entre latence et précision, en fonction de votre application.

Si votre modèle contient d'autres couches au comportement particulier pendant l'entraînement (comme des couches BatchNormalization), vous ne devez pas forcer le mode d'entraînement comme nous venons de le faire. À la place, vous devez remplacer les couches Dropout par la classe MCDropout suivante⁹⁹:

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

Vous créez une sous-classe de la couche Dropout et surchargez la méthode `call()` pour forcer son argument `training` à `True` (voir le chapitre 4). De manière comparable, vous pourriez également définir une classe MCAlphaDropout

99. Cette classe MCDropout est compatible avec toutes les API Keras, y compris l'API Sequential. Si vous vous intéressez uniquement aux API Functional ou Subclassing, il est inutile de créer une classe MCDropout. Vous pouvez simplement créer une couche Dropout normale et l'invoquer avec `training=True`.

en dérivant de AlphaDropout. Si le modèle est créé à partir de zéro, il suffit de remplacer Dropout par MCDropout. En revanche, si le modèle a déjà été entraîné avec Dropout, vous devez créer un nouveau modèle identique à celui existant, mais en remplaçant les couches Dropout par MCDropout, puis copier les poids du modèle existant dans votre nouveau modèle.

En résumé, MC Dropout est une technique fantastique qui améliore les modèles dropout et fournit de meilleures estimations d'incertitude. Bien entendu, puisqu'il s'agit d'un dropout normal pendant l'entraînement, elle sert aussi de régulariseur.

3.4.4 Régularisation max-norm

Les réseaux de neurones mettent également souvent en œuvre une autre technique appelée *régularisation max-norm*. Pour chaque neurone, elle contraint les poids \mathbf{w} des connexions entrantes de sorte que $\|\mathbf{w}\|_2 \leq r$, où r est l'hyperparamètre max-norm et $\|\cdot\|_2$ est la norme ℓ_2 .

La régularisation max-norm n'ajoute aucun terme de perte de régularisation à la fonction de perte globale. À la place, elle est généralement mise en œuvre par le calcul de $\|\mathbf{w}\|_2$ après chaque étape d'entraînement et le redimensionnement de \mathbf{w} si nécessaire ($\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|\mathbf{w}\|_2}$).

En diminuant r , on augmente le niveau de régularisation et on réduit le risque de surajustement. La régularisation max-norm permet également d'atténuer les problèmes d'instabilité des gradients (si la normalisation par lots n'est pas utilisée).

Pour implémenter la régularisation max-norm dans Keras, l'argument `kernel_constraint` de chaque couche cachée doit être fixé à une contrainte `max_norm()` ayant la valeur maximale appropriée :

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
    kernel_constraint=keras.constraints.max_norm(1.))
```

Après chaque itération entraînement, la méthode `fit()` du modèle appelle l'objet retourné par `max_norm()`, en lui passant les poids de la couche, et reçoit en retour les poids redimensionnés, qui vont ensuite remplacer les poids de la couche. Au chapitre 4, nous verrons comment définir notre propre fonction de contrainte, si nécessaire, et l'utiliser comme `kernel_constraint`. Les termes constants peuvent également être contraints en précisant l'argument `bias_constraint`.

La fonction `max_norm()` définit l'argument `axis`, dont la valeur par défaut est 0. Une couche `Dense` possède généralement des poids de la forme [nombre d'entrées, nombre de neurones]. Par conséquent, utiliser `axis=0` signifie que la contrainte max-norm sera appliquée indépendamment à chaque vecteur de poids d'un neurone. Pour utiliser max-norm avec les couches de convolution (voir le chapitre 6), il faut s'assurer que l'argument `axis` de la contrainte `max_norm()` est défini de façon appropriée (en général `axis=[0, 1, 2]`).

3.5 RÉSUMÉ ET CONSEILS PRATIQUES

Dans ce chapitre, nous avons présenté une grande variété de techniques et vous vous demandez peut-être lesquelles vous devez appliquer. Cela dépend de la tâche concernée et il n'existe encore aucun consensus clair, mais j'ai pu déterminer que la configuration donnée au tableau 3.3 fonctionnera très bien dans la plupart des cas, sans exiger un réglage compliqué des hyperparamètres. Cela dit, vous ne devez pas considérer ces valeurs par défaut comme gravées dans le marbre.

Tableau 3.3 – Configuration par défaut d'un réseau de neurones profond

Hyperparamètre	Valeur par défaut
Initialisation du noyau	Initialisation de He
Fonction d'activation	ELU
Normalisation	Aucune si superficiel; normalisation par lots si profond
Régularisation	Arrêt prématué (et régularisation ℓ_2 si nécessaire)
Optimiseur	Optimisation avec inertie (ou RMSProp ou Nadam)
Échéancier d'apprentissage	1 cycle

Si le réseau est un simple empilage de couches denses, il peut alors profiter de l'autonormalisation et la configuration donnée au tableau 3.4 doit être employée à la place.

Tableau 3.4 – Configuration par défaut d'un réseau de neurones profond autonormalisant

Hyperparamètre	Valeur par défaut
Initialisation du noyau	Initialisation de LeCun
Fonction d'activation	SELU
Normalisation	Aucune (autonormalisation)
Régularisation	Alpha dropout si nécessaire
Optimiseur	Optimisation avec inertie (ou RMSProp ou Nadam)
Échéancier d'apprentissage	1 cycle

N'oubliez pas de normaliser les caractéristiques d'entrée ! Vous devez également tenter de réutiliser des parties d'un réseau de neurones préentraîné si vous en trouvez un qui résout un problème comparable, ou utiliser un préentraînement non supervisé si les données non étiquetées sont nombreuses, ou utiliser un préentraînement sur une tâche secondaire si vous disposez d'un grand nombre de données étiquetées pour une tâche similaire.

Les recommandations précédentes devraient couvrir la majorité des cas, mais voici quelques exceptions :

- Si vous avez besoin d'un modèle creux, vous pouvez employer la régularisation ℓ_1 (et, éventuellement, la mise à zéro des poids très faibles après l'entraînement). Si vous avez besoin d'un modèle encore plus creux, vous pouvez utiliser TF-MOT. Puisque cela mettra fin à l'autonormalisation, vous devez, dans ce cas, opter pour la configuration par défaut.
- Si vous avez besoin d'un modèle à faible latence (s'il doit effectuer des prédictions à la vitesse de la lumière), vous devez utiliser un nombre plus faible de couches, intégrer les couches de normalisation par lots dans les couches précédentes et utiliser si possible une fonction d'activation plus rapide comme leaky ReLU ou simplement ReLU. Un modèle creux pourra également aider. Enfin, vous pouvez passer la précision des nombres à virgule flottante de 32 à 16, voire 8, bits (voir § 11.2). Là aussi, envisagez l'utilisation de TF-MOT.
- Si vous développez une application sensible aux risques ou si la latence des inférences n'est pas très importante, vous pouvez utiliser MC Dropout pour augmenter les performances et obtenir des estimations de probabilités plus fiables, avec des estimations d'incertitude.

Avec ces conseils, vous êtes paré pour entraîner des réseaux très profonds ! Nous espérons que vous êtes à présent convaincu que vous pouvez aller très loin avec Keras. Toutefois, le moment viendra probablement où vous aurez besoin d'un contrôle plus important, par exemple pour écrire une fonction de perte personnalisée ou pour adapter l'algorithme d'entraînement. Dans de tels cas, vous devrez vous tourner vers l'API de bas niveau de TensorFlow, qui sera décrite au prochain chapitre.

3.6 EXERCICES

1. Est-il bon de donner la même valeur initiale à tous les poids tant que celle-ci est choisie aléatoirement avec l'initialisation de He ?
2. Est-il bon d'initialiser les termes constants à zéro ?
3. Citez trois avantages de la fonction d'activation ELU par rapport à la fonction ReLU.
4. Dans quels cas utiliseriez-vous chacune des fonctions d'activation suivantes : SELU, leaky ReLU (et ses variantes), ReLU, tanh, logistique et softmax ?
5. Que peut-il se passer lorsqu'un optimiseur SGD est utilisé et que l'hyperparamètre momentum est trop proche de 1 (par exemple, 0,99999) ?
6. Donnez trois façons de produire un modèle creux.
7. La régularisation dropout ralentit-elle l'entraînement ? Ralentit-elle l'inférence (c'est-à-dire les prédictions sur de nouvelles instances) ? Qu'en est-il de MC Dropout ?
8. Entraînement d'un réseau de neurones profond sur le jeu de données d'images CIFAR10 :

- a. Construisez un RNP constitué de 20 couches cachées, chacune avec 100 neurones (c'est trop, mais c'est la raison de cet exercice). Utilisez l'initialisation de He et la fonction d'activation ELU.
- b. En utilisant l'optimisation Nadam et l'arrêt prématué, entraînez-le sur le jeu de données CIFAR10. Vous pouvez le charger avec `keras.datasets.cifar10.load_data()`. Le jeu de données est constitué de 60 000 images en couleur de 32×32 pixels (50 000 pour l'entraînement, 10 000 pour les tests) avec 10 classes. Vous aurez donc besoin d'une couche de sortie softmax avec 10 neurones. N'oubliez pas de rechercher le taux d'apprentissage approprié chaque fois que vous modifiez l'architecture ou les hyperparamètres du modèle.
- c. Ajoutez à présent la normalisation par lots et comparez les courbes d'apprentissage. La convergence se fait-elle plus rapidement ? Le modèle obtenu est-il meilleur ? En quoi la rapidité d'entraînement est-elle affectée ?
- d. Remplacez la normalisation par lots par SELU et procédez aux ajustements nécessaires pour que le réseau s'autonormalise (autrement dit, standardisez les caractéristiques d'entrée, utilisez l'initialisation normale de LeCun, assurez-vous que le RNP contient uniquement une suite de couches denses, etc.).
- e. Essayez de régulariser le modèle avec alpha dropout. Puis, sans réentraîner votre modèle, voyez si vous pouvez obtenir une meilleure précision avec MC Dropout.
- f. Réentraînez votre modèle en utilisant la planification 1cycle et voyez si cela améliore la vitesse d'entraînement et la précision.

Les solutions de ces exercices sont données à l'annexe A.

4

Modèles personnalisés et entraînement avec TensorFlow

Depuis le début de cet ouvrage, nous avons employé uniquement l'API de haut niveau de TensorFlow, tf.keras. Elle nous a permis d'aller déjà assez loin : nous avons construit plusieurs architectures de réseaux de neurones, notamment des réseaux de régression et de classification, des réseaux Wide & Deep et des réseaux autonormalisant, en exploitant différentes techniques, comme la normalisation par lots, le dropout et les échéanciers de taux d'apprentissage.

En réalité, 95 % des cas d'utilisation que vous rencontrerez n'exigeront rien de plus que tf.keras (et tf.data ; voir le chapitre 5). Toutefois, il est temps à présent de plonger au cœur de TensorFlow et d'examiner son API Python de bas niveau (<https://hml.info/tf2api>). Elle devient indispensable lorsque nous avons besoin d'un contrôle supplémentaire pour écrire des fonctions de perte personnalisées, des indicateurs personnalisés, des couches, des modèles, des initialiseurs, des régulariseurs, des contraintes de poids, etc. Il peut même arriver que nous ayons besoin d'un contrôle total sur la boucle d'entraînement, par exemple pour appliquer des transformations ou des contraintes particulières sur les gradients (au-delà du simple écrêtage) ou pour utiliser plusieurs optimiseurs dans différentes parties du réseau.

Dans ce chapitre, nous allons examiner tous ces cas et nous verrons également comment améliorer nos modèles personnalisés et nos algorithmes d'entraînement en exploitant la fonctionnalité TensorFlow de génération automatique d'un graphe. Mais commençons par un tour rapide de TensorFlow.



TensorFlow 2.0 (rc1) a été rendu public en septembre 2019. Cette version est beaucoup plus facile à utiliser que la précédente. La première édition de cet ouvrage était fondée sur TF 1, tandis que celle-ci profite de TF 2.

4.1 PRÉSENTATION RAPIDE DE TENSORFLOW

TensorFlow est une bibliothèque logicielle puissante destinée au calcul numérique. Elle est particulièrement bien adaptée et optimisée pour l'apprentissage automatique (*Machine Learning*, ou *ML*) à grande échelle, mais elle peut être employée à toute tâche qui nécessite des calculs lourds. Elle a été développée par l'équipe Google Brain et se trouve au cœur de nombreux services à grande échelle de Google, comme Google Cloud Speech, Google Photos et Google Search. Elle est passée en open source en novembre 2015 et fait à présent partie des bibliothèques de Deep Learning les plus populaires (en termes de citation dans les articles, d'adoption dans les entreprises, d'étoiles sur GitHub, etc.). Un nombre incalculable de projets utilisent TensorFlow pour toutes sortes de tâches d'apprentissage automatique, comme la classification d'images, le traitement automatique du langage naturel, les systèmes de recommandation et les prévisions de séries chronologiques.

Voici un récapitulatif des possibilités offertes par TensorFlow :

- Son noyau est très similaire à NumPy, mais il reconnaît les processeurs graphiques.
- Il prend en charge l'informatique distribuée (sur plusieurs processeurs et serveurs).
- Il dispose d'un compilateur à la volée (JIT, *just-in-time*) capable d'optimiser la rapidité et l'empreinte mémoire des calculs. Ce compilateur se fonde sur un *graphe de calcul* extrait d'une fonction Python, sur l'optimisation de ce graphe (par exemple, en élaguant les nœuds inutilisés) et sur son exécution efficace (par exemple, en exécutant automatiquement en parallèle les opérations indépendantes).
- Les graphes de calcul peuvent être exportés dans un format portable, ce qui nous permet d'entraîner un modèle TensorFlow dans un environnement (par exemple, en utilisant Python sur Linux) et de l'exécuter dans un autre (par exemple, en utilisant Java sur un appareil Android).
- Il implémente la différentiation automatique (voir le chapitre 2 et l'annexe B) et fournit plusieurs optimiseurs excellents, comme RMSProp et Nadam (voir le « »). Il est donc facile de minimiser toutes sortes de fonctions de perte.

TensorFlow offre de nombreuses autres fonctionnalités construites au-dessus de ces fonctions de base, la plus importante étant évidemment `tf.keras`¹⁰⁰, mais on trouve également des outils de chargement et de prétraitement des données (`tf.data`, `tf.io`, etc.), des outils de traitement d'images (`tf.image`), des outils de traitement du signal (`tf.signal`), et d'autres encore (la figure 4.1 donne une vue d'ensemble de l'API Python de TensorFlow).

100. TensorFlow comprend une autre API de Deep Learning appelée *Estimators API*, mais l'équipe de TensorFlow conseille d'employer plutôt `tf.keras`.



Nous examinerons plusieurs packages et fonctions de l'API de TensorFlow, mais il est impossible d'en étudier l'intégralité. Prenez le temps de parcourir l'API, vous découvrirez combien elle est riche et documentée.

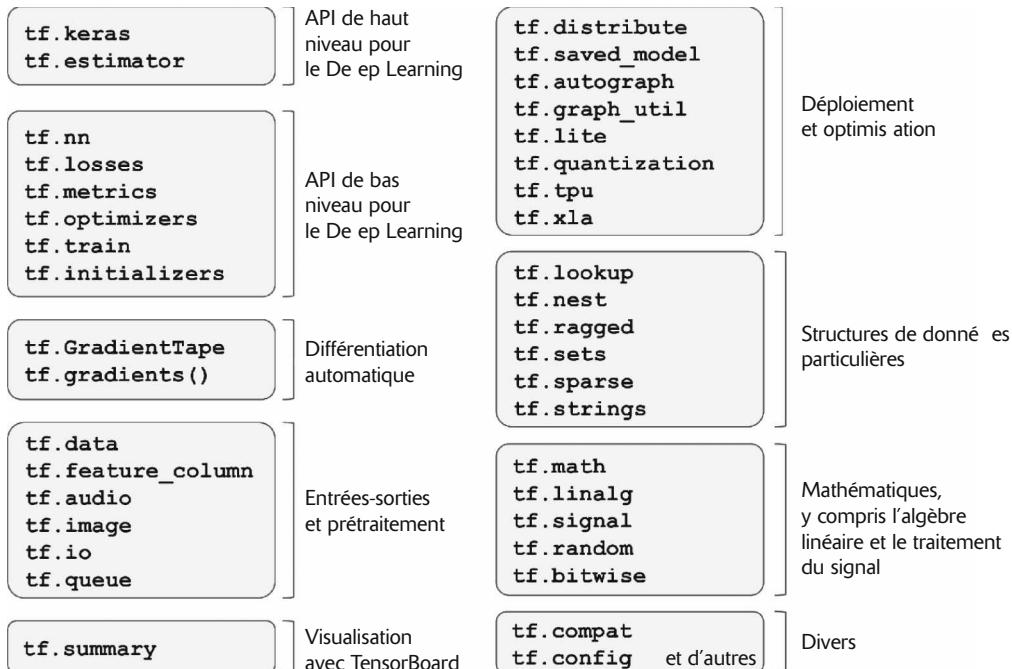


Figure 4.1 – API Python de TensorFlow

Au niveau le plus bas, chaque opération TensorFlow est implémentée par du code C++ extrêmement efficace¹⁰¹.

De nombreuses opérations disposent de plusieurs implémentations appelées noyaux (*kernels*) : chaque noyau est dédié à un type de processeur spécifique, comme des CPU, des GPU ou même des TPU (*tensor processing unit*). Les GPU sont capables d'accélérer énormément les calculs en les découpant en portions plus petites et en les exécutant en parallèle sur de nombreux threads de GPU. Les TPU sont encore plus efficaces : il s'agit de circuits intégrés ASIC spécialisés dans les opérations de Deep Learning¹⁰² (le chapitre 11 explique comment utiliser TensorFlow avec des GPU et des TPU).

101. Si jamais vous en avez besoin, mais ce ne sera probablement pas le cas, vous pouvez écrire vos propres opérations à l'aide de l'API C++.

102. Pour de plus amples informations sur les TPU et leur fonctionnement, consultez la page <https://homl.info/tpus>.

L'architecture de TensorFlow est illustrée à la figure 4.2. La plupart du temps, le code exploitera les API de haut niveau, en particulier tf.keras et tf.data, mais, si un plus grand besoin de flexibilité survient, vous emploierez l'API Python de plus bas niveau, en gérant directement des tenseurs. Il existe également des API pour d'autres langages. Dans de nombreux cas, le moteur d'exécution de TensorFlow se charge d'exécuter efficacement les opérations, même sur plusieurs processeurs et machines si vous le lui demandez.

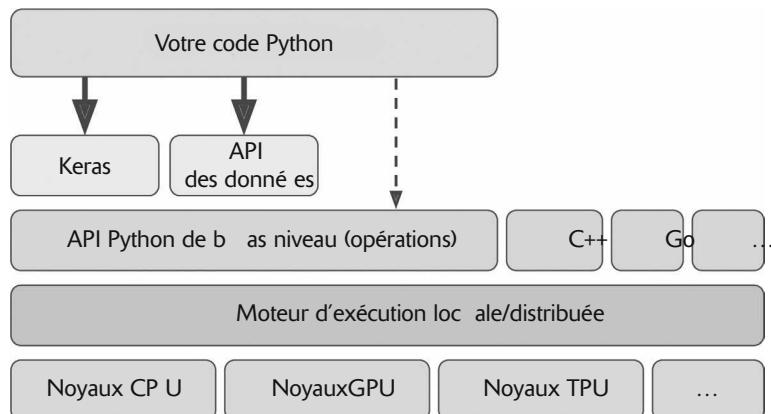


Figure 4.2 – Architecture de TensorFlow

TensorFlow est disponible non seulement sur Windows, Linux et macOS, mais également sur les appareils mobiles (version *TensorFlow Lite*), y compris iOS et Android (voir le chapitre 11). Si vous ne souhaitez pas utiliser l'API Python, il existe des API C++, Java, Go et Swift. Il existe même une version JavaScript appelée *TensorFlow.js* ; elle permet d'exécuter des modèles directement dans un navigateur.

TensorFlow ne se limite pas à sa bibliothèque. Il est au cœur d'un vaste écosystème de bibliothèques. On trouve en premier lieu TensorBoard pour la visualisation (voir le chapitre 2). TensorFlow Extended (TFX) (<https://tensorflow.org/tfx>) est un ensemble de bibliothèques développées par Google pour la mise en production des projets TensorFlow : il fournit des outils de validation des données de prétraitement, d'analyse des modèles et de service (avec TF Serving ; voir le chapitre 11). *TensorFlow Hub* de Google apporte une solution simple de téléchargement et de réutilisation de réseaux de neurones préentraînés. Vous pouvez également trouver de nombreuses architectures de réseaux de neurones, certains préentraînés, dans le « jardin » de modèles TensorFlow (<https://github.com/tensorflow/models/>). D'autres projets fondés sur TensorFlow sont disponibles sur le site TensorFlow Resources (<https://www.tensorflow.org/resources>) et à l'adresse <https://github.com/jtjoy/awesome-tensorflow>. Des centaines de projets TensorFlow sont présents sur GitHub et permettent de trouver facilement du code existant quels que soient les besoins de vos tâches.



De plus en plus d'articles sur l'apprentissage automatique proposent leurs implémentations et, parfois, des modèles préentraînés. Pour les retrouver facilement, consultez le site <https://paperswithcode.com/>.

Dernier point, mais non le moindre, une équipe de passionnés, des développeurs serviables, ainsi qu'une large communauté, contribuent à l'amélioration de TensorFlow. Les questions techniques doivent être posées sur <http://stackoverflow.com/> en ajoutant les étiquettes *tensorflow* et *python*. Pour signaler des bogues ou demander des fonctionnalités, il faut passer par GitHub (<https://github.com/tensorflow/tensorflow>). Les discussions générales se font sur le groupe Google dédié à TensorFlow (<https://groups.google.com/forum/#!topic/tensorflow/41>).

Il est temps à présent de commencer à coder !

4.2 UTILISER TENSORFLOW COMME NUMPY

L'API de TensorFlow s'articule autour des *tenseurs*, qui circulent d'une opération à l'autre, d'où le nom « flux de tenseurs », ou *TensorFlow*. Un tenseur est très similaire à un `ndarray` dans NumPy: c'est habituellement un tableau multidimensionnel, mais il peut également contenir un scalaire (une simple valeur comme 42). Puisque ces tenseurs seront importants lors de la création de fonctions de coût personnalisées, d'indicateurs personnalisés, de couches personnalisées, etc., voyons comment les créer et les manipuler.

4.2.1 Tenseurs et opérations

Nous pouvons créer un tenseur avec `tf.constant()`. Par exemple, voici un tenseur qui représente une matrice de deux lignes et trois colonnes de nombres à virgule flottante :

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrice
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # scalaire
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

À l'instar d'un `ndarray`, un `tf.Tensor` possède une forme et un type de données (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

L'utilisation des indices est comparable à celle de NumPy:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Mais le plus important est que des opérations de toutes sortes sur les tenseurs sont possibles :

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,   4.,   9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Écrire `t + 10` équivaut à invoquer `tf.add(t, 10)` (bien entendu, Python appelle la méthode magique `t.__add__(10)`, qui appelle simplement `tf.add(t, 10)`). D'autres opérateurs, comme `-` et `*` sont également reconnus. L'opérateur `@` a été ajouté dans Python 3.5 pour la multiplication de matrices ; il équivaut à un appel à la fonction `tf.matmul()`.

Toutes les opérations mathématiques de base dont nous avons besoin (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`, etc.) et la plupart des opérations existantes dans NumPy (par exemple, `tf.reshape()`, `tf.squeeze()`, `tf.tile()`) sont disponibles. Certaines fonctions n'ont pas le même nom que dans NumPy ; par exemple, `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()` et `tf.math.log()` sont les équivalents de `np.mean()`, `np.sum()`, `np.max()` et `np.log()`. Lorsque les noms diffèrent, c'est souvent pour une bonne raison. Par exemple, dans TensorFlow, nous devons écrire `tf.transpose(t)` et non pas simplement `t.T` comme dans NumPy. En effet, la fonction `tf.transpose()` ne fait pas exactement la même chose que l'attribut `T` de NumPy. Dans TensorFlow, un nouveau tenseur est créé avec sa propre copie des données permutees, tandis que, dans NumPy, `t.T` n'est qu'une vue permutee sur les mêmes données. De façon comparable, l'opération `tf.reduce_sum()` se nomme ainsi car son noyau GPU (c'est-à-dire son implémentation pour processeur graphique) utilise un algorithme de réduction qui ne garantit pas l'ordre dans lequel les éléments sont ajoutés : en raison de la précision limitée des nombres à virgule flottante sur 32 bits, le résultat peut changer très légèrement chaque fois que nous appelons cette opération. C'est la même chose pour `tf.reduce_mean()` (mais `tf.reduce_max()` est évidemment déterministe).



Des alias sont définis pour de nombreuses fonctions et classes. Par exemple, `tf.add()` et `tf.math.add()` correspondent à la même fonction. Ainsi, TensorFlow peut donner des noms concis à la plupart des opérations communes¹⁰³, tout en conservant une parfaite organisation des packages.

API de bas niveau de Keras

L'API de Keras possède sa propre API de bas niveau, dans le package `keras.backend`. Elle offre des fonctions comme `square()`, `exp()` et `sqrt()`. En général, dans `tf.keras`, ces fonctions appellent simplement les opérations TensorFlow correspondantes. Si vous souhaitez écrire du code portable sur d'autres implémentations de Keras, vous devez employer ces fonctions Keras. Cependant, puisqu'elles ne couvrent qu'un sous-ensemble de toutes les fonctions disponibles dans TensorFlow, nous utiliserons dans cet ouvrage directement les opérations TensorFlow. Voici un exemple simple d'utilisation de `keras.backend`, souvent abrégé K:

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

4.2.2 Tenseurs et NumPy

Les tenseurs travaillent parfaitement de concert avec NumPy : nous pouvons créer un tenseur à partir d'un tableau NumPy, et inversement. Nous pouvons même appliquer des opérations TensorFlow à des tableaux NumPy et des opérations NumPy à des tenseurs :

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # ou np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

¹⁰³. L'exception remarquable est `tf.math.log()`, qui, malgré son usage fréquent, ne possède pas d'alias `tf.log()` (car il pourrait être confondu avec la journalisation).



Notez que NumPy utilise par défaut une précision sur 64 bits, tandis que celle de TensorFlow se limite à 32 bits. En effet, une précision sur 32 bits est généralement suffisante pour les réseaux de neurones, sans compter que l'exécution est ainsi plus rapide et l'empreinte mémoire plus faible. Par conséquent, lorsque vous créez un tenseur à partir d'un tableau NumPy, n'oubliez pas d'indiquer `dtype=tf.float32`.

4.2.3 Conversions de type

Les conversions de type peuvent impacter fortement les performances et elles sont facilement invisibles lorsqu'elles se font automatiquement. Pour éviter cela, TensorFlow n'effectue aucune conversion de type de manière automatique. Si nous tentons d'exécuter une opération sur des tenseurs de types incompatibles, une exception est simplement lancée. Par exemple, l'addition d'un tenseur réel et d'un tenseur entier n'est pas possible, pas plus que celle d'un nombre à virgule flottante sur 32 bits et d'un autre sur 64 bits :

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]
```

Au premier abord, cela peut sembler quelque peu gênant, mais il faut se rappeler que c'est pour la bonne cause ! Et, bien entendu, nous pouvons toujours utiliser `tf.cast()` si nous avons également besoin de convertir des types :

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

4.2.4 Variables

Les valeurs `tf.Tensor` que nous avons vues jusqu'à présent sont immuables : elles ne sont pas modifiables. Cela signifie que les poids d'un réseau de neurones ne peuvent pas être représentés par des tenseurs normaux car la rétropropagation doit être en mesure de les ajuster. Par ailleurs, d'autres paramètres doivent pouvoir évoluer au fil du temps (par exemple, un optimiseur à inertie conserve la trace des gradients antérieurs). Nous avons donc besoin d'un `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Un `tf.Variable` fonctionne de manière comparable à `tf.Tensor`: vous pouvez réaliser les mêmes opérations, il s'accorde parfaitement avec NumPy et il est aussi exigeant avec les types. En revanche, il peut également être modifié en place à l'aide de la méthode `assign()` (ou `assign_add()` ou `assign_sub()`, qui incrémente ou décrémente la variable de la valeur indiquée). Vous pouvez également modifier des cellules, ou des tranches, individuelles avec la méthode `assign()`

d'une cellule ou d'une tranche (l'affectation directe d'un élément ne fonctionne pas), ou en utilisant les méthodes `scatter_update()` et `scatter_nd_update()`:

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])
# => [[100., 42., 0.], [8., 10., 200.]]
```



En pratique, la création manuelle de variables sera plutôt rare, car Keras fournit une méthode `add_weight()` qui s'en charge. Par ailleurs, puisque les paramètres d'un modèle seront généralement actualisés directement par les optimiseurs, vous aurez rarement besoin d'actualiser manuellement des variables.

4.2.5 Autres structures de données

TensorFlow prend en charge plusieurs autres structures de données, notamment les suivantes (pour de plus amples détails, consultez la section « Tensors and Operations » dans le notebook¹⁰⁴ ou l'annexe D):

- *Tenseurs creux (`tf.SparseTensor`)*

Une représentation efficace de tenseurs qui contiennent principalement des zéros. Le package `tf.sparse` fournit des opérations destinées aux tenseurs creux (*sparse tensor*).

- *Tableaux de tenseurs (`tf.TensorArray`)*

Il s'agit de listes de tenseurs. Ces tableaux ont une taille fixée par défaut, mais ils peuvent éventuellement être rendus dynamiques. Tous les tenseurs qu'ils contiennent doivent avoir la même forme et le même type de données.

- *Tenseurs irréguliers (`tf.RaggedTensor`)*

Ils représentent des listes statiques de listes de tenseurs, dans lesquelles chaque tenseur a la même forme et le même type de données. Le `tf.ragged` fournit plusieurs opérations pour des opérations avec les tenseurs irréguliers (*ragged tensor*).

- *Tenseurs chaînes de caractères*

Ces tenseurs réguliers sont de type `tf.string`. Ils représentent non pas des chaînes de caractères Unicode mais des chaînes de caractères d'octets. Par conséquent, si nous créons un tel tenseur à partir d'une chaîne de caractères Unicode (par exemple, une chaîne Python 3 normale comme "café"), elle sera encodée automatiquement au format UTF-8 (par exemple, `b"caf\xc3\xa9"`). Nous pouvons également représenter des chaînes de caractères Unicode en utilisant des tenseurs de type `tf.int32`, où chaque élément représente un point de code Unicode (par exemple, `[99, 97, 102, 233]`). Le package `tf.strings` (avec un `s`) dispose d'opérations pour les deux types de chaînes

104. Voir « `12_custom_models_and_training_with_tensorflow.ipynb` » sur <https://github.com/ageron/handson-ml2>.

de caractères, et pour les conversions de l'une à l'autre. Il est important de noter qu'un `tf.string` est atomique, c'est-à-dire que sa longueur n'apparaît pas dans la forme du tenseur. Après qu'il a été converti en un tenseur Unicode (autrement dit, un tenseur de type `tf.int32` contenant des points de code), la longueur est présente dans la forme.

- **Ensembles**

Ils sont représentés sous forme de tenseurs normaux (ou tenseurs creux). Par exemple, `tf.constant([[1, 2], [3, 4]])` représente les deux ensembles {1, 2} et {3, 4}. Plus généralement, chaque ensemble est représenté par un vecteur dans le dernier axe du tenseur. Les ensembles se manipulent à l'aide des opérations provenant du package `tf.sets`.

- **Files d'attente**

Les files d'attente stockent des tenseurs au cours de plusieurs étapes. TensorFlow propose différentes sortes de files d'attente: FIFO (*First In, First Out*) simples (`FIFOQueue`), files permettant de donner la priorité à certains éléments (`PriorityQueue`), de mélanger leurs éléments (`RandomShuffleQueue`), et de grouper des éléments de formes différentes par remplissage (`PaddingFIFOQueue`). Toutes ces classes sont fournies par le package `tf.queue`.

Armés des tenseurs, des opérations, des variables et des différentes structures de données, vous pouvez à présent personnaliser vos modèles et entraîner des algorithmes !

4.3 PERSONNALISER DES MODÈLES ET ENTRAÎNER DES ALGORITHMES

Nous allons commencer par créer une fonction de perte personnalisée, car il s'agit d'un cas d'utilisation simple et fréquent.

4.3.1 Fonctions de perte personnalisées

Supposez que vous souhaitez entraîner un modèle de régression, mais que votre jeu d'entraînement soit légèrement bruyant. Bien entendu, vous commencez par tenter un nettoyage du jeu de données en supprimant ou en corrigéant les cas particuliers, mais cela se révèle insuffisant ; le jeu de donné reste bruyant. Quelle fonction de perte devez-vous employer ? L'erreur quadratique moyenne risque de pénaliser de façon trop importante les grandes erreurs et de rendre votre modèle imprécis. L'erreur absolue moyenne ne va pas pénaliser les cas particuliers, mais la convergence de l'entraînement va probablement prendre du temps et le modèle entraîné risque d'être peu précis. C'est peut-être le bon moment de vous tourner vers la perte de Huber (introduite au chapitre 2) à la place de la bonne vieille MSE. La perte de Huber ne fait pas partie de l'API officielle de Keras, mais elle est disponible dans `tf.keras` (il suffit d'utiliser une instance de la classe `keras.losses.Huber`). Supposons toutefois qu'elle ne s'y trouve pas : son implémentation est simple comme bonjour ! Vous créez

une fonction qui prend en arguments les étiquettes et les prédictions, et vous utilisez des opérations TensorFlow pour calculer la perte de chaque instance :

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```



Pour de meilleures performances, vous devez utiliser une implémentation vectorisée, comme c'est le cas dans cet exemple. Par ailleurs, pour bénéficier des fonctionnalités des graphes de TensorFlow, il faut employer uniquement des opérations TensorFlow.

Il est également préférable de retourner un tenseur qui contient une perte par instance, plutôt que la perte moyenne. De cette manière, Keras est capable d'appliquer des poids de classe ou des poids d'échantillon en fonction de la demande (voir le chapitre 2).

Vous pouvez à présent utiliser cette perte pour la compilation du modèle Keras, puis entraîner celui-ci :

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

Et voilà ! Pendant l'entraînement, Keras appellera la fonction `huber_fn()` pour chaque lot de façon à calculer la perte et utilisera celle-ci pour procéder à la descente de gradient. Il conservera également une trace de la perte totale depuis le début de l'époque et affichera la perte moyenne.

Mais que devient cette perte personnalisée lorsque vous enregistrez le modèle ?

4.3.2 Enregistrer et charger des modèles contenant des composants personnalisés

L'enregistrement d'un modèle qui contient une fonction de perte personnalisée ne pose pas de problème, car Keras enregistre le nom de la fonction. Lors du chargement du modèle, vous devez fournir un dictionnaire qui associe le nom de la fonction à la fonction réelle. Plus généralement, lorsque vous chargez un modèle qui contient des objets personnalisés, vous devez établir le lien entre les noms et les objets :

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

Dans l'implémentation actuelle, toute erreur entre -1 et 1 est considérée comme « petite ». Mais, comment mettre en place un seuil différent ? Une solution consiste à créer une fonction qui configure une fonction de perte :

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn
```

```

        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="adam")

```

Malheureusement, lorsque vous enregistrez le modèle, le seuil `threshold` n'est pas sauvegardé. Autrement dit, vous devrez fixer la valeur de `threshold` au moment du chargement du modèle (notez que le nom à utiliser est "`huber_fn`", c'est-à-dire le nom de la fonction que vous avez donnée à Keras, non celui de la fonction qui l'a créée) :

```

model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})

```

Pour résoudre ce problème, vous pouvez créer une sous-classe de `keras.losses.Loss`, puis implémenter sa méthode `get_config()` :

```

class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}

```



Pour le moment, l'API de Keras précise uniquement comment utiliser l'héritage pour la définition de couches, de modèles, de rappels et de régulariseurs. Si vous souhaitez construire d'autres éléments en utilisant des sous-classes, par exemple des pertes, des indicateurs, des initialiseurs ou des contraintes, ils risquent d'être incompatibles avec d'autres implémentations de Keras. L'API sera très certainement revue de manière à spécifier l'héritage également pour l'ensemble de ces composants.

Examinons ce code :

- Le constructeur accepte `**kwargs` et les passe au constructeur parent, qui s'occupe des hyperparamètres standard: le nom de la perte (`name`) et l'algorithme de réduction (`reduction`) à utiliser pour agréger les pertes d'instance individuelle. Par défaut, il s'agit de "`sum_over_batch_size`". Autrement dit, la perte sera la somme des pertes d'instance, pondérée par les poids d'échantillonnage, le cas échéant, et divisée par la taille du lot (non par la somme des poids; ce n'est donc pas la moyenne pondérée)¹⁰⁵. "`sum`" et "`none`" sont d'autres valeurs possibles.

¹⁰⁵. L'utilisation d'une moyenne pondérée n'est pas une bonne idée. En effet, deux instances de même poids mais provenant de lots différents auraient alors un impact différent sur l'entraînement, en fonction du poids total de chaque lot.

- La méthode `call()` prend les étiquettes et des prédictions, calcule toutes les pertes d'instance et les retourne.
- La méthode `get_config()` retourne un dictionnaire qui associe chaque nom d'hyperparamètre à sa valeur. Elle commence par invoquer la méthode `get_config()` de la classe parent, puis ajoute les nouveaux hyperparamètres à ce dictionnaire (la syntaxe pratique `{**x}` a été ajoutée dans Python 3.5).

Vous pouvez utiliser une instance de cette classe pendant la compilation du modèle :

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

Lorsque vous enregistrez le modèle, le seuil est également sauvegardé. Lorsque vous le chargez, vous devez simplement lier le nom de la classe à la classe elle-même :

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

À l'enregistrement d'un modèle, Keras appelle la méthode `get_config()` de l'instance de la perte et sauvegarde la configuration au format JSON dans le fichier HDF5. Au chargement du modèle, il invoque la méthode de classe `from_config()` sur la classe `HuberLoss` : elle est implémentée par la classe de base (`Loss`) et crée une instance de la classe, en passant `**config` au constructeur.

Voilà tout pour les pertes ! On ne peut pas dire que c'était trop difficile. Il en va de même pour les fonctions d'activation, les initialiseurs, les régulariseurs et les contraintes personnalisés. Voyons cela.

4.3.3 Fonctions d'activation, initialiseurs, régulariseurs et contraintes personnalisés

La plupart des fonctionnalités de Keras, comme les pertes, les régulariseurs, les contraintes, les initialiseurs, les indicateurs, les fonctions d'activation, les couches et même les modèles complets, peuvent être personnalisés en procédant de façon très similaire.

En général, il suffit d'écrire une simple fonction disposant des entrées et des sorties appropriées. Voici des exemples de personnalisation d'une fonction d'activation (équivalente à `keras.activations.softplus()` ou `tf.nn.softplus()`), d'un initialiseur de Glorot (équivalent à `keras.initializers.glorot_normal()`), d'un régulariseur ℓ_1 (équivalent à `keras.regularizers.l1(0.01)`) et d'une contrainte qui s'assure que tous les poids sont positifs (équivalente à `keras.constraints.nonneg()` ou `tf.nn.relu()`) :

```
def my_softplus(z): # La valeur de retour est simplement tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
```

```
def my_positive_weights(weights): # La valeur de retour est simplement
    # tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

Vous le constatez, les arguments dépendent du type de la fonction personnalisée. Ces fonctions personnalisées s'utilisent comme n'importe quelle autre fonction, par exemple :

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

La fonction d'activation sera appliquée à la sortie de cette couche Dense et son résultat sera transmis à la couche suivante. Les poids de la couche seront initialisés à l'aide de la valeur renvoyée par l'initialiseur. À chaque étape de l'entraînement, les poids seront passés à la fonction de régularisation afin de calculer la perte de régularisation, qui sera ajoutée à la perte principale pour obtenir la perte finale utilisée pour l'entraînement. Enfin, la fonction de contrainte sera appelée après chaque étape d'entraînement et les poids de la couche seront remplacés par les poids contraints.

Si une fonction dispose d'hyperparamètres devant être enregistrés avec le modèle, vous créerez une sous-classe de la classe appropriée, comme keras.regularizers.Regularizer, keras.constraints.Constraint, keras.initializers.Initializer ou keras.layers.Layer (pour n'importe quelle couche, y compris les fonctions d'activation). De façon comparable à la perte personnalisée, voici une classe simple pour la régularisation ℓ_1 qui sauvegarde son hyperparamètre factor (cette fois-ci, nous n'avons pas besoin d'appeler le constructeur parent ni la méthode get_config(), car la classe parent ne les définit pas) :

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

Vous devez implémenter la méthode call() pour les pertes, les couches (y compris les fonctions d'activation) et les modèles, ou la méthode __call__() pour les régulariseurs, les initialiseurs et les contraintes. Le cas des indicateurs est légèrement différent, comme nous allons le voir.

4.3.4 Indicateurs personnalisés

Conceptuellement, les pertes et les indicateurs ne sont pas la même chose. Les pertes (par exemple, l'entropie croisée) sont utilisées par la descente de gradient pour entraîner un modèle. Elles doivent donc être différentiables et leurs gradients ne doivent pas être égaux à 0 en tout point. Par ailleurs, rien ne les oblige à être facilement interprétables par des humains. À l'opposé, les indicateurs (par exemple, la précision) servent à évaluer un modèle. Ils doivent être plus facilement interprétables, n'ont pas besoin d'être différentiables et peuvent avoir des gradients égaux à 0 partout.

Cela dit, dans la plupart des cas, définir une fonction pour un indicateur personnalisé revient au même que définir une fonction de perte personnalisée. En réalité, nous pouvons même utiliser la fonction de perte de Huber créée précédemment comme indicateur¹⁰⁶; cela fonctionnerait parfaitement et la persistance se ferait de la même manière, dans ce cas en enregistrant uniquement le nom de la fonction, "huber_fn":

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Pendant l'entraînement, Keras calculera cet indicateur pour chaque lot et conservera une trace de sa moyenne depuis le début de l'époque. La plupart du temps, c'est précisément ce que nous souhaitons. Mais pas toujours ! Prenons, par exemple, la précision d'un classificateur binaire. La précision correspond au nombre de vrais positifs divisé par le nombre de prédictions positives (y compris les vrais et les faux positifs)¹⁰⁷. Supposons que le modèle ait effectué cinq prédictions positives dans le premier lot, quatre d'entre elles étant correctes : la précision est de 80 %. Supposons que le modèle ait ensuite effectué trois prédictions positives dans le second lot, mais que toutes étaient incorrectes : la précision est alors de 0 % sur le second lot. Si nous calculons simplement la moyenne des deux précisions, nous obtenons 40 %. Cependant, il ne s'agit pas de la précision du modèle sur ces deux lots ! En réalité, il y a eu un total de quatre vrais positifs ($4 + 0$) sur huit prédictions positives ($5 + 3$). La précision globale est donc non pas de 40 %, mais de 50 %. Nous avons besoin d'un objet capable de conserver une trace du nombre de vrais positifs et du nombre de faux positifs, et de calculer sur demande leur rapport. C'est précisément ce que réalise la classe `keras.metrics.Precision` :

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

Dans cet exemple, nous créons un objet `Precision`, puis nous l'utilisons comme une fonction en lui passant les étiquettes et les prédictions du premier lot puis du second (nous pourrions également passer des poids d'échantillonnage). Nous utilisons le même nombre de vrais et de faux positifs que dans l'exemple décrit. Après le premier lot, l'objet retourne une précision de 80 %, et de 50 % après le second (elle correspond non pas à la précision du second lot mais à la précision globale). Il s'agit d'un *indicateur en continu (streaming metric)*, ou *indicateur à états (stateful metric)*, qui est actualisé progressivement, lot après lot.

Nous pouvons appeler la méthode `result()` à n'importe quel moment de façon à obtenir la valeur courante de l'indicateur. Nous pouvons également examiner ses variables (suivre le nombre de vrais et de faux positifs) au travers de l'attribut

106. Cependant, la perte de Huber est rarement utilisée comme indicateur (la préférence va à la MAE ou à la MSE).

107. Voir le chapitre 3 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

variables. Et nous pouvons les réinitialiser en invoquant la méthode `reset_states()`:

```
>>> precision.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> precision.reset_states() # Les deux variables sont remises à 0.0
```

Si nous avons besoin d'un tel indicateur en continu, il suffit de créer une sous-classe de `keras.metrics.Metric`. Voici un exemple simple, qui suit la perte totale de Huber et le nombre d'instances rencontrées. Sur demande du résultat, elle retourne le rapport, qui correspond à la perte moyenne de Huber:

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # Traiter les arguments de base (comme dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Examinons en détail ce code¹⁰⁸:

- Le constructeur se sert de la méthode `add_weight()` pour créer les variables qui permettront d'assurer le suivi de l'état de l'indicateur sur plusieurs lots – dans ce cas, la somme des pertes de Huber (`total`) et le nombre d'instances traitées jusqu'à présent (`count`). Nous aurions également pu créer ces variables manuellement. Keras suit tout `tf.Variable` défini comme un attribut (et plus généralement, tout objet « traçable », comme des couches ou des modèles).
- La méthode `update_state()` est invoquée lorsque nous instancions cette classe en tant que fonction (comme nous l'avons fait avec l'objet `Precision`). Elle actualise les variables à partir des étiquettes et des prédictions d'un lot (et des poids d'échantillonnage, mais, dans ce cas, nous les ignorons).
- La méthode `result()` calcule et retourne le résultat final: l'indicateur de Huber moyen sur toutes les instances. Lorsque l'indicateur est employé comme

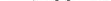
108. Cette classe n'est là qu'à titre d'illustration. Une meilleure implémentation plus simple serait de créer une sous-classe de `keras.metrics.Mean`; voir l'exemple donné dans la section « Streaming metrics » du notebook (voir « `12_custom_models_and_training_with_tensorflow.ipynb` » sur <https://github.com/ageron/handson-ml2>).

une fonction, la méthode `update_state()` est appelée en premier, puis c'est au tour de la méthode `result()`, et la sortie est retournée.

- Nous implémentons également la méthode `get_config()` pour assurer l'enregistrement de `threshold` avec le modèle.
- L'implémentation par défaut de la méthode `reset_states()` réinitialise toutes les variables à `0.0` (mais elle peut être redéfinie si nécessaire).



Keras s'occupera sans difficulté de la persistance des variables ; aucune action n'est requise.



Lorsque nous définissons un indicateur à l'aide d'une simple fonction, Keras l'appelle automatiquement pour chaque lot et conserve la trace de la moyenne sur chaque époque, comme nous l'avons fait manuellement. Le seul avantage de notre classe `HuberMetric` réside donc dans la sauvegarde de `threshold`. Mais, évidemment, certains indicateurs, comme la précision, ne peuvent pas être obtenus par une simple moyenne sur l'ensemble des lots. Dans de tels cas, il n'y a pas d'autre option que d'implémenter un indicateur en continu.

Maintenant que nous savons construire un indicateur en continu, la création d'une couche personnalisée sera une promenade de santé !

4.3.5 Couches personnalisées

Il pourrait arriver que vous ayez à construire une architecture dont une couche se révèle plutôt exotique et pour laquelle TensorFlow ne propose aucune implémentation par défaut. Dans ce cas, vous devez créer une couche personnalisée. Vous pourriez également avoir à construire une architecture excessivement répétitive, qui contient des blocs identiques de couches répétés à plusieurs reprises ; il serait alors pratique de traiter chaque bloc de couches comme une seule couche. Par exemple, si le modèle est une suite de couches A, B, C, A, B, C, A, B, C, vous pourriez souhaiter définir une couche D personnalisée qui contient les couches A, B, C, de sorte que votre modèle serait simplement D, D, D. Voyons donc comment construire des couches personnalisées.

Tout d'abord, certaines couches n'ont pas de poids, comme `keras.layers.Flatten` ou `keras.layers.ReLU`. Si vous voulez créer une couche personnalisée dépourvue de poids, l'option la plus simple consiste à écrire une fonction et à l'intégrer dans une couche `keras.layers.Lambda`. Par exemple, la couche suivante applique la fonction exponentielle à ses entrées :

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

Cette couche personnalisée peut ensuite être utilisée comme n'importe quelle autre couche, que ce soit avec l'API Sequential, Functional ou Subclassing. Vous pouvez également vous en servir comme fonction d'activation (ou vous pouvez utiliser `activation=tf.exp`, `activation=keras.activations.exponential` ou `activation="exponential"`). L'exponentielle

est parfois employée dans la couche de sortie d'un modèle de régression lorsque les échelles des valeurs à prédire sont très différentes (par exemple, 0,001, 10, 1 000).

Vous l'avez probablement deviné, pour construire une couche personnalisée avec état (c'est-à-dire une couche qui possède des poids), vous devez créer une sous-classe de `keras.layers.Layer`. Par exemple, la classe suivante implémente une version simplifiée de la couche `Dense`:

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # Doit se trouver à la fin

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

Étudions ce code:

- Le constructeur prend tous les hyperparamètres en arguments (dans cet exemple, `units` et `activation`) et, plus important, il prend également un argument `**kwargs`. Il invoque le constructeur parent, en lui passant les `kwargs`; cela permet de prendre en charge les arguments standard, comme `input_shape`, `trainable` et `name`. Il enregistre ensuite les hyperparamètres sous forme d'attributs, en convertissant l'argument `activation` en une fonction d'activation appropriée grâce à la fonction `keras.activations.get()` (elle accepte des fonctions, des chaînes de caractères standard comme "`relu`" ou "`selu`", ou juste `None`)¹⁰⁹.
- La méthode `build()` a pour rôle de créer les variables de la couche en invoquant la méthode `add_weight()` pour chaque poids. `build()` est appelée lors de la première utilisation de la couche. À ce stade, Keras connaît

109. Cette fonction est propre à `tf.keras`. Vous pouvez également utiliser à la place `keras.layers.Activation`.

la forme des entrées de la couche et la passera à la méthode `build()`¹¹⁰. Elle est souvent indispensable à la création de certains poids. Par exemple, nous devons connaître le nombre de neurones de la couche précédente de façon à créer la matrice des poids des connexions (autrement dit, le "kernel") : il correspond à la taille de la dernière dimension des entrées. À la fin de la méthode `build()` (et uniquement à la fin), nous devons appeler la méthode `build()` du parent pour indiquer à Keras que la couche est construite (elle fixe simplement `self.built` à True).

- La méthode `call()` réalise les opérations souhaitées. Dans ce cas, nous multiplions la matrice des entrées `X` et le noyau de la couche, nous ajoutons le vecteur de termes constants et nous appliquons la fonction d'activation au résultat. Nous obtenons alors la sortie de la couche.
- La méthode `compute_output_shape()` retourne simplement la forme des sorties de cette couche. Dans cet exemple, elle est identique à la forme des entrées, excepté que la dernière dimension est remplacée par le nombre de neurones de la couche. Dans `tf.keras`, les formes sont des instances de la classe `tf.TensorShape`, que nous pouvons convertir en listes Python avec `as_list()`.
- La méthode `get_config()` est identique à celle des classes personnalisées précédentes. Mais nous enregistrons l'intégralité de la configuration de la fonction d'activation en invoquant `keras.activations.serialize()`.

Nous pouvons à présent utiliser une couche `MyDense` comme n'importe quelle autre couche !



En général, vous pouvez passer outre la méthode `compute_output_shape()`, car `tf.keras` détermine automatiquement la forme de la sortie, excepté lorsque la couche est dynamique (nous y reviendrons plus loin). Dans d'autres implémentations de Keras, soit cette méthode est requise, soit son implémentation par défaut suppose que la forme de la sortie est identique à celle de l'entrée.

Pour créer une couche ayant de multiples entrées (par exemple, `Concatenate`), l'argument de la méthode `call()` doit être un tuple qui contient toutes les entrées et, de façon comparable, l'argument de la méthode `compute_output_shape()` doit être un tuple qui contient la forme du lot de chaque entrée. Pour créer une couche avec de multiples sorties, la méthode `call()` doit retourner la liste des sorties, et `compute_output_shape()` doit retourner la liste des formes des sorties du lot (une par sortie). Par exemple, la couche suivante possède deux entrées et retourne trois sorties :

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]
```

¹¹⁰ L'API de Keras nomme cet argument `input_shape`, mais, puisqu'il comprend également la dimension du lot, je préfère l'appeler `batch_input_shape`. Il en va de même pour `compute_output_shape()`.

```
def compute_output_shape(self, batch_input_shape):
    b1, b2 = batch_input_shape
    return [b1, b1, b1] # Devrait probablement gérer les règles
                      # de diffusion
```

Cette couche peut à présent être utilisée comme n'importe quelle autre couche, mais, évidemment, uniquement avec les API Functional et Subclassing. Elle est incompatible avec l'API Sequential, qui n'accepte que des couches ayant une entrée et une sortie.

Si la couche doit afficher des comportements différents au cours de l'entraînement et des tests (par exemple, si elle utilise les couches Dropout ou BatchNormalization), nous devons ajouter un argument `training` à la méthode `call()` et nous en servir pour décider du comportement approprié. Par exemple, créons une couche qui ajoute un bruit gaussien pendant l'entraînement (pour la régularisation), mais rien pendant les tests (Keras dispose d'une couche ayant ce comportement, `keras.layers.GaussianNoise`):

```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

Avec tous ces éléments, vous pouvez construire n'importe quelle couche personnalisée ! Voyons à présent comment créer des modèles personnalisés.

4.3.6 Modèles personnalisés

Nous avons déjà examiné la création de classes d'un modèle personnalisé au chapitre 2 dans le cadre de la présentation de l'API Subclassing¹¹¹. La procédure est simple : créer une sous-classe de `keras.Model`, créer des couches et des variables dans le constructeur, et implémenter les actions du modèle dans la méthode `call()`. Supposons que nous souhaitions construire le modèle représenté à la figure 4.3.

¹¹¹. En général, « API Subclassing » fait référence uniquement à la création de modèles personnalisés en utilisant l'héritage. Mais, comme nous l'avons vu dans ce chapitre, bien d'autres structures peuvent être créées de cette manière.

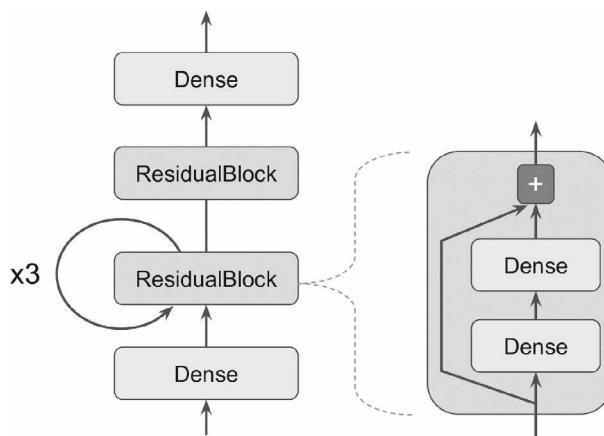


Figure 4.3 – Exemple de modèle personnalisé: un modèle quelconque avec une couche personnalisée ResidualBlock qui contient une connexion de saut

Les entrées passent par une première couche dense, puis par un *bloc résiduel* constitué de deux couches denses et d'une opération d'addition (nous le verrons au chapitre 6, un bloc résiduel additionne ses entrées à ses sorties), puis de nouveau par ce bloc résiduel à trois reprises, puis par un second bloc résiduel, et le résultat final traverse une couche de sortie dense. Notez que ce modèle n'a pas beaucoup de sens ; il s'agit simplement d'un exemple qui nous permet d'illustrer la facilité de construction de n'importe quel modèle, même ceux contenant des boucles et des connexions de saut. Pour l'implémenter, il est préférable de commencer par créer une couche ResidualBlock, car nous allons créer deux blocs identiques (et nous pourrions souhaiter réutiliser un tel bloc dans un autre modèle) :

```

class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         kernel_initializer="he_normal")
                      for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z

```

Cette couche est un peu particulière, car elle en contient deux autres. Cette configuration est gérée de façon transparente par Keras : il détecte automatiquement que l'attribut `hidden` contient des objets traçables (dans ce cas, des couches) et leurs variables sont ajoutées automatiquement à la liste des variables de cette couche. Le reste du code de la classe se comprend par lui-même. Ensuite, nous utilisons l'API Subclassing pour définir le modèle :

```

class ResidualRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)

```

```

    self.hidden1 = keras.layers.Dense(30, activation="elu",
                                     kernel_initializer="he_normal")
    self.block1 = ResidualBlock(2, 30)
    self.block2 = ResidualBlock(2, 30)
    self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)

```

Nous créons les couches dans le constructeur et les utilisons dans la méthode `call()`. Ce modèle peut être employé comme n'importe quel autre modèle : compilation, ajustement, évaluation et exécution pour effectuer des prédictions. Pour que nous puissions l'enregistrer avec la méthode `save()` et le charger avec la fonction `keras.models.load_model()`, nous devons implémenter la méthode `get_config()` (comme nous l'avons fait précédemment) dans les classes `ResidualBlock` et `ResidualRegressor`. Nous pouvons également enregistrer et charger les poids à l'aide de méthodes `save_weights()` et `load_weights()`.

Puisque la classe `Model` dérive de la classe `Layer`, les modèles peuvent être définis et employés exactement comme des couches. Toutefois, un modèle dispose de fonctionnalités supplémentaires, notamment ses méthodes `compile()`, `fit()`, `evaluate()` et `predict()` (ainsi que quelques variantes), mais aussi la méthode `get_layers()` (qui renvoie des couches du modèle d'après leur nom ou leur indice) et la méthode `save()` (sans oublier la prise en charge de `keras.models.load_model()` et de `keras.models.clone_model()`).



Si les modèles offrent plus de fonctionnalités que les couches, pourquoi ne pas simplement définir chaque couche comme un modèle ? Techniquement c'est possible, mais il est en général plus propre de faire la distinction entre les composants internes du modèle (c'est-à-dire les couches ou les blocs de couches réutilisables) et le modèle lui-même (c'est-à-dire l'objet qui sera entraîné). Dans le premier cas, il faut créer une sous-classe de `Layer`, et de `Model` dans le second.

Avec tous ces outils, vous pouvez construire naturellement et synthétiquement presque n'importe quel modèle décrit dans un article, en utilisant les API Sequential, Functional et Subclassing, voire même en les mélangeant. « Presque » n'importe quel modèle ? Oui, car il reste encore quelques points à examiner : premièrement la définition de pertes ou d'indicateurs en fonction des éléments internes du modèle, et deuxièmement la construction d'une boucle d'entraînement personnalisée.

4.3.7 Pertes et indicateurs fondés sur les éléments internes du modèle

Les pertes et les indicateurs personnalisés que nous avons définis précédemment se fondaient sur les étiquettes et les prédictions (et, en option, les poids d'échantillonage). Nous pourrions également avoir besoin de définir des pertes en fonction

d'autres éléments du modèle, comme les poids ou les activations de ses couches cachées. Cela sera notamment utile pour la régularisation ou pour la supervision des aspects internes du modèle.

Pour définir une perte personnalisée qui dépend des éléments internes du modèle, nous la calculons en fonction des parties du modèle concernées, puis nous passons le résultat à la méthode `add_loss()`. Par exemple, construisons un modèle personnalisé pour un PMC de régression constitué d'une pile de cinq couches cachées et d'une couche de sortie. Le modèle disposera également d'une sortie auxiliaire au-dessus de la couche cachée supérieure. La perte associée à cette sortie supplémentaire sera appelée *perte de reconstruction* (voir le chapitre 9) : il s'agit de l'écart quadratique moyen entre la reconstruction et les entrées. En ajoutant cette perte de reconstruction à la perte principale, nous encouragerons le modèle à conserver autant d'informations que possible au travers des couches cachées – même celles qui ne sont pas directement utiles à la tâche de régression. En pratique, cette perte améliore parfois la généralisation (c'est une perte de régularisation). Voici le code de ce modèle personnalisé doté d'une perte de reconstruction personnalisée :

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                         kernel_initializer="lecun_normal")
                      for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

Examinons-le :

- Le constructeur crée le RNP constitué de cinq couches cachées denses et d'une couche de sortie dense.
- La méthode `build()` crée une couche dense supplémentaire qui servira à reconstruire les entrées du modèle. Sa création doit se faire à cet endroit car son nombre d'unités doit être égal au nombre d'entrées et cette valeur est inconnue jusqu'à l'invocation de la méthode `build()`.
- La méthode `call()` traite les entrées au travers des cinq couches cachées, puis passe le résultat à la couche de reconstruction, qui produit la reconstruction.
- Ensuite, la méthode `call()` calcule la perte de reconstruction (l'écart quadratique moyen entre la reconstruction et les entrées) et l'ajoute à la liste

des pertes du modèle en invoquant `add_loss()`¹¹². Nous réduisons la perte de reconstruction en la multipliant par 0,05 (un hyperparamètre ajustable) de façon qu'elle n'écrase pas la perte principale.

- Enfin, la méthode `call()` transmet la sortie des couches cachées à la couche de sortie et retourne le résultat.

En procédant de manière comparable, nous pouvons ajouter un indicateur personnalisé dont le calcul se fonde sur les éléments internes du modèle, tant que le résultat est la sortie d'un objet de mesure. Par exemple, nous pouvons créer un objet `keras.metrics.Mean` dans le constructeur, l'invoquer depuis la méthode `call()` en lui passant `recon_loss`, et finalement l'ajouter au modèle à l'aide de la méthode `add_metric()` du modèle. Ainsi, pendant l'entraînement du modèle, Keras affichera la perte moyenne sur chaque époque (la somme de la perte principale et de 0,05 fois la perte de reconstruction) ainsi que l'erreur de reconstruction moyenne sur chaque époque. Toutes deux doivent diminuer au cours de l'entraînement :

```
Epoch 1/5
11610/11610 [=====] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [=====] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

Dans plus de 99 % des cas, tout ce que nous venons de présenter suffira à implémenter le modèle dont vous avez besoin, même pour des architectures, des pertes et des indicateurs complexes. Toutefois, dans quelques rares cas, vous pourriez avoir besoin de personnaliser la boucle d'entraînement elle-même. Avant d'en expliquer la procédure, nous devons décrire le fonctionnement du calcul automatique des gradients dans TensorFlow.

4.3.8 Calculer des gradients en utilisant la différentiation automatique

La petite fonction suivante va nous servir de support à la description du calcul automatique des gradients avec la différentiation automatique (voir le chapitre 2 et l'annexe B) :

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

Si vous avez quelques connaissances en algèbre, vous pouvez déterminer que la dérivée partielle de cette fonction par rapport à w_1 est $6 \cdot w_1 + 2 \cdot w_2$. Vous pouvez également trouver sa dérivée partielle par rapport à w_2 : $2 \cdot w_1$. Par exemple, au point $(w_1, w_2) = (5, 3)$, ces dérivées partielles sont respectivement égales à 36 et à 10 ; le vecteur de gradient à ce point est donc (36, 10). Toutefois, s'il s'agissait d'un réseau de neurones, la fonction serait beaucoup plus complexe, en général avec des dizaines de milliers de paramètres. Dans ce cas, déterminer manuellement les dérivées partielles de façon analytique serait une tâche quasi impossible. Une solution serait

112. Nous pouvons également appeler `add_loss()` sur n'importe quelle couche interne au modèle, car celui-ci collecte de façon récursive les pertes depuis toutes ses couches.

de calculer une approximation de chaque dérivée partielle en mesurant les changements en sortie de la fonction lorsque le paramètre correspondant est ajusté :

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.00000003174137
```

Cela semble plutôt bien ! C'est le cas, et la solution est facile à implémenter. Cela dit, il ne s'agit que d'une approximation et, plus important encore, nous devons appeler `f()` au moins une fois par paramètre (non pas deux, car nous pouvons calculer `f(w1, w2)` juste une fois). En raison de cette obligation, cette approche n'est pas envisageable avec les grands réseaux de neurones. À la place, nous devons employer la différentiation automatique. Grâce à TensorFlow, c'est assez simple :

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

Nous définissons tout d'abord les variables `w1` et `w2`, puis nous créons un contexte `tf.GradientTape` qui enregistrera automatiquement chaque opération impliquant une variable, et nous demandons à cet enregistrement de calculer les gradients du résultat `z` en rapport avec les deux variables `[w1, w2]`. Examinons les gradients calculés par TensorFlow :

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Parfait ! Non seulement le résultat est précis (la précision est uniquement limitée par les erreurs des calculs en virgule flottante), mais la méthode `gradient()` est également passée une seule fois sur les calculs enregistrés (en ordre inverse), quel que soit le nombre de variables existantes. Elle est donc incroyablement efficace. C'est magique !



Pour économiser la mémoire, le bloc `tf.GradientTape()` ne doit contenir que le strict minimum. Vous pouvez également faire une pause dans l'enregistrement en créant un bloc `with tape.stop_recording()` à l'intérieur du bloc `tf.GradientTape()`.

L'enregistrement est effacé automatiquement juste après l'invocation de sa méthode `gradient()`. Nous recevrons donc une exception si nous tentons d'appeler `gradient()` à deux reprises :

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError !
```

S'il nous faut invoquer `gradient()` plus d'une fois, nous devons rendre l'enregistrement persistant et le supprimer chaque fois que nous n'en avons plus besoin de façon à libérer ses ressources¹¹³:

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, OK à présent !
del tape
```

Par défaut, l'enregistrement ne conserve que les opérations qui impliquent des variables. Si nous essayons de calculer le gradient de `z` par rapport à autre chose qu'une variable, le résultat est `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # Retourne [None, None]
```

Cependant, nous pouvons imposer à l'enregistrement la surveillance de n'importe quel tenseur afin d'enregistrer les opérations qui le concernent. Nous pouvons ensuite calculer des gradients en lien avec ces tenseurs, comme s'ils étaient des variables:

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # Retourne [tensor 36., tensor 10.]
```

Cette possibilité se révélera parfois utile, par exemple pour implémenter une perte de régularisation qui pénalise les activations qui varient beaucoup alors que les entrées varient peu: la perte sera fondée sur le gradient des activations en lien avec les entrées. Puisque les entrées ne sont pas des variables, nous devons indiquer à l'enregistrement de les surveiller.

Le plus souvent, un enregistrement de gradients sert à calculer les gradients d'une seule valeur (en général la perte) par rapport à un ensemble de valeurs (en général les paramètres du modèle). C'est dans ce contexte que la différentiation automatique en mode inverse brille, car elle doit uniquement effectuer une passe en avant et une passe en arrière pour obtenir tous les gradients en une fois. Si nous calculons les gradients d'un vecteur, par exemple un vecteur qui contient plusieurs pertes, TensorFlow va calculer des gradients de la somme du vecteur. Par conséquent, si nous avons besoin d'obtenir des gradients individuels (par exemple, les gradients de chaque perte en lien avec les paramètres du modèle), nous devons appeler la méthode `jacobian()` de l'enregistrement. Elle effectuera une différentiation automatique en mode inverse pour chaque perte du vecteur (par défaut, toutes exécutées en parallèle). Il est même possible de calculer des dérivées partielles de deuxième ordre (les hessiens, c'est-à-dire les dérivées

113. Si l'enregistrement n'est plus accessible, par exemple lorsque la fonction qui l'a utilisé se termine, le ramasse-miettes de Python le supprimera pour nous.

partielles des dérivées partielles), mais, en pratique, ce besoin est rare (un exemple est donné dans la section « Computing Gradients with Autodiff » du notebook¹¹⁴).

Parfois, il faudra arrêter la rétropropagation des gradients dans certaines parties du réseau de neurones. Pour cela, nous devons utiliser la fonction `tf.stop_gradient()`. Au cours de la passe en avant, elle retourne son entrée (comme `tf.identity()`), mais, pendant la rétropropagation, elle ne laisse pas passer les gradients (elle agit comme une constante) :

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # Même résultat que sans stop_gradient()

gradients = tape.gradient(z, [w1, w2]) # => Retourne [tensor 30., None]
```

Enfin, nous pourrions occasionnellement rencontrer des problèmes numériques lors du calcul des gradients. Par exemple, si nous calculons les gradients de la fonction `my_softplus()` pour de grandes entrées, le résultat est NaN :

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

En effet, le calcul des gradients de cette fonction avec la différentiation automatique conduit à certaines difficultés numériques : en raison des erreurs de précision des calculs en virgule flottante, autodiff en arrive à diviser l'infini par l'infini (ce qui retourne NaN). Heureusement, nous sommes capables de déterminer de façon analytique la dérivée de la fonction softplus : $1 / (1 + 1 / \exp(x))$; elle est stable numériquement. Ensuite, nous pouvons indiquer à TensorFlow d'utiliser cette fonction stable pour le calcul des gradients de la fonction `my_softplus()` en la décorant de `@tf.custom_gradient` et en lui faisant retourner à la fois sa sortie normale et la fonction qui calcule les dérivées (notez qu'elle recevra en entrée les gradients qui ont été rétropropagés jusqu'à présent, jusqu'à la fonction softplus ; et, conformément à la règle de la chaîne, nous devons les multiplier avec les gradients de cette fonction) :

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

À présent, le calcul des gradients de la fonction `my_better_softplus()` nous donne le résultat correct, même pour les grandes valeurs d'entrée (toutefois,

114. Voir « 12_custom_models_and_training_with_tensorflow.ipynb » sur <https://github.com/ageron/handson-ml2>.

la sortie principale continue à exploser en raison de l'exponentielle ; une solution consiste à retourner les entrées avec `tf.where()` lorsqu'elles sont grandes.

Félicitations ! Vous savez désormais calculer les gradients de n'importe quelle fonction (à condition qu'elle soit différentiable au point du calcul), bloquer la rétro-propagation au besoin, et écrire vos propres fonctions de gradient ! Cette souplesse va probablement bien au-delà de ce dont vous aurez besoin, même si vous construisez vos propres boucles d'entraînement personnalisées, comme nous allons le voir.

4.3.9 Boucles d'entraînement personnalisées

Dans quelques cas rares, la méthode `fit()` pourrait ne pas être suffisamment souple pour répondre à vos besoins. Par exemple, l'article sur Wide & Deep (<https://homl.info/widedeep>), dont nous avons parlé au chapitre 2, exploite deux optimiseurs différents : l'un pour le chemin large et l'autre pour le chemin profond. Puisque la méthode `fit()` n'utilise qu'un seul optimiseur (celui indiqué lors de la compilation du modèle), vous devez développer votre propre boucle personnalisée pour mettre en œuvre la technique décrite dans l'article.

Vous pourriez également écrire vos propres boucles d'entraînement uniquement pour être sûrs qu'elles procèdent comme vous le souhaitez (si vous doutez de certains détails de la méthode `fit()`) ; on peut se sentir plus en confiance lorsque tout est explicite. Toutefois, l'écriture d'une boucle d'entraînement personnalisée rendra le code plus long, plus sujet aux erreurs et plus difficile à maintenir.



À moins d'avoir réellement besoin d'une souplesse supplémentaire, vous devez opter de préférence pour la méthode `fit()` plutôt que pour l'implémentation de votre propre boucle d'entraînement, en particulier si vous travaillez au sein d'une équipe.

Commençons par construire un modèle simple. Il est inutile de le compiler, car nous allons gérer manuellement la boucle d'entraînement :

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                      kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Créons ensuite une petite fonction qui échantillonne de façon aléatoire un lot d'instances à partir du jeu d'entraînement (au chapitre 5 nous verrons l'API Data, qui offre une bien meilleure alternative) :

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Définissons également une fonction qui affichera l'état de l'entraînement, y compris le nombre d'étapes, le nombre total d'étapes, la perte moyenne depuis le début de l'époque (autrement dit, nous utilisons l'indicateur `Mean` pour le calculer) et d'autres indicateurs :

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}{}".format(m.name, m.result())
                           for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{} / {} - ".format(iteration, total) + metrics,
          end=end)
```

Ce code n'a pas besoin d'explication, sauf si le formatage des chaînes de caractères dans Python vous est inconnu: `{:.4f}` formate un nombre à virgule flottante avec quatre chiffres après la virgule, et la combinaison de `\r` (retour-chariot) et de `end=""` garantit que la barre d'état sera toujours affichée sur la même ligne. Le notebook¹¹⁵ définit la fonction `print_status_bar()` qui comprend une barre de progression, mais vous pouvez utiliser à la place la bibliothèque `tqdm` très pratique.

Revenons au sujet principal! Tout d'abord, nous devons définir les divers paramètres et choisir l'optimiseur, la fonction de perte et les indicateurs (dans cet exemple, uniquement la MAE):

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

Voilà, nous sommes prêts à construire la boucle personnalisée :

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {} / {}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

Ce code réalisant de nombreuses opérations, détaillons-le :

- Nous créons deux boucles imbriquées : l'une pour les époques, l'autre pour les lots à l'intérieur d'une époque.
- Ensuite, nous échantillonons un lot aléatoire à partir du jeu d'entraînement.
- À l'intérieur du bloc `tf.GradientTape()`, nous effectuons une prédiction pour un lot (en utilisant le modèle comme une fonction) et nous calculons

la perte : elle est égale à la perte principale plus les autres pertes (ce modèle comprend une perte de régularisation par couche). Puisque la fonction `mean_squared_error()` retourne une perte par instance, nous calculons la moyenne sur le lot à l'aide de `tf.reduce_mean()` (si nous voulons appliquer différents poids à chaque instance, c'est là qu'il faut le faire). Les pertes de régularisation étant déjà réduites à un seul scalaire chacune, il nous faut simplement les additionner (avec `tf.add_n()`, qui effectue la somme de plusieurs tenseurs de mêmes forme et type de données).

- Puis nous demandons à l'enregistrement de calculer le gradient de la perte en rapport avec chaque variable entraînable (*non toutes les variables !*) et nous les appliquons à l'optimiseur de manière à réaliser l'étape de descente de gradient.
- Nous actualisons la perte moyenne et les indicateurs (sur l'époque en cours) et nous affichons la barre d'état.
- À la fin de chaque époque, nous affichons de nouveau la barre d'état, afin qu'elle soit complète¹¹⁶, et un saut de ligne. Nous réinitialissons également les états de la perte moyenne et des indicateurs.

Si nous fixons l'hyperparamètre `clipnorm` ou `clipvalue` de l'optimiseur, il s'en chargera à notre place. Si nous souhaitons appliquer d'autres transformations au gradient, il suffit de les effectuer avant d'invoquer la méthode `apply_gradients()`.

Si nous ajoutons des contraintes de poids au modèle (par exemple, en précisant `kernel_constraint` ou `bias_constraint` au moment de la création d'une couche), nous devons revoir la boucle d'entraînement afin qu'elle applique ces contraintes juste après `apply_gradients()` :

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

Point important, cette boucle d'entraînement ne gère pas les couches qui se comportent différemment pendant l'entraînement et pendant les tests (par exemple, BatchNormalization ou Dropout). Pour traiter ces situations, nous devons appeler le modèle avec `training=True` et faire en sorte qu'il propage cette information à chaque couche qui en a besoin.

Vous le voyez, pour que tout fonctionne correctement, il faut faire attention à de nombreux aspects et il est facile de se tromper. En revanche, vous obtenez un contrôle total. C'est donc à vous de décider.

Puisque vous savez désormais personnaliser tous les éléments de vos modèles¹¹⁷, voyons comment utiliser la fonctionnalité TensorFlow de génération automatique

116. En vérité, nous n'avons pas traité toutes les instances du jeu d'entraînement, car nous en avons extrait un échantillon de façon aléatoire : certaines ont été traitées plusieurs fois, tandis que d'autres ne l'ont pas été du tout. De manière comparable, si la taille du jeu d'entraînement n'est pas un multiple de la taille du lot, nous manquerons quelques instances. En pratique, ce n'est pas un problème.

117. À l'exception des optimiseurs, mais rares sont les personnes qui les personnalisent ; un exemple est donné dans la section « Custom Optimizers » du notebook (voir « `12_custom_models_and_training_with_tensorflow.ipynb` » sur <https://github.com/ageron/handson-ml2>).

d'un graphe. Elle peut accélérer considérablement le code personnalisé et le rendre portable sur toutes les plateformes reconnues par TensorFlow.

4.4 FONCTIONS ET GRAPHES TENSORFLOW

Dans TensorFlow 1, les graphes étaient incontournables (tout comme l'étaient les complexités qui les accompagnaient), car ils étaient au cœur de l'API de TensorFlow. Dans TensorFlow 2, ils sont toujours présents, sans être autant centraux, et sont beaucoup plus simples à utiliser. Pour l'illustrer, commençons par une fonction triviale qui calcule le cube de son entrée :

```
def cube(x):
    return x ** 3
```

Nous pouvons évidemment appeler cette fonction en lui passant une valeur Python, comme un entier ou réel, ou bien un tenseur :

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

Voyons comment `tf.function()` permet de convertir cette fonction Python en une fonction *TensorFlow* :

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

La fonction TF obtenue peut être employée exactement comme la fonction Python d'origine. Elle retournera le même résultat, mais sous forme de tenseur :

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

`tf.function()` a analysé les calculs réalisés par la fonction `cube()` et a généré un graphe de calcul équivalent ! Vous le constatez, cela ne nous a pas demandé trop d'efforts (nous verrons le fonctionnement plus loin). Une autre solution, plus répandue, consiste à utiliser `tf.function` comme décorateur :

```
@tf.function
def tf_cube(x):
    return x ** 3
```

En cas de besoin, la fonction Python d'origine reste disponible par l'intermédiaire de l'attribut `python_function` de la fonction TF :

```
>>> tf_cube.python_function(2)
8
```

TensorFlow optimise le graphe de calcul, en élaguant les nœuds inutilisés, en simplifiant les expressions (par exemple, $1 + 2$ est remplacé par 3), etc. Lorsque le graphe optimisé est prêt, la fonction TF exécute efficacement les opérations qu'il contient, dans l'ordre approprié (et en parallèle si possible). En conséquence, l'exécution d'une

fonction TF sera souvent beaucoup plus rapide que celle de la fonction Python d'origine, en particulier lorsque des calculs complexes sont impliqués¹¹⁸. De façon générale, vous pouvez simplement considérer que, pour améliorer les performances d'une fonction Python, il suffit de la transformer en une fonction TF. Voilà tout !

Lorsque nous personnalisons une fonction de perte, un indicateur, une couche ou toute autre fonction, et l'utilisons dans un modèle Keras (comme nous l'avons fait tout au long de ce chapitre), Keras convertit automatiquement notre fonction en une fonction TF – nous n'avons pas à utiliser `tf.function()`. La plupart du temps, toute cette magie est donc totalement transparente.



Pour demander à Keras de ne *pas* convertir vos fonctions Python en fonctions TF, indiquez `dynamic=True` lors de la création d'une couche ou d'un modèle personnalisé. Vous pouvez également préciser `run_eagerly=True` lors de l'appel à la méthode `compile()` du modèle.

Par défaut, une fonction TF génère un nouveau graphe pour chaque ensemble unique de formes et de types de données d'entrée, et le met en cache pour les appels suivants. Par exemple, si nous appelons `tf_cube(tf.constant(10))`, un graphe est généré pour des tenseurs de type int32 et de forme `[]`. Si, par la suite, nous appelons `tf_cube(tf.constant(20))`, ce même graphe est réutilisé. En revanche, si nous appelons `tf_cube(tf.constant([10, 20]))`, un nouveau graphe est généré pour des tenseurs de type int32 et de forme `[2]`. C'est de cette manière que les fonctions TF prennent en charge le polymorphisme (c'est-à-dire les arguments de type et de forme variables). Toutefois, cela ne concerne que les arguments qui sont des tenseurs. Si nous passons des valeurs Python numériques à une fonction TF, un nouveau graphe est généré pour chaque valeur distincte. Par exemple, appeler `tf_cube(10)` et `tf_cube(20)` produit deux graphes.



Si vous appelez une fonction TF à de nombreuses reprises avec différentes valeurs Python numériques, de nombreux graphes seront générés. Cela va ralentir le programme et utiliser une grande quantité de mémoire RAM (vous devez détruire la fonction TF pour libérer cette mémoire). Les valeurs Python doivent être réservées aux arguments qui n'auront que quelques valeurs uniques, comme les hyperparamètres, par exemple le nombre de neurones par couche. Cela permet à TensorFlow de mieux optimiser chaque variante du modèle.

4.4.1 AutoGraph et traçage

Comment TensorFlow procède-t-il pour générer des graphes ? Il commence par analyser le code source de la fonction Python de manière à repérer toutes les instructions de contrôle du flux, comme les boucles `for`, les boucles `while` et les instructions `if`, sans oublier les instructions `break`, `continue` et `return`. Cette première étape

¹¹⁸. Dans notre exemple simple, le graphe de calcul est trop petit pour qu'une optimisation quelconque puisse être mise en place. `tf_cube()` s'exécute donc plus beaucoup lentement que `cube()`.

se nomme *AutoGraph*. TensorFlow n'a d'autre choix que d'analyser le code source, car Python n'offre aucune autre manière d'obtenir les instructions de contrôle du flux. S'il propose bien des méthodes magiques comme `__add__()` et `__mul__()` pour capturer les opérateurs `+` et `*`, il ne définit aucune méthode magique comme `__while__()` ou `__if__()`. À la suite de l'analyse du code de la fonction, AutoGraph produit une version actualisée de cette fonction dans laquelle toutes les instructions de contrôle du flux sont remplacées par des opérations TensorFlow appropriées, comme `tf.while_loop()` pour les boucles et `tf.cond()` pour les instructions `if`.

Par exemple, dans le cas illustré à la figure 4.4, AutoGraph analyse le code source de la fonction Python `sum_squares()` et génère la fonction `tf_sum_squares()`. Dans celle-ci, la boucle `for` est remplacée par la définition de la fonction `loop_body()` (qui contient le corps de la boucle `for` d'origine), suivie d'un appel à `for_stmt()`. Cet appel va construire l'opération `tf.while_loop()` appropriée dans le graphe de calcul.

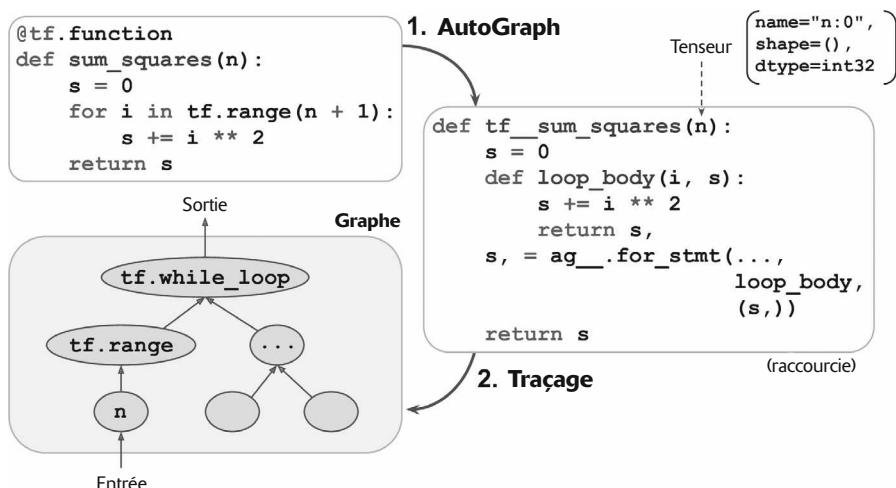


Figure 4.4 – Processus de génération des graphes par TensorFlow avec AutoGraph et le traçage

Ensuite, TensorFlow appelle cette fonction « modernisée », mais, à la place de l'argument, il passe un *tenseur symbolique* – un tenseur sans valeur réelle, uniquement un nom, un type de données et une forme. Par exemple, dans le cas de l'appel `sum_squares(tf.constant(10))`, la fonction `tf_sum_squares()` est appelée avec un tenseur symbolique de type `int32` et de forme `[]`. Elle sera exécutée en *mode graphe* (*graph mode*), dans lequel chaque opération TensorFlow ajoute un noeud au graphe pour se représenter elle-même et son ou ses tenseurs de sortie (*a contrario* du mode normal, appelé *exécution pressée* [*eager execution*] ou *mode pressé* [*eager mode*]). En mode graphe, les opérations TF n'effectuent aucun calcul. Si vous connaissez TensorFlow 1, cela devrait vous être familier, car le mode graphe était actif par défaut.

La figure 4.4 montre la fonction `tf._sum_squares()` appelée avec un tenseur symbolique comme argument (dans ce cas, un tenseur de type `int32` et de forme `[]`) et le graphe final généré pendant le traçage. Les noeuds représentent des opérations, les flèches, des tenseurs (la fonction et le graphe générés sont tous deux simplifiés).



`tf.autograph.to_code(sum_squares.python_function)` affiche le code source de la fonction générée. Le code n'est peut-être pas joli, mais il peut parfois aider au débogage.

4.4.2 Règles d'une fonction TF

La plupart du temps, la conversion d'une fonction Python qui réalise des opérations TensorFlow en une fonction TF est triviale : il suffit de la décorer avec `@tf.function` ou de laisser Keras s'en charger à notre place. Cependant, voici quelques règles qui devront être respectées :

- Si nous faisons appel à une bibliothèque externe, y compris NumPy ou même la bibliothèque standard, cet appel s'exécutera uniquement pendant le traçage ; il ne fera pas partie du graphe. Un graphe TensorFlow ne peut contenir que des constructions TensorFlow (tenseurs, opérations, variables, jeux de données, etc.). Par conséquent, il faut s'assurer d'utiliser `tf.reduce_sum()` à la place de `np.sum()`, `tf.sort()` à la place de la fonction `sorted()` intégrée, et ainsi de suite (sauf si le code doit s'exécuter uniquement pendant le traçage). Cette contrainte a quelques implications supplémentaires :
 - Si nous définissons une fonction TF `f(x)` qui retourne simplement `np.random.rand()`, un nombre aléatoire sera généré uniquement lors du traçage de la fonction. Par conséquent, `f(tf.constant(2.))` et `f(tf.constant(3.))` retourneront la même valeur aléatoire, mais elle sera différente avec `f(tf.constant([2., 3.]))`. Si nous remplaçons `np.random.rand()` par `tf.random.uniform([])`, un nouveau nombre aléatoire est généré à chaque appel, car l'opération fait alors partie du graphe.
 - Si le code non-TensorFlow a des effets secondaires (comme la journalisation d'informations ou l'actualisation d'un compteur Python), nous ne devons pas supposer qu'ils se produiront chaque fois que nous appelons la fonction TF, car les opérations se feront uniquement lorsque la fonction est tracée.
 - Nous pouvons inclure n'importe quel code Python dans une fonction `tf.py_function()`, mais cela va compromettre les performances, car TensorFlow ne saura pas optimiser le graphe pour ce code. Par ailleurs, la portabilité s'en trouvera réduite, car le graphe s'exécutera uniquement sur les plateformes qui disposent de Python (et des bibliothèques adéquates).
- Nous pouvons appeler d'autres fonctions Python ou TF, mais elles doivent respecter les mêmes règles, car TensorFlow capturera leurs opérations dans le graphe de calcul. Toutefois, ces autres fonctions n'ont pas besoin d'être décorées avec `@tf.function`.

- Si la fonction crée une variable TensorFlow (ou tout autre objet TensorFlow ayant un état, comme un jeu de données ou une file d'attente), elle doit le faire au premier appel, et uniquement à ce moment-là, sinon une exception sera lancée. En général, il est préférable de créer les variables en dehors de la fonction TF (par exemple, dans la méthode `build()` d'une couche personnalisée). Pour affecter une nouvelle valeur à la variable, nous devons non pas utiliser l'opérateur `=`, mais invoquer sa méthode `assign()`.
- Le code source de la fonction Python doit être accessible à TensorFlow. S'il est indisponible (par exemple, si nous définissons la fonction dans le shell Python, ce qui ne donne pas accès au code source, ou si nous déployons en production uniquement les fichiers Python *.pyc compilés), le processus de génération du graphe échouera ou aura une fonctionnalité limitée.
- TensorFlow ne capture que les boucles `for` dont l'itération se fait sur un tenseur ou un jeu de données. Nous devons donc employer `for i in tf.range(x)` à la place de `for i in range(x)` ; dans le cas contraire, la boucle ne sera pas capturée dans le graphe et s'exécutera pendant le traçage. (Ce comportement peut être celui attendu si la boucle `for` est là pour construire le graphe, par exemple pour créer chaque couche d'un réseau de neurones.)
- Comme toujours, pour des questions de performances, nous devons donner la priorité à une implémentation vectorisée plutôt qu'à l'utilisation des boucles.

Il est temps de résumer tout ce que nous avons vu ! Dans ce chapitre, nous avons commencé par un bref aperçu de TensorFlow, puis nous avons examiné l'API de bas niveau de TensorFlow, notamment les tenseurs, les opérations, les variables et les structures de données particulières. Nous avons employé ces outils pour personnaliser presque tous les composants de `tf.keras`. Enfin, nous avons expliqué comment les fonctions TF peuvent améliorer les performances, comment les graphes sont générés avec AutoGraph et le traçage, et les règles à respecter lors de l'écriture de fonctions TF (si vous souhaitez entrer un peu plus dans la boîte noire, par exemple pour explorer les graphes générés, vous trouverez les détails techniques dans l'annexe E).

Dans le chapitre suivant, nous verrons comment charger et prétraiter efficacement les données avec TensorFlow.

4.5 EXERCICES

1. Comment décririez-vous TensorFlow en quelques mots ? Quelles sont ses principales fonctionnalités ? Pouvez-vous nommer d'autres bibliothèques de Deep Learning connues ?
2. TensorFlow est-il un remplaçant direct de NumPy ? Quelles sont leurs principales différences ?
3. Les appels `tf.range(10)` et `tf.constant(np.arange(10))` produisent-ils le même résultat ?
4. Hormis les tenseurs classiques, nommez six autres structures de données disponibles dans TensorFlow.

5. Une fonction personnalisée peut être définie en écrivant une fonction ou une sous-classe de `keras.losses.Loss`. Dans quels cas devez-vous employer chaque option ?
6. De façon comparable, un indicateur personnalisé peut être défini dans une fonction ou une sous-classe de `keras.metrics.Metric`. Dans quels cas devez-vous utiliser chaque option ?
7. Quand devez-vous créer une couche personnalisée plutôt qu'un modèle personnalisé ?
8. Donnez quelques cas d'utilisation qui nécessitent l'écriture d'une boucle d'entraînement personnalisée ?
9. Les composants Keras personnalisés peuvent-ils contenir du code Python quelconque ou doivent-ils être convertibles en fonctions TF ?
10. Quelles sont les principales règles à respecter si une fonction doit être convertible en une fonction TF ?
11. Quand devez-vous créer un modèle Keras dynamique ? Comment procédez-vous ? Pourquoi ne pas rendre dynamiques tous les modèles ?
12. Implémentation d'une couche personnalisée qui effectue une *normalisation de couche* (nous étudierons ce type de couches au chapitre 7) :
 - a. La méthode `build()` doit définir deux poids entraînables α et β , tous deux ayant la forme `input_shape[-1:]` et le type de données `tf.float32`. α doit être initialisé avec des 1, et β avec des 0.
 - b. La méthode `call()` doit calculer la moyenne μ et l'écart-type σ de chaque caractéristique d'instance. Pour cela, vous pouvez utiliser `tf.nn.moments(inputs, axes=-1, keepdims=True)`, qui retourne la moyenne μ et la variance σ^2 de toutes les instances (prenez la racine carrée de la variance pour obtenir l'écart-type). Ensuite, la fonction doit calculer et retourner $\alpha \otimes (\mathbf{X} - \mu) / (\sigma + \epsilon) + \beta$, où \otimes représente la multiplication ($*$) terme à terme et ϵ est un terme de lissage (une petite constante qui évite la division par zéro, par exemple 0,001).
 - c. Vérifiez que votre couche personnalisée produit la même (ou vraiment très proche) sortie que la couche `keras.layers.LayerNormalization`.
13. Entraînement d'un modèle sur le jeu de données Fashion MNIST (voir le chapitre 2) en utilisant une boucle d'entraînement personnalisée :
 - a. Affichez l'époque, l'itération, la perte d'entraînement moyenne et la précision moyenne sur chaque époque (avec une actualisation à chaque itération), ainsi que la perte de validation et la précision à la fin de chaque époque.
 - b. Essayez un autre optimiseur avec un taux d'apprentissage différent pour les couches supérieures et les couches inférieures.

Les solutions de ces exercices sont données à l'annexe A.

5

Chargement et prétraitement de données avec TensorFlow

Les jeux de données que nous avons utilisés jusqu'à présent tenaient en mémoire. Cependant, les systèmes de Deep Learning sont souvent entraînés sur des jeux de données trop volumineux pour tenir intégralement en RAM. Le chargement d'un tel jeu de données et son prétraitement efficace peuvent être difficiles à mettre en place avec certaines bibliothèques de Deep Learning. Au travers de son *API Data*, TensorFlow apporte une solution simple : nous créons simplement un objet qui représente un jeu de données et lui indiquons où trouver les données et comment les transformer. TensorFlow se charge de tous les détails d'implémentation, comme les traitements en parallèle, la mise en file d'attente, la mise en lots et le prétraitement. Par ailleurs, l'*API Data* fonctionne parfaitement avec `tf.keras` !

En standard, l'*API Data* est capable de lire des fichiers texte (comme des fichiers CSV), des fichiers binaires constitués d'enregistrements de taille fixe, et des fichiers binaires fondés sur le format TFRecord de TensorFlow, qui prend en charge les enregistrements de taille variable. TFRecord est un format binaire souple et efficace fondé sur Protocol Buffers (un format binaire open source). L'*API Data* gère également la lecture à partir de bases de données SQL. De nombreuses extensions open source permettent de lire d'autres sources de données, comme le service BigQuery de Google.

Outre le problème de lecture efficace d'un jeu de données volumineux, nous devons également résoudre une autre difficulté : le prétraitement des données (le plus souvent une normalisation). Par ailleurs, le jeu de données n'est pas toujours exclusivement constitué de champs numériques pratiques. Il peut contenir des caractéristiques sous forme de texte, de catégories, etc. Ces informations doivent être encodées, par exemple à l'aide d'un encodage 1 parmi n (*one-hot*), d'un encodage par

sac de mots (*bag of words*) ou d'un *plongement*¹¹⁹ (*embedding*). Pour mettre en place tous ces prétraitements, une solution consiste à écrire des couches de prétraitement personnalisées. Une autre est d'utiliser celles fournies en standard par Keras.

Dans ce chapitre, nous décrivons l'API Data, le format TFRecord, la création de couches de prétraitement personnalisées et l'utilisation de celles disponibles dans Keras. Nous examinerons également brièvement quelques projets connexes de l'écosystème TensorFlow :

- **TF Transform (tf.Transform)**

Cette bibliothèque permet d'écrire une fonction de prétraitement unique qui peut être exécutée en mode batch sur l'intégralité du jeu d'entraînement, avant l'entraînement (afin de l'accélérer), puis être exportée sous forme de fonction TF et incorporée au modèle entraîné de sorte qu'après son déploiement en production il puisse gérer à la volée le prétraitement de nouvelles instances.

- **TF Datasets (TFDS)**

Cette bibliothèque fournit une fonction pratique pour télécharger de nombreux jeux de données de toutes sortes, même les plus volumineux comme ImageNet, ainsi que des objets `Dataset` pour les manipuler à l'aide de l'API Data.

C'est parti !

5.1 L'API DATA

Toute l'API Data s'articule autour du concept de *dataset*. Il s'agit de la représentation d'une suite d'éléments de données. En général, nous utilisons des datasets qui lisent les données progressivement à partir du disque, mais, pour plus de simplicité, créons un dataset entièrement en RAM avec `tf.data.Dataset.from_tensor_slices()` :

```
>>> X = tf.range(10) # N'importe quel tenseur de données
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

La fonction `from_tensor_slices()` prend un tenseur et crée un `tf.data.Dataset` dont les éléments sont tous des tranches de X (sur la première dimension). Ce dataset contient donc dix éléments : les tenseurs 0, 1, 2, ..., 9. Dans ce cas précis, nous aurions obtenu le même dataset avec `tf.data.Dataset.range(10)`.

Voici comment itérer simplement sur les éléments d'un dataset :

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
```

119. Comme nous le verrons, un plongement est un vecteur dense entraînable qui représente une catégorie ou un jeton.

```
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

5.1.1 Enchaîner des transformations

Après que nous avons obtenu un dataset, nous pouvons lui appliquer toutes sortes de transformations en invoquant ses méthodes de transformation. Puisque chaque méthode retourne un nouveau dataset, nous pouvons enchaîner des transformations (la chaîne créée par le code suivant est illustrée à la figure 5.1):

```
>>> dataset = dataset.repeat(3).batch(7)

>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

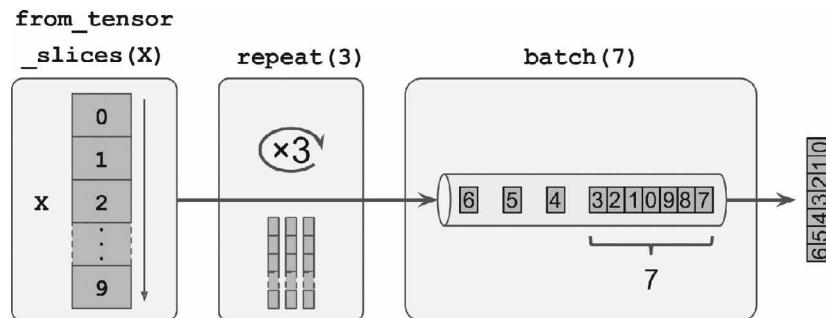


Figure 5.1 – Enchaînement de transformations sur le dataset

Dans cet exemple, nous commençons par invoquer la méthode `repeat()` sur le dataset d'origine. Elle retourne un nouveau dataset qui contient trois répétitions des éléments du dataset initial. Bien évidemment, toutes les données ne sont pas copiées trois fois en mémoire ! (Si nous appelons cette méthode sans argument, le nouveau dataset répète indéfiniment le dataset source. Dans ce cas, le code qui parcourt le dataset décide quand il doit s'arrêter.) Ensuite nous invoquons la méthode `batch()` sur ce nouveau dataset, et obtenons encore un nouveau dataset. Celui-ci répartit les éléments du précédent dans des lots de sept éléments. Enfin, nous itérons sur les éléments de ce dataset final. La méthode `batch()` a été obligée de produire un lot final dont la taille est non pas sept mais deux. Si nous préférons écarter ce lot final de sorte qu'ils aient tous la même taille, nous pouvons invoquer cette méthode avec `drop_remainder=True`.



Les méthodes ne modifient pas les datasets, mais en créent de nouveaux. Vous devez donc faire attention à conserver une référence à ces nouveaux datasets (par exemple avec `dataset = ...`), sinon rien ne se passera.

Nous pouvons également transformer les éléments en appelant la méthode `map()`. Par exemple, le code suivant crée un nouveau dataset dont la valeur de tous les éléments est multipliée par deux :

```
>>> dataset = dataset.map(lambda x: x * 2) # Éléments : [0,2,4,6,8,10,12]
```

C'est la fonction que nous appellerons pour appliquer un prétraitement à nos données. Puisqu'il impliquera parfois des calculs intensifs, comme le changement de forme d'une image ou sa rotation, mieux vaudra lancer plusieurs threads pour accélérer son exécution : il suffit de fixer l'argument `num_parallel_calls`. La fonction passée à la méthode `map()` doit être convertible en fonction TF (voir le chapitre 4).

Alors que la méthode `map()` applique une transformation à chaque élément, la méthode `apply()` applique une transformation au dataset dans son ensemble. Par exemple, le code suivant applique la fonction `unbatch()` au dataset (elle est actuellement en version expérimentale, mais elle va probablement rejoindre l'API de base dans une prochaine version). Chaque élément du nouveau dataset sera un tenseur d'un seul entier au lieu d'un lot de sept entiers :

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Éléments :  
# 0,2,4,...
```

Pour filtrer le dataset, il suffit simplement d'utiliser la méthode `filter()` :

```
>>> dataset = dataset.filter(lambda x: x < 10) # Éléments :  
# 0 2 4 6 8 0 2 4 6...
```

Nous souhaiterons souvent examiner uniquement quelques éléments d'un dataset. Pour cela, nous avons la méthode `take()` :

```
>>> for item in dataset.take(3):  
...     print(item)  
...  
tf.Tensor(0, shape=(), dtype=int64)  
tf.Tensor(2, shape=(), dtype=int64)  
tf.Tensor(4, shape=(), dtype=int64)
```

5.1.2 Mélanger les données

La descente de gradient est plus efficace lorsque les instances du jeu d'entraînement sont indépendantes et distribuées de façon identique¹²⁰. Pour obtenir cette configuration, une solution simple consiste à mélanger les instances à l'aide de la méthode `shuffle()`. Elle crée un nouveau dataset qui commencera par remplir un tampon avec les premiers éléments du dataset source. Ensuite, lorsqu'un élément est demandé, ce dataset le prendra de façon aléatoire dans le tampon et le remplacera par un nouvel élément provenant du dataset source. Le processus se répète jusqu'à

120. Voir le chapitre 1.

l'itération complète sur le dataset source. À ce moment-là, les éléments continueront à être extraits aléatoirement du tampon, jusqu'à ce que celui-ci soit vide. La taille du tampon doit être précisée et doit être suffisamment importante pour que le mélange soit efficace¹²¹. Toutefois, il ne faut pas dépasser la quantité de RAM disponible et, même si elle est importante, il est inutile d'aller au-delà de la taille du dataset.

Nous pouvons fournir un germe aléatoire de façon à obtenir le même ordre aléatoire à chaque exécution du programme. Par exemple, le code suivant crée et affiche un dataset qui contient les chiffres 0 à 9, répétés trois fois, mélangés avec un tampon de taille 5 et un germe aléatoire de 42, et regroupés en lots de taille 7 :

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 à 9, trois fois
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```



Si vous appelez `repeat()` sur un dataset mélangé, elle générera par défaut un nouvel ordre à chaque itération. Cela correspond en général au comportement souhaité. Mais, si vous préférez réutiliser le même ordre à chaque itération (par exemple pour les tests ou le débogage), vous pouvez indiquer `reshuffle_each_iteration=False`.

Lorsque le dataset est volumineux et ne tient pas en mémoire, cette solution de mélange à partir d'un tampon risque d'être insuffisante, car le tampon sera trop petit en comparaison du dataset. Une solution consiste à mélanger les données sources elles-mêmes (par exemple, la commande `shuf` de Linux permet de mélanger des fichiers texte). La qualité du mélange en sera énormément améliorée ! Mais, même si les données sources sont mélangées, nous voudrons généralement les mélanger un peu plus pour éviter que le même ordre ne se répète à chaque époque et que le modèle ne finisse par être biaisé (par exemple, en raison de faux motifs présents par hasard dans l'ordre des données sources). Pour cela, une solution classique consiste à découper des données sources en plusieurs fichiers, qui seront lus dans un ordre aléatoire au cours de l'entraînement. Cependant, les instances qui se trouvent dans le même fichier resteront proches l'une de l'autre. Pour éviter cette situation, nous pouvons choisir aléatoirement plusieurs fichiers et les lire simultanément, en entrelaçant leurs enregistrements.

121. Imaginons un jeu de cartes triées à notre gauche. Nous prenons uniquement les trois premières cartes et les mélangeons, puis en tirons une de façon aléatoire et la plaçons à notre droite, en gardant les deux autres cartes dans nos mains. Nous prenons une autre carte à gauche, mélangeons les trois cartes dans nos mains et en prenons une de façon aléatoire pour la placer à notre droite. Lorsque nous avons appliqué ce traitement à toutes les cartes, nous obtenons un jeu de cartes à notre droite. Pensez-vous qu'il soit parfaitement mélangé ?

Ensuite, par-dessus tout cela, nous pouvons ajouter un tampon de mélange en utilisant la méthode `shuffle()`. Si l'ensemble du processus vous semble complexe, ne vous inquiétez pas, l'API Data est là pour le réaliser en quelques lignes de code.

Entrelacer des lignes issues de plusieurs fichiers

Supposons tout d'abord que nous ayons chargé le jeu de données California Housing, l'ayons mélangé (sauf s'il l'était déjà) et l'ayons séparé en un jeu d'entraînement, un jeu de validation et un jeu de test. Nous découpons ensuite chaque jeu en de nombreux fichiers CSV, chacun ayant la forme suivante (chaque ligne contient huit caractéristiques d'entrée plus la valeur médiane cible d'une maison) :

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]
```

Supposons également que `train_filepaths` contienne la liste des chemins des fichiers d'entraînement (nous avons également `valid_filepaths` et `test_filepaths`) :

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]
```

Nous pourrions également utiliser des motifs de fichiers, par exemple `train_filepaths = "datasets/housing/my_train_*.csv"`. Nous créons à présent un dataset qui ne contient que ces chemins de fichiers :

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Par défaut, la fonction `list_files()` retourne un dataset qui mélange les chemins de fichiers. En général c'est plutôt une bonne chose, mais nous pouvons indiquer `shuffle=False` si, quelle qu'en soit la raison, nous ne le souhaitons pas.

Nous appelons ensuite la méthode `interleave()` pour effectuer la lecture à partir de cinq fichiers à la fois et entrelacer leurs lignes (en sautant la ligne d'en-tête de chaque fichier avec la méthode `skip()`) :

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

La méthode `interleave()` crée un dataset qui extrait cinq chemins de fichiers à partir de `filepath_dataset` et qui, pour chacun, appelle la fonction indiquée (un `lambda` dans cet exemple) de façon à créer un nouveau dataset (dans ce cas un `TextLineDataset`). À ce stade, nous avons sept datasets en tout : le dataset des chemins de fichiers, le dataset entrelacé et les cinq datasets `TextLineDataset` créés en interne par le dataset entrelacé. Lorsque nous itérons sur le dataset entrelacé, celui-ci alterne entre les cinq `TextLineDataset`, lisant une ligne à la fois à partir de chacun, jusqu'à ce qu'ils ne contiennent plus d'éléments. Il prend ensuite les cinq chemins de fichiers suivants à partir du `filepath_dataset` et les entrelace de la même manière. Il procède ainsi jusqu'à épuisement des chemins de fichiers.



Pour que l'entrelacement soit efficace, il est préférable que les fichiers soient de la même longueur. Dans le cas contraire, les fins des fichiers les plus longs ne seront pas entrelacées.

Par défaut, `interleave()` n'exploite pas le parallélisme ; elle se contente de lire une ligne à la fois à partir de chaque fichier, séquentiellement. Pour qu'elle procède en parallèle, nous pouvons fixer l'argument `num_parallel_calls` au nombre de threads souhaité (la méthode `map()` dispose également de cet argument). Nous pouvons même le fixer à `tf.data.experimental.AUTOTUNE` pour que TensorFlow choisisse dynamiquement le nombre de threads approprié en fonction du processeur disponible (cette fonctionnalité est encore expérimentale). Voyons ce que le dataset contient à présent :

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

Cela correspond aux premières lignes (la ligne d'en-tête étant ignorée) de cinq fichiers CSV choisies aléatoirement. C'est plutôt pas mal ! Mais, vous le constatez, il s'agit de chaînes d'octets. Nous devons donc les analyser et redimensionner les données.

5.1.3 Prétraiter les données

Implémentons une petite fonction qui effectue ce prétraitement :

```
X_mean, X_std = [...] # Moyenne et échelle de chaque caractéristique
# dans le jeu d'entraînement
n_inputs = 8

def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

Parcourons ce code :

- Tout d'abord, le code suppose que nous avons précalculé la moyenne et l'écart-type de chaque caractéristique dans le jeu d'entraînement. `X_mean` et `X_std` sont de simples tenseurs à une dimension (ou des tableaux NumPy) qui contiennent huit nombres à virgule flottante, un par caractéristique d'entrée.
- La fonction `preprocess()` prend une ligne CSV et commence son analyse. Pour cela, elle se sert de la fonction `tf.io.decode_csv()`, qui attend deux arguments : le premier est la ligne à analyser, le second est un tableau

contenant la valeur par défaut de chaque colonne du fichier CSV. Ce tableau indique à TensorFlow non seulement la valeur par défaut de chaque colonne, mais également le nombre de colonnes et leur type. Dans cet exemple, nous précisons que toutes les colonnes de caractéristiques sont des nombres à virgule flottante et que les valeurs manquantes doivent être fixées à 0. Toutefois, pour la dernière colonne (la cible), nous fournissons un tableau vide de type `tf.float32` comme valeur par défaut. Il indique à TensorFlow que cette colonne contient des nombres à virgule flottante, sans aucune valeur par défaut. Une exception sera donc lancée en cas de valeur manquante.

- La fonction `decode_csv()` retourne une liste de tenseurs de type scalaire (un par colonne), alors que nous devons retourner des tableaux de tenseurs à une dimension. Nous appelons donc `tf.stack()` sur tous les tenseurs à l'exception du dernier (la cible) : cette opération empile les tenseurs dans un tableau à une dimension. Nous faisons ensuite de même pour la valeur cible (elle devient un tableau de tenseurs à une dimension avec une seule valeur à la place d'un tenseur de type scalaire).
- Enfin, nous redimensionnons les caractéristiques d'entrée en leur soustrayant les moyennes des caractéristiques et en divisant le résultat par leur écart-type. Nous retournons un tuple qui contient les caractéristiques mises à l'échelle et la cible.

Testons cette fonction de prétraitement :

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
array([ 0.16579159,  1.216324 , -0.05204564, -0.39215982, -0.5277444 ,
       -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: [...], numpy=array([2.782], dtype=float32)>)
```

Le résultat semble bon ! Nous pouvons à présent l'appliquer au dataset.

5.1.4 Réunir le tout

Pour que le code soit réutilisable, nous allons réunir tout ce dont nous avons discuté jusqu'à présent dans une petite fonction utilitaire. Elle va créer et retourner un dataset qui chargera efficacement le jeu de données California Housing à partir de plusieurs fichiers CSV, le prétraitera, le mélangera, le répétera éventuellement et le divisera en lots (voir la figure 5.2) :

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                      n_read_threads=None, shuffle_buffer_size=10000,
                      n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    return dataset.batch(batch_size).prefetch(1)
```

Comprendre ce code ne devrait pas vous poser de difficultés, excepté peut-être pour la dernière ligne (`prefetch(1)`), qui est importante pour les performances.

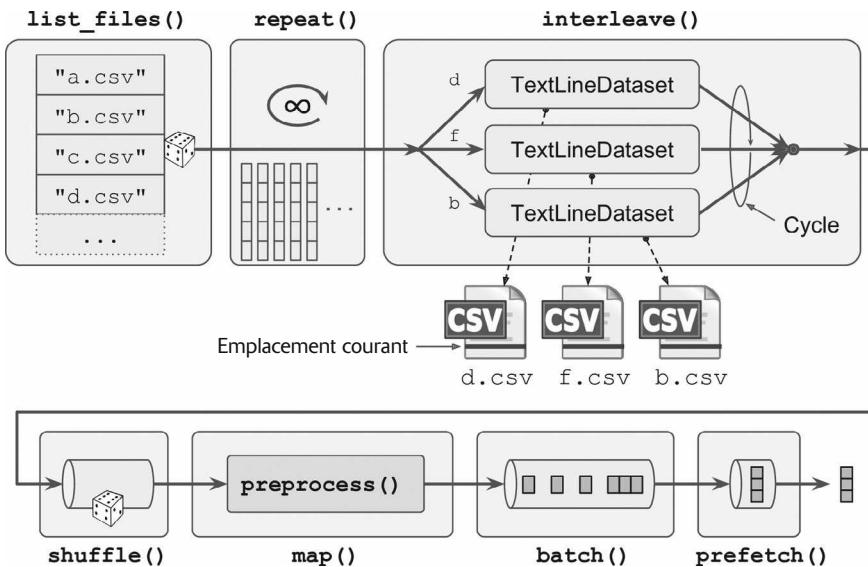


Figure 5.2 – Chargement et prétraitement de données provenant de plusieurs fichiers CSV

5.1.5 Prélecture

En appelant `prefetch(1)` à la fin du code, nous créons un dataset qui fera de son mieux pour rester toujours en avance d'un lot¹²². Autrement dit, pendant que notre algorithme d'entraînement travaille sur un lot, le dataset s'occupe en parallèle de préparer le lot suivant (par exemple, lire les données à partir du disque et les prétraiter). Cette approche peut améliorer énormément les performances, comme l'illustre la figure 5.3. Si nous faisons également en sorte que le chargement et le prétraitement soient multithreads (en fixant `num_parallel_calls` lors des appels à `interleave()` et à `map()`), nous pouvons exploiter les différents coeurs du processeur et, potentiellement, rendre le temps de préparation d'un lot de données plus court que l'exécution d'une étape d'entraînement sur le GPU. Celui-ci sera ainsi utilisé à 100 %, ou presque en raison du temps de transfert des données depuis le CPU vers le GPU¹²³, et l'entraînement sera beaucoup plus rapide.

122. En général, la prélecture d'un lot suffit. Mais, dans certains cas, il peut être bon d'effectuer la prélecture d'un plus grand nombre de lots. Il est également possible de laisser TensorFlow décider automatiquement en indiquant `tf.data.experimental.AUTOTUNE` (pour le moment, cette fonctionnalité est expérimentale).

123. Consultez la fonction `tf.data.experimental.prefetch_to_device()`, qui est capable de précharger directement des données dans le GPU.

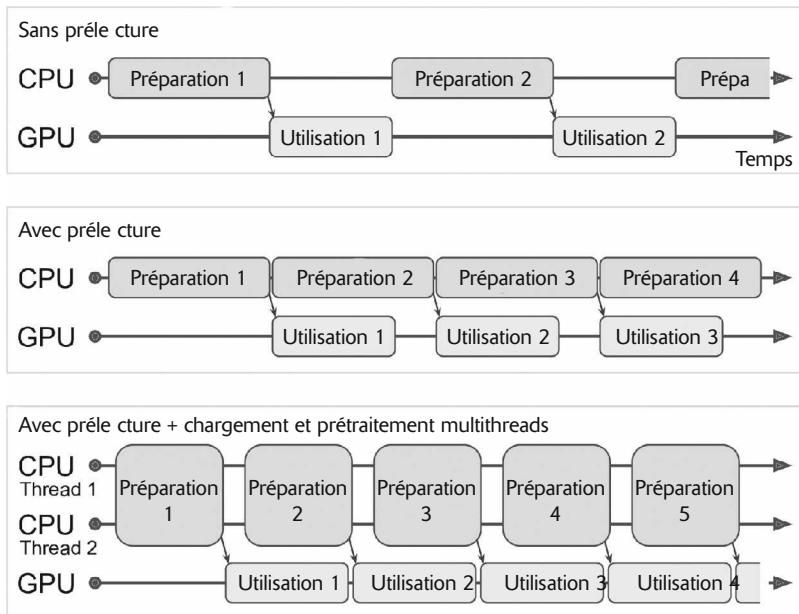


Figure 5.3 – Avec la prélecture, le CPU et le GPU fonctionnent en parallèle : pendant que le GPU travaille sur un lot, le CPU prépare le suivant



Si vous prévoyez d'acheter une carte graphique, sa puissance de traitement et la taille de sa mémoire sont évidemment très importantes (en particulier, une grande quantité de RAM est indispensable à la vision par ordinateur). Sa *bande passante mémoire* doit elle aussi être performante, car elle correspond au nombre de giga-octets de données qui peuvent entrer ou sortir de sa mémoire RAM à chaque seconde.

Si le dataset est suffisamment petit pour tenir en mémoire, nous pouvons accélérer l'entraînement en utilisant sa méthode `cache()` : elle place son contenu dans un cache en mémoire RAM. Cette opération se fait généralement après le chargement et le prétraitement des données, mais avant leur mélange, leur répétition, leur mise en lot et leur prélecture. De cette manière, chaque instance ne sera lue et prétraitée qu'une seule fois (au lieu d'une fois par époque), mais les données resteront mélangées différemment à chaque époque, et le lot suivant sera toujours préparé à l'avance.

Vous savez à présent construire un pipeline d'entrée efficace pour charger et prétraiter des données provenant de plusieurs fichiers texte. Nous avons décrit les méthodes les plus utilisées sur un dataset, mais quelques autres méritent votre attention: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()` et `padded_batch()`. Par ailleurs, deux méthodes de classe supplémentaires, `from_generator()` et `from_tensors()`, permettent de créer un nouveau dataset à partir, respectivement, d'un générateur Python ou d'une liste de tenseurs. Les détails se trouvent dans la documentation de l'API. Des fonctionnalités

expérimentales sont disponibles dans `tf.data.experimental` et nombre d'entre elles devraient rejoindre l'API de base dans les prochaines versions (par exemple, jetez un œil à la classe `CsvDataset` et à la méthode `make_csv_dataset()`, qui se charge de deviner le type de chaque colonne).

5.1.6 Utiliser le dataset avec tf.keras

Nous pouvons à présent utiliser la fonction `csv_reader_dataset()` pour créer un dataset associé au jeu d'entraînement. Il est inutile de le répéter nous-mêmes, car `tf.keras` va s'en occuper. Nous créons également des datasets pour les jeux de validation et de test :

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

Il ne nous reste plus qu'à construire et à entraîner un modèle Keras avec ces datasets¹²⁴. Pour cela, nous passons simplement les datasets d'entraînement et de validation à la méthode `fit()`, à la place de `x_train`, `y_train`, `x_valid` et `y_valid`¹²⁵:

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

De manière comparable, nous passons un dataset aux méthodes `evaluate()` et `predict()` :

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) # Prétendre que nous avons
                                                # 3 nouvelles instances
model.predict(new_set) # Un dataset contenant de nouvelles instances
```

Contrairement aux autres jeux, `new_set` ne contiendra généralement aucune étiquette (dans le cas contraire, Keras les ignore). Si nous le souhaitons, dans tous ces cas, nous pouvons toujours utiliser des tableaux NumPy à la place des datasets (ils devront cependant être préalablement chargés et prétraités).

Si nous voulons construire notre propre boucle d'entraînement personnalisée (comme au chapitre 4), il nous suffit d'itérer très naturellement sur le jeu d'entraînement :

```
for X_batch, y_batch in train_set:
    [...] # Effectuer une étape de descente de gradient
```

En réalité, il est même possible de créer une fonction TF (voir le chapitre 4) qui réalise l'intégralité de la boucle d'entraînement :

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
```

124. La prise en charge des datasets est spécifique à `tf.keras`. La procédure ne fonctionnera pas dans d'autres implémentations de l'API de Keras.

125. La méthode `fit()` se charge de répéter le dataset d'entraînement. Mais nous pouvons également appeler `repeat()` sur ce dataset afin d'avoir une répétition infinie et fixer l'argument `steps_per_epoch` lors de l'appel à `fit()`. Cette solution pourrait être utile dans quelques cas rares, par exemple si nous devons utiliser un tampon de mélange qui s'étale sur plusieurs époques.

```

for X_batch, y_batch in train_set:
    with tf.GradientTape() as tape:
        y_pred = model(X_batch)
        main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
        loss = tf.add_n([main_loss] + model.losses)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Et voilà, vous savez à présent construire des pipelines d'entrée puissants à l'aide de l'API Data ! Mais, pour le moment, nous n'avons utilisé que des fichiers CSV. S'ils sont répandus, simples et pratiques, ils sont en revanche peu efficaces et ont quelques difficultés avec les structures de données volumineuses ou complexes (comme les images ou l'audio). Nous allons voir comment utiliser à la place des fichiers au format TFRecord.



Si les fichiers CSV (ou de tout autre format) vous conviennent parfaitement, rien ne vous oblige à employer le format TFRecord. Comme le dit le dicton, «si ça marche, il ne faut pas y toucher !» Les fichiers TFRecord seront utiles lorsque le goulot d'étranglement pendant l'entraînement réside dans le chargement et l'analyse des données.

5.2 LE FORMAT TFRECORD

Le format TFRecord de TensorFlow doit, de préférence, être adopté si l'on veut un stockage et une lecture efficace de grandes quantités de données. Il s'agit d'un format binaire très simple qui contient uniquement une suite d'enregistrements binaires de taille variable (chaque enregistrement est constitué d'une longueur, d'une somme de contrôle CRC pour vérifier que la longueur n'a pas été corrompue, des données réelles et d'une somme de contrôle CRC pour les données). La création d'un fichier TFRecord se fait simplement à l'aide de la classe `tf.io.TFRecordWriter`:

```

with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")

```

Nous pouvons ensuite utiliser un `tf.data.TFRecordDataset` pour lire un ou plusieurs fichiers TFRecord :

```

filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)

```

Nous obtenons le résultat suivant :

```

tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)

```



Par défaut, un `TFRecordDataset` lit les fichiers un par un, mais vous pouvez lui demander de lire plusieurs fichiers en parallèle et d'entrelacer leurs enregistrements en fixant `num_parallel_reads`. Vous pouvez également obtenir le même résultat en utilisant `list_files()` et `interleave()`, comme nous l'avons fait précédemment pour la lecture de plusieurs fichiers CSV.

5.2.1 Fichiers TFRecord compressés

La compression des fichiers TFRecord pourrait se révéler utile, notamment s'ils doivent être chargés au travers d'une connexion réseau. Pour créer un fichier TFRecord compressé, nous devons fixer l'argument `options`:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    [...]
```

Pour sa lecture, nous devons préciser le type de compression:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

5.2.2 Introduction à Protocol Buffers

Bien que chaque enregistrement puisse avoir n'importe quel format binaire, les fichiers TFRecord contiennent habituellement des objets sérialisés au format Protocol Buffers (également appelés *protobufs*, et respectant un format particulier défini dans un fichier de définition de Protocol Buffer, ou fichier `.proto`). Protocol Buffers est un format binaire portable, extensible et efficace développé par Google en 2001 et mis en open source en 2008. Les protobufs sont aujourd'hui très utilisés, en particulier dans gRPC (<https://grpc.io>), le système d'appel de procédure à distance de Google. Ils sont définis à l'aide d'un langage simple :

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

Cette définition indique que nous utilisons la version 3 du format du protobuf et précise que chaque objet `Person`¹²⁶ peut (potentiellement) avoir un nom `name` de type `string`, un identifiant `id` de type `int32`, et zéro champ `email` ou plus, chacun de type `string`. Les chiffres 1, 2 et 3 sont des identifiants de champs qui seront utilisés dans la représentation binaire de chaque enregistrement. Après que nous avons défini un fichier `.proto`, nous le compilons. Pour cela, nous avons besoin de `protoc`, le compilateur protobuf, qui génère des classes d'accès en Python (ou dans un autre langage). Les définitions protobuf que nous allons utiliser ont déjà été compilées pour nous et leurs classes Python font partie de TensorFlow. Vous n'aurez donc pas besoin d'utiliser `protoc`, juste de savoir comment utiliser les classes d'accès protobuf en Python.

Pour illustrer les bases, examinons un exemple simple qui exploite les classes d'accès générées pour le protobuf `Person` (les commentaires expliquent le code):

```
>>> from person_pb2 import Person # Importer la classe d'accès générée
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # Créer un objet
# Person
>>> print(person) # Afficher l'objet Person
```

126. Puisque les objets protobuf sont destinés à être sérialisés et transmis, ils sont appelés *messages*.

```

name: "Al"
id: 123
email: "a@b.com"
>>> person.name # Lire un champ
"Al"
>>> person.name = "Alice" # Modifier un champ
>>> person.email[0] # L'accès aux champs répétés peut se faire
# comme pour des tableaux
"a@b.com"
>>> person.email.append("c@d.com") # Ajouter une adresse e-mail
>>> s = person.SerializeToString() # Sérialiser l'objet dans une chaîne
# d'octets
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # Créer un nouveau Person
>>> person2.ParseFromString(s) # Analyser la chaîne d'octets
# (longue de 27 octets)
27
>>> person == person2 # À présent égaux
True

```

En bref, nous importons la classe `Person` produite par `protoc`, nous en créons une instance, que nous manipulons et affichons, et dont nous lisons et modifions certains champs, puis nous la sérialisons avec la méthode `SerializeToString()`. Nous obtenons des données binaires prêtes à être enregistrées ou transmises sur le réseau. Lors de la lecture ou de la réception de ces données binaires, nous pouvons les analyser à l'aide de la méthode `ParseFromString()`, pour obtenir une copie de l'objet qui avait été sérialisé¹²⁷.

Nous pouvons enregistrer l'objet `Person` sérialisé dans un fichier `TFRecord`, pour ensuite le charger et l'analyser; tout devrait bien se passer. Toutefois, puisque `SerializeToString()` et `ParseFromString()` ne sont pas des opérations TensorFlow (pas plus que ne le sont les autres opérations de ce code), elles ne peuvent pas être incluses dans une fonction TensorFlow (sauf en les enveloppant dans une opération `tf.py_function()`, qui rendrait le code plus long et moins portable, comme nous l'avons vu au chapitre 4). Heureusement, TensorFlow fournit des définitions protobuf spéciales avec des opérations d'analyse.

5.2.3 Protobufs de TensorFlow

Le principal protobuf utilisé dans un fichier `TFRecord` est le protobuf `Example`, qui représente une instance d'un dataset. Il contient une liste de caractéristiques nommées, chacune pouvant être une liste de chaînes d'octets, une liste de nombres à virgule flottante ou une liste d'entiers. Voici sa définition :

```

syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }

```

127. Ce chapitre présente le strict minimum sur les protobufs dont vous avez besoin pour utiliser des fichiers `TFRecord`. Pour de plus amples informations, consultez le site <https://homl.info/protobuf>.

```

message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};

message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };

```

Les définitions de `BytesList`, `FloatList` et `Int64List` sont faciles à comprendre. `[packed = true]` est employé avec les champs numériques répétés pour obtenir un encodage plus efficace. Un `Feature` contient un `BytesList`, un `FloatList` ou un `Int64List`. Un `Features` (avec un `s`) contient un dictionnaire qui met en correspondance un nom de caractéristique et sa valeur. Enfin, un `Example` contient simplement un objet `Features`¹²⁸. Le code suivant crée un `tf.train.Example` qui représente la même personne que précédemment, et l'enregistre dans un fichier TFRecord :

```

from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                b"c@d.com"]))
        })
)

```

Ce code est un tantinet verbeux et répétitif, mais il est relativement simple (et il est facile à placer dans une petite fonction utilitaire). Nous disposons à présent d'un protobuf `Example`, que nous pouvons sérialiser en invoquant sa méthode `SerializeToString()`, et écrire les données obtenues dans un fichier TFRecord :

```

with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())

```

Normalement, nous devrions écrire plusieurs `Example`! En effet, nous voudrons généralement créer un script de conversion qui lit les éléments dans leur format actuel (par exemple, des fichiers CSV), crée un protobuf `Example` pour chaque instance, les sérialise et les enregistre dans plusieurs fichiers TFRecord, idéalement en les mélangeant au passage. Puisque cela demande un peu de travail, demandez-vous s'il est absolument nécessaire (il est possible que votre pipeline fonctionne parfaitement avec des fichiers CSV).

Essayons à présent de charger ce fichier TFRecord qui contient un `Example` sérialisé.

128. Pourquoi avoir défini `Example` s'il ne contient rien de plus qu'un objet `Features`? La raison est simple : les développeurs de TensorFlow pourraient décider un jour de lui ajouter d'autres champs. Tant que la nouvelle définition `Example` contient le champ `features`, avec le même identifiant, elle conserve sa rétrocompatibilité. Cette extensibilité est l'une des grandes caractéristiques des protobufs.

5.2.4 Charger et analyser des Example

Pour charger les protobufs `Example` sérialisés, nous allons utiliser à nouveau un `tf.data.TFRecordDataset`, et nous analyserons chaque `Example` avec `tf.io.parse_single_example()`. Puisqu'il s'agit d'une opération TensorFlow, elle peut être incluse dans une fonction TF. Elle attend au moins deux arguments : un tenseur scalaire de type chaîne qui contient les données sérialisées et une description de chaque caractéristique. La description est un dictionnaire qui associe un nom de caractéristique soit à un descripteur `tf.io.FixedLenFeature` indiquant la forme, le type et la valeur par défaut de la caractéristique, soit à un descripteur `tf.io.VarLenFeature` précisant uniquement le type (si la longueur de la liste de la caractéristique peut varier, comme dans le cas de la caractéristique "emails").

Le code suivant définit un dictionnaire de description, puis itère sur le `TFRecordDataset` et analyse le protobuf `Example` sérialisé contenu dans ce dataset :

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

Les caractéristiques de taille fixe sont analysées comme des tenseurs normaux, tandis que les caractéristiques de taille variable sont analysées comme des tenseurs creux. La conversion d'un tenseur creux en un tenseur dense se fait avec `tf.sparse.to_dense()`, mais, dans ce cas, il est plus simple d'accéder directement à ses valeurs :

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

Un `BytesList` peut contenir n'importe quelles données binaires, y compris un objet sérialisé. Par exemple, nous pouvons appeler `tf.io.encode_jpeg()` pour encoder une image au format JPEG et placer ces données binaires dans un `BytesList`. Plus tard, lorsque notre code lira le `TFRecord`, il commencera par analyser l'`Example`, puis il devra appeler `tf.io.decode_jpeg()` pour analyser les données et obtenir l'image d'origine (nous pouvons également appeler `tf.io.decode_image()`, capable de décoder n'importe quelle image BMP, GIF, JPEG ou PNG). Nous pouvons également stocker n'importe quel tenseur dans un `BytesList` en le sérialisant avec `tf.io.serialize_tensor()`, puis en plaçant la chaîne d'octets résultante dans une caractéristique `BytesList`. Ensuite, lors de l'analyse du `TFRecord`, il suffit de traiter ces données avec `tf.io.parse_tensor()`.

Au lieu d'analyser les *Example* unitairement avec `tf.io.parse_single_example()`, nous pourrions préférer les analyser lot par lot avec `tf.io.parse_example()` :

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)
```

Dans la plupart des cas d'utilisation, le protobuf *Example* devrait donc suffire. Toutefois, son utilisation risque d'être un peu lourde si nous devons manipuler des listes de listes. Par exemple, supposons que nous souhaitions classifier des documents texte. Chaque document peut se présenter sous forme d'une liste de phrases, où chaque phrase est représentée par une liste de mots. De plus, chaque document peut également avoir une liste de commentaires, où chaque commentaire est représenté par une liste de mots. À cela peuvent s'ajouter des données contextuelles, comme l'auteur, le titre et la date de publication du document. Le protobuf *SequenceExample* de TensorFlow est conçu pour prendre en charge de tels cas.

5.2.5 Gérer des listes de listes avec le protobuf *SequenceExample*

Voici la définition du protobuf *SequenceExample*:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

Un *SequenceExample* est constitué d'un objet *Features* pour les données contextuelles et d'un objet *FeatureLists* qui contient un ou plusieurs objets *FeatureList* nommés (par exemple, un *FeatureList* nommé "content" et un autre nommé "comments"). Chaque *FeatureList* contient une liste d'objets *Feature*, chacun pouvant être une liste de chaînes d'octets, une liste d'entiers sur 64 bits ou une liste de nombres à virgule flottante (dans notre exemple, chaque *Feature* représenterait une phrase ou un commentaire, éventuellement sous la forme d'une liste d'identifiants de mots). Construire, sérialiser et analyser un *SequenceExample* ne sont pas très différents de ces mêmes opérations sur un *Example*, mais nous devons employer `tf.io.parse_single_sequence_example()` pour analyser un seul *SequenceExample* ou `tf.io.parse_sequence_example()` pour un traitement par lot. Les deux fonctions retournent un tuple qui contient les caractéristiques de contexte (sous forme de dictionnaire) et les listes de caractéristiques (également un dictionnaire). Si les listes de caractéristiques contiennent des séquences de taille variable (comme dans l'exemple précédent), nous pouvons les convertir en tenseurs irréguliers à l'aide de `tf.RaggedTensor.from_sparse()` (code complet dans le notebook¹²⁹):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
```

129. Voir « 13_loading_and_preprocessing_data.ipynb » sur <https://github.com/ageron/handson-ml2>.

```

    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])

```

Puisque vous savez à présent comment stocker, charger et analyser efficacement des données, l'étape suivante consiste à les préparer afin qu'elles puissent alimenter un réseau de neurones.

5.3 PRÉTRAITER LES CARACTÉRISTIQUES D'ENTRÉE

La préparation des données destinées à un réseau de neurones demande notamment de convertir toutes les caractéristiques en caractéristiques numériques, généralement en les normalisant. Plus précisément, si les données contiennent des caractéristiques de catégorie ou de texte, elles doivent être converties en nombres. Cela peut se faire à l'avance, pendant la préparation des fichiers de données, à l'aide de n'importe quel outil, comme NumPy, pandas ou Scikit-Learn. Nous pouvons également prétraiter les données à la volée pendant leur chargement avec l'API Data (par exemple, en utilisant la méthode `map()` du dataset, comme nous l'avons vu précédemment) ou inclure une couche de prétraitement directement dans le modèle. Examinons cette dernière approche.

Voici par exemple comment implémenter une couche de standardisation en utilisant une couche Lambda. Pour chaque caractéristique, elle soustrait la moyenne et divise le résultat par son écart-type (plus un petit terme de lissage pour éviter la division par zéro) :

```

means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...] # Autres couches
])

```

Ce n'était pas trop difficile ! Cependant, nous pourrions préférer utiliser une couche personnalisée autonome (à l'instar de `StandardScaler` de Scikit-Learn) plutôt que gérer des variables comme `means` et `stds` :

```

class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())

```

Avant de pouvoir utiliser cette couche de standardisation, nous devons l'adapter à notre jeu de données en invoquant la méthode `adapt()` et en lui passant un échantillon de données. Elle est ainsi en mesure d'appliquer la moyenne et l'écart-type appropriés pour chaque caractéristique :

```

std_layer = Standardization()
std_layer.adapt(data_sample)

```

L'échantillon doit être suffisamment grand pour être représentatif du jeu de données, mais il est inutile de passer le jeu d'entraînement complet. En général, quelques

centaines d'instances choisies aléatoirement suffiront, mais cela dépend de la tâche. Nous pouvons ensuite employer cette couche de prétraitement comme une couche normale :

```
model = keras.Sequential()
model.add(std_layer)
[...] # Créer le reste du modèle
model.compile([...])
model.fit([...])
```

Si vous pensez que Keras devrait proposer une couche de standardisation comme celle-ci, voici de bonnes nouvelles : au moment où vous lirez ces lignes, la couche `keras.layers.Normalization` sera probablement disponible. Elle opérera de façon comparable à notre couche `Standardization` personnalisée : créer la couche, l'adapter au jeu de données en passant un échantillon à la méthode `adapt()`, et l'utiliser de façon normale.

Voyons à présent les caractéristiques de catégorie. Nous commencerons par les encoder sous forme de vecteurs *one-hot*.

5.3.1 Encoder des caractéristiques de catégorie dans des vecteurs *one-hot*

Attardons-nous sur la caractéristique `ocean_proximity` du jeu de données California Housing¹³⁰. Il s'agit d'une caractéristique de catégorie ayant cinq valeurs possibles : "`<1H OCEAN`", "`INLAND`", "`NEAR OCEAN`", "`NEAR BAY`" et "`ISLAND`". Nous devons l'encoder avant de la transmettre à un réseau de neurones. Puisque les valeurs sont peu nombreuses, un encodage *one-hot* conviendra. Pour cela, nous commençons par associer chaque catégorie à son indice (0 à 4) à l'aide d'une table de correspondance :

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

Détaillons ce code :

- Nous définissons tout d'abord le *vocabulaire*, c'est-à-dire la liste de toutes les catégories possibles.
- Nous créons ensuite un tenseur avec les indices correspondants (0 à 4).
- Puis nous créons un initialiseur pour la table de correspondance, en lui passant la liste des catégories et les indices associés. Dans cet exemple, puisqu'en nous disposons déjà de ces données, nous utilisons un `KeyValueTensorInitializer`. Si les catégories se trouvaient dans un fichier texte (avec une catégorie par ligne), nous aurions utilisé à la place un `TextFileInitializer`.
- Les deux dernières lignes créent la table de correspondance, en lui passant l'initialiseur et en précisant le nombre d'emplacements hors *vocabulaire*. Si nous

¹³⁰. Voir le chapitre 2 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

recherchons une catégorie absente du vocabulaire, la table de correspondance calcule un code de hachage pour cette catégorie et l'utilise pour affecter la catégorie inconnue à l'un des emplacements hors vocabulaire. Leurs indices débutent après les catégories connues, c'est-à-dire, dans cet exemple, les indices 5 et 6.

Pourquoi utiliser des emplacements hors vocabulaire ? Si le nombre de catégories est élevé (par exemple, des codes postaux, des noms de villes, des mots, des produits ou des utilisateurs) et si le jeu de données est également volumineux ou change constamment, il ne sera pas facile d'obtenir la liste complète des catégories. Une solution consiste à définir le vocabulaire à partir d'un échantillon des données (à la place du jeu d'entraînement complet) et d'ajouter quelques emplacements hors vocabulaire pour les catégories qui seraient absentes de l'échantillon. Le nombre d'emplacements dépend du nombre de catégories inconnues que nous pensons trouver pendant l'entraînement. S'il est insuffisant, il y aura évidemment des collisions : des catégories différentes vont se retrouver dans le même emplacement et le réseau de neurones sera incapable de les différencier (tout au moins avec cette caractéristique).

Utilisons la table de correspondance pour encoder un petit lot de caractéristiques de catégorie dans des vecteurs *one-hot* :

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

"NEAR BAY" a été associé à l'indice 3, la catégorie inconnue "DESERT", à l'un des deux emplacements hors vocabulaire (à l'indice 5), et "INLAND", à l'indice 1, deux fois. L'encodage de ces indices dans des vecteurs *one-hot* a été effectué par `tf.one_hot()`. Nous avons dû indiquer à cette fonction le nombre total d'indices, qui correspond à la taille du vocabulaire plus le nombre d'emplacements hors vocabulaire. Voilà comment encoder des caractéristiques de catégorie sous forme de vecteurs *one-hot* avec TensorFlow !

Comme précédemment, il n'est pas trop difficile de réunir toute cette logique dans une classe indépendante. Sa méthode `adapt()` prendrait un échantillon de données et en extrairait les différentes catégories qu'il contient. Elle créerait une table de correspondance pour associer chaque catégorie à son indice (y compris les catégories inconnues à l'aide d'emplacements hors vocabulaire). Puis sa méthode `call()` se servirait de la table de correspondance pour associer les catégories d'entrée à leur indice. Voici une bonne nouvelle : au moment où vous lirez ces lignes, Keras proposera probablement une couche nommée `keras.layers.TextVectorization`, capable de réaliser tout cela. Sa méthode `adapt()` extraira le vocabulaire à partir d'un échantillon de données et sa méthode `call()` convertira chaque catégorie en son indice dans le vocabulaire. Pour

convertir ces indices en vecteurs *one-hot*, vous pouvez ajouter cette couche au début du modèle, suivie d'une couche Lambda qui applique la fonction `tf.one_hot()`.

Mais ce n'est peut-être pas la meilleure solution. La taille de chaque vecteur *one-hot* est égale à la longueur du vocabulaire plus le nombre d'emplacements hors vocabulaire. Si cette approche convient pour un nombre de catégories possibles faible, lorsque le vocabulaire est grand, il devient beaucoup plus efficace d'opter à la place pour les *plongements*.



En règle générale, si le nombre de catégories est inférieur à 10, l'encodage *one-hot* constitue souvent la meilleure solution (mais votre seuil peut varier!). Si le nombre de catégories est supérieur à 50 (ce qui est souvent le cas avec les codes de hachage), les plongements deviennent souvent incontournables. Entre 10 et 50 catégories, vous pouvez tester les deux solutions et voir celle qui convient.

5.3.2 Encoder les caractéristiques de catégorie avec des plongements

Un plongement (*embedding*) est un vecteur dense entraînable qui représente une catégorie. Par défaut, les plongements sont initialisés de façon aléatoire. Par exemple, la catégorie "NEAR BAY" pourrait être représentée initialement par un vecteur aléatoire comme [0.131, 0.890], tandis que la catégorie "NEAR OCEAN" le serait par un autre vecteur aléatoire comme [0.631, 0.791]. Cet exemple utilise des plongements à deux dimensions, mais le nombre de dimensions est un hyperparamètre ajustable. Puisque ces plongements sont entraînables, ils s'amélioreront progressivement au cours de l'entraînement; et s'ils représentent des catégories relativement similaires, la descente de gradient les rapprochera sûrement l'un de l'autre, tandis qu'ils auront tendance à s'éloigner du plongement de la catégorie "INLAND" (voir la figure 5.4).

Évidemment, plus la représentation est bonne, plus il est facile pour le réseau de neurones d'effectuer des prédictions précises. L'entraînement tend donc à faire en sorte que les plongements soient des représentations utiles des catégories. C'est ce que l'on appelle l'*apprentissage de représentations* (nous verrons d'autres types d'apprentissage de représentations au chapitre 9).

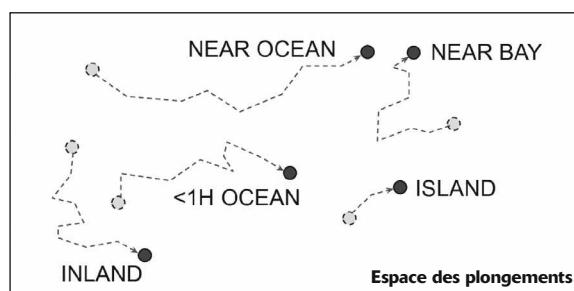


Figure 5.4 – Les plongements vont s'améliorer progressivement au cours de l'entraînement

Plongements de mots

Les plongements seront généralement des représentations utiles pour la tâche en cours. Mais, assez souvent, ces mêmes plongements pourront également être réutilisés avec succès dans d'autres tâches. L'exemple le plus répandu est celui des *plongements de mots* (c'est-à-dire, des plongements de mots individuels) : lorsqu'on travaille sur un problème de traitement automatique du langage naturel, il est souvent préférable de réutiliser des plongements de mots préentraînés plutôt que d'entrainer nos propres plongements.

L'idée d'employer des vecteurs pour représenter des mots remonte aux années 1960 et de nombreuses techniques élaborées ont été mises en œuvre pour générer des vecteurs utiles, y compris à l'aide de réseaux de neurones. Mais l'approche a réellement décollé en 2013, lorsque Tomáš Mikolov et d'autres chercheurs chez Google ont publié un article¹³¹ décrivant une technique d'apprentissage des plongements de mots avec des réseaux de neurones qui surpassait largement les tentatives précédentes. Elle leur a permis d'apprendre des plongements sur un corpus de texte très large : ils ont entraîné un réseau de neurones pour qu'il prédise les mots proches de n'importe quel mot donné et ont obtenu des plongements de mots stupéfiants. Par exemple, les synonymes ont des plongements très proches et des mots liés sémantiquement, comme France, Espagne et Italie, finissent par être regroupés.

Toutefois, ce n'est pas qu'une question de proximité. Les plongements de mots sont également organisés selon des axes significatifs dans l'espace des plongements. Voici un exemple bien connu : si nous calculons Roi – Homme + Femme (en ajoutant et en soustrayant les vecteurs des plongements de ces mots), le résultat sera très proche du plongement du mot Reine (voir la figure 5.5). Autrement dit, les plongements de mots permettent d'encoder le concept de genre ! De façon comparable, le calcul de Madrid – Espagne + France donne un résultat proche de Paris, ce qui semble montrer que la notion de capitale a également été encodée dans les plongements.

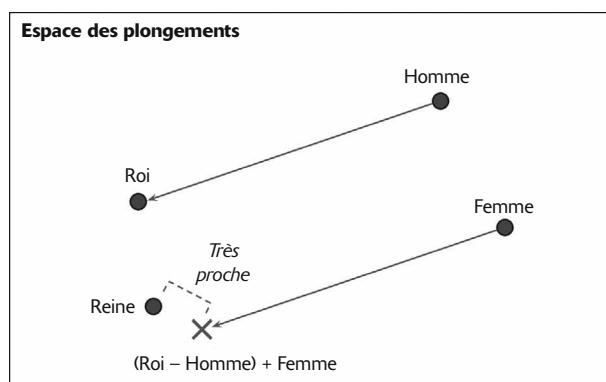


Figure 5.5 – Pour des mots similaires, les plongements de mots ont tendance à se rapprocher, et certains axes semblent encoder des concepts significatifs

131. Tomas Mikolov et al., « Distributed Representations of Words and Phrases and Their Compositionality », *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013), 3111-3119 : <https://homl.info/word2vec>.

Malheureusement, les plongements de mots traduisent parfois nos pires préjugés. Par exemple, même s'ils apprennent correctement que l'homme est roi comme la femme est reine, ils semblent également apprendre que l'homme est médecin comme la femme est infirmière : un parti pris plutôt sexiste ! Pour être honnête, cet exemple précis est probablement exagéré, comme l'ont souligné Malvina Nissim et al. dans un article¹³² publié en 2019. Quoi qu'il en soit, garantir l'impartialité dans les algorithmes de Deep Learning est un sujet de recherche important et actif.

Pour comprendre le fonctionnement des plongements, voyons comment les mettre en œuvre de façon manuelle (par la suite, nous utiliserons à la place une couche Keras simple). Tout d'abord, nous devons créer une *matrice de plongements* qui contiendra chaque plongement de catégorie initialisé aléatoirement. Cette matrice est constituée d'une ligne par catégorie et par emplacement hors vocabulaire, et d'une colonne par dimension de plongement :

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

Cet exemple utilise des plongements à deux dimensions, mais, en règle générale, ils ont entre 10 et 300 dimensions, en fonction de la tâche et la taille du vocabulaire (cet hyperparamètre devra être calibré).

Notre matrice de plongements est une matrice aléatoire 6×2 stockée dans une variable (elle peut donc être ajustée par la descente de gradient au cours de l'entraînement) :

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Encodons le lot de caractéristiques de catégorie précédent, cette fois-ci en utilisant les plongements :

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

132. Malvina Nissim et al., « Fair is Better than Sensational: Man is to Doctor as Woman is to Doctor » (2019) : <https://hml.info/fairembeds>.

La fonction `tf.nn.embedding_lookup()` recherche dans la matrice de plongements les lignes qui se trouvent aux indices donnés – elle ne fait rien d'autre. Par exemple, la table de correspondance stipule que la catégorie "INLAND" a l'indice 1 et la fonction `tf.nn.embedding_lookup()` retourne le plongement qui se trouve à la ligne 1 de la matrice de plongements (deux fois): [0.3528825, 0.46448255].

Keras fournit la couche `keras.layers.Embedding` qui prend en charge la matrice de plongements (par défaut entraînable). Au moment de sa création, la couche initialise la matrice de plongements de façon aléatoire, puis, lors de son appel avec des indices de catégories, elle retourne les lignes stockées à ces indices dans la matrice :

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
...                                         output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

En combinant tout cela, nous pouvons créer un modèle Keras capable de gérer des caractéristiques de catégorie (ainsi que les caractéristiques numériques classiques) et d'apprendre un plongement pour chaque catégorie (ainsi que pour chaque emplacement hors vocabulaire) :

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

Ce modèle possède deux entrées : une entrée normale contenant huit caractéristiques numériques par instance, et une entrée de catégorie contenant une caractéristique de catégorie par instance. Il se sert d'une couche `Lambda` pour déterminer l'indice de chaque catégorie, puis il recherche les plongements correspondant à ces indices. Ensuite, il associe les plongements et les entrées normales de façon à produire des entrées encodées, prêtes à alimenter un réseau de neurones. À ce stade, nous pourrions ajouter n'importe quel type de réseau de neurones, mais nous ajoutons simplement une couche de sortie dense et créons le modèle Keras.

Si la couche `keras.layers.TextVectorization` est disponible, nous pouvons invoquer sa méthode `adapt()` pour qu'elle extraie le vocabulaire à partir d'un échantillon de données (elle se chargera de la création de la table de correspondance à notre place). Nous pouvons ensuite l'ajouter au modèle pour qu'elle effectue la recherche des indices (en remplaçant la couche `Lambda` dans le code précédent).



L'encodage *one-hot* suivi d'une couche `Dense` (sans fonction d'activation ni terme constant) équivaut à une couche `Embedding`. Cependant, la couche `Embedding` réalise beaucoup moins de calculs (la différence de performance apparaît nettement lorsque la taille de la matrice de plongements augmente). La matrice des poids de la couche `Dense` a le même rôle que la matrice de plongements. Par exemple, avec des vecteurs *one-hot* de taille 20 et une couche `Dense` de 10 unités, nous avons l'équivalent d'une couche `Embedding` avec `input_dim=20` et `output_dim=10`. C'est pourquoi il serait superflu d'utiliser un nombre de dimensions de plongements supérieur au nombre d'unités dans la couche qui suit la couche `Embedding`.

À présent, examinons de plus près les couches de prétraitement de Keras.

5.3.3 Couches de prétraitement de Keras

L'équipe de TensorFlow travaille sur la mise à disposition de couches standard de prétraitement dans Keras (<https://homl.info/preproc>). Elles seront peut-être disponibles au moment où vous lirez ces lignes, mais l'API actuelle peut évoluer. Si vous constatez des dysfonctionnements, reportez-vous au notebook de ce chapitre. La nouvelle API remplacera probablement l'API Feature Columns existante, qui est difficile à utiliser et moins intuitive (si vous souhaitez néanmoins plus d'informations sur l'API Feature Columns, consultez le notebook de ce chapitre¹³³).

Nous avons déjà abordé deux de ces couches: la couche `keras.layers.Normalization` qui réalisera une standardisation de caractéristique (elle équivaut à la couche `Standardization` définie précédemment), et la couche `layer TextVectorization` qui sera capable d'encoder chaque mot d'entrée en son indice dans le vocabulaire. Dans les deux cas, nous créons la couche, nous invoquons sa méthode `adapt()` avec un échantillon de données, et nous l'utilisons normalement dans notre modèle. Les autres couches de prétraitement suivront ce même schéma.

L'API inclura également une couche `keras.layers.Discretization` qui découpera les données en continu dans différentes corbeilles et encodera chaque corbeille sous forme d'un vecteur *one-hot*. Par exemple, nous pouvons l'employer pour répartir des prix en trois catégories (bas, moyen, haut), qui seront encodées sous la forme [1, 0, 0], [0, 1, 0] et [0, 0, 1], respectivement. Cette méthode perd effectivement une grande quantité d'informations, mais, dans certains cas, elle peut aider le modèle à détecter des motifs qui resteraient invisibles en examinant simplement les valeurs continues.



La couche `Discretization` ne sera pas différentiable et ne doit être utilisée qu'au début du modèle. Puisque les couches de prétraitement du modèle seront figées pendant l'entraînement, leurs paramètres ne seront pas affectés par la descente de gradient et n'ont donc pas besoin d'être différentiables. Cela signifie également que vous ne devez pas employer une couche `Embedding` directement dans une couche de prétraitement personnalisée si vous souhaitez qu'elle soit entraînable. À la place, il faut l'ajouter séparément dans le modèle, comme dans l'exemple de code précédent.

133. Voir « 13_loading_and_preprocessing_data.ipynb » sur <https://github.com/ageron/handson-ml2>.

Il sera également possible de chaîner plusieurs couches de prétraitement avec la classe `PreprocessingStage`. Par exemple, le code suivant crée un pipeline de prétraitement qui commencera par normaliser les entrées, puis les séparera (cela pourrait vous rappeler les pipelines Scikit-Learn). Après l'adaptation du pipeline à un échantillon de données, nous pouvons l'utiliser comme une couche normale dans nos modèles (mais, une fois encore, uniquement au début du modèle, car elle contient une couche de prétraitement non différentiable) :

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

La couche `TextVectorization` aura également la possibilité de produire des vecteurs de compteurs de mots à la place d'indices de mots. Par exemple, si le vocabulaire contient quatre mots, disons `["en", "football", "plus", "de"]`, alors l'expression `"de plus en plus de football"` sera associée au vecteur `[1, 1, 2, 2]`. En effet, `"en"` apparaît une fois, `"football"` apparaît une fois, `"plus"` apparaît deux fois et `"de"` apparaît deux fois. Cette représentation du texte est appelée *sac de mots* (*bag of words*), car elle perd complètement l'ordre des mots. Les mots répandus, comme `"de"`, auront une valeur importante dans la plupart des textes, même s'ils sont généralement les moins intéressants (par exemple, dans le texte `"de plus en plus de football"`, le mot `"football"` est clairement le plus important, précisément parce qu'il n'est pas un terme fréquent). Par conséquent, les compteurs de mots doivent être normalisés de manière à réduire l'importance des mots fréquents. Pour cela, une solution classique consiste à diviser chaque compteur de mots par le logarithme du nombre total d'instances d'entraînement dans lesquelles le mot apparaît¹³⁴. Cette technique est appelée TF-IDF (*Term-Frequency × Inverse-Document-Frequency*). Par exemple, imaginons que les mots `"en"`, `"football"`, `"plus"` et `"de"` apparaissent, respectivement, dans 200, 10, 120 et 300 instances de texte du jeu d'entraînement. Dans ce cas, le vecteur final sera `[1/log(200), 1/log(10), 2/log(120), 2/log(300)]`, c'est-à-dire environ `[0.19, 0.43, 0.42, 0.35]`. La couche `TextVectorization` devrait proposer une option pour appliquer la TF-IDF.



Si les couches de prétraitement standard ne suffisent pas à remplir votre tâche, vous pouvez toujours, comme nous l'avons fait avec la classe `Standardization`, créer votre propre couche de prétraitement personnalisée. Il suffit pour cela de créer une sous-classe de `keras.layers.PreprocessingLayer` dotée d'une méthode `adapt()` qui prend en argument un `data_sample` et, éventuellement, un argument supplémentaire `reset_state`. Si ce dernier vaut `True`, alors la méthode `adapt()` doit réinitialiser tout état existant avant de calculer le nouvel état; s'il vaut `False`, elle doit essayer d'actualiser l'état existant.

¹³⁴. J'utilise ici le logarithme népérien, mais beaucoup de gens utilisent plutôt le logarithme décimal. Cela n'a pas grande importance car le logarithme décimal est proportionnel au logarithme népérien. Notez qu'il existe de nombreuses variantes de TF-IDF (voir la page Wikipédia pour plus de détails : <https://fr.wikipedia.org/wiki/TF-IDF>). Il est probable que Keras utilisera une autre variante que celle décrite ici, mais le principe général sera le même.

Les couches de prétraitement proposées par Keras vont énormément faciliter le prétraitement ! Désormais, que vous décidiez d'écrire vos propres couches de prétraitement ou d'employer celles de Keras (voire même d'utiliser l'API Feature Columns), tout le prétraitement sera effectué à la volée. En revanche, pendant l'entraînement, il peut être préférable de le réaliser à l'avance. Voyons pourquoi et voyons comment procéder.

5.4 TF TRANSFORM

Si le prétraitement exige des calculs intensifs, le réaliser avant l'entraînement plutôt qu'à la volée pourra améliorer énormément les performances : les données seront prétraitées une seule fois par instance *avant* l'entraînement au lieu d'une seule fois par instance et par époque *pendant* l'entraînement. Nous l'avons mentionné précédemment, si le dataset est suffisamment petit pour tenir en mémoire, nous pouvons profiter de sa méthode `cache()`. Mais s'il est trop volumineux, il faudra alors se tourner vers des outils comme Apache Beam et Spark. Ils permettent d'exécuter des pipelines de traitement efficaces sur de grandes quantités de données, même réparties sur plusieurs serveurs. Nous pouvons donc nous en servir pour traiter toutes les données d'entraînement avant l'entraînement.

Cette approche fonctionne bien et accélère évidemment l'entraînement, mais elle présente un inconvénient. Supposons que nous souhaitions déployer le modèle entraîné dans une application mobile. Dans ce cas, nous devons écrire dans notre application du code qui prend en charge le prétraitement des données avant qu'elles ne soient transmises au modèle. Supposons de plus que nous souhaitions également le déployer dans TensorFlow.js afin qu'il s'exécute dans un navigateur web. Là encore, nous devons écrire du code de prétraitement. La maintenance du projet risque de devenir un vrai cauchemar. Dès que nous modifions la logique de prétraitement, nous devons actualiser le code Apache Beam, le code de l'application mobile et le code JavaScript. Cette configuration non seulement demande du temps mais est aussi sujette aux erreurs : nous risquons d'introduire de subtiles différences entre les opérations de prétraitement effectuées avant l'entraînement et celles réalisées dans l'application mobile ou dans le navigateur. Cette *asymétrie entraînement/service (training/serving skew)* va conduire à des bogues ou à des performances dégradées.

Une amélioration possible serait de prendre le modèle entraîné (entraîné sur les données prétraitées par le code Apache Beam ou Spark) et, avant de le déployer dans l'application mobile ou le navigateur, d'ajouter des couches de prétraitement supplémentaires pour effectuer le prétraitement à la volée. Le résultat est bien meilleur, puisque nous avons à présent seulement deux versions du code de prétraitement : celui d'Apache Beam ou de Spark, et celui des couches de prétraitement.

Mais il serait encore mieux que nos opérations de prétraitement ne soient définies qu'une seule fois. C'est là que TF Transform entre en scène. Cette bibliothèque fait partie de TensorFlow Extended (TFX) (<https://tensorflow.org/tfx>), une plateforme complète pour la mise en production de modèles TensorFlow (malheureusement,

à l'heure de l'écriture de ces lignes, TFX n'est pas encore disponible sous Windows, mais cela ne saurait tarder). Tout d'abord, pour utiliser un composant TFX comme TF Transform, il faut l'installer (nous l'avons fait au chapitre 1, lors de la création de l'environnement conda `tf2`). Nous définissons ensuite notre fonction de prétraitement (en Python) une seule fois, en appelant des fonctions de TF Transform pour le redimensionnement, le découpage, etc. Nous pouvons également utiliser n'importe quelle opération TensorFlow utile. Voici un exemple de fonction de prétraitement dans le cas où nous avons deux caractéristiques :

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = un lot de caractéristiques d'entrée
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

TF Transform nous permet ensuite d'appliquer cette fonction `preprocess()` à l'intégralité du jeu d'entraînement en utilisant Apache Beam (la bibliothèque fournit la classe `AnalyzeAndTransformDataset` que nous pouvons employer à cela dans le pipeline Apache Beam). Pendant le traitement, elle calculera également les statistiques indispensables sur l'ensemble du jeu d'entraînement. Dans cet exemple, il s'agit de la moyenne et de l'écart-type de la caractéristique `housing_median_age`, et du vocabulaire pour la caractéristique `ocean_proximity`. Les composants qui établissent ces statistiques sont appelés *analyseurs*.

Il est important de savoir que TF Transform va également générer une fonction TensorFlow équivalente que nous pourrons intégrer au modèle à déployer. Cette fonction TF comprend des constantes qui correspondent à toutes les statistiques indispensables calculées par Apache Beam (la moyenne, l'écart-type et le vocabulaire).

Armés de l'API Data, des fichiers TFRecord, des couches de prétraitement de Keras et de TF Transform, nous sommes en mesure de construire des pipelines d'entrée fortement évolutifs pour l'entraînement et de bénéficier d'un prétraitement des données rapide et portable en production.

Mais, si nous voulons juste utiliser un jeu de donné standard, les choses sont beaucoup plus simples : il suffit d'avoir recours à TFDS !

5.5 LE PROJET TENSORFLOW DATASETS (TFDS)

Le projet TensorFlow Datasets (<https://tensorflow.org/datasets>) a pour objectif de faciliter le téléchargement de jeux de données classiques, des plus petits, comme MNIST ou Fashion MNIST, aux très volumineux, comme ImageNet. Les jeux de données disponibles concernent les images, le texte (y compris les traductions), l'audio et la

vidéo. Pour en consulter la liste complète, avec la description de chacun, rendez-vous sur <https://homl.info/tfds>.

TFDS n'étant pas livré avec TensorFlow, il faut installer la bibliothèque `tensorflow-datasets` (c'est l'une des librairies installées au chapitre 1 lors de la création de l'environnement `conda tf2`). Il suffit d'appeler ensuite la fonction `tfds.load()` pour que les données demandées soient téléchargées (excepté si elles l'ont déjà été précédemment) et pour les obtenir sous forme d'un dictionnaire de datasets (en général un pour l'entraînement et un pour les tests, mais cela dépend du jeu de données choisi). Prenons MNIST comme exemple :

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

Nous pouvons ensuite appliquer n'importe quelle transformation (le plus souvent mélange, mise en lot et prélecture), avant d'être prêts à entraîner le modèle :

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```



La fonction `load()` mélange chaque portion de données qu'elle télécharge (uniquement pour le jeu d'entraînement). Puisque cela pourrait ne pas suffire, il est préférable de mélanger un peu plus les données d'entraînement.

Chaque élément du dataset est un dictionnaire qui contient les caractéristiques et les étiquettes. Cependant, Keras attend que chaque élément soit un tuple contenant deux éléments (de nouveau les caractéristiques et les étiquettes). Nous pouvons transformer le dataset en invoquant la méthode `map()` :

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Mais il est plus simple de laisser fonction `load()` s'en charger à notre place, en indiquant `as_supervised=True` (évidemment, cela ne vaut que pour les jeux de données étiquetés). Si nécessaire, nous pouvons également préciser la taille du lot. Ensuite, nous passons le dataset directement au modèle `tf.keras` :

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

En raison du contenu plutôt technique de ce chapitre, vous vous sentez peut-être un peu loin de la beauté abstraite des réseaux de neurones. Il n'en reste pas moins que le Deep Learning implique souvent de grandes quantités de données et qu'il est indispensable de savoir comment les charger, les analyser et les prétraiter efficacement.

Dans le chapitre suivant, nous étudierons les réseaux de neurones convolutifs, qui font partie des architectures parfaitement adaptées au traitement d'images et à de nombreuses autres applications.

5.6 EXERCICES

1. Pourquoi voudriez-vous utiliser l'API Data ?
2. Quels sont les avantages du découpage d'un jeu de données volumineux en plusieurs fichiers ?
3. Pendant l'entraînement, comment pouvez-vous savoir que le pipeline d'entrée constitue le goulot d'étranglement ? Comment pouvez-vous corriger le problème ?
4. Un fichier TFRecord peut-il contenir n'importe quelles données binaires ou uniquement des protobufs sérialisés ?
5. Pourquoi vous embêter à convertir toutes vos données au format du protobuf `Example` ? Pourquoi ne pas utiliser votre propre définition de protobuf ?
6. Avec les fichiers TFRecord, quand faut-il activer la compression ? Pourquoi ne pas le faire systématiquement ?
7. Les données peuvent être prétraitées directement pendant l'écriture des fichiers de données, au sein du pipeline `tf.data`, dans des couches de prétraitement à l'intérieur du modèle ou en utilisant TF Transform. Donnez quelques avantages et inconvénients de chaque approche.
8. Nommez quelques techniques classiques d'encodage des caractéristiques de catégorie. Qu'en est-il du texte ?
9. Chargez le jeu de données Fashion MNIST (présenté au chapitre 2). Séparez-le en un jeu d'entraînement, un jeu de validation et un jeu de test. Mélangez le jeu d'entraînement. Enregistrez les datasets dans plusieurs fichiers TFRecord. Chaque enregistrement doit être un protobuf `Example` sérialisé avec deux caractéristiques : l'image sérialisée (utilisez pour cela `tf.io.serialize_tensor()`) et l'étiquette¹³⁵. Servez-vous ensuite de `tf.data` pour créer un dataset efficace correspondant à chaque jeu. Enfin, employez un modèle Keras pour entraîner ces datasets, sans oublier une couche de prétraitement afin de standardiser chaque caractéristique d'entrée. Essayez de rendre le pipeline d'entrée aussi efficace que possible, en visualisant les données de profilage avec TensorBoard.
10. Dans cet exercice, vous allez télécharger un jeu de données, le découper, créer un `tf.data.Dataset` pour le charger et le prétraiter efficacement, puis vous construirez et entraînerez un modèle de classification binaire contenant une couche `Embedding` :

135. Pour les grandes images, vous pouvez utiliser à la place `tf.io.encode_jpeg()`. Vous économisez ainsi beaucoup d'espace, mais vous perdrez un peu en qualité d'image.

- a. Téléchargez le jeu de données Large Movie Review Dataset (<https://homl.info/imdb>), qui contient 50 000 critiques de films provenant de la base de données Internet Movie Database (<https://imdb.com/>). Les données sont réparties dans deux dossiers, *train* et *test*, chacun contenant un sous-dossier *pos* de 12 500 critiques positives et un sous-dossier *neg* de 12 500 critiques négatives. Chaque avis est stocké dans un fichier texte séparé. Le jeu de données comprend d'autres fichiers et dossiers (y compris des sacs de mots prétraités), mais nous allons les ignorer dans cet exercice.
- b. Découpez le jeu de test en un jeu de validation (15 000) et un jeu de test (10 000).
- c. Utilisez `tf.data` afin de créer un dataset efficace pour chaque jeu.
- d. Créez un modèle de classification binaire, en utilisant une couche `TextVectorization` pour prétraiter chaque critique. Si cette couche n'est pas encore disponible (ou si vous aimez les défis), développez votre propre couche de prétraitement personnalisée : vous pouvez employer les fonctions du package `tf.strings`, par exemple `lower()` pour tout mettre en minuscules, `regex_replace()` pour remplacer la ponctuation par des espaces, et `split()` pour scinder le texte en fonction des espaces. Vous devez utiliser une table de correspondance afin de produire des indices de mots, qui doivent être préparés dans la méthode `adapt()`.
- e. Ajoutez une couche `Embedding` et calculez le plongement moyen pour chaque critique, multiplié par la racine carrée du nombre de mots (voir le chapitre 8). Ce plongement moyen redimensionné peut ensuite être passé au reste du modèle.
- f. Entraînez le modèle et voyez la précision obtenue. Essayez d'optimiser les pipelines pour que l'entraînement soit le plus rapide possible.
- g. Utilisez TFDS pour charger plus facilement le même jeu de données : `tfds.load("imdb_reviews")`.

Les solutions de ces exercices sont données à l'annexe A.

6

Vision par ordinateur et réseaux de neurones convolutifs

Si la victoire du superordinateur Deep Blue d'IBM sur le champion du monde des échecs Garry Kasparov remonte à 1996, ce n'est que récemment que des ordinateurs ont été capables d'effectuer des tâches d'apparence triviale, comme repérer un chat sur une photo ou reconnaître les mots prononcés. Pourquoi sommes-nous capables, nous êtres humains, d'effectuer ces tâches sans efforts ? La réponse tient dans le fait que cela se passe largement en dehors du domaine de notre conscience, à l'intérieur de modules sensoriels spécialisés de notre cerveau, par exemple dans la vision ou l'audition. Au moment où cette information sensorielle atteint notre conscience, elle a déjà été largement ornée de caractéristiques de haut niveau. Par exemple, lorsque nous regardons la photo d'un mignon petit chaton, nous ne pouvons pas décider de ne *pas* voir le chaton ou de ne *pas* remarquer qu'il est mignon. Nous ne pouvons pas plus expliquer *comment* reconnaître un mignon chaton : c'est juste évident pour nous. Ainsi, nous ne pouvons pas faire confiance à notre expérience subjective. La perception n'est pas du tout triviale et, pour la comprendre, nous devons examiner le fonctionnement des modules sensoriels.

Les réseaux de neurones convolutifs (CNN, *Convolutional Neural Network*) sont apparus à la suite de l'étude du cortex visuel du cerveau et sont utilisés dans la reconnaissance d'images depuis les années 1980. Ces quelques dernières années, grâce à l'augmentation de la puissance de calcul, à la quantité de données d'entraînement disponibles et aux astuces présentées au chapitre 3 pour l'entraînement des réseaux profonds, les CNN ont été capables de performances surhumaines sur des tâches visuelles complexes. Ils sont au cœur des services de recherche d'images, des voitures autonomes, des systèmes de classification automatique des vidéos, etc. De plus, ils ne se limitent pas à la perception visuelle, mais servent également dans

d'autres domaines, comme la reconnaissance vocale ou le traitement automatique du langage naturel (TALN) ; nous nous focaliserons sur les applications visuelles.

Dans ce chapitre, nous présenterons l'origine des CNN, les éléments qui les constituent et leur implémentation avec TensorFlow et Keras. Puis nous décrirons certaines des meilleures architectures de CNN, ainsi que d'autres tâches visuelles, notamment la détection d'objets (la classification de plusieurs objets dans une image et leur délimitation par des rectangles d'encadrement) et la segmentation sémantique (la classification de chaque pixel en fonction de la classe de l'objet auquel il appartient).

6.1 L'ARCHITECTURE DU CORTEX VISUEL

En 1958¹³⁶ et en 1959¹³⁷, David H. Hubel et Torsten Wiesel ont mené une série d'expériences sur des chats (et, quelques années plus tard, sur des singes¹³⁸), apportant des informations essentielles sur la structure du cortex visuel (en 1981, ils ont reçu le prix Nobel de physiologie ou médecine pour leurs travaux). Ils ont notamment montré que de nombreux neurones du cortex visuel ont un petit *champ récepteur local* et qu'ils réagissent donc uniquement à un stimulus visuel qui se trouve dans une région limitée du champ visuel (voir la figure 6.1, sur laquelle les champs récepteurs locaux de cinq neurones sont représentés par les cercles en pointillé). Les champs récepteurs des différents neurones peuvent se chevaucher et ils couvrent ensemble l'intégralité du champ visuel.

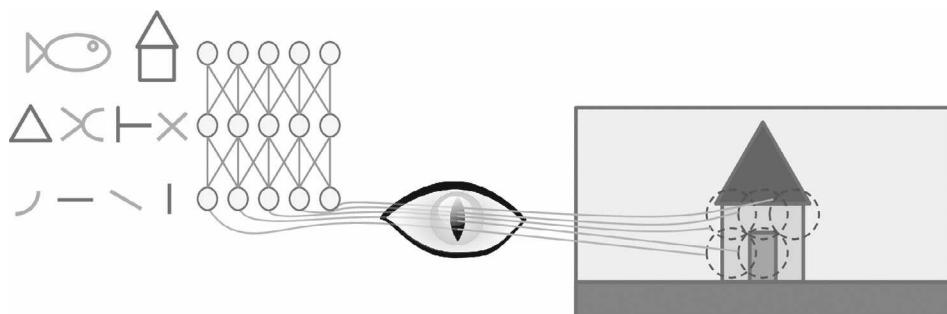


Figure 6.1 – Les neurones biologiques du cortex visuel répondent à des motifs spécifiques dans de petites régions du champ visuel appelées champs récepteurs; au fur et à mesure que le signal visuel traverse les modules cérébraux consécutifs, les neurones répondent à des motifs plus complexes dans des champs récepteurs plus larges

136. David H. Hubel, « Single Unit Activity in Striate Cortex of Unrestrained Cats », *The Journal of Physiology*, 147 (1959), 226-238 : <https://homl.info/71>.

137. David H. Hubel et Torsten N. Wiesel, « Receptive Fields of Single Neurons in the Cat's Striate Cortex », *The Journal of Physiology*, 148 (1959), 574-591 : <https://homl.info/72>.

138. David H. Hubel et Torsten N. Wiesel, « Receptive Fields and Functional Architecture of Monkey Striate Cortex », *The Journal of Physiology*, 195 (1968), 215-243 : <https://homl.info/73>.

Ils ont également montré que certains neurones réagissent uniquement aux images de lignes horizontales, tandis que d'autres réagissent uniquement aux lignes ayant d'autres orientations (deux neurones peuvent avoir le même champ récepteur mais réagir à des orientations de lignes différentes). Ils ont remarqué que certains neurones ont des champs récepteurs plus larges et qu'ils réagissent à des motifs plus complexes, correspondant à des combinaisons de motifs de plus bas niveau. Ces observations ont conduit à l'idée que les neurones de plus haut niveau se fondent sur la sortie des neurones voisins de plus bas niveau (sur la figure 6.1, chaque neurone est connecté uniquement à quelques neurones de la couche précédente). Cette architecture puissante est capable de détecter toutes sortes de motifs complexes dans n'importe quelle zone du champ visuel.

Ces études du cortex visuel sont à l'origine du neocognitron¹³⁹, présenté en 1980, qui a progressivement évolué vers ce que nous appelons aujourd'hui *réseau de neurones convolutif*. Un événement majeur a été la publication d'un article¹⁴⁰ en 1998 par Yann LeCun *et al.*, dans lequel les auteurs ont présenté la célèbre architecture LeNet-5, largement employée par les banques pour reconnaître les numéros de chèques écrits à la main. Nous connaissons déjà quelques blocs de construction de cette architecture, comme les couches intégralement connectées (FC, *fully connected*) et les fonctions d'activation sigmoïdes, mais elle en ajoute deux nouveaux : les *couches de convolution* et les *couches de pooling*. Étudions-les.



Pourquoi, pour les tâches de reconnaissance d'images, ne pas simplement utiliser un réseau de neurones profond avec des couches intégralement connectées ? Malheureusement, bien qu'un tel réseau convienne parfaitement aux petites images (par exemple, celles du jeu MNIST), il n'est pas adapté aux images plus grandes en raison de l'énorme quantité de paramètres qu'il exige. Par exemple, une image 100×100 est constituée de 10 000 pixels et, si la première couche comprend uniquement 1 000 neurones (ce qui limite déjà beaucoup la quantité d'informations transmises à la couche suivante), cela donne 10 millions de connexions. Et nous ne parlons que de la première couche. Les CNN résolvent ce problème en utilisant des couches partiellement connectées et en partageant des poids.

6.2 COUCHES DE CONVOLUTION

La *couche de convolution*¹⁴¹ est le bloc de construction le plus important d'un CNN. Dans la première couche de convolution, les neurones ne sont pas connectés à

139. Kunihiko Fukushima, « Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position », *Biological Cybernetics*, 36 (1980), 193-202 : <https://homl.info/74>.

140. Yann LeCun *et al.*, « Gradient-Based Learning Applied to Document Recognition », *Proceedings of the IEEE*, 86, n° 11 (1998), 2278-2324 : <https://homl.info/75>.

141. Une convolution est une opération mathématique qui fait glisser une fonction par-dessus une autre et mesure l'intégrale de leur multiplication ponctuelle. Ses liens avec les transformées de Fourier et de Laplace sont étroits. Elle est souvent employée dans le traitement du signal. Les couches de convolution utilisent en réalité des corrélations croisées, qui sont très similaires aux convolutions (pour de plus amples informations, voir <https://homl.info/76>).

chaque pixel de l'image d'entrée (comme c'était le cas dans les chapitres précédents), mais uniquement aux pixels dans leurs champs récepteurs (voir la figure 6.2). À leur tour, les neurones de la deuxième couche de convolution sont chacun connectés uniquement aux neurones situés à l'intérieur d'un petit rectangle de la première couche. Cette architecture permet au réseau de se focaliser sur des caractéristiques de bas niveau dans la première couche cachée, puis de les assembler en caractéristiques de plus haut niveau dans la couche cachée suivante, etc. Cette structure hiérarchique est récurrente dans les images réelles et c'est l'une des raisons des bons résultats des CNN pour la reconnaissance d'images.

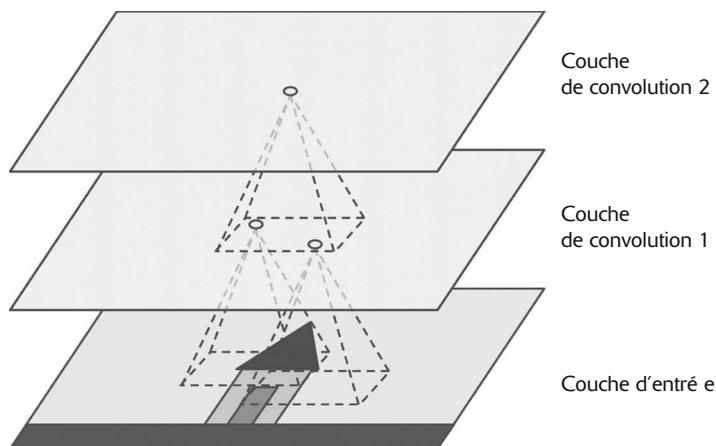


Figure 6.2 – Couches d'un CNN avec des champs récepteurs locaux rectangulaires



Jusque-là, tous les réseaux de neurones multicouches que nous avons examinés avaient des couches constituées d'une longue suite de neurones et nous devions donc aplatis les images d'entrée en une dimension avant de les transmettre au réseau. Dans un CNN, chaque couche étant représentée en deux dimensions, il est plus facile de faire correspondre les neurones aux entrées associées.

Un neurone situé en ligne i et colonne j d'une couche donnée est connecté aux sorties des neurones de la couche précédente situés aux lignes i à $i + f_h - 1$ et aux colonnes j à $j + f_w - 1$, où f_h et f_w sont la hauteur et la largeur du champ récepteur (voir la figure 6.3). Pour qu'une couche ait les mêmes hauteur et largeur que la couche précédente, on ajoute des zéros autour des entrées (marge), comme l'illustre la figure. Cette opération se nomme *remplissage par zéros* (*zero padding*), mais nous dirons plutôt *ajout d'une marge de zéros* pour éviter toute confusion (car seule la marge est remplie par des zéros).

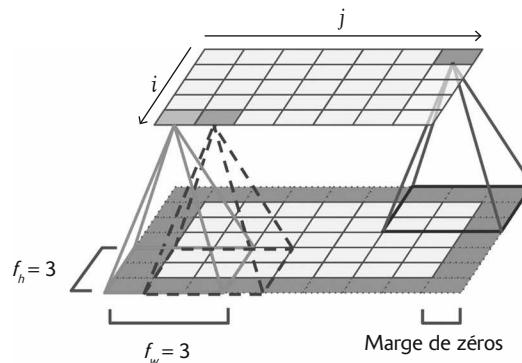


Figure 6.3 – Connexions entre les couches et marge de zéros

Il est également possible de connecter une large couche d'entrée à une couche plus petite en espaçant les champs récepteurs (voir la figure 6.4). Cela permet de réduire énormément la complexité des calculs du modèle. L'espacement de deux champs récepteurs consécutifs est appelé le *pas* (*stride* en anglais). Sur la figure, une couche d'entrée 5×7 (plus la marge) est connectée à une couche 3×4 , en utilisant des champs récepteurs 3×3 et un pas de 2 (dans cet exemple, le pas est identique dans les deux directions, mais ce n'est pas obligatoire). Un neurone situé en ligne i et colonne j dans la couche supérieure est connecté aux sorties des neurones de la couche précédente situés aux lignes $i \times s_h$ à $i \times s_h + f_h - 1$ et colonnes $j \times s_w$ à $j \times s_w + f_w - 1$, où s_h et s_w sont les pas vertical et horizontal.

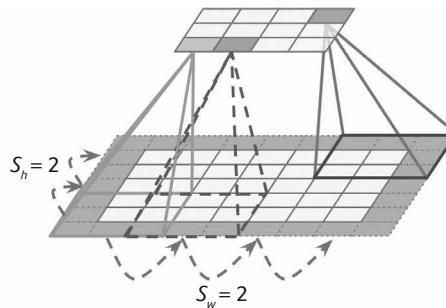


Figure 6.4 – Diminution de la dimensionnalité grâce à un pas de 2

6.2.1 Filtres

Les poids d'un neurone peuvent être représentés sous la forme d'une petite image de la taille du champ récepteur. Par exemple, la figure 6.5 montre deux ensembles de poids possibles, appelés *filtres* (ou *noyaux de convolution*). Le premier est un carré noir avec une ligne blanche verticale au milieu (il s'agit d'une matrice 7×7 remplie de 0, à l'exception de la colonne centrale, pleine de 1). Les neurones qui utilisent

ces poids ignoreront tout ce qui se trouve dans leur champ récepteur, à l'exception de la ligne verticale centrale (puisque toutes les entrées seront multipliées par zéro, excepté celles situées sur la ligne verticale centrale). Le second filtre est un carré noir traversé en son milieu par une ligne blanche horizontale. À nouveau, les neurones qui utilisent ces poids ignoreront tout ce qui se trouve dans leur champ récepteur, hormis cette ligne horizontale centrale.

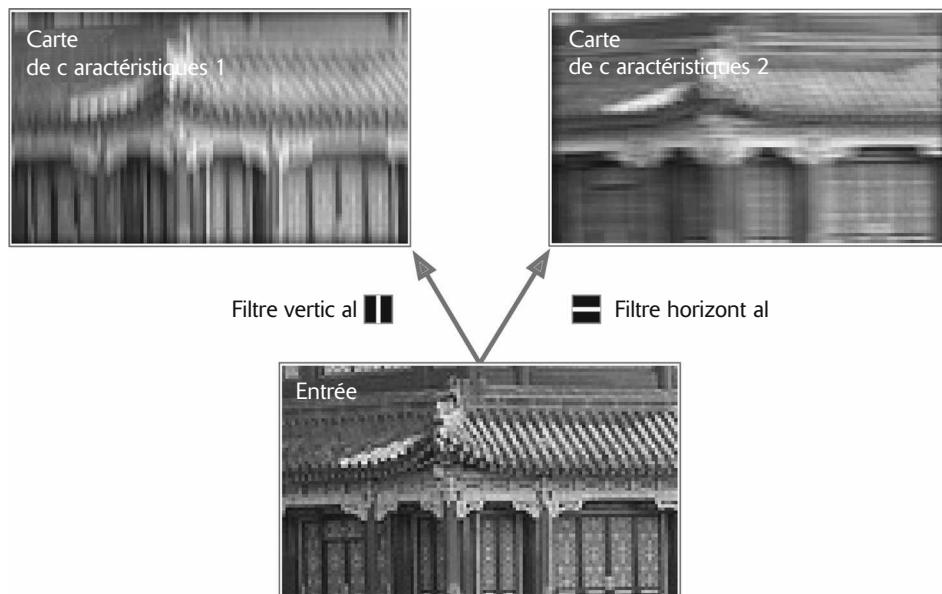


Figure 6.5 – Application de deux filtres différents pour obtenir deux cartes de caractéristiques

Si tous les neurones d'une couche utilisent le même filtre à ligne verticale (ainsi que le même terme constant) et si nous fournissons en entrée du réseau l'image illustrée à la figure 6.5 (image du bas), la couche sortira l'image située en partie supérieure gauche. Les lignes blanches verticales sont mises en valeur, tandis que le reste devient flou. De façon comparable, l'image supérieure droite est obtenue lorsque tous les neurones utilisent le filtre à ligne horizontale. Les lignes blanches horizontales sont améliorées, tandis que le reste est flou. Une couche remplie de neurones qui utilisent le même filtre nous donne ainsi une *carte de caractéristiques* (*feature map*) qui fait ressortir les zones d'une image qui se rapprochent le plus du filtre. Évidemment, nous n'avons pas à définir des filtres manuellement. À la place, au cours de l'entraînement, la couche de convolution apprend automatiquement les filtres qui seront les plus utiles à sa tâche, et les couches supérieures apprennent à les combiner dans des motifs plus complexes.

6.2.2 Empiler plusieurs cartes de caractéristiques

Jusqu'à présent, pour une question de simplicité, nous avons représenté la sortie de chaque couche de convolution comme une couche à deux dimensions, mais, en réalité, elle est constituée de plusieurs filtres (vous décidez du nombre) et produit une carte de caractéristiques par filtre. Il est donc plus correct de la représenter en trois dimensions (voir la figure 6.6). Dans chaque carte de caractéristiques, nous avons un neurone par pixel et tous les neurones d'une carte donnée partagent les mêmes paramètres (poids et terme constant). Mais les neurones dans différentes cartes de caractéristiques utilisent des paramètres distincts. Le champ récepteur d'un neurone est tel que nous l'avons décrit précédemment, mais il s'étend sur toutes les cartes de caractéristiques des couches précédentes. En résumé, une couche de convolution applique simultanément plusieurs filtres entraînables à ses entrées, ce qui lui permet de détecter plusieurs caractéristiques n'importe où dans ses entrées.

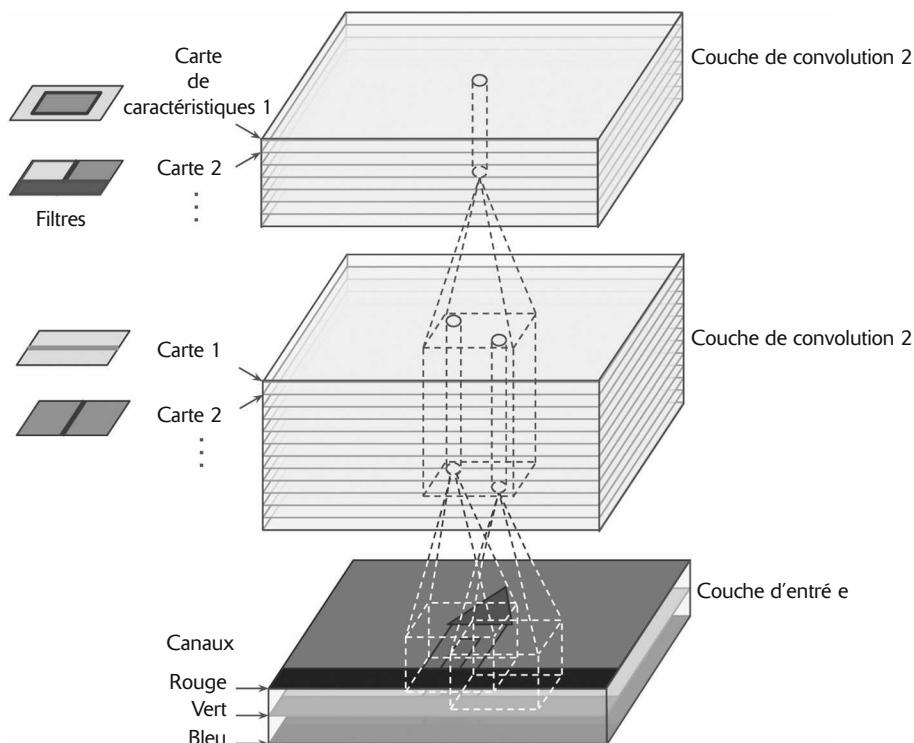


Figure 6.6 – Couches de convolution avec plusieurs cartes de caractéristiques et images à trois canaux de couleur



Puisque tous les neurones d'une carte de caractéristiques partagent les mêmes paramètres, le nombre de paramètres du modèle s'en trouve considérablement réduit. Dès que le CNN a appris à reconnaître un motif en un endroit, il peut le reconnaître partout ailleurs. À l'opposé, lorsqu'un RNP classique a appris à reconnaître un motif en un endroit, il ne peut le reconnaître qu'en cet endroit précis.

Les images d'entrée sont également constituées de multiples sous-couches, une par *canal de couleur*. Classiquement, il en existe trois: rouge, vert et bleu (RVB, ou RGB de l'anglais Red, Green, Blue). Les images en niveaux de gris en possèdent un seul. D'autres types d'images en ont beaucoup plus, par exemple les images satellite qui capturent des fréquences lumineuses supplémentaires comme l'infrarouge.

Plus précisément, un neurone situé en ligne i et colonne j de la carte de caractéristiques k dans une couche de convolution l est relié aux sorties des neurones de la couche précédente $l - 1$, situés aux lignes $i \times s_h$ à $i \times s_h + f_h - 1$ et aux colonnes $j \times s_w$ à $j \times s_w + f_w - 1$, sur l'ensemble des cartes de caractéristiques (de la couche $l - 1$). Tous les neurones situés sur les mêmes ligne i et colonne j , mais dans des cartes de caractéristiques différentes, sont connectés aux sorties des mêmes neurones dans la couche précédente.

L'équation 6.1 résume toutes les explications précédentes en une unique équation mathématique. Elle montre comment calculer la sortie d'un neurone donné dans une couche de convolution. Elle est un peu laide en raison de tous les indices, mais elle ne fait que calculer la somme pondérée de toutes les entrées, plus le terme constant.

Équation 6.1 – Calcul de la sortie d'un neurone dans une couche de convolution

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n'-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{avec} \quad \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

Dans cette équation :

- $z_{i,j,k}$ correspond à la sortie du neurone situé en ligne i et colonne j dans la carte de caractéristiques k de la couche de convolution (couche l).
- Comme nous l'avons expliqué, s_h et s_w sont les pas vertical et horizontal, f_h et f_w sont la hauteur et la largeur du champ récepteur, et f_n' est le nombre de cartes de caractéristiques dans la couche précédente (couche $l - 1$).
- $x_{i',j',k'}$ correspond à la sortie du neurone situé dans la couche $l - 1$, ligne i' , colonne j' , carte de caractéristiques k' (ou canal k' si la couche précédente est la couche d'entrée).
- b_k est le terme constant de la carte de caractéristiques k (dans la couche l). Il peut être vu comme un réglage de la luminosité globale de la carte de caractéristiques k .
- $w_{u,v,k',k}$ correspond au poids de la connexion entre tout neurone de la carte de caractéristiques k de la couche l et son entrée située ligne u , colonne v (relativement au champ récepteur du neurone) dans la carte de caractéristiques k' .

6.2.3 Implémentation avec TensorFlow

Dans TensorFlow, chaque image d'entrée est généralement représentée par un tenseur à trois dimensions de forme [*hauteur*, *largeur*, *nombre de canaux*]. Un mini-lot est représenté par un tenseur à quatre dimensions de forme [*taille du mini-lot*, *hauteur*, *largeur*, *nombre de canaux*]. Les poids d'une couche de convolution sont représentés par un tenseur à quatre dimensions de forme [f_h, f_w, f_n, f_n], et les termes constants, par un tenseur à une dimension de forme [f_n].

Examinons un exemple simple. Le code suivant charge deux images en couleur (l'une d'un temple chinois, l'autre d'une fleur) à l'aide de la fonction `load_sample_images()` de Scikit-Learn, puis crée deux filtres, les applique aux deux images, et termine en affichant l'une des cartes de caractéristiques résultantes (notez que la fonction `load_sample_images()` repose sur la librairie Pillow, installée au chapitre 1):

```
from sklearn.datasets import load_sample_image

# Charger les images d'exemple
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Créer deux filtres
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # Ligne verticale
filters[3, :, :, 1] = 1 # Ligne horizontale

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

# Afficher la 2e carte de caractéristiques de la 1re image
plt.imshow(outputs[0, :, :, 1], cmap="gray")
plt.show()
```

Examinons ce code :

- Pour chaque canal de couleur, l'intensité d'un pixel est représentée par un octet dont la valeur est entre 0 et 255. Nous redimensionnons donc ces caractéristiques en les divisant simplement par 255, pour obtenir des nombres réels entre 0 et 1.
- Nous créons ensuite des filtres 7×7 (l'un avec une ligne blanche verticale centrale, l'autre avec une ligne blanche horizontale centrale).
- Nous les appliquons aux deux images avec la fonction `tf.nn.conv2d()`, qui fait partie de l'API de bas niveau pour le Deep Learning de TensorFlow. Dans cet exemple, nous utilisons une marge de zéros (`padding="SAME"`) et un pas de 1.
- Enfin, nous affichons l'une des cartes de caractéristiques résultantes (comparable à l'image en partie supérieure droite de la figure 6.5).

La ligne `tf.nn.conv2d()` mérite quelques explications supplémentaires :

- `images` est le mini-lot d'entrée (un tenseur à quatre dimensions, comme expliqué précédemment).
- `filters` correspond à l'ensemble des filtres à appliquer (également un tenseur à quatre dimensions, comme expliqué précédemment).

- `strides` est égal à 1, mais il pourrait s'agir d'un tableau à une dimension de quatre éléments, dans lequel les deux éléments centraux donnent les pas vertical et horizontal (s_h et s_w). Les premier et dernier éléments doivent être, pour le moment, égaux à 1. Ils seront peut-être utilisés un jour pour préciser un pas de lot (pour sauter certaines instances) et un pas de canal (pour sauter certaines cartes de caractéristiques ou canaux de la couche précédente).
- `padding` doit avoir pour valeur "SAME" ou "VALID" :
 - S'il est fixé à "SAME", la couche de convolution ajoute une marge de zéros si nécessaire. La taille de la sortie est fixée au nombre de neurones d'entrée divisé par le pas et arrondi à l'entier supérieur. Par exemple, si la taille d'entrée est 13 et le pas est 5 (voir la figure 6.7), alors la sortie aura une taille de 3 (c'est-à-dire, $13/5 = 2,6$, arrondi à 3). Puis des zéros sont ajoutés aussi uniformément que possible autour des entrées. Si `strides` vaut 1, les sorties de la couche ont les mêmes dimensions spatiales (largeur et hauteur) que ses entrées, d'où le nom *same*.
 - S'il est fixé à "VALID", la couche de convolution n'utilise *pas* de marge de zéros et peut ignorer certaines lignes et colonnes dans les parties inférieure et droite de l'image d'entrée, en fonction du pas, comme illustré à la figure 6.7 (pour des raisons de simplicité, nous montrons uniquement la dimension horizontale, mais la même logique s'applique évidemment à la dimension verticale). Cela signifie que le champ récepteur de chaque neurone se trouve strictement dans les positions valides au sein de l'entrée (il ne dépasse pas les limites), d'où le nom *valid*.

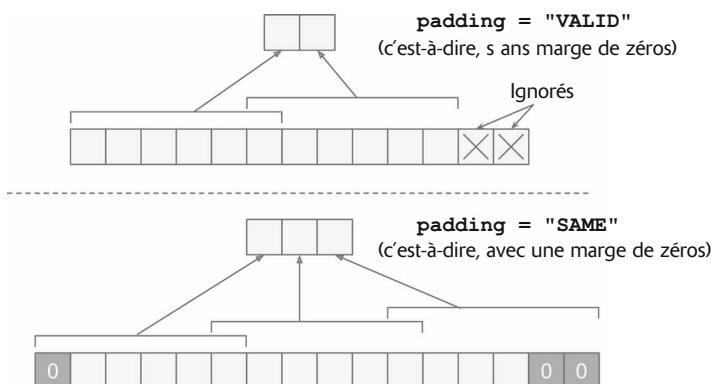


Figure 6.7 – Options de remplissage "SAME" et "VALID", avec une largeur d'entrée de 13, une largeur de filtre de 6, et un pas de 5

Dans cet exemple, nous avons créé les filtres manuellement, mais dans un vrai CNN ils seraient définis comme des variables entraînables afin que le réseau de neurones apprenne ceux qui conviennent. Au lieu de créer les variables manuellement, nous pouvons utiliser la couche `keras.layers.Conv2D` (notez que TensorFlow attend des majuscules tandis que l'API Keras utilise des arguments en minuscules, tels que "same", mais `tf.keras` supporte également les majuscules "SAME") :

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

Ce code crée une couche Conv2D constituée de 32 filtres, chacun de taille 3×3 , avec un pas de 1 (tant horizontalement que verticalement) et le remplissage "same". La fonction d'activation ReLU est appliquée à ses sorties. Les couches de convolution font intervenir un nombre d'hyperparamètres non négligeable. Il faut choisir le nombre de filtres, leur hauteur et largeur, les pas et le type de remplissage. Comme toujours, la validation croisée peut aider à trouver la bonne valeur d'un hyperparamètre, mais cela prend beaucoup de temps. Nous étudierons plus loin des architectures de CNN répandues afin d'avoir une idée des valeurs des hyperparamètres qui conviennent bien en pratique.

6.2.4 Besoins en mémoire

Les CNN posent un autre problème : les couches de convolution ont besoin d'une grande quantité de RAM, notamment au cours de l'entraînement, car la rétro-propagation utilise toutes les valeurs intermédiaires calculées pendant la passe en avant.

Prenons, par exemple, une couche de convolution dotée de filtres 5×5 , produisant 200 cartes de caractéristiques de taille 150×100 , avec un pas de 1 et un remplissage "same". Si l'entrée est une image RVB (trois canaux) de 150×100 , alors le nombre de paramètres est $(5 \times 5 \times 3 + 1) \times 200 = 15\,200$ (le + 1 correspond au terme constant), ce qui est relativement faible par rapport à une couche intégralement connectée¹⁴². Cependant, chacune des 200 cartes de caractéristiques contient 150×100 neurones, et chacun de ces neurones doit calculer une somme pondérée de ses $5 \times 5 \times 3 = 75$ entrées. Cela fait un total de 225 millions de multiplications de nombres à virgule flottante. C'est mieux qu'une couche intégralement connectée mais cela reste un calcul assez lourd. Par ailleurs, si les cartes de caractéristiques sont représentées par des flottants sur 32 bits, alors la sortie de la couche de convolution occupera $200 \times 150 \times 100 \times 32 = 96$ millions de bits (environ 12 Mo) de RAM¹⁴³. Et cela ne concerne qu'une seule instance ! Si un lot d'entraînement comprend 100 instances, alors cette couche aura besoin de 1,2 Go de RAM !

Au cours d'une inférence (c'est-à-dire lors d'une prédiction pour une nouvelle instance), la RAM occupée par une couche sera libérée aussitôt que la couche suivante aura été calculée. Nous n'avons donc besoin que de la quantité de RAM nécessaire à deux couches consécutives. Mais, au cours de l'entraînement, tous les calculs effectués pendant la passe en avant doivent être conservés pour la passe en arrière. La quantité de RAM requise est alors au moins égale à la quantité totale de RAM nécessaire à toutes les couches.

142. Une couche intégralement connectée avec 150×100 neurones, chacun connecté à l'ensemble des $150 \times 100 \times 3$ entrées, aurait $150^2 \times 100^2 \times 3 = 675$ millions de paramètres !

143. Dans le système international d'unités (SI), 1 Mo = 1000 ko = 1000×1000 octets = $1000 \times 1000 \times 8$ bits.



Si l'entraînement échoue à cause d'un manque de mémoire, vous avez plusieurs options : essayer de réduire la taille du mini-lot, tenter de diminuer la dimensionnalité en appliquant un pas ou en retirant quelques couches, passer à des nombres à virgule flottante sur 16 bits à la place de 32 bits, ou encore distribuer le CNN sur plusieurs processeurs.

Exammons à présent le deuxième bloc de construction le plus répandu dans les CNN : la *couche de pooling*.

6.3 COUCHE DE POOLING

Lorsque le fonctionnement des couches de convolution est compris, celui des couches de pooling ne devrait poser aucune difficulté. Ces couches ont pour objectif de sous-échantillonner (c'est-à-dire rétrécir) l'image d'entrée afin de réduire la charge de calcul, l'utilisation de la mémoire et le nombre de paramètres (limitant ainsi le risque de surajustement).

Comme dans les couches de convolution, chaque neurone d'une couche de pooling est connecté aux sorties d'un nombre limité de neurones de la couche précédente, situés à l'intérieur d'un petit champ récepteur rectangulaire. On doit à nouveau définir sa taille, le pas et le type de remplissage. En revanche, un neurone de pooling ne possède aucun poids et se contente d'agrégier les entrées en utilisant une fonction d'agrégation, comme la valeur maximale ou la moyenne.

La figure 6.8 illustre une *couche de pooling maximum (max pooling)*, qui est la plus répandue. Dans cet exemple, on utilise un *noyau de pooling* de 2×2^{144} , un pas de 2 et aucune marge de zéros. Dans chaque champ récepteur, seule la valeur d'entrée maximale permet le passage à la couche suivante. Les autres entrées sont ignorées. Par exemple, dans le champ récepteur inférieur gauche de la figure 6.8, les valeurs d'entrée sont 1, 5, 3, 2, et seule la valeur maximale, 5, est propagée à la couche suivante. En raison du pas de 2, l'image de sortie a une hauteur et une largeur de moitié inférieures à celles de l'image d'entrée (arrondies à l'entier inférieur puisque aucune marge n'est utilisée).

Outre la réduction des calculs nécessaires, de la quantité de mémoire requise et du nombre de paramètres, une couche de pooling maximum ajoute également un certain degré d'*invariance* envers les petites translations (voir la figure 6.9). Dans ce cas, nous supposons que les pixels clairs ont une valeur inférieure aux pixels sombres. Nous prenons trois images (A, B, C) transmises à une couche de pooling maximum, avec un noyau 2×2 et un pas de 2. Les images B et C sont identiques à l'image A, mais sont décalées d'un et de deux pixels vers la droite. Pour les images A et B, les sorties de la couche de pooling maximum sont identiques. Voilà la signification de l'invariance de translation. Pour l'image C, la sortie est différente : elle est décalée d'un pixel vers la droite (mais il reste toujours 75 % d'invariance). En insérant dans un CNN une couche de pooling maximum toutes les deux ou trois couches, il est possible d'obtenir un certain degré d'invariance de translation à plus grande échelle.

¹⁴⁴. Les noyaux de pooling sont simplement des fenêtres glissantes sans état. Ils n'ont pas de poids, contrairement aux autres noyaux que nous avons présentés jusqu'ici.

Par ailleurs, une couche de pooling maximum permet également d'obtenir une petite quantité d'invariance de rotation et une légère invariance d'échelle. De telles invariances, bien que limitées, peuvent se révéler utiles dans les cas où la prédiction ne doit pas dépendre de ces détails, comme dans les tâches de classification.

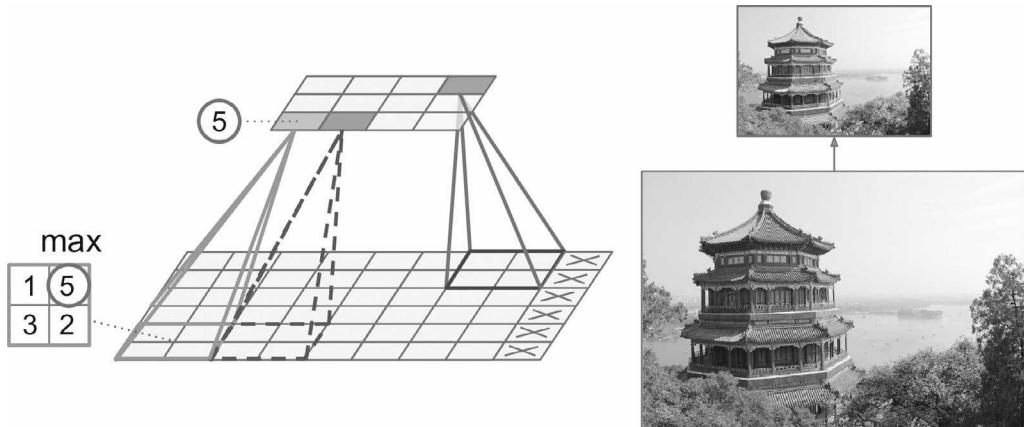


Figure 6.8 – Couche de pooling maximum (noyau de pooling 2×2 , pas de 2, aucune marge de zéros)



Puisqu'une couche de pooling travaille généralement de façon indépendante sur chaque canal d'entrée, la profondeur de la sortie est identique à celle de l'entrée.

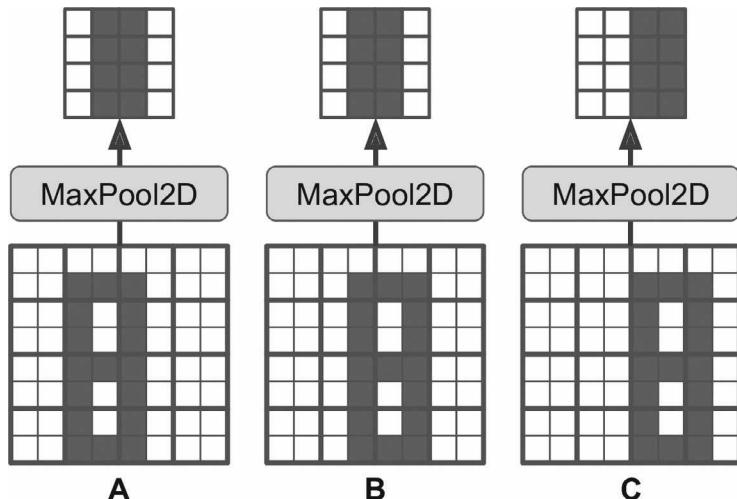


Figure 6.9 – Invariance envers les petites translations

Cependant, le pooling maximum souffre de quelques inconvénients. Tout d'abord, une telle couche est évidemment très destructrice. Même avec un minuscule noyau 2×2 et un pas de 2, la sortie est deux fois plus petite dans les deux directions (la surface est donc quatre fois inférieure), ce qui élimine 75 % des valeurs d'entrée. Par ailleurs, dans certaines applications, l'invariance n'est pas souhaitable. Prenons une segmentation sémantique (c'est-à-dire une classification de chaque pixel d'une image en fonction de l'objet auquel ce pixel appartient ; nous y reviendrons plus loin) : il est clair que si l'image d'entrée est décalée d'un pixel vers la droite, la sortie doit l'être également. Dans ce cas, l'objectif est non pas l'invariance mais l'*équivariance* : un petit changement sur les entrées doit conduire au petit changement correspondant sur la sortie.

6.3.1 Implémentation avec TensorFlow

La mise en œuvre d'une couche de pooling maximum avec TensorFlow pose peu de difficultés. Le code suivant crée une telle couche avec un noyau 2×2 . Puisque le pas correspond par défaut à la taille du noyau, il est donc de 2 (horizontalement et verticalement) dans cette couche. Le remplissage "valid" (c'est-à-dire aucune marge de zéros) est actif par défaut :

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

Pour créer une *couche de pooling moyen* (*average pooling*), il suffit de remplacer MaxPool2D par AvgPool2D. Elle opère exactement comme une couche de pooling maximum, excepté qu'elle calcule la moyenne à la place du maximum. Les couches de pooling moyen ont connu leur moment de gloire, mais elles sont aujourd'hui supplantées par les couches de pooling maximum, qui affichent généralement de meilleures performances. Cela peut sembler surprenant car calculer la moyenne conduit à une perte d'informations moindre que calculer le maximum. Mais le pooling maximum conserve uniquement les caractéristiques les plus fortes, écartant les moins pertinentes. La couche suivante travaille donc sur un signal plus propre. Par ailleurs, le pooling maximum offre une invariance de translation plus importante que le pooling moyen et demande des calculs légèrement moins intensifs.

Le pooling maximum et le pooling moyen peuvent également être appliqués sur la profondeur à la place de la hauteur et de la largeur, mais cette utilisation est plus rare. Elle permet néanmoins au CNN d'apprendre à devenir invariant à diverses caractéristiques. Par exemple, il peut apprendre plusieurs filtres, chacun détectant une rotation différente du même motif (comme les chiffres manuscrits ; voir la figure 6.10), et la couche de pooling maximum en profondeur garantit que la sortie reste la même quelle que soit la rotation. De manière comparable, le CNN pourrait apprendre à devenir invariant à d'autres caractéristiques : épaisseur, luminosité, inclinaison, couleur, etc.

Keras n'offre pas de couche de pooling maximum en profondeur, contrairement à l'API de bas niveau pour le Deep Learning de TensorFlow : il suffit d'utiliser la fonction `tf.nn.max_pool()` et de préciser la taille du noyau et les pas sous forme de tuples de quatre éléments (c'est-à-dire des tuples de taille 4). Les trois

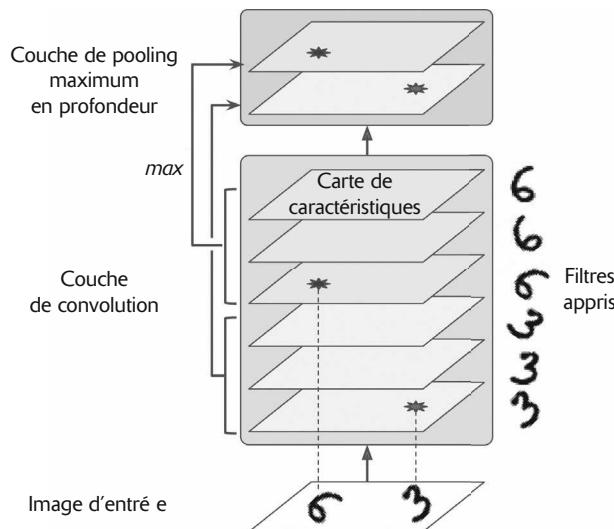


Figure 6.10 – Un pooling maximum en profondeur peut aider à apprendre n'importe quelle invariance

premières valeurs doivent être égales à 1 : elles indiquent que la taille du noyau et le pas sur le lot, la hauteur et la largeur doivent valoir 1. La dernière valeur correspond à la taille du noyau et au pas à appliquer à la profondeur – par exemple 3 (ce doit être un diviseur de la profondeur de l’entrée ; il ne conviendra pas si la couche précédente produit 20 cartes de caractéristiques, car 20 n’est pas un multiple de 3) :

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="VALID")
```

Pour inclure ce code sous forme de couche dans vos modèles Keras, enveloppez-la dans une couche Lambda (ou créer une couche Keras personnalisée) :

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                            padding="VALID"))
```

Dans les architectures modernes, vous rencontrerez souvent un dernier type de couche de pooling : la *couche de pooling moyen global*. Elle fonctionne très différemment, en se limitant à calculer la moyenne sur l’intégralité de chaque carte de caractéristiques (cela équivaut à une couche de pooling moyen qui utilise un noyau de pooling ayant les mêmes dimensions spatiales que les entrées). Autrement dit, elle produit simplement une seule valeur par carte de caractéristiques et par instance. Bien que cette approche soit extrêmement destructrice (la majorité des informations présentes dans la carte de caractéristiques est perdue), elle peut se révéler utile en tant que couche de sortie, comme nous le verrons plus loin dans

ce chapitre. Pour créer une telle couche, utilisez simplement la classe `keras.layers.GlobalAvgPool2D`:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

Cela équivaut à la simple couche `Lambda` suivante, qui calcule la moyenne sur les dimensions spatiales (hauteur et largeur):

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

Puisque vous connaissez à présent tous les blocs qui permettent de construire des réseaux de neurones convolutifs, voyons comment les assembler.

6.4 ARCHITECTURES DE CNN

Les architectures classiques de CNN empilent quelques couches de convolution (chacune généralement suivie d'une couche ReLU), puis une couche de pooling, puis quelques autres couches de convolution (+ ReLU), puis une autre couche de pooling, et ainsi de suite. L'image rétrécit au fur et à mesure qu'elle traverse le réseau, mais elle devient également de plus en plus profonde (le nombre de cartes de caractéristiques augmente), grâce aux couches de convolution (voir la figure 6.11). Au sommet de la pile, on ajoute souvent un réseau de neurones non bouclé classique, constitué de quelques couches entièrement connectées (+ ReLU), et la couche finale produit la prédiction (par exemple, une couche softmax qui génère les probabilités de classe estimées).

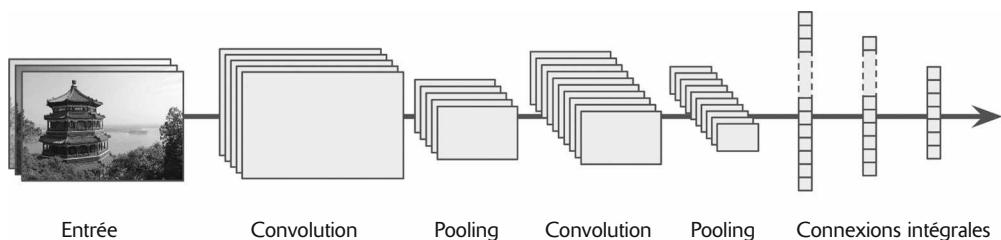


Figure 6.11 – Architecture de CNN classique



Une erreur fréquente est d'utiliser des noyaux de convolution trop grands. Par exemple, au lieu d'utiliser une couche de convolution avec un noyau 5×5 , mieux vaut empiler deux couches ayant des noyaux 3×3 : la charge de calcul et le nombre de paramètres seront bien inférieurs, et les performances seront généralement meilleures. L'exception concerne la première couche de convolution: elle a typiquement un grand noyau (par exemple, 5×5), avec un pas de 2 ou plus. Cela permet de diminuer la dimension spatiale de l'image sans perdre trop d'informations, et, puisque l'image d'entrée n'a en général que trois canaux, les calculs ne seront pas trop lourds.

Voici comment implémenter un CNN simple pour traiter le jeu de données Fashion MNIST (présenté au chapitre 2):

```

model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])

```

Détaillons ce modèle :

- La première couche utilise 64 filtres assez larges (7×7) mais aucun pas, car les images d'entrée ne sont pas très grandes. Elle précise également `input_shape=[28, 28, 1]`, car les images font 28×28 pixels, avec un seul canal de couleur (elles sont en niveaux de gris).
- Ensuite, nous avons une couche de pooling maximum, qui utilise un pool de taille 2 et qui divise donc chaque dimension spatiale par un facteur 2.
- Nous répétons ensuite deux fois la même structure : deux couches de convolution suivies d'une couche de pooling maximum. Pour des images plus grandes, nous pourrions répéter cette structure un plus grand nombre de fois (ce nombre est un hyperparamètre ajustable).
- Le nombre de filtres augmente plus nous progressons dans le CNN vers la couche de sortie (il est initialement de 64, puis de 128, puis de 256) : cette augmentation a un sens, car le nombre de caractéristiques de bas niveau est souvent assez bas (par exemple, de petits cercles, des lignes horizontales), mais il existe de nombreuses manières différentes de les combiner en caractéristiques de plus haut niveau. La pratique courante consiste à doubler le nombre de filtres après chaque couche de pooling : puisqu'une couche de pooling divise chaque dimension spatiale par un facteur 2, nous pouvons doubler le nombre de cartes de caractéristiques de la couche suivante sans craindre de voir exploser le nombre de paramètres, l'empreinte mémoire ou la charge de calcul.
- Vient ensuite le réseau intégralement connecté, constitué de deux couches cachées denses et d'une couche de sortie dense. Nous devons aplatisir les entrées, car un réseau dense attend un tableau de caractéristiques à une dimension pour chaque instance. Pour réduire le surajustement, nous ajoutons également deux couches de dropout, chacune avec un taux d'extinction de 50 %.

Ce CNN permet d'atteindre une précision de 92 % sur le jeu de test. Il n'est pas le plus performant, mais le résultat est plutôt bon, en tout cas bien meilleur que celui obtenu au chapitre 2 avec des réseaux denses.

Au fil des années, des variantes de cette architecture fondamentale sont apparues, conduisant à des avancées impressionnantes dans le domaine. Pour bien mesurer cette progression, il suffit de prendre en référence le taux d'erreur obtenu dans différentes compétitions, comme le défi ILSVRC ImageNet (<http://image-net.org/>). Dans cette compétition, le taux d'erreur top-5 pour la classification d'images est passé de plus de 26 % à un peu moins de 2,3 % en six ans. Le taux d'erreur top-5 correspond au nombre d'images test pour lesquelles les cinq premières prédictions du système ne comprenaient pas la bonne réponse. Les images sont grandes (hautes de 256 pixels) et il existe 1 000 classes, certaines d'entre elles étant réellement subtiles (par exemple, distinguer 120 races de chiens). Pour mieux comprendre le fonctionnement des CNN, nous allons regarder l'évolution des propositions gagnantes.

Nous examinerons tout d'abord l'architecture LeNet-5 classique (1998), puis trois des gagnants du défi ILSVRC: AlexNet (2012), GoogLeNet (2014) et ResNet (2015).

6.4.1 LeNet-5

L'architecture LeNet-5¹⁴⁵ est probablement la plus connue des architectures de CNN. Nous l'avons indiqué précédemment, elle a été créée en 1998 par Yann LeCun et elle est largement utilisée pour la reconnaissance des chiffres écrits à la main (MNIST). Elle est constituée de plusieurs couches, énumérées au tableau 6.1.

Tableau 6.1 – Architecture LeNet-5

Couche	Type	Cartes	Taille	Taille de noyau	Pas	Activation
Out	Intégralement connectée	–	10	–	–	RBF
F6	Intégralement connectée	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Pooling moyen	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Pooling moyen	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Entrée	1	32×32	–	–	–

Voici quelques détails supplémentaires :

- Les images MNIST font 28×28 pixels, mais une marge de zéros est rajoutée pour arriver à une taille de 32×32 pixels, et le résultat est normalisé avant d'entrer dans le réseau. Puisque les autres parties du réseau n'ajoutent pas de marge de

145. Yann LeCun *et al.*, « Gradient-Based Learning Applied to Document Recognition », *Proceedings of the IEEE*, 86, n° 11 (1998), 2278-2324 : <https://hml.info/lenet5>.

zéros, la taille diminue continuellement au fur et à mesure de la progression de l'image dans le réseau.

- Les couches de pooling moyen sont légèrement plus complexes qu'à l'habitude. Chaque neurone calcule la moyenne de ses entrées, multiplie le résultat par un coefficient (un par carte de caractéristiques), ajoute un terme constant (de nouveau, un par carte) et termine en appliquant la fonction d'activation. Les coefficients et le terme constant sont appris lors de l'entraînement.
- La plupart des neurones des cartes de C3 sont connectés aux neurones de seulement trois ou quatre cartes de S2 (à la place des six cartes de S2). Les détails se trouvent dans le tableau 1 (page 8) de l'article d'origine.
- La couche de sortie est un peu particulière. Au lieu de calculer le produit scalaire des entrées et du vecteur des poids, chaque neurone génère le carré de la distance euclidienne entre son vecteur des entrées et son vecteur des poids. Chaque sortie mesure le degré d'appartenance de l'image à une classe de chiffre particulière. L'entropie croisée est à présent la fonction de coût préférée, car elle pénalise davantage les mauvaises prédictions, en produisant des gradients plus importants et en convergeant donc plus rapidement.

Le site web de Yann LeCun (<http://yann.lecun.com/exdb/lenet/index.html>) propose des démonstrations de la classification des chiffres avec LeNet-5.

6.4.2 AlexNet

En 2012, l'architecture de CNN AlexNet¹⁴⁶ a remporté le défi ImageNet ILSVRC avec une bonne longueur d'avance. Elle est arrivée à un taux d'erreur top-5 de 17 %, tandis que le deuxième n'a obtenu que 26 % ! Elle a été développée par Alex Krizhevsky (d'où son nom), Ilya Sutskever et Geoffrey Hinton. Elle ressemble énormément à LeNet-5, en étant plus large et profonde, et a été la première à empiler des couches de convolution directement les unes au-dessus des autres, sans intercaler des couches de pooling. Le tableau 6.2 résume cette architecture.

Tableau 6.2 – Architecture AlexNet

Nom	Type	Cartes	Taille	Taille de noyau	Pas	Rémpissage	Activation
Out	Intégralement connectée	–	1 000	–	–	–	Softmax
F10	Intégralement connectée	–	4 096	–	–	–	ReLU
F9	Intégralement connectée	–	4 096	–	–	–	ReLU

146. Alex Krizhevsky *et al.*, « ImageNet Classification with Deep Convolutional Neural Networks », *Proceedings of the 25th International Conference on Neural Information Processing Systems*, 1 (2012), 1097-1105 : <https://homl.info/80>.

Nom	Type	Cartes	Taille	Taille de noyau	Pas	Rémpissage	Activation
S8	Pooling maximum	256	6×6	3×3	2	valid	–
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Pooling maximum	256	13×13	3×3	2	valid	–
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Pooling maximum	96	27×27	3×3	2	valid	–
C1	Convolution	96	55×55	11×11	4	same	ReLU
In	Entrée	3 (RVB)	227×227	–	–	–	–

Pour diminuer le surajustement, les auteurs ont employé deux techniques de régularisation. Premièrement, ils ont, au cours de l'entraînement, appliqué un dropout (voir le chapitre 3) avec un taux d'extinction de 50 % aux sorties des couches F9 et F10. Deuxièmement, ils ont effectué une *augmentation des données* en décalant aléatoirement les images d'entraînement, en les retournant horizontalement et en changeant les conditions d'éclairage.

AlexNet utilise également une étape de normalisation immédiatement après l'étape ReLU des couches C1 et C3, appelée *normalisation de réponse locale* (LRN, Local Response Normalization). Cette forme de normalisation conduit les neurones qui s'activent le plus fortement à inhiber les neurones situés au même endroit dans les cartes de caractéristiques voisines (une telle rivalité d'activation a été observée avec les neurones biologiques). Les différentes cartes de caractéristiques ont ainsi tendance à se spécialiser, en s'éloignant et en s'obligeant à explorer une plage plus large de caractéristiques. Cela finit par améliorer la généralisation. L'équation 6.2 montre comment appliquer la LRN.

Équation 6.2 – Normalisation de réponse locale

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{bas}}}^{j_{\text{haut}}} a_j^2 \right)^{-\beta} \quad \text{avec} \quad \begin{cases} j_{\text{haut}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{bas}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

Dans cette équation :

- b_i est la sortie normalisée du neurone situé dans la carte de caractéristiques i , en ligne u et colonne v (dans cette équation, puisque nous considérons uniquement les neurones qui se trouvent sur cette ligne et cette colonne, u et v ne sont pas mentionnés).

- a_i est l'activation de ce neurone après l'étape ReLU, mais avant la normalisation.
- k , α , β et r sont des hyperparamètres. k est appelé le *terme constant*, et r , le *rayon de profondeur*.
- f_n est le nombre de cartes de caractéristiques.

Augmentation des données

L'augmentation des données est une technique qui consiste à augmenter artificiellement la taille du jeu d'entraînement en générant de nombreuses variantes réalistes de chaque instance d'entraînement. Elle permet de réduire le surajustement, ce qui en fait une technique de régularisation. Les instances générées doivent être aussi réalistes que possible. Idéalement, un être humain ne devrait pas être capable de faire la distinction entre les images artificielles et les autres. On pourrait imaginer l'ajout d'un simple bruit blanc, mais cela ne suffit pas : les modifications effectuées doivent contribuer à l'apprentissage, ce qui n'est pas le cas du bruit blanc.

Par exemple, vous pouvez légèrement décaler, pivoter et redimensionner chaque image du jeu d'entraînement, en variant l'importance des modifications, puis ajouter les images résultantes au jeu d'entraînement (voir la figure 6.12). Le modèle devrait ainsi être plus tolérant aux variations de position, d'orientation et de taille des objets sur les photos. Si vous voulez qu'il soit également plus tolérant vis-à-vis des conditions d'éclairage, vous pouvez générer d'autres images en variant le contraste. En général, vous pouvez également retourner horizontalement les images (excepté pour le texte et les autres objets asymétriques). En combinant toutes ces transformations, vous augmentez considérablement la taille du jeu d'entraînement.

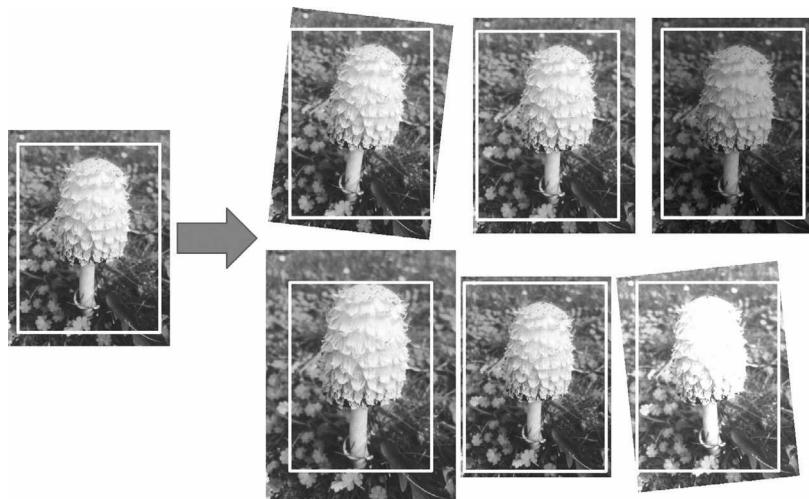


Figure 6.12 – Génération de nouvelles instances d'entraînement à partir de celles existantes

Par exemple, si $r = 2$ et qu'un neurone possède une forte activation, il inhibe l'activation des neurones qui se trouvent dans les cartes de caractéristiques immédiatement au-dessus et en dessous de la sienne.

Dans AlexNet, les hyperparamètres prennent les valeurs suivantes: $r = 2$, $\alpha = 0,00002$, $\beta = 0,75$ et $k = 1$. Cette étape peut être implémentée avec la fonction `tf.nn.local_response_normalization()` de TensorFlow (que vous pouvez envelopper dans une couche Lambda pour l'utiliser dans un modèle Keras).

ZF Net¹⁴⁷, une variante d'AlexNet développée par Matthew Zeiler et Rob Fergus, a remporté le défi ILSVRC en 2013. Il s'agit essentiellement d'une version d'AlexNet avec quelques hyperparamètres ajustés (nombre de cartes de caractéristiques, taille du noyau, pas, etc.).

6.4.3 GoogLeNet

L'architecture GoogLeNet (<https://homl.info/81>) a été proposée par Christian Szegedy *et al.* de Google Research¹⁴⁸. Elle a gagné le défi ILSVRC en 2014, en repoussant le taux d'erreur top-5 sous les 7 %. Cette excellente performance vient principalement du fait que le réseau était beaucoup plus profond que les CNN précédents (voir la figure 6.14). Cela a été possible grâce à la présence de sous-réseaux nommés modules *Inception*¹⁴⁹, qui permettent à GoogLeNet d'utiliser les paramètres beaucoup plus efficacement que dans les architectures précédentes. GoogLeNet possède en réalité dix fois moins de paramètres qu'AlexNet (environ 6 millions à la place de 60 millions).

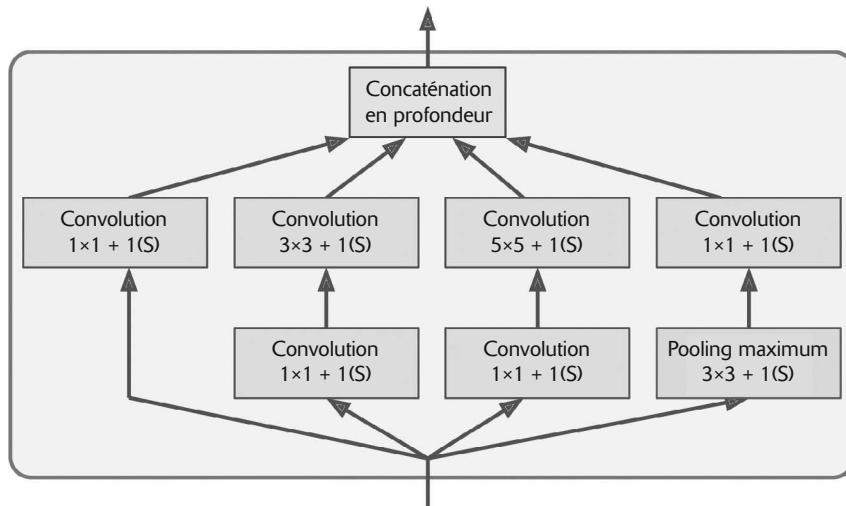


Figure 6.13 – Module Inception

147. Matthew D. Zeiler et Rob Fergus, « Visualizing and Understanding Convolutional Networks », *Proceedings of the European Conference on Computer Vision* (2014), 818-833 : <https://homl.info/zfnet>.

148. Christian Szegedy *et al.*, « Going Deeper with Convolutions », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), 1-9.

149. Dans le film *Inception*, de 2001, les personnages vont de plus en plus profond dans les multiples strates des rêves, d'où le nom de ces modules.

La figure 6.13 présente l'architecture d'un module Inception. La notation « $3 \times 3 + 1(S)$ » signifie que la couche utilise un noyau 3×3 , un pas de 1 et le remplissage "same". Le signal d'entrée est tout d'abord copié et transmis à quatre couches différentes. Toutes les couches de convolution utilisent la fonction d'activation ReLU. Le deuxième jeu de couches de convolution choisit des tailles de noyau différentes (1×1 , 3×3 et 5×5), ce qui leur permet de détecter des motifs à des échelles différentes. Puisque chaque couche utilise également un pas de 1 et un remplissage "same" (même la couche de pooling maximum), leurs sorties conservent la hauteur et la largeur de leurs entrées. Cela permet de concaténer toutes les entrées dans le sens de la profondeur au sein de la dernière couche, appelée *couche de concaténation en profondeur* (elle empile les cartes de caractéristiques des quatre couches de convolution sur lesquelles elle repose). Cette couche de concaténation peut être implémentée avec TensorFlow via l'opération `tf.concat()`, en choisissant `axis=3` (la dimension 3 correspond à la profondeur).

Pourquoi les modules Inception ont-ils des couches de convolution avec des noyaux 1×1 ? À quoi peuvent-elles servir puisque, en n'examinant qu'un seul pixel à la fois, elles ne peuvent capturer aucune caractéristique? En réalité, ces couches ont trois objectifs:

- Même si elles ne peuvent pas capturer des motifs spatiaux, elles peuvent capturer des motifs sur la profondeur.
- Elles sont configurées pour produire moins de cartes de caractéristiques que leurs entrées et servent donc de *couches de rétrécissement* qui réduisent la dimensionnalité. Cela permet d'abaisser la charge de calcul et le nombre de paramètres, et donc d'accélérer l'entraînement et d'améliorer la généralisation.
- Chaque couple de couches de convolution ($[1 \times 1, 3 \times 3]$ et $[1 \times 1, 5 \times 5]$) agit comme une seule couche de convolution puissante, capable de capturer des motifs plus complexes. Au lieu d'appliquer un simple classificateur linéaire à chaque région de l'image (comme le fait une seule couche de convolution), ce couple de couches de convolution applique un réseau de neurones à deux couches sur l'image.

En résumé, le module Inception peut être vu comme une couche de convolution survitaminée, capable de sortir des cartes de caractéristiques qui identifient des motifs complexes à différentes échelles.



Dans chaque couche de convolution, le nombre de noyaux de convolution est un hyperparamètre. Malheureusement, cela signifie qu'il y a six hyperparamètres supplémentaires à ajuster pour chaque couche Inception ajoutée.

Étudions à présent l'architecture du CNN GoogLeNet (voir la figure 6.14). Le nombre de cartes de caractéristiques produites par chaque couche de convolution et chaque couche de pooling est indiqué avant la taille du noyau. L'architecture est si profonde que nous avons dû la représenter sur trois colonnes, mais GoogLeNet est en réalité une grande pile, comprenant neuf modules Inception (les rectangles accompagnés d'une toupie). Les six valeurs données dans les modules Inception représentent le nombre de cartes de caractéristiques produites par chaque couche de convolution

dans le module (dans le même ordre qu'à la figure 6.13). Toutes les couches de convolution sont suivies de la fonction d'activation ReLU.

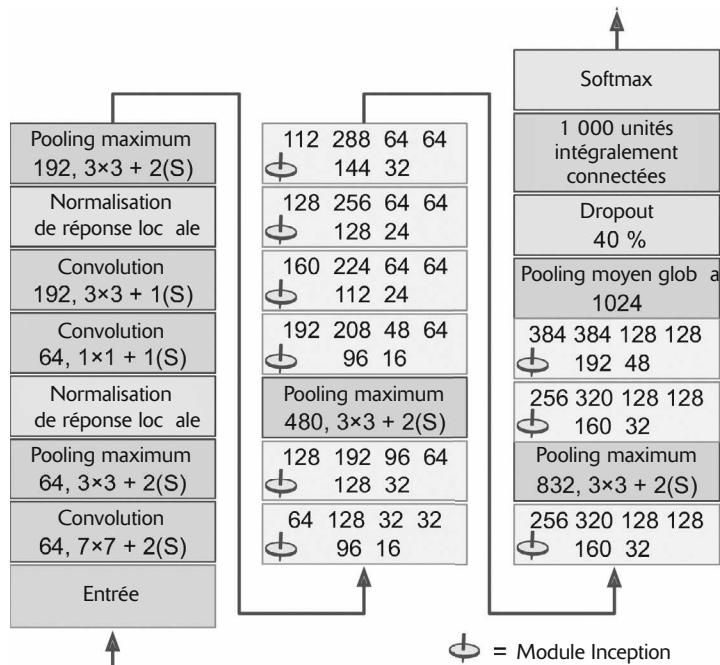


Figure 6.14 – Architecture GoogLeNet

Passons en revue ce réseau :

- Les deux premières couches divisent la hauteur et la largeur de l'image par 4 (sa surface est donc divisée par 16), afin de réduire la charge de calcul. La première couche utilise une grande taille de noyau afin de conserver une grande partie des informations.
 - Puis, la couche de normalisation de réponse locale s'assure que les couches précédentes apprennent une grande diversité de caractéristiques (comme nous l'avons expliqué précédemment).
 - Suivent deux couches de convolution, dont la première agit comme une couche de rétrécissement. Nous l'avons expliqué précédemment, elles peuvent être vues comme une seule couche de convolution plus intelligente.
 - À nouveau, une couche de normalisation de réponse locale s'assure que les couches précédentes capturent une grande diversité de motifs.
 - Ensuite, une couche de pooling maximum divise la hauteur et la largeur de l'image par 2, accélérant encore les calculs.
 - Puis arrive la grande pile de neuf modules Inception, dans laquelle s'immiscent deux couches de pooling maximum pour réduire la dimensionnalité et accélérer le réseau.

- Ensuite, la couche de pooling moyen global produit la moyenne de chaque carte de caractéristiques. Cela permet de retirer les informations spatiales restantes, ce qui convient parfaitement car, à ce stade, elles sont peu nombreuses. Évidemment, les images attendues en entrée de GoogLeNet font 224×224 pixels. Par conséquent, après cinq couches de pooling maximum, chacune divisant la hauteur et la largeur par deux, les cartes de caractéristiques sont réduites à 7×7 . Par ailleurs, puisqu'il s'agit non pas d'une tâche de localisation mais de classification, l'emplacement de l'objet n'a pas d'importance. Grâce à la réduction de la dimensionnalité apportée par cette couche, il est inutile d'avoir plusieurs couches intégralement connectées au sommet du CNN (comme dans l'architecture AlexNet). Cela réduit énormément le nombre de paramètres dans le réseau et limite le risque de surajustement.
- Les dernières couches n'ont pas besoin d'explications: dropout pour la régularisation, puis une couche intégralement connectée de 1 000 unités (puisque il y a 1 000 classes) avec une fonction d'activation softmax pour sortir les probabilités de classe estimées.

Notre schéma est légèrement simplifié par rapport à l'architecture GoogLeNet d'origine, qui comprenait également deux classificateurs secondaires ajoutés au-dessus des troisième et sixième modules Inception. Ils étaient tous deux constitués d'une couche de pooling moyen, d'une couche de convolution, de deux couches intégralement connectées et d'une couche d'activation softmax. Au cours de l'entraînement, leur perte (réduite de 70 %) était ajoutée à la perte globale. L'objectif était de combattre le problème de disparition des gradients et de régulariser le réseau. Toutefois, il a été montré que leurs effets étaient relativement mineurs.

Les chercheurs de Google ont ensuite proposé plusieurs variantes de l'architecture GoogLeNet, notamment Inception-v3 et Inception-v4. Elles utilisent des modules Inception différents et obtiennent des performances meilleures encore.

6.4.4 VGGNet

L'autre finaliste du défi ILSVRC 2014 était VGGNet¹⁵⁰, développé par Karen Simonyan et Andrew Zisserman du laboratoire de recherche VGG (Visual Geometry Group) à l'université d'Oxford. Leur architecture classique très simple se fondait sur une répétition de blocs de deux ou trois couches de convolution et d'une couche de pooling (pour atteindre un total de 16 ou 19 couches de convolution selon les variantes), ainsi qu'un réseau final dense de deux couches cachées et la couche de sortie. Elle utilisait uniquement des filtres 3×3 , mais en grand nombre.

6.4.5 ResNet

En 2015, Kaiming He *et al.* ont gagné le défi ILSVRC en proposant un *réseau résiduel*¹⁵¹ (*residual network* ou *ResNet*) dont le taux d'erreur top-5 stupéfiant a été

150. Karen Simonyan et Andrew Zisserman, « Very Deep Convolutional Networks for Large-Scale Image Recognition » (2014) : <https://homl.info/83>.

151. Kaiming He *et al.*, « Deep Residual Learning for Image Recognition » (2015) : <https://homl.info/82>.

inférieur à 3,6 %. La variante gagnante se fondait sur un CNN extrêmement profond constitué de 152 couches (d'autres variantes en possédaient 34, 50 et 101). Elle a confirmé la tendance générale : les modèles sont de plus en plus profonds, avec de moins en moins de paramètres. Pour entraîner un réseau aussi profond, l'astuce a été d'utiliser des *connexions de saut* (également appelées *connexions de raccourci*, ou *skip connections* en anglais) : le signal fourni à une couche est également ajouté à la sortie d'une couche qui se trouve un peu plus haut dans la pile. Voyons pourquoi cette technique est utile.

L'objectif de l'entraînement d'un réseau de neurones est que celui-ci modélise une fonction cible $h(\mathbf{x})$. Si vous ajoutez l'entrée \mathbf{x} à la sortie du réseau (autrement dit, vous ajoutez une connexion de saut), alors le réseau doit modéliser $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ à la place de $h(\mathbf{x})$. C'est ce que l'on appelle l'*apprentissage résiduel* (*residual learning*), illustré à la figure 6.15.

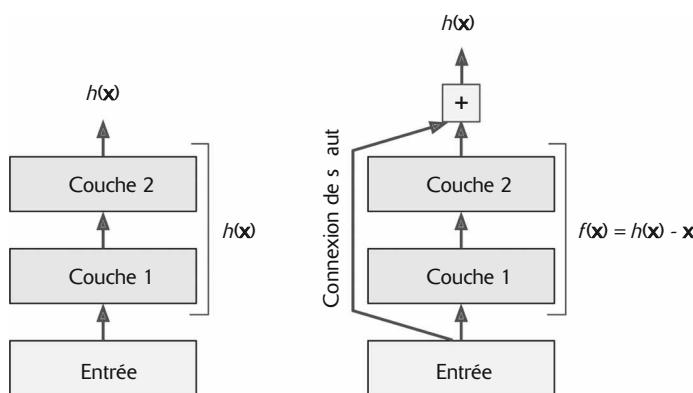


Figure 6.15 – Apprentissage résiduel

Lorsque vous initialisez un réseau de neurones ordinaire, ses poids sont proches de zéro et il produit donc des valeurs proches de zéro. Si vous ajoutez une connexion de saut, le réseau obtenu produit une copie de ses entrées. Autrement dit, il modélise initialement la fonction identité. Si la fonction cible est assez proche de la fonction identité (ce qui est souvent le cas), l'entraînement s'en trouve considérablement accéléré.

Par ailleurs, en ajoutant de nombreuses connexions de saut, le réseau peut commencer à faire des progrès même si l'apprentissage de plusieurs couches n'a pas encore débuté (voir la figure 6.16). Grâce aux connexions de saut, un signal peut aisément cheminer au travers de l'intégralité du réseau. Le réseau résiduel profond peut être vu comme une pile d'*unités résiduelles*, chacune étant un petit réseau de neurones avec une connexion de saut.

Étudions à présent l'architecture ResNet (voir la figure 6.17). Elle est en fait étonnamment simple. Elle commence et se termine exactement comme GoogLeNet (à l'exception d'une couche dropout absente), et, au milieu, ce n'est qu'une pile très

profonde de simples unités résiduelles. Chaque unité résiduelle est constituée de deux couches de convolution (sans couche de pooling!), avec une normalisation par lots (BN, *Batch Normalization*) et une activation ReLU, utilisant des noyaux 3×3 et conservant les dimensions spatiales (pas de 1, remplissage "same").

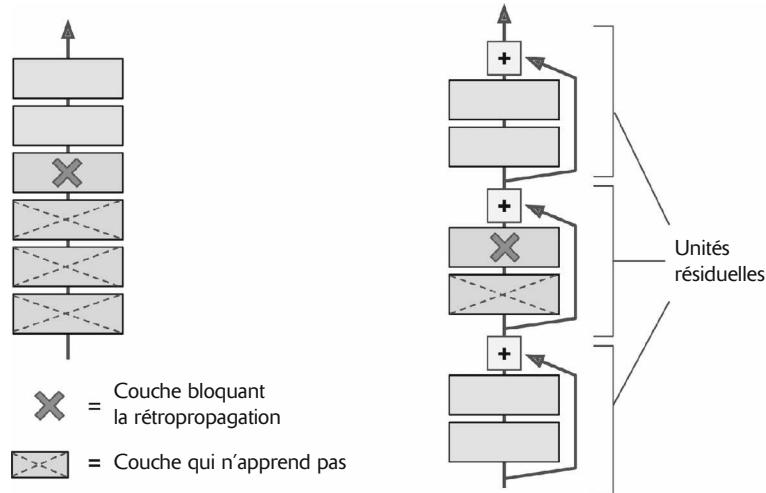


Figure 6.16 – Réseau de neurones profond classique (à gauche) et réseau résiduel profond (à droite)

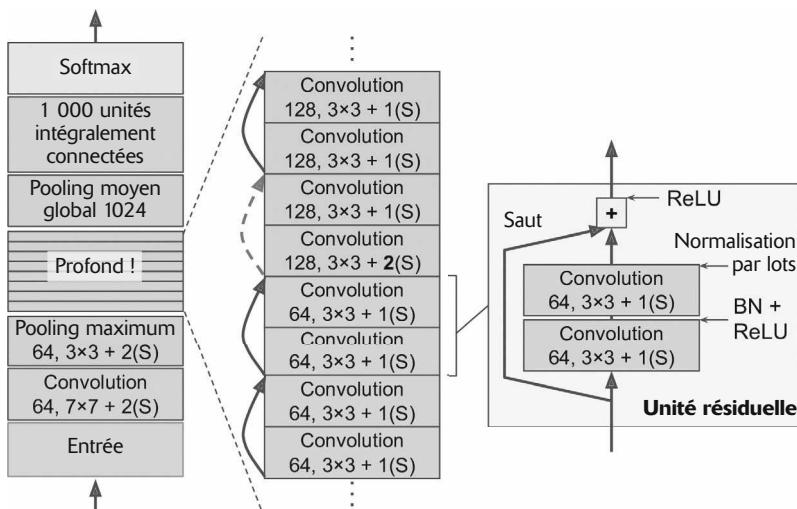


Figure 6.17 – Architecture ResNet

Le nombre de cartes de caractéristiques double toutes les quelques unités résiduelles, en même temps que leur hauteur et largeur sont divisées par deux (à l'aide d'une couche de convolution dont le pas est égal à 2). Lorsque cela se produit, les

entrées ne peuvent pas être ajoutées directement aux sorties de l'unité résiduelle car elles n'ont pas la même forme (par exemple, ce problème affecte la connexion de saut représentée par la flèche en pointillé sur la figure 6.17). Pour résoudre ce problème, les entrées sont passées au travers d'une couche de convolution 1×1 avec un pas de 2 et le nombre approprié de cartes de caractéristiques en sortie (voir la figure 6.18).

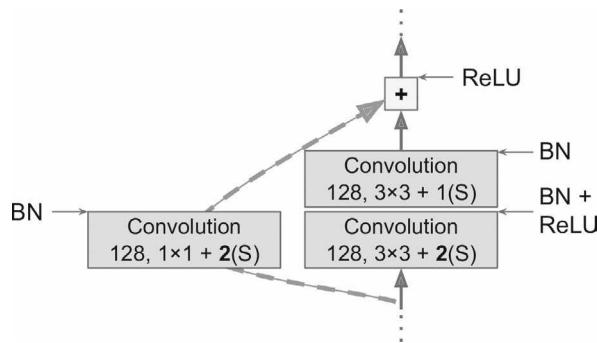


Figure 6.18 – Connexion de saut lors d'un changement de taille et de profondeur d'une carte de caractéristiques

ResNet-34, l'architecture ResNet avec 34 couches (en comptant uniquement les couches de convolution principales et la couche intégralement connectée)¹⁵² comprend trois unités résiduelles qui produisent 64 cartes de caractéristiques, quatre unités résiduelles avec 128 cartes, six unités résiduelles avec 256 cartes et trois unités résiduelles avec 512 cartes. Nous implémenterons cette architecture plus loin dans ce chapitre.

Les ResNet encore plus profonds, comme ResNet-152, utilisent des unités résiduelles légèrement différentes. À la place de deux couches de convolution 3×3 avec, par exemple, 256 cartes de caractéristiques, elles utilisent trois couches de convolution. La première est une couche de convolution 1×1 avec uniquement 64 cartes de caractéristiques (quatre fois moins) qui sert de couche de rétrécissement (comme nous l'avons expliqué précédemment). Vient ensuite une couche 3×3 avec 64 cartes de caractéristiques. La dernière est une autre couche de convolution 1×1 avec 256 cartes de caractéristiques (4 fois 64) qui restaure la profondeur d'origine. ResNet-152 comprend trois unités résiduelles de ce type, qui produisent 256 cartes, puis huit unités avec 512 cartes, 36 unités avec 1 024 cartes, et enfin trois unités avec 2 048 cartes.



L'architecture Inception-v4¹⁵³ de Google fusionne les idées de GoogLeNet et de ResNet. Elle permet d'obtenir un taux d'erreur top-5 proche de 3 % sur une classification ImageNet.

152. Pour décrire un réseau de neurones, il est fréquent de compter uniquement les couches qui possèdent des paramètres.

153. Christian Szegedy *et al.*, « Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning » (2016) : <https://hml.info/84/>.

6.4.6 Xception

Xception¹⁵⁴ est une variante très intéressante de l'architecture GoogLeNet. Proposée en 2016 par François Chollet (le créateur de Keras), elle a surpassé largement Inception-v3 sur une grande tâche de traitement d'images (350 millions d'images et 17 000 classes). À l'instar d'Inception-v4, elle fusionne les idées de GoogLeNet et de ResNet, mais elle remplace les modules Inception par une couche spéciale appelée *couche de convolution séparable en profondeur* (*depthwise separable convolution layer*) ou, plus simplement, *couche de convolution séparable*¹⁵⁵.

Ces couches avaient déjà été employées dans certaines architectures de CNN, mais elles n'étaient pas aussi importantes que dans l'architecture Xception. Alors qu'une couche de convolution ordinaire utilise des filtres qui tentent de capturer simultanément des motifs spatiaux (par exemple, un ovale) et des motifs multicanaux (par exemple, une bouche + un nez + des yeux = un visage), une couche de convolution séparable suppose que les motifs spatiaux et les motifs multicanaux peuvent être modélisés séparément (voir la figure 6.19). Elle est donc constituée de deux parties : la première partie applique un seul filtre spatial pour chaque carte de caractéristiques, puis la deuxième recherche exclusivement les motifs multicanaux – il s'agit d'une simple couche de convolution avec des filtres 1×1 .

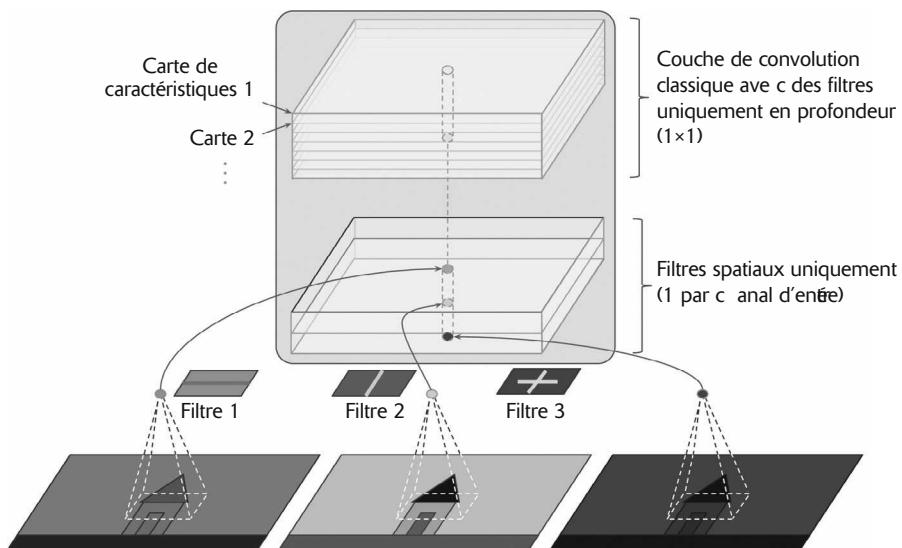


Figure 6.19 – Couche de convolution séparable en profondeur

Puisque les couches de convolution séparable ne disposent que d'un filtre spatial par canal d'entrée, vous devez éviter de les placer après des couches qui possèdent

154. François Chollet, «Xception: Deep Learning with Depthwise Separable Convolutions» (2016): <https://homl.info/xception>.

155. Ce nom peut parfois être ambigu, car les convolutions séparables spatialement sont souvent elles aussi appelées «convolutions séparables».

trop peu de canaux, comme la couche d'entrée (d'accord c'est le cas sur la figure 6.19, mais elle n'est là qu'à titre d'illustration). Voilà pourquoi l'architecture Xception commence par deux couches de convolution ordinaires et le reste de l'architecture n'utilise que des convolutions séparables (34 en tout), plus quelques couches de pooling maximum et les couches finales habituelles (une couche de pooling moyen global et une couche de sortie dense).

Vous pourriez vous demander pourquoi Xception est considérée comme une variante de GoogLeNet alors qu'elle ne comprend aucun module Inception. Nous l'avons expliqué précédemment, un module Inception contient des couches de convolution avec des filtres 1×1 : ils recherchent exclusivement des motifs multicanaux. Cependant, les couches de convolution situées au-dessus sont classiques et recherchent des motifs tant spatiaux que multicanaux. Vous pouvez donc voir un module Inception comme un intermédiaire entre une couche de convolution ordinaire (qui considère conjointement les motifs spatiaux et les motifs multicanaux) et une couche de convolution séparable (qui les considère séparément). En pratique, il semble que les couches de convolution séparable donnent généralement de meilleures performances.



Les couches de convolution séparable impliquent moins de paramètres, moins de mémoire et moins de calculs que les couches de convolution classique. Par ailleurs, elles affichent généralement de meilleures performances. Vous devez donc les envisager par défaut (excepté après des couches qui possèdent peu de canaux).

Le défi ILSVRC 2016 a été gagné par l'équipe CUImage de l'université chinoise de Hong Kong. Ils ont appliqué diverses techniques, dont un système sophistiqué de détection d'objets appelé GBD-Net¹⁵⁶, de manière à obtenir un taux d'erreur top-5 inférieure à 3 %. Bien que ce résultat soit incontestablement impressionnant, la complexité de la solution contraste avec la simplicité des architectures ResNet. Par ailleurs, un an plus tard, une autre architecture plutôt simple a obtenu de meilleures performances encore, comme nous allons le voir à présent.

6.4.7 SENet

En 2017, SENet (*Squeeze-and-Excitation Network*)¹⁵⁷ a été l'architecture gagnante du défi ILSVRC. Elle étend les architectures existantes, comme les réseaux Inception et les ResNet, en améliorant leurs performances. Son approche lui a permis de gagner la compétition avec un taux d'erreur top-5 stupéfiant de 2,25 % ! Les versions étendues des réseaux Inception et des ResNet sont nommées, respectivement, SE-Inception et SE-ResNet. Les améliorations proviennent d'un petit réseau de neurones, appelé *bloc SE*, ajouté par un SENet à chaque unité de l'architecture d'origine (autrement dit, chaque module Inception ou chaque unité résiduelle), comme l'illustre la figure 6.20.

156. Xingyu Zeng *et al.*, « Crafting GBD-Net for Object Detection », *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40, n° 9 (2018), 2109-2123 : <https://homl.info/gbdnet>.

157. Jie Hu *et al.*, « Squeeze-and-Excitation Networks », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), 7132-7141 : <https://homl.info/senet>.

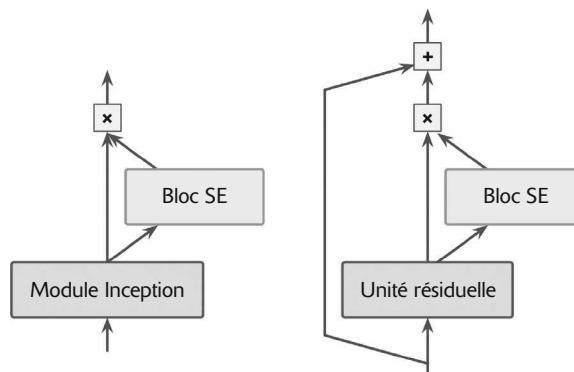


Figure 6.20 – Module SE-Inception (à gauche) et unité SE-ResNet (à droite)

Un bloc SE analyse la sortie de l'unité à laquelle il est attaché, en se focalisant exclusivement sur la profondeur (il ne recherche aucun motif spatial), et apprend les caractéristiques qui sont généralement les plus actives ensemble. Il se sert ensuite de cette information pour recalibrer les cartes de caractéristiques (voir la figure 6.21). Par exemple, un bloc SE peut apprendre que des bouches, des nez et des yeux se trouvent généralement proches les uns des autres dans des images : si vous voyez une bouche et un nez, vous devez également vous attendre à voir des yeux. Par conséquent, si le bloc constate une activation forte dans les cartes de caractéristiques des bouches et des nez, mais une activation moyenne dans celle des yeux, il stimulera celle-ci (plus précisément, il réduira les cartes de caractéristiques non pertinentes). Si les yeux étaient en quelque sorte masqués par d'autres éléments, ce recalibrage de la carte de caractéristiques aidera à résoudre l'ambiguïté.

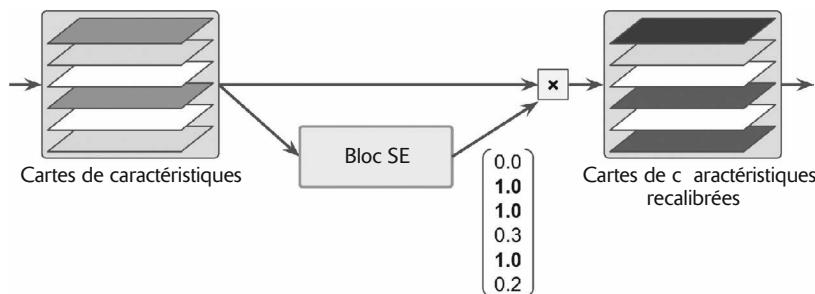


Figure 6.21 – Un bloc SE effectue un recalibrage de la carte de caractéristiques

Un bloc SE n'est constitué que de trois couches : une couche de pooling moyen global, une couche cachée dense avec une fonction d'activation ReLU et une couche de sortie dense avec la fonction d'activation sigmoïde (voir la figure 6.22).

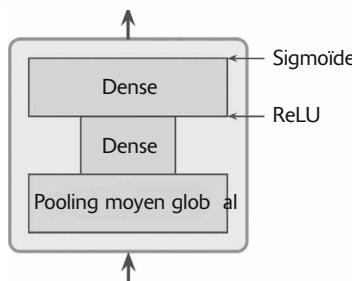


Figure 6.22 – Architecture d'un bloc SE

La couche de pooling moyen global calcule l'activation moyenne pour chaque carte de caractéristiques. Par exemple, si son entrée contient 256 cartes de caractéristiques, elle produit en sortie 256 valeurs qui représentent le niveau global de réponse à chaque filtre. C'est dans la couche suivante que se fait la « compression » : le nombre de neurones de cette couche est bien inférieur à 256 – en général seize fois moins que le nombre de cartes de caractéristiques (par exemple, seize neurones) – et les 256 valeurs sont réduites en un petit vecteur (par exemple, seize dimensions). Il s'agit d'une représentation vectorielle de faible dimension (autrement dit, un plongement) de la distribution des réponses aux caractéristiques. Cette étape de rétrécissement oblige le bloc SE à apprendre une représentation générale des combinaisons de caractéristiques (nous reverrons ce principe en action lors de la présentation des autoencodeurs au chapitre 9). Enfin, la couche de sortie prend le plongement et génère un vecteur de recalibrage avec une valeur par carte de caractéristiques (par exemple, 256), chacune entre 0 et 1. Les cartes de caractéristiques sont ensuite multipliées par ce vecteur de recalibrage de sorte que les caractéristiques non pertinentes (ayant une valeur de recalibrage faible) sont réduites, tandis que les caractéristiques importantes (avec une valeur de recalibrage proche de 1) sont maintenues.

6.5 IMPLÉMENTER UN CNN RESNET-34 AVEC KERAS

La plupart des architectures de CNN décrites jusqu'à présent sont relativement simples à implémenter (même si, en général, vous chargerez de préférence un réseau préentraîné, comme nous le verrons). Pour illustrer la procédure, implémentons un réseau ResNet-34 à partir de zéro en utilisant Keras. Commençons par créer une couche ResidualUnit :

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
```

```

        keras.layers.Conv2D(filters, 3, strides=1,
                            padding="same", use_bias=False),
        keras.layers.BatchNormalization())
self.skip_layers = []
if strides > 1:
    self.skip_layers = [
        keras.layers.Conv2D(filters, 1, strides=strides,
                            padding="same", use_bias=False),
        keras.layers.BatchNormalization()]

def call(self, inputs):
    Z = inputs
    for layer in self.main_layers:
        Z = layer(Z)
    skip_Z = inputs
    for layer in self.skip_layers:
        skip_Z = layer(skip_Z)
    return self.activation(Z + skip_Z)

```

Ce code est assez proche de l'architecture illustrée à la figure 6.18. Dans le constructeur, nous créons toutes les couches dont nous avons besoin : les couches principales sont celles données en partie droite de la figure, les couches de saut sont celles placées à gauche (requises uniquement si le pas est supérieur à 1). Ensuite, dans la méthode `call()`, nous faisons passer les entrées par les couches principales et les couches de saut (si présentes), puis nous additionnons les deux résultats et appliquons la fonction d'activation.

Nous pouvons à présent construire le ResNet-34 avec un modèle `Sequential`, car il s'agit en réalité d'une simple longue suite de couches (nous pouvons traiter chaque unité résiduelle comme une seule couche puisque nous disposons de la classe `ResidualUnit`):

```

model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                            padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))

```

Dans ce cas, la seule petite difficulté réside dans la boucle qui ajoute les couches `ResidualUnit` au modèle : les trois premières unités résiduelles comprennent 64 filtres, les quatre suivantes en ont 128, etc. Ensuite, si le nombre de filtres est identique à celui de l'unité résiduelle précédente, nous fixons le pas à 1, sinon à 2. Puis nous ajoutons la couche `ResidualUnit` et terminons en actualisant `prev_filters`.

Il est surprenant de voir qu'à peine 40 lignes de code permettent de construire le modèle qui a gagné le défi ILSVRC en 2015 ! Cela montre parfaitement l'élégance du modèle ResNet et la capacité d'expression de l'API de Keras. L'implémentation des autres architectures de CNN n'est pas beaucoup plus complexe. Mais Keras fournit déjà plusieurs de ces architectures en standard, alors pourquoi ne pas les employer directement ?

6.6 UTILISER DES MODÈLES PRÉENTRAÎNÉS DE KERAS

En général, vous n'aurez pas à implémenter manuellement des modèles standard comme GoogLeNet ou ResNet, car des réseaux préentraînés sont disponibles dans le package `keras.applications`; une ligne de code suffira. Par exemple, vous pouvez charger le modèle ResNet-50, préentraîné sur ImageNet, avec la ligne de code suivante :

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

Voilà tout ! Elle crée un modèle ResNet-50 et télécharge des poids préentraînés sur le jeu de données ImageNet. Pour l'exploiter, vous devez commencer par vérifier que les images ont la taille appropriée. Puisqu'un modèle ResNet-50 attend des images de 224×224 pixels (pour d'autres modèles, la taille peut être différente, par exemple 299×299), nous utilisons la fonction `tf.image.resize()` de TensorFlow pour redimensionner les images chargées précédemment :

```
images_resized = tf.image.resize(images, [224, 224])
```



`tf.image.resize()` ne conserve pas le rapport entre la hauteur et la largeur de l'image. Si cela pose un problème, essayez de rogner les images aux proportions appropriées avant de les redimensionner. Les deux opérations peuvent se faire d'un coup avec `tf.image.crop_and_resize()`.

Les modèles préentraînés supposent que les images sont prétraitées de manière spécifique. Dans certains cas, ils peuvent attendre que les entrées soient redimensionnées entre 0 et 1, ou entre -1 et 1, etc. Chaque modèle fournit une fonction `preprocess_input()` qui vous permet de prétraiter vos images. Puisque ces fonctions supposent que les valeurs de pixels se trouvent dans la plage 0 à 255, nous devons les multiplier par 255 (puisque elles ont été précédemment redimensionnées dans la plage 0 à 1) :

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Nous pouvons à présent utiliser le modèle préentraîné pour effectuer des prédictions :

```
Y_proba = model.predict(inputs)
```

La sortie `Y_proba` est une matrice constituée d'une ligne par image et d'une colonne par classe (dans ce cas, nous avons 1 000 classes). Pour afficher les K premières prédictions, accompagnées du nom de la classe et de la probabilité estimée pour chaque classe prédite, utilisez la fonction `decode_predictions()`. Pour chaque image, elle retourne un tableau contenant les K premières prédictions, où

chaque prédiction est représentée sous forme d'un tableau qui contient l'identifiant de la classe¹⁵⁸, son nom et le score de confiance correspondant :

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print(" {} - {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

Voici la sortie que nous avons obtenue :

```
Image #0
n03877845 - palace      42.87%
n02825657 - bell_cote   40.57%
n03781244 - monastery   14.56%

Image #1
n04522168 - vase         46.83%
n07930864 - cup          7.78%
n11939491 - daisy        4.87%
```

Les classes correctes («monastery» et «daisy») apparaissent dans les trois premiers résultats pour les deux images. C'est plutôt pas mal, si l'on considère que le modèle devait choisir parmi 1 000 classes.

Vous le constatez, un modèle préentraîné permet de créer très facilement un classificateur d'images plutôt bon. D'autres modèles pour le traitement d'images sont disponibles dans `keras.applications`, notamment plusieurs variantes de ResNet, de GoogLeNet, comme Inception-v3 et Xception, de VGGNet, ainsi que MobileNet et MobileNetV2 (des modèles légers destinés à des applications mobiles).

Est-il possible d'utiliser un classificateur d'images pour des classes d'images qui ne font pas partie d'ImageNet ? Oui, en profitant des modèles préentraînés pour mettre en place un transfert d'apprentissage.

6.7 MODÈLES PRÉENTRAÎNÉS POUR UN TRANSFERT D'APPRENTISSAGE

Si vous souhaitez construire un classificateur d'images sans disposer de données d'entraînement suffisantes, il est souvent préférable de réutiliser les couches basses d'un modèle préentraîné (voir le chapitre 3). Par exemple, entraînons un modèle de classification d'images de fleurs, en réutilisant un modèle Xception préentraîné. Commençons par charger le jeu de données à l'aide d'un dataset TensorFlow (voir le chapitre 5) :

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

158. Dans le jeu de données ImageNet, chaque image est associée à un mot du jeu de données WordNet (<https://wordnet.princeton.edu/>) : l'identifiant de classe est simplement un identifiant WordNet ID.

Vous pouvez obtenir des informations sur le dataset en précisant `with_info=True`. Dans cet exemple, nous obtenons la taille du jeu de données et le nom des classes. Malheureusement, il n'existe qu'un dataset "train", sans jeu de test ni de validation. Nous devons donc découper le jeu d'entraînement. Pour cela, le projet TF Datasets fournit une API. Par exemple, nous prenons les premiers 10 % du jeu de données pour les tests, les 15 % suivants pour la validation et les 75 % restants pour l'entraînement:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])
```

```
test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Ensuite, nous devons prétraiter les images. Puisque le CNN attend des images de 224×224 pixels, Nous devons les redimensionner. Nous devons également les passer à la fonction `preprocess_input()` du Xception:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Appliquons cette fonction de prétraitement aux trois datasets, mélangeons le jeu d'entraînement et ajoutons la mise en lots et la prélecture aux trois datasets:

```
batch_size = 32
train_set = train_set.shuffle(1000)
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

Si vous souhaitez effectuer une augmentation des données, modifiez la fonction de prétraitement pour le jeu d'entraînement, en ajoutant des transformations aléatoires sur les images correspondantes. Par exemple, utilisez `tf.image.random_crop()` pour rogner aléatoirement les images, `tf.image.random_flip_left_right()` pour les retourner horizontalement, etc. (un exemple est donné dans la section « Pretrained Models for Transfer Learning » du notebook¹⁵⁹).



La classe `keras.preprocessing.image.ImageDataGenerator` simplifie le chargement des images à partir du disque et leur augmentation de diverses manières: vous pouvez décaler chaque image, la pivoter, la redimensionner, la retourner horizontalement ou verticalement, l'incliner, ou lui appliquer toute autre fonction de transformation souhaitée. Elle est très pratique pour des projets simples, mais la construction d'un pipeline `tf.data` présente de nombreux avantages. Il peut lire les images de façon efficace (par exemple, en parallèle) à partir de n'importe quelle source et non simplement depuis le disque local. Vous pouvez manipuler le Dataset comme bon vous semble. Si vous écrivez une fonction de prétraitement fondée sur des opérations `tf.image`, elle peut être utilisée à la fois dans le pipeline `tf.data` et dans le modèle qui sera déployé en production (voir le chapitre 11).

159. Voir « 14_deep_computer_vision_with_cnns.ipynb » sur <https://github.com/ageron/handson-ml2>.

Nous chargeons ensuite un modèle Xception, préentraîné sur ImageNet. Nous excluons la partie supérieure du réseau en indiquant `include_top=False`; c'est-à-dire la couche de pooling moyen global et la couche de sortie dense. Nous ajoutons ensuite notre propre couche de pooling moyen global, fondée sur la sortie du modèle de base, suivie d'une couche de sortie dense avec une unité par classe et la fonction d'activation softmax. Pour finir, nous créons le modèle Keras :

```
base_model = keras.applications.Xception(weights="imagenet",
                                         include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

Nous l'avons expliqué au chapitre 3, il est préférable de figer les poids des couches préentraînées, tout au moins au début de l'entraînement :

```
for layer in base_model.layers:
    layer.trainable = False
```



Puisque notre modèle utilise directement les couches du modèle de base, plutôt que l'objet `base_model` lui-même, régler `base_model.trainable` à `False` n'aurait aucun effet.

Enfin, nous pouvons compiler le modèle et démarrer l'entraînement :

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```



Tout cela sera extrêmement lent, sauf si vous disposez d'un processeur graphique. Dans le cas contraire, exécutez le notebook de ce chapitre¹⁶⁰ dans Colab, en utilisant un système d'exécution sous GPU (c'est gratuit !). Les instructions sont données sur la page <https://github.com/ageron/handson-ml2>.

Après quelques époques d'entraînement du modèle, sa précision de validation doit atteindre entre 75 et 80 %, avec peu de progression ensuite. Cela signifie que les couches supérieures sont alors plutôt bien entraînées et que nous pouvons libérer toutes les couches (ou vous pouvez essayer de libérer uniquement les premières couches) et poursuivre l'entraînement (n'oubliez pas de compiler le modèle lorsque vous figez ou libérez des couches). Cette fois-ci, nous utilisons un taux d'apprentissage beaucoup plus faible afin d'éviter d'endommager les poids préentraînés :

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

160. Voir « 14_deep_computer_vision_with_cnns.ipynb » sur <https://github.com/ageron/handson-ml2>.

Il lui faudra du temps, mais ce modèle doit atteindre une précision d'environ 95 % sur le jeu de test. Avec cela, vous pouvez déjà commencer à entraîner des classificateurs d'images impressionnantes ! Mais la vision par ordinateur ne se limite pas à la classification. Par exemple, comment pouvez-vous déterminer l'emplacement de la fleur dans l'image ? C'est ce que nous allons voir à présent.

6.8 CLASSIFICATION ET LOCALISATION

Localiser un objet dans une image peut s'exprimer sous forme d'une tâche de régression (voir le chapitre 2) : pour prédire un rectangle d'encadrement autour de l'objet, une approche classique consiste à prédire les coordonnées horizontale et verticale du centre de l'objet, ainsi que sa hauteur et sa largeur. Autrement dit, nous avons quatre valeurs à prédire. Cela n'implique que peu de modification du modèle. Nous devons simplement ajouter une deuxième couche de sortie dense avec quatre unités (le plus souvent au-dessus de la couche de pooling moyen global) et l'entraîner avec la perte MSE :

```
base_model = keras.applications.Xception(weights="imagenet",
                                           include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.Model(inputs=base_model.input,
                     outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # Dépend de ce qui importe
              optimizer=optimizer, metrics=["accuracy"])
```

Mais nous avons un problème : le jeu de données des fleurs ne contient aucun rectangle d'encadrement autour des fleurs. Nous devons donc les ajouter nous-mêmes. Puisque obtenir les étiquettes est souvent l'une des parties les plus difficiles et les plus coûteuses d'un projet de Machine Learning, mieux vaut passer du temps à rechercher des outils appropriés. Pour annoter des images avec des rectangles d'encadrement, vous pouvez utiliser un outil open source d'étiquetage des images comme VGG Image Annotator, LabelImg, OpenLabeler ou ImgLab, ou bien un outil commercial, comme LabelBox ou Supervisely.

Vous pouvez également envisager d'avoir recours à des plateformes de *crowdsourcing*, comme Amazon Mechanical Turk si le nombre d'images à annoter est très grand. Cependant, la mise en place d'une telle plateforme demande beaucoup de travail, avec la préparation du formulaire à envoyer aux intervenants, leur supervision et la vérification de la qualité des rectangles d'encadrement produits. Assurez-vous que cela en vaut la peine. Si vous n'avez que quelques milliers d'images à étiqueter et si vous ne prévoyez pas d'y procéder fréquemment, il peut être plus intéressant de le faire vous-même. Adriana Kovashka *et al.* ont rédigé un article¹⁶¹ très pratique sur

161. Adriana Kovashka *et al.*, « CrowdSourcing in Computer Vision », *Foundations and Trends in Computer Graphics and Vision*, 10, n° 3 (2014), 177-243 : <https://hml.info/crowd>.

le crowdsourcing dans la vision par ordinateur. Je vous conseille de le lire, même si vous ne prévoyez pas d'employer cette solution.

Supposons que vous ayez pu obtenir les rectangles d'encadrement de chaque image dans le jeu de données des fleurs (pour le moment, nous supposerons que chaque image possède un seul rectangle). Vous devez alors créer un dataset dont les éléments seront des lots d'images prétraitées accompagnées de leur étiquette de classe et de leur rectangle d'encadrement. Chaque élément doit être un tuple de la forme (`images, (étiquettes_de_classe, rectangles_d'encadrement)`). Ensuite, il ne vous reste plus qu'à entraîner votre modèle !



Les rectangles d'encadrement doivent être normalisés de sorte que les coordonnées horizontale et verticale, ainsi que la hauteur et la largeur, soient toutes dans la plage 0 à 1. De plus, les prédictions se font souvent sur la racine carrée de la hauteur et de la largeur plutôt que directement sur ces valeurs. Ainsi, une erreur de 10 pixels sur un grand rectangle d'encadrement ne sera pas pénalisée autant qu'une erreur de 10 pixels sur un petit rectangle d'encadrement.

La MSE est souvent une fonction de coût bien adaptée à l'entraînement du modèle, mais constitue un indicateur médiocre pour l'évaluation de sa capacité à prédire les rectangles d'encadrement. Dans ce cas, l'indicateur le plus répandu est l'*indice de Jaccard* (également appelé *Intersection over Union*, IoU) : l'étendue de l'intersection entre le rectangle d'encadrement prédit et le rectangle d'encadrement cible, divisée par l'étendue de leur union (voir la figure 6.23). Dans tf.keras, cette métrique est implémentée par la classe `tf.keras.metrics.MeanIOU`.



Figure 6.23 – Indicateur IoU pour les rectangles d'encadrement

Classier et localiser un seul objet est intéressant, mais que pouvons-nous faire si les images contiennent plusieurs objets (comme c'est souvent le cas dans le jeu de données des fleurs) ?

6.9 DÉTECTION D'OBJETS

L'opération de classification et de localisation de multiples objets dans une image se nomme détection d'objets. Il y a quelques années encore, une approche répandue consistait à prendre un CNN qui avait été entraîné pour classifier et localiser un seul objet, puis à le déplacer sur l'image (voir la figure 6.24). Dans notre exemple, l'image est découpée en une grille 6×8 , et le CNN (le rectangle noir au contour épais) est déplacé sur des zones 3×3 . Lorsque le CNN fait sa recherche dans l'angle supérieur gauche de l'image, il détecte une partie de la rose à gauche, puis, après son décalage d'un pas sur la droite, il détecte de nouveau la même rose. Lors de l'étape suivante, il commence par détecter une partie de la rose la plus haute, puis il la détecte de nouveau lors de son décalage d'un pas de plus vers la droite. Le CNN est déplacé sur l'intégralité de l'image pour un examen de toutes les zones 3×3 . Par ailleurs, puisque tous les objets n'ont pas la même taille, vous devez également déplacer le CNN sur des zones de tailles différentes. Par exemple, après en avoir terminé avec les zones 3×3 , il faudrait également déplacer le CNN sur toutes les zones 4×4 .

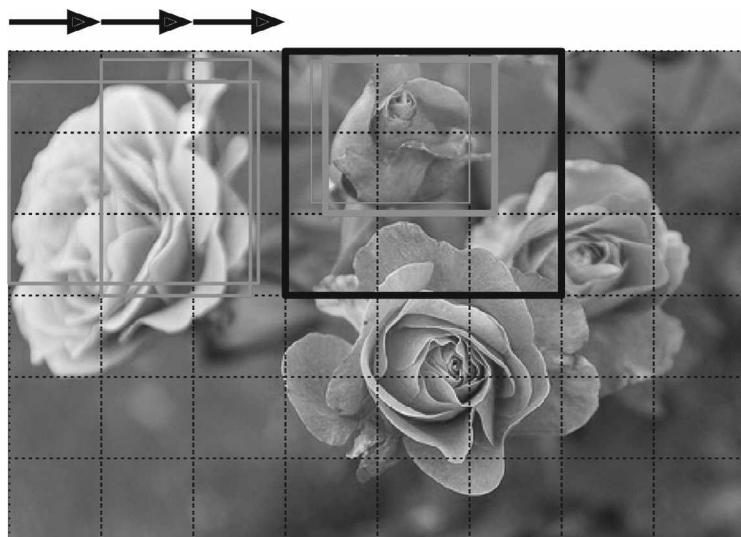


Figure 6.24 – Détection de plusieurs objets par déplacement d'un CNN sur l'image

Cette méthode est relativement simple, mais elle détectera le même objet à plusieurs reprises, à des endroits légèrement différents. Un post-traitement sera nécessaire pour retirer tous les rectangles d'encadrement inutiles. Pour cela, la *suppression des non-maxima* (NMS, *non-max suppression*) est souvent employée :

1. Tout d'abord, vous devez ajouter au CNN une sortie d'*objectness*¹⁶² afin d'estimer la probabilité qu'une fleur est bien présente dans l'image (une autre

162. Ce mot signifie littéralement le « caractère objet » d'une région de l'image : il mesure la probabilité qu'un objet soit bien présent dans cette région.

possibilité consiste à ajouter une classe « non-fleur », mais, en général, elle ne fonctionne pas aussi bien). Elle doit utiliser la fonction d'activation sigmoïde et vous pouvez l'entraîner avec une perte d'entropie croisée binaire. Ensuite, retirez tous les rectangles d'encadrement pour lesquels le score d'*objectness* est inférieur à un certain seuil: cela supprime tous les rectangles d'encadrement qui ne contiennent pas réellement une fleur.

2. Recherchez le rectangle d'encadrement dont le score d'*objectness* est le plus élevé et retirez tous les autres rectangles d'encadrement qui le chevauchent de façon importante (par exemple, avec un IoU supérieur à 60 %). Pour reprendre l'exemple de la figure 6.24, le rectangle d'encadrement affichant le meilleur score d'*objectness* est celui au bord épais qui entoure la rose du haut (le score d'*objectness* est représenté par l'épaisseur du contour des rectangles d'encadrement). Puisque l'intersection entre l'autre rectangle d'encadrement de cette rose et le rectangle d'encadrement maximal est importante, nous le supprimerons.
3. Répétez l'étape 2 jusqu'à la suppression de tous les rectangles d'encadrement inutiles.

Cette méthode simple de détection d'objets fonctionne plutôt bien, mais elle demande de nombreuses exécutions du CNN. Elle est donc assez lente. Heureusement, il existe une manière beaucoup plus rapide de déplacer un CNN sur une image: l'utilisation d'un *réseau entièrement convolutif* (FCN, *fully convolutional network*).

6.9.1 Réseaux entièrement convolutifs (FCN)

Le principe du FCN a été initialement décrit dans un article¹⁶³ publié en 2015 par Jonathan Long *et al.*, dans le cadre de la segmentation sémantique (la classification de chaque pixel d'une image en fonction de la classe de l'objet auquel il appartient). Les auteurs ont montré que vous pouvez remplacer les couches denses supérieures d'un CNN par des couches de convolution.

Prenons un exemple: supposons qu'une couche dense de 200 neurones soit placée au-dessus d'une couche de convolution produisant 100 cartes de caractéristiques, chacune de taille 7×7 (taille de la carte de caractéristiques, non celle du noyau). Chaque neurone va calculer une somme pondérée des $100 \times 7 \times 7$ activations de la couche de convolution (plus un terme constant). Voyons à présent ce qui se passe si nous remplaçons la couche dense par une couche de convolution qui utilise 200 filtres, chacun de taille 7×7 , et le remplissage "valid". Cette couche produira 200 cartes de caractéristiques, chacune de taille 1×1 (puisque le noyau correspond exactement à la taille des cartes de caractéristiques d'entrée et que nous utilisons un remplissage "valid"). Autrement dit, elle générera 200 valeurs, comme le faisait la couche dense; si vous examinez attentivement les calculs effectués par une couche de convolution, vous noterez que ces valeurs seront exactement celles qu'aurait produites la couche dense. La seule différence vient du fait que la sortie de la couche dense était

163. Jonathan Long *et al.*, « Fully Convolutional Networks for Semantic Segmentation », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), 3431-3440: <https://homl.info/fcn>.

un tenseur de forme [*taille de lot*, 200], alors que la couche de convolution retourne un tenseur de forme [*taille de lot*, 1, 1, 200].



Pour transformer une couche dense en une couche de convolution, il faut que le nombre de filtres de la couche de convolution soit égal au nombre d'unités de la couche dense, que la taille du filtre soit égale à celle des cartes de caractéristiques d'entrée et qu'un remplissage "valid" soit appliqué. Le pas doit être fixé à 1, ou plus, comme nous le verrons plus loin.

En quoi est-ce important ? Alors qu'une couche dense attend des entrées de taille spécifique (puisque elle possède un poids par caractéristiques d'entrée), une couche de convolution est en mesure de traiter des images de n'importe quelle taille¹⁶⁴ (toutefois, elle attend de ses entrées qu'elles aient un nombre précis de canaux, puisque chaque noyau contient un ensemble de poids différent pour chaque canal d'entrée). Puisqu'un FCN ne comprend que des couches de convolution (et des couches de pooling, qui ont la même propriété), il peut être entraîné et exécuté sur des images de n'importe quelle taille !

Par exemple, supposons que nous ayons déjà entraîné un CNN pour la classification et la localisation de fleurs. Il a été entraîné sur des images 224×224 et produit dix valeurs : les sorties 0 à 4 sont envoyées à la fonction d'activation softmax, pour nous donner les probabilités de classe (une par classe) ; la sortie 5 est envoyée à la fonction d'activation logistique, pour nous donner le score d'*objectness* ; les sorties 6 à 9 n'utilisent aucune fonction d'activation et représentent les coordonnées du centre du rectangle d'encadrement, ainsi que sa hauteur et sa largeur. Nous pouvons à présent convertir ses couches denses en couches de convolution. Il est même inutile de recommencer son entraînement, nous pouvons simplement copier les poids des couches denses vers les couches de convolution ! Une autre solution pourrait être de convertir le CNN en FCN avant l'entraînement.

Supposons à présent que la dernière couche de convolution située avant la couche de sortie (également appelée couche de rétrécissement) produise des cartes de caractéristiques 7×7 lorsque le réseau reçoit une image 224×224 (voir la partie gauche de la figure 6.25). Si nous passons au FCN une image 448×448 (voir la partie droite de la figure 6.25), la couche de rétrécissement génère alors des cartes de caractéristiques 14×14¹⁶⁵. Puisque la couche de sortie dense a été remplacée par une couche de convolution qui utilise dix filtres de taille 7×7, avec un remplissage "valid" et un pas de 1, la sortie sera constituée de dix cartes de caractéristiques, chacune de taille 8×8 (puisque $14 - 7 + 1 = 8$). Autrement dit, le FCN va traiter

164. À une exception près : une couche de convolution qui utilise le remplissage "valid" manifestera son mécontentement si la taille de l'entrée est inférieure à celle du noyau.

165. Cela suppose que le réseau utilise uniquement un remplissage "same". Un remplissage "valid" réduirait évidemment la taille des cartes de caractéristiques. De plus, 448 se divise parfaitement par 2 à plusieurs reprises, jusqu'à atteindre 7, sans aucune erreur d'arrondi. Si une couche utilise un pas différent de 1 ou 2, des erreurs d'arrondi peuvent se produire et les cartes de caractéristiques peuvent finir par être plus petites.

l'intégralité de l'image une seule fois et produira une grille 8×8 dont chaque cellule contient dix valeurs (cinq probabilités de classe, un score d'*objectness* et quatre coordonnées de rectangle d'encadrement). Cela revient exactement à prendre le CNN d'origine et à le déplacer sur l'image en utilisant huit pas par ligne et huit par colonne.

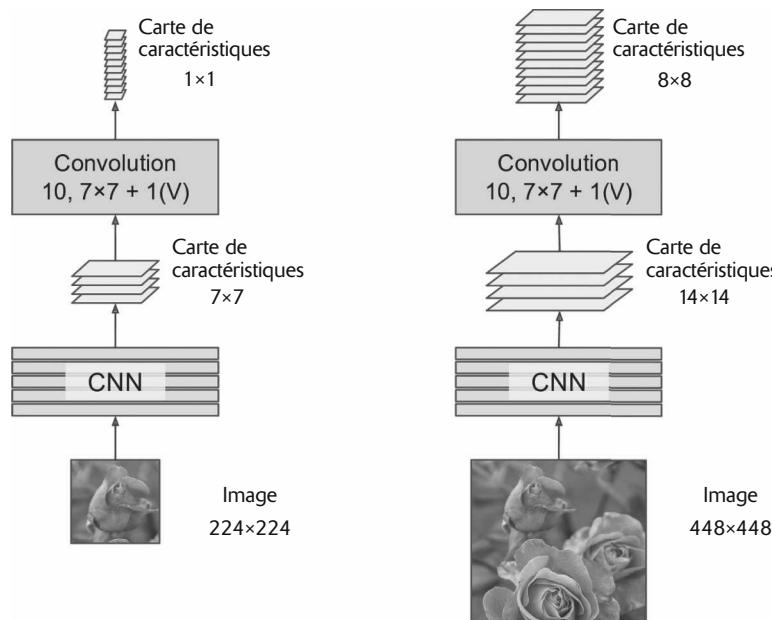


Figure 6.25 – Le même réseau entièrement convolutif traitant une petite (à gauche) et une grande (à droite) image

Pour visualiser tout cela, imaginez le découpage de l'image d'origine en une grille 14×14 , puis le déplacement d'une fenêtre 7×7 sur cette grille; il y aura $8 \times 8 = 64$ emplacements possibles pour cette fenêtre, d'où 8×8 prédictions. La solution fondée sur le FCN est *beaucoup* plus efficace, car le réseau n'examine l'image qu'une seule fois. À ce propos, «on ne regarde qu'une fois» est la traduction du nom d'une architecture de détection d'objet très répandue que nous allons maintenant décrire: YOLO (You Only Look Once).

6.9.2 YOLO (You Only Look Once)

YOLO est une architecture de détection d'objet extrêmement rapide et précise proposée par Joseph Redmon *et al.* dans un article¹⁶⁶ publié en 2015. Elle a été ensuite

166. Joseph Redmon *et al.*, « You Only Look Once: Unified, Real-Time Object Detection », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), 779-788 : <https://homl.info/yolo>.

améliorée en 2016¹⁶⁷ (YOLOv2), puis de nouveau en 2018¹⁶⁸ (YOLOv3). Elle est si rapide qu'elle peut travailler en temps réel sur une vidéo, comme vous pouvez le voir sur la démonstration mise en place par Redmon (<https://homl.info/yolodemo>).

L'architecture de YOLOv3 est proche de celle que nous venons de présenter, mais avec quelques différences importantes :

- Elle produit cinq rectangles d'encadrement pour chaque cellule de la grille (au lieu d'un seul), chacun avec un score d'*objectness*. Elle génère également vingt probabilités de classe par cellule, car elle a été entraînée sur le jeu de données PASCAL VOC qui définit vingt classes. Cela fait un total de 45 valeurs par cellule : cinq rectangles d'encadrement, chacun avec quatre coordonnées, plus cinq scores d'*objectness*, plus vingt probabilités de classe.
- Au lieu de prédire les coordonnées absolues du centre des rectangles d'encadrement, YOLOv3 prédit un décalage relativement aux coordonnées de la cellule de la grille, où (0, 0) représente l'angle supérieur gauche de cette cellule et (1, 1) l'angle inférieur droit. Pour chaque cellule, YOLOv3 est entraîné à prédire uniquement les rectangles d'encadrement dont le centre se trouve dans cette cellule (mais le cadre lui-même s'étend généralement bien au-delà de la cellule). YOLOv3 applique la fonction d'activation logistique aux coordonnées du rectangle d'encadrement afin qu'elles restent dans la plage 0 à 1.
- Avant d'entraîner le réseau de neurones, YOLOv3 trouve cinq dimensions représentatives d'un rectangle d'encadrement, appelées *rectangles d'ancrage* (*anchor boxes* ou *bounding box priors*). Pour cela, il applique l'algorithme des k-moyennes¹⁶⁹ à la hauteur et la largeur des rectangles d'encadrement du jeu d'entraînement. Par exemple, si les images d'entraînement comportent de nombreux piétons, l'un des rectangles d'ancrage aura probablement les dimensions d'un piéton type. Ensuite, lorsque le réseau de neurones prédit cinq rectangles d'encadrement par cellule, il prédit en réalité le redimensionnement de chaque rectangle d'ancrage. Supposons que la hauteur d'un rectangle d'ancrage soit de 100 pixels et sa largeur de 50 pixels, et que le réseau prédise un facteur de redimensionnement vertical de 1,5 et horizontal de 0,9 (pour une des cellules de la grille). La prédiction est un rectangle d'encadrement de taille 150×45 pixels.

De façon plus précise, pour chaque cellule et chaque rectangle d'ancrage, le réseau prédit le logarithme des facteurs de redimensionnement vertical et horizontal. En disposant de ces rectangles antérieurs, le réseau est plus à même de prédire des rectangles d'encadrement aux dimensions appropriées. De plus, cette approche accélère l'entraînement car l'apprentissage de la forme des rectangles d'encadrement que l'on peut considérer comme raisonnables se fera plus rapidement.

167. Joseph Redmon et Ali Farhadi, « YOLO9000: Better, Faster, Stronger », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), 6517-6525 : <https://homl.info/yolo2>.

168. Joseph Redmon et Ali Farhadi, « YOLOv3: An Incremental Improvement » (2018) : <https://homl.info/yolo3>.

169. Voir le chapitre 9 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

- Le réseau est entraîné avec des images de tailles différentes. Tous les quelques lots d'entraînement, il choisit aléatoirement une nouvelle dimension d'image (de 330x330 à 608x608 pixels). Cela lui permet d'apprendre à détecter des objets de taille variable. De plus, YOLOv3 peut alors être utilisé avec différentes échelles. Une échelle plus petite sera moins précise mais plus rapide que l'échelle plus grande. Vous pourrez choisir le bon compromis en fonction de votre utilisation.

Quelques autres innovations pourraient vous intéresser, comme l'utilisation de connexions de saut pour récupérer une partie de la perte en résolution spatiale due au CNN (nous y reviendrons plus loin dans le cadre de la segmentation sémantique). Dans l'article publié en 2016, les auteurs présentent le modèle YOLO9000, qui utilise la classification hiérarchique : il prédit une probabilité pour chaque nœud d'une hiérarchie visuelle appelée *WordTree*. Ainsi, le réseau peut prédire avec un niveau de certitude assez élevé qu'une image représente, par exemple, un chien, même s'il n'est pas certain de sa race. Je vous encourage à aller plus loin en lisant les trois articles : ils sont plutôt bien rédigés et donnent de bons exemples de l'amélioration progressive des systèmes de Deep Learning.

Moyenne de la précision moyenne (mAP)

La *moyenne de la précision moyenne* (mAP, *mean Average Precision*) est un indicateur très utilisé dans les opérations de détection d'objets. Deux « moyennes », cela pourrait sembler quelque peu redondant. Pour comprendre cet indicateur, revenons sur deux métriques de classification, la précision et le rappel, et sur le compromis : plus le rappel est élevé, plus la précision est faible¹⁷⁰. Pour résumer la courbe de précision/rappel en un seul chiffre, nous pouvons mesurer l'aire sous la courbe (AUC, *area under the curve*). Mais cette courbe peut contenir quelques portions où la précision monte lorsque le rappel augmente, en particulier pour les valeurs de rappel basses¹⁷¹. Voilà l'une des raisons d'être de l'indicateur mAP.

Supposons que le classificateur ait une précision de 90 % pour un rappel de 10 %, mais une précision de 96 % pour un rappel de 20 %. Dans ce cas, il n'y a pas réellement de compromis : il est tout simplement plus sensé d'utiliser le classificateur avec un rappel de 20 % plutôt que celui à 10 %, car vous aurez à la fois une précision et un rappel plus élevés. Par conséquent, au lieu d'examiner la précision avec un rappel à 10 %, nous devons en réalité rechercher la précision *maximale* que le classificateur peut offrir avec un rappel *d'au moins* 10 %. Elle serait non pas de 90 %, mais de 96 %. Pour avoir une bonne idée des performances du modèle, une solution consiste à calculer la précision maximale que vous pouvez obtenir avec un rappel d'au moins 0 %, puis de 10 %, de 20 %, et ainsi de suite jusqu'à 100 %, et de calculer la moyenne de

170. Voir le § 3.3.4 du chapitre 3 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

171. Ce phénomène est visible en partie supérieure gauche de la figure 3.5 du chapitre 3, *op. cit.*

ces précisions maximales. C'est ce que l'on appelle la *précision moyenne* (AP, *Average Precision*). Lorsque le nombre de classes est supérieur à deux, nous pouvons calculer l'AP de chaque classe, puis calculer l'AP moyenne (mAP). Et voilà !

Dans un système de détection d'objets, nous avons un niveau de complexité supplémentaire : le système peut détecter la classe appropriée, mais au mauvais emplacement (autrement dit, le rectangle d'encadrement est totalement à côté). Dans ce cas, cette prédiction ne doit pas être considérée comme positive. Une solution consiste à fixer un seuil d'IoU. Par exemple, nous pourrions considérer qu'une prédiction est correcte uniquement si l'IoU est supérieur à 0,5 et que la classe prédite est juste. La mAP correspondante est généralement notée mAP@0.5 (ou mAP@50%, ou, parfois, simplement AP₅₀). Cette approche est mise en œuvre dans certaines compétitions (par exemple, le défi PASCAL VOC). Dans d'autres, comme la compétition COCO, la mAP est calculée pour différents seuils d'IoU (0,50, 0,55, 0,60, ..., 0,95) et l'indicateur final est la moyenne de toutes ces mAP (noté AP@[.50:.95] ou AP@[.50:0.05:.95]). Oui, il s'agit bien d'une moyenne de moyennes moyennes.

Plusieurs implémentations de YOLO avec TensorFlow sont disponibles sur GitHub. Intéressez-vous notamment à celle de Zihao Zang avec TensorFlow 2 (<https://homl.info/yolotf2>). D'autres modèles de détection d'objets sont disponibles au sein du projet TensorFlow Models, nombre d'entre eux avec des poids préentraînés, et certains ayant été portés sur TF Hub, comme SSD¹⁷² et Faster-RCNN¹⁷³, tous deux assez connus. SSD est également un modèle de détection « à un coup » similaire à YOLO. Faster R-CNN est plus complexe : l'image passe d'abord par un CNN, puis la sortie est transmise à un *réseau de propositions de régions* (RPN, *Region Proposal Network*) qui propose des rectangles d'encadrement ayant le plus de chances de contenir un objet, et un classificateur est exécuté pour chaque rectangle d'encadrement, à partir de la sortie rognée du CNN.

Le choix du système de détection approprié dépend de nombreux facteurs : vitesse, précision, disponibilité de modèles préentraînés, durée de l'entraînement, complexité, etc. Les différents articles donnent bien des tableaux de mesure, mais les environnements de test sont très variables et les technologies évoluent si rapidement qu'il est difficile d'effectuer une comparaison juste qui pourra servir au plus grand nombre et qui restera valide plus de quelques mois.

Voilà, nous sommes donc en mesure de localiser des objets en les entourant de rectangles. C'est très bien ! Mais peut-être voudriez-vous être un peu plus précis. Voyons comment descendre au niveau du pixel.

172. Wei Liu *et al.*, « SSD: Single Shot Multibox Detector », *Proceedings of the 14th European Conference on Computer Vision*, 1 (2016), 21-37 : <https://homl.info/ssd>.

173. Shaoqing Ren *et al.*, « Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks », *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 1 (2015), 91-99 : <https://homl.info/fasterrcnn>.

6.10 SEGMENTATION SÉMANTIQUE

Dans une *segmentation sémantique*, chaque pixel est classifié en fonction de la classe de l'objet auquel il appartient (par exemple, route, voitures, piétons, bâtiment, etc.), comme l'illustre la figure 6.26. Aucune distinction n'est faite entre les différents objets d'une même classe. Par exemple, tous les vélos sur le côté droit de l'image segmentée terminent dans un gros paquet de pixels. La principale difficulté de cette tâche vient du passage des images dans un CNN classique, car elles perdent progressivement leur résolution spatiale (en raison des couches dont les pas sont supérieurs à 1). Un tel CNN pourra conclure qu'une personne se trouve quelque part dans la partie inférieure gauche de l'image, mais il ne sera pas plus précis que cela.



Figure 6.26 – Segmentation sémantique

À l'instar de la détection d'objets, différentes approches permettent d'aborder ce problème, certaines étant assez complexes. Toutefois, dans leur article publié en 2015, Jonathan Long *et al.* ont proposé une solution relativement simple. Ils ont commencé par prendre un CNN préentraîné et l'ont converti en un FCN. Le CNN applique un pas global de 32 sur l'image d'entrée (c'est-à-dire la somme de tous les pas supérieurs à 1), ce qui donne des cartes de caractéristiques en sortie 32 fois plus petites que l'image d'entrée. Puisque cela est clairement trop grossier, ils ajoutent une seule *couche de suréchantillonnage (upsampling)* qui multiplie la résolution par 32.

Il existe différentes méthodes de suréchantillonnage (c'est-à-dire l'augmentation de la taille d'une image), comme l'interpolation binaire, mais elle ne donne des résultats acceptables qu'avec des facteurs 4 ou 8. À la place, ils ont employé une *couche de convolution transposée*¹⁷⁴. Elle équivaut à un étirement de l'image par insertion de lignes et de colonnes vides (remplies de zéros), suivi d'une convolution normale (voir la figure 6.27). Certains préfèrent la voir comme une couche de convolution ordinaire qui utilise des pas fractionnaires (par exemple, 1/2 à la figure 6.27). La couche de convolution transposée peut être initialisée afin d'effectuer une opération

174. Ce type de couche est parfois appelé *couche de déconvolution*, mais elle n'effectue aucunement ce que les mathématiciens nomment déconvolution. Ce terme doit donc être évité.

proche de l'interpolation linéaire, mais, puisqu'il s'agit d'une couche entraînable, elle apprendra à mieux travailler pendant l'entraînement. Dans tf.keras, vous pouvez utiliser la couche Conv2DTranspose.

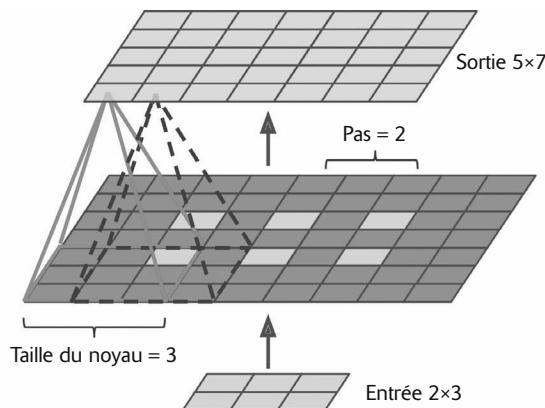


Figure 6.27 – Suréchantillonnage à l'aide d'une couche de convolution transposée



Dans une couche de convolution transposée, le pas correspond non pas à la taille des filtres mais à l'étiement de l'entrée. Par conséquent, plus le pas est grand, plus la sortie est large (contrairement aux couches de convolution ou aux couches de pooling).

Opérations de convolution de TensorFlow

TensorFlow offre également d'autres sortes de couches de convolution :

`keras.layers.Conv1D`

Crée une couche de convolution pour des entrées à une dimension, comme des séries temporelles ou du texte (suite de lettres ou de mots), comme nous le verrons au chapitre 7.

`keras.layers.Conv3D`

Crée une couche de convolution pour des entrées à trois dimensions, comme des PET-Scans en 3D.

`dilation_rate`

En fixant l'hyperparamètre `dilation_rate` de n'importe quelle couche de convolution à une valeur supérieure ou égale à 2, nous créons une *couche de convolution à trous*. Cela revient à utiliser une couche de convolution normale avec un filtre dilaté par insertion de lignes et de colonnes de zéros (les trous). Par exemple, si nous dilatons d'un *facteur de dilation* 4 un filtre 1×3 égal à $\begin{bmatrix} 1, 2, 3 \end{bmatrix}$, nous obtenons un *filtre dilaté* égal à $\begin{bmatrix} 1, 0, 0, 0, 2, 0, 0, 0, 3 \end{bmatrix}$. La couche de convolution dispose ainsi d'un champ

récepteur plus large sans augmentation des calculs ni paramètres supplémentaires.

`tf.nn.depthwise_conv2d()`

Peut servir à créer une *couche de convolution en profondeur* (mais vous devez créer les variables vous-même). Elle applique chaque filtre de façon indépendante à chaque canal d'entrée individuel. Par conséquent, si nous avons f_n filtres et $f_{n'}$ canaux d'entrée, elle produira $f_n \times f_{n'}$ cartes de caractéristiques.

Cette solution peut convenir, mais elle reste encore trop imprécise. Pour l'améliorer, les auteurs ont ajouté des connexions de saut à partir des couches inférieures. Par exemple, ils ont suréchantillonné l'image de sortie d'un facteur 2 (à la place de 32) et ont ajouté la sortie d'une couche inférieure qui avait cette résolution double. Ils ont ensuite suréchantillonné le résultat d'un facteur 16, pour arriver à un facteur total de 32 (voir la figure 6.28). Cela permet de récupérer une partie de la résolution spatiale qui avait été perdue dans les couches de pooling précédentes. Dans leur meilleure architecture, ils ont employé une deuxième connexion de saut comparable pour récupérer des détails encore plus fins à partir d'une couche encore plus basse. En résumé, la sortie du CNN d'origine passe par les étapes supplémentaires suivantes : conversion ascendante $\times 2$, ajout de la sortie d'une couche inférieure (d'échelle appropriée), conversion ascendante $\times 2$, ajout de la sortie d'une couche plus inférieure encore, et, pour finir, conversion ascendante $\times 8$. Il est même possible d'effectuer un agrandissement au-delà de la taille de l'image d'origine : cette technique, appelée *super-résolution*, permet d'augmenter la résolution d'une image.

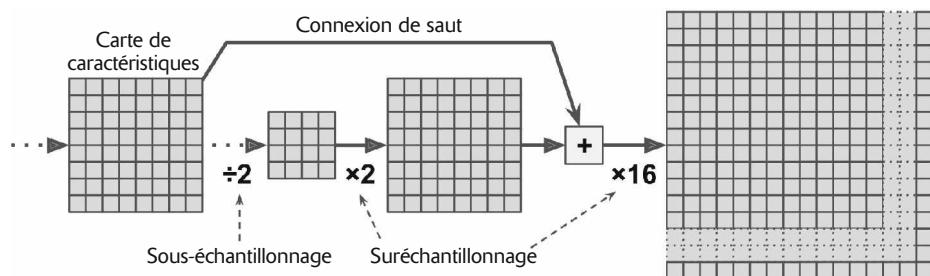


Figure 6.28 – Des couches de saut récupèrent une certaine résolution spatiale à partir de couches inférieures

Plusieurs dépôts GitHub proposent des implémentations TensorFlow de la segmentation sémantique (TensorFlow 1, pour le moment), et vous trouverez même des modèles de *segmentation d'instance* préentraînés dans le projet TensorFlow Models. La segmentation d'instance est comparable à la segmentation sémantique, mais, au lieu de fusionner tous les objets de la même classe dans un gros bloc, ils sont tous distingués (par exemple, chaque vélo est identifié individuellement). Les modèles de segmentation d'instance actuellement disponibles dans le projet TensorFlow Models se

fondent sur l'architecture *Mask R-CNN*, qui a été proposée dans un article¹⁷⁵ publié en 2017. Elle étend le modèle Faster R-CNN en produisant un masque de pixels pour chaque rectangle d'encadrement. Ainsi, vous obtenez non seulement un rectangle d'encadrement autour de chaque objet, avec un ensemble de probabilités de classe estimées, mais également un masque de pixels qui localise, dans le rectangle d'encadrement, les pixels appartenant à l'objet.

Vous le constatez, l'application du Deep Learning à la vision par ordinateur est un domaine vaste et en constante évolution, avec divers types d'architectures arrivant chaque année, tous fondés sur des réseaux de neurones convolutifs. Les progrès réalisés en quelques années sont stupéfiants et les chercheurs se concentrent à présent sur des problèmes de plus en plus complexes, comme l'*apprentissage antagoniste* (qui tente de rendre le réseau plus résistant aux images conçues pour le tromper), la capacité d'explication (comprendre pourquoi le réseau effectue une classification précise), la génération d'*images* réalistes (sur laquelle nous reviendrons au chapitre 9) et l'*apprentissage single-shot* (un système qui reconnaît un objet après l'avoir vu une seule fois). Certains explorent même des architectures totalement nouvelles, comme les réseaux à *capsules*¹⁷⁶ de Geoffrey Hinton (je les ai présentés dans deux vidéos [<https://youtu.be/BJBxae3c1H4>], avec le code correspondant dans un notebook). Dans le chapitre suivant, nous verrons comment traiter des données séquentielles, comme les suites temporelles, avec des réseaux de neurones récurrents et des réseaux de neurones convolutifs.

6.11 EXERCICES

1. Dans le contexte de la classification d'*images*, quels sont les avantages d'un CNN par rapport à un RNP intégralement connecté ?
2. Prenons un CNN constitué de trois couches de convolution, chacune avec des noyaux 33, un pas de 2 et un remplissage "same". La couche inférieure produit 100 cartes de caractéristiques, la couche intermédiaire, 200, et la couche supérieure, 400. L'entrée est constituée d'*images* RVB de 200×300 pixels.
Quel est le nombre total de paramètres du CNN ? Si l'on utilise des nombres à virgule flottante sur 32 bits, quelle quantité de RAM minimale faut-il à ce réseau lorsqu'il effectue une prédiction pour une seule instance ? Qu'en est-il pour l'entraînement d'un mini-lot de cinquante images ?
3. Si la carte graphique vient à manquer de mémoire pendant l'entraînement d'un CNN, quelles sont les cinq actions que vous pourriez effectuer pour tenter de résoudre le problème ?
4. Pourquoi voudriez-vous ajouter une couche de pooling maximum plutôt qu'une couche de convolution avec le même pas ?

175. Kaiming He *et al.*, « Mask R-CNN » (2017) : <https://homl.info/maskrcnn>.

176. Geoffrey Hinton *et al.*, « Matrix Capsules with EM Routing », *Proceedings of the International Conference on Learning Representations* (2018) : <https://homl.info/capsnet>.

5. Quand devriez-vous ajouter une couche de normalisation de réponse locale ?
6. Citez les principales innovations d'AlexNet par rapport à LeNet-5 ? Quelles sont celles de GoogLeNet, de ResNet, de SENet et de Xception ?
7. Qu'est-ce qu'un réseau entièrement convolutif ? Comment pouvez-vous convertir une couche dense en une couche de convolution ?
8. Quelle est la principale difficulté technique de la segmentation sémantique ?
9. Construisez votre propre CNN à partir de zéro et tentez d'obtenir la meilleure précision possible sur le jeu MNIST.
10. Utilisez le transfert d'apprentissage pour la classification de grandes images :
 - a. Créez un jeu d'entraînement contenant au moins 100 images par classe. Vous pouvez, par exemple, classer vos propres photos en fonction du lieu (plage, montagne, ville, etc.), ou utiliser simplement un jeu de données existant (par exemple, venant de TensorFlow Datasets).
 - b. Découpez-le en un jeu d'entraînement, un jeu de validation et un jeu de test.
 - c. Entraînez le modèle sur le jeu d'entraînement et évaluez-le sur le jeu de test.
 - d. Construisez le pipeline d'entrée, en y incluant les opérations de prétraitement appropriées, et ajoutez éventuellement une augmentation des données.
 - e. Ajustez un modèle préentraîné sur ce jeu de données.
11. Consultez le tutoriel Style Transfer de TensorFlow (<https://homl.info/styletuto>). Il décrit une manière amusante d'utiliser le Deep Learning de façon artistique.

Les solutions de ces exercices sont données à l'annexe A.

7

Traitements des séries avec des RNR et des CNN

L'attaquant frappe le ballon, vous commencez immédiatement à courir, en anticipant la trajectoire du ballon, vous le suivez des yeux et ajustez vos déplacements, pour finalement l'intercepter (sous un tonnerre d'applaudissements). Nous prédisons le futur en permanence, que ce soit en terminant la phrase d'un ami ou en anticipant l'odeur du café au petit-déjeuner. Dans ce chapitre, nous allons étudier les *réseaux de neurones récurrents* (RNR, ou RNN en anglais), une classe de réseaux qui permettent de prédire le futur (jusqu'à un certain point, évidemment). Ils sont capables d'analyser des séries chronologiques, comme le prix des actions, et de vous indiquer s'il faut acheter ou vendre. Dans les systèmes de conduite autonome, ils peuvent anticiper la trajectoire d'une voiture et aider à éviter les accidents. Plus généralement, ils peuvent travailler sur des séries de longueur quelconque, plutôt que sur des entrées de taille figée comme les réseaux examinés jusqu'à présent. Par exemple, ils peuvent prendre en entrée des phrases, des documents ou des échantillons audio, ce qui les rend très utiles pour le traitement automatique du langage naturel (TALN), comme les systèmes de traduction automatique ou de reconnaissance automatique de la parole.

Dans ce chapitre, nous commencerons par présenter les concepts fondamentaux des RNR et la manière de les entraîner en utilisant la rétropropagation dans le temps, puis nous les utiliserons pour effectuer des prévisions sur des séries chronologiques. Ensuite, nous explorerons les deux principales difficultés auxquelles font face les RNR :

- l'instabilité des gradients (décrise au chapitre 3), qui peut être réduite à l'aide de diverses techniques, notamment le dropout récurrent et la normalisation de couche récurrente ;
- une mémoire à court terme (très) limitée, qui peut être étendue en utilisant les cellules LSTM et GRU.

Les RNR ne sont pas les seuls types de réseaux de neurones capables de traiter des données séquentielles. Pour les petites séries, un réseau dense classique peut faire l'affaire. Pour les séries très longues, comme des échantillons audio ou du texte, les réseaux de neurones convolutifs fonctionnent plutôt bien. Nous examinerons également ces deux approches, et nous terminerons ce chapitre par l'implémentation d'un *WaveNet*. Cette architecture de CNN est capable de traiter des séries constituées de dizaines de milliers d'étapes temporelles. Au chapitre 8, nous poursuivrons notre exploration des RNR et nous verrons comment les utiliser dans le traitement automatique du langage naturel, au côté d'architectures plus récentes fondées sur des mécanismes d'attention. Allons-y !

7.1 NEURONES ET COUCHES RÉCURRENTS

Jusqu'à présent, nous avons décrit principalement des réseaux de neurones non bouclés, dans lesquels le flux des activations allait dans un seul sens, depuis la couche d'entrée vers la couche de sortie (à l'exception de quelques réseaux décrits dans l'annexe C). Un réseau de neurones récurrents est très semblable à un réseau de neurones non bouclé, mais certaines connexions reviennent en arrière dans le réseau. Examinons le RNR le plus simple possible. Il est constitué d'un seul neurone qui reçoit des entrées, produit une sortie et se renvoie celle-ci (voir en partie gauche de la figure 7.1). À chaque *étape temporelle* t (également appelée *trame*), ce *neurone récurrent* reçoit les entrées $x_{(t)}$, ainsi que sa propre sortie produite à l'étape temporelle précédente, $y_{(t-1)}$. Nous pouvons représenter ce petit réseau le long d'un axe du temps (voir en partie droite de la figure 7.1). Cette procédure se nomme *déplier le réseau dans le temps* (il s'agit du même neurone récurrent représenté une fois par étape temporelle).

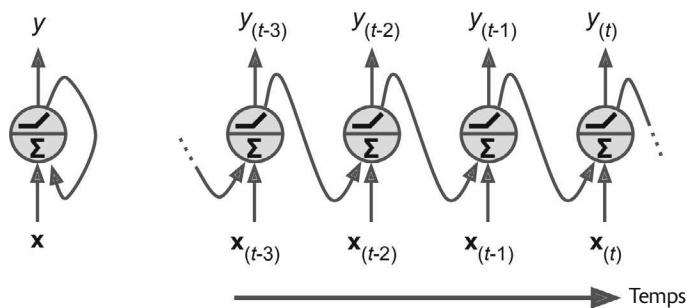


Figure 7.1 – Un neurone récurrent (à gauche), déplié dans le temps (à droite)

La création d'une couche de neurones récurrents n'est pas bien compliquée. À chaque étape temporelle t , chaque neurone reçoit à la fois le vecteur d'entrée $x_{(t)}$ et le vecteur de sortie de l'étape temporelle précédente $y_{(t-1)}$ (voir la figure 7.2). Notez que les entrées et les sorties sont à présent des vecteurs (avec un seul neurone, la sortie était un scalaire).

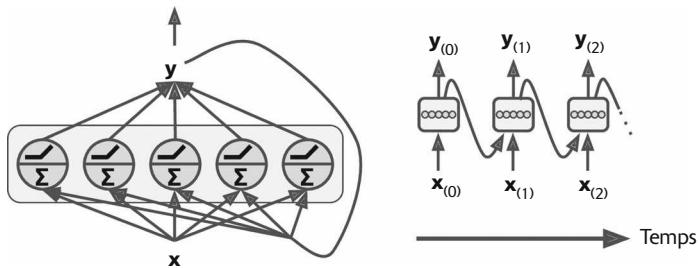


Figure 7.2 – Une couche de neurones récurrents (à gauche), dépliée dans le temps (à droite)

Chaque neurone récurrent possède deux jeux de poids : un premier pour les entrées, $x_{(t)}$, et un second pour les sorties de l'étape temporelle précédente, $y_{(t-1)}$. Appelons ces vecteurs poids W_x et W_y , respectivement. Si l'on considère maintenant la couche complète, nous pouvons regrouper les vecteurs poids de tous les neurones en deux matrices poids W_x et W_y . Le vecteur de sortie de la couche récurrente complète peut alors être calculé selon l'équation 7.1 (b est le vecteur des termes constants et $\phi(\cdot)$ est la fonction d'activation, par exemple ReLU¹⁷⁷).

Équation 7.1 – Sortie d'une couche récurrente pour une seule instance

$$y_{(t)} = \phi(W_x^T x_{(t)} + W_y^T y_{(t-1)} + b)$$

Comme pour les réseaux de neurones non bouclés, nous pouvons calculer d'un seul coup la sortie d'une couche pour un mini-lot entier en plaçant toutes les entrées à l'étape temporelle t dans une matrice d'entrées $X_{(t)}$ (voir l'équation 7.2).

Équation 7.2 – Sorties d'une couche de neurones récurrents pour toutes les instances d'un mini-lot

$$\begin{aligned} Y_{(t)} &= \phi\left(X_{(t)} W_x + Y_{(t-1)} W_y + b\right) \\ &= \phi\left[\begin{bmatrix} X_{(t)} & Y_{(t-1)} \end{bmatrix} W + b\right] \text{ avec } W = \begin{bmatrix} W_x \\ W_y \end{bmatrix} \end{aligned}$$

Dans cette équation :

- $Y_{(t)}$ est une matrice $m \times n_{\text{neurones}}$ qui contient les sorties de la couche à l'étape temporelle t pour chaque instance du mini-lot (m est le nombre d'instances dans le mini-lot et n_{neurones} le nombre de neurones).
- $X_{(t)}$ est une matrice $m \times n_{\text{entrées}}$ qui contient les entrées de toutes les instances ($n_{\text{entrées}}$ est le nombre de caractéristiques d'entrée).

177. De nombreux chercheurs préfèrent employer la tangente hyperbolique (tanh) dans les RNR plutôt que la fonction ReLU, comme l'explique l'article de Vu Pham *et al.* publié en 2013 et intitulé « Dropout Improves Recurrent Neural Networks for Handwriting Recognition » (<https://homl.info/91>). Mais les RNR fondés sur ReLU sont également employés, comme l'expliquent Quoc V. Le *et al.* dans leur article « A Simple Way to Initialize Recurrent Networks of Rectified Linear Units » publié en 2015 (<https://homl.info/92>).

- \mathbf{W}_x est une matrice $n_{\text{entrées}} \times n_{\text{neurones}}$ qui contient les poids des connexions des entrées pour l'étape temporelle courante.
- \mathbf{W}_y est une matrice $n_{\text{neurones}} \times n_{\text{neurones}}$ qui contient les poids des connexions des sorties pour l'étape temporelle précédente.
- \mathbf{b} est un vecteur de taille n_{neurones} qui contient le terme constant de chaque neurone.
- Les matrices de poids \mathbf{W}_x et \mathbf{W}_y sont souvent concaténées verticalement dans une seule matrice de poids \mathbf{W} de forme $(n_{\text{entrées}} + n_{\text{neurones}}) \times n_{\text{neurones}}$ (voir la deuxième ligne de l'équation 7.2).
- La notation $[\mathbf{X}_{(t)} \ \mathbf{Y}_{(t-1)}]$ représente la concaténation horizontale des matrices $\mathbf{X}_{(t)}$ et $\mathbf{Y}_{(t-1)}$.

$\mathbf{Y}_{(t)}$ est une fonction de $\mathbf{X}_{(t)}$ et de $\mathbf{Y}_{(t-1)}$, qui est une fonction de $\mathbf{X}_{(t-1)}$ et de $\mathbf{Y}_{(t-2)}$, qui est une fonction de $\mathbf{Y}_{(t-2)}$ et de $\mathbf{Y}_{(t-3)}$, etc. Par conséquent, $\mathbf{Y}_{(t)}$ est une fonction de toutes les entrées depuis l'instant $t = 0$ (c'est-à-dire $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$). Lors de la première étape temporelle, à $t = 0$, les sorties précédentes n'existent pas et sont, en général, supposées être toutes à zéro.

7.1.1 Cellules de mémoire

Puisque la sortie d'un neurone récurrent à l'étape temporelle t est une fonction de toutes les entrées des étapes temporelles précédentes, on peut considérer que ce neurone possède une forme de *mémoire*. Une partie d'un réseau de neurones qui conserve un état entre plusieurs étapes temporelles est appelée *cellule de mémoire* (ou, plus simplement, *cellule*). Un seul neurone récurrent, ou une couche de neurones récurrents, forme une *cellule de base*, capable d'apprendre uniquement des motifs courts (long d'environ dix étapes, en général, mais cela varie en fonction de la tâche). Nous le verrons plus loin dans ce chapitre, il existe des cellules plus complexes et plus puissantes capables d'apprendre des motifs plus longs (environ dix fois plus longs, mais, de nouveau, cela dépend de la tâche).

En général, l'état d'une cellule à l'étape temporelle t , noté $\mathbf{h}_{(t)}$ (« h » pour « *hidden* », c'est-à-dire caché), est une fonction de certaines entrées à cette étape temporelle et de son état à l'étape temporelle précédente : $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Sa sortie à l'étape temporelle t , notée $\mathbf{y}_{(t)}$, est également une fonction de l'état précédent et des entrées courantes. Dans le cas des cellules de base, la sortie est simplement égale à l'état. En revanche, ce n'est pas toujours le cas avec les cellules plus complexes (voir la figure 7.3).

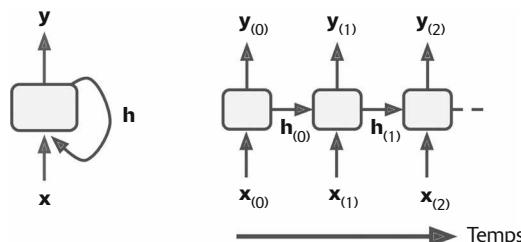


Figure 7.3 – L'état caché d'une cellule et sa sortie peuvent être différents

7.1.2 Séries en entrée et en sortie

Un RNR peut simultanément prendre une série d'entrées et produire une série de sorties (voir le réseau en partie supérieure gauche de la figure 7.4). Ce type de réseau *série-vers-série* est utile pour prédire des séries chronologiques, comme la valeur des actions. On lui fournit leurs valeurs sur les N derniers jours et il doit produire les valeurs décalées d'un jour dans le futur (c'est-à-dire depuis les $N - 1$ jours jusqu'à demain).

On peut également fournir au réseau une série d'entrées et ignorer toutes les sorties, à l'exception de la dernière (voir le réseau en partie supérieure droite de la figure 7.4). Autrement dit, il s'agit d'un réseau *série-vers-vecteur*. Par exemple, l'entrée du réseau peut être une suite de mots qui correspondent aux critiques d'un film, sa sortie sera une note d'opinion (par exemple de -1 [j'ai détesté] à $+1$ [j'ai adoré]).

À l'inverse, on peut lui donner le même vecteur d'entrée encore et encore à chaque étape temporelle et le laisser produire en sortie une série (voir le réseau en partie inférieure gauche de la figure 7.4). Il s'agit alors d'un réseau *vecteur-vers-série*. Par exemple, l'entrée peut être une image (ou la sortie d'un CNN) et la sortie une légende pour cette image.

Enfin, on peut avoir un réseau *série-vers-vecteur*, appelé *encodeur*, suivi d'un réseau *vecteur-vers-série*, appelé *décodeur* (voir le réseau en partie inférieure droite de la figure 7.4). Ce type d'architecture peut, par exemple, servir à la traduction d'une phrase d'une langue vers une autre. On fournit en entrée une phrase dans une langue, l'*encodeur* la convertit en une seule représentation vectorielle, puis le *décodeur* transforme ce vecteur en une phrase dans une autre langue. Ce modèle en deux étapes, appelé *encodeur-décodeur*, permet d'obtenir de bien meilleurs résultats qu'une traduction à la volée avec un seul RNR *série-vers-série* (comme celui représenté en partie supérieure gauche). En effet, les derniers mots d'une phrase pouvant influencer la traduction des premiers, il vaut mieux attendre d'avoir reçu l'intégralité de la phrase avant de la traduire. Nous étudierons l'implémentation d'un encodeur-décodeur au chapitre 8 (vous le verrez, c'est un peu plus complexe que ne le suggère la figure 7.4).

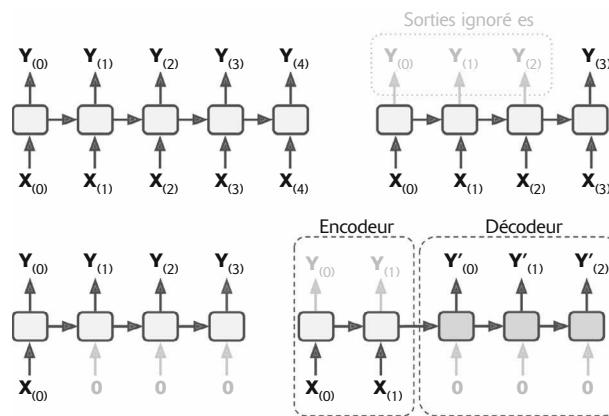


Figure 7.4 – Réseaux série-vers-série (en haut à gauche), série-vers-vecteur (en haut à droite), vecteur-vers-série (en bas à gauche) et encodeur-décodeur (en bas à droite)

Tout cela semble bien prometteur, mais comment entraîne-t-on un réseau de neurones récurrents ?

7.2 ENTRAÎNER DES RNR

Pour entraîner un RNR, l'astuce consiste à le déplier dans le temps (comme nous l'avons fait), puis à simplement employer une rétropropagation classique (voir la figure 7.5). Cette stratégie est appelée *rétropropagation dans le temps* (BPTT, *backpropagation through time*).

À l'instar de la rétropropagation classique, il existe une première passe vers l'avant au travers du réseau déplié (représentée par les flèches en pointillé). La série de sortie est ensuite évaluée à l'aide d'une fonction de coût $C(Y_{(0)}, Y_{(1)}, \dots, Y_{(T)})$ (où T est la dernière étape temporelle). Notez que cette fonction de coût peut ignorer certaines sorties, comme le montre la figure 7.5 (par exemple, dans un RNR série-vers-vecteur, toutes les sorties sont ignorées à l'exception de la dernière). Les gradients de cette fonction de coût sont rétropropagés au travers du réseau déplié (représenté par les flèches pleines). Pour finir, les paramètres du modèle sont actualisés à l'aide des gradients calculés au cours de la BPTT.

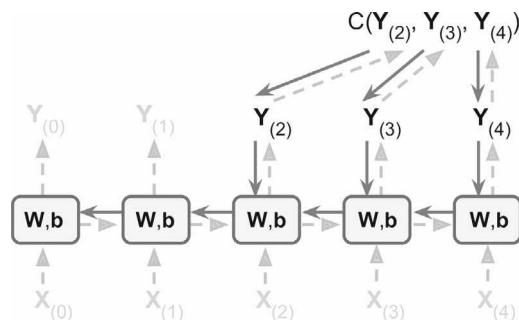


Figure 7.5 – Rétropropagation dans le temps

Il faut noter que les gradients sont transmis en arrière au travers de toutes les sorties utilisées par la fonction de coût et pas uniquement au travers de la sortie finale. Par exemple, à la figure 7.5, puisque la fonction de coût est calculée en utilisant les trois dernières sorties du réseau, $Y_{(2)}$, $Y_{(3)}$ et $Y_{(4)}$, les gradients passent par ces trois sorties, mais pas par $Y_{(0)}$ et $Y_{(1)}$. Par ailleurs, puisque les mêmes paramètres W et b sont utilisés à chaque étape temporelle, la rétropropagation opérera de la bonne manière et effectuera un cumul sur toutes les étapes temporelles.

Heureusement, tf.keras va s'occuper de toute cette complexité à notre place, alors commençons à coder !

7.3 PRÉVOIR DES SÉRIES CHRONOLOGIQUES

Supposons que vous soyez en train d'étudier le nombre d'utilisateurs actifs chaque heure sur votre site web, ou les températures quotidiennes dans votre ville, ou la santé financière de votre entreprise, évaluée trimestriellement d'après plusieurs indicateurs. Dans tous ces cas, les données seront une série d'une ou plusieurs valeurs pour chaque étape temporelle. C'est ce que nous appelons une *série chronologique*. Dans les deux premiers exemples, chaque étape temporelle comprend une seule valeur et nous avons donc des *séries chronologiques univariées*. En revanche, dans l'exemple financier, nous avons plusieurs valeurs à chaque étape temporelle (par exemple, les revenus de l'entreprise, la dette, etc.), il s'agit donc de *séries chronologiques multivariées*.

L'objectif classique est de prédire des valeurs futures, autrement dit faire des *prévisions*. Un autre peut être de remplir les vides, afin de prédire (ou plutôt « postdire ») les valeurs passées manquantes: une *imputation*. Par exemple, la figure 7.6 montre trois séries chronologiques univariées, chacune longue de 50 étapes temporelles, avec pour objectif de prévoir la valeur de l'étape temporelle suivante (représentée par la croix) dans chaque cas.

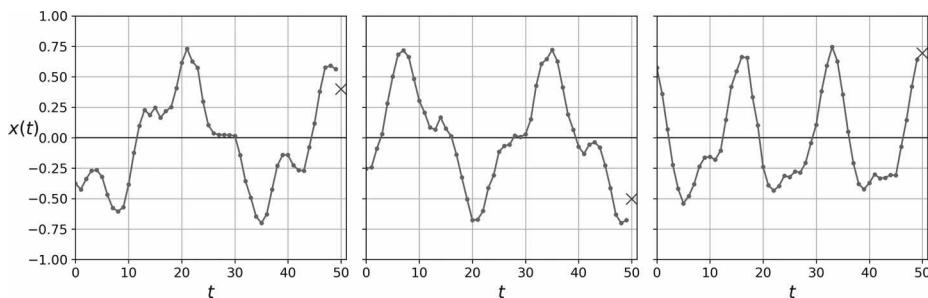


Figure 7.6 – Prévision de séries chronologiques

Pour une question de simplicité, nous utilisons des séries chronologiques générées par la fonction `generate_time_series()`:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) # courbe 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + courbe 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) # + bruit
    return series[...], np.newaxis).astype(np.float32)
```

Elle crée autant de séries chronologiques que demandé (à l'aide de l'argument `batch_size`), chacune de longueur `n_steps`. Chaque série comprend une seule valeur par étape temporelle (toutes les séries sont univariées). La fonction retourne un tableau NumPy de forme [taille du lot, étapes temporelles, 1], où chaque série est la somme de deux courbes sinusoïdales d'amplitude fixe, mais de fréquences et de phases aléatoires, auxquelles est ajouté un peu de bruit.



Lorsqu'on manipule des séries chronologiques (ainsi que d'autres types de séquences, comme des phrases), les caractéristiques d'entrée sont généralement représentées par des tableaux à trois dimensions ayant la forme [taille du lot, étapes temporelles, dimensionnalité], où dimensionnalité est égale à 1 pour les séries chronologiques univariées et à une valeur supérieure pour les séries chronologiques multivariées.

Créons à présent un jeu d'entraînement, un jeu de validation et un jeu de test à l'aide de cette fonction :

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

`X_train` contient 7 000 séries chronologiques (autrement dit, sa forme est [7 000, 50, 1]), tandis que `X_valid` en contient 2 000 (de la 7 000^e série chronologique à la 8 999^e) et `X_test`, 1 000 (de la 9 000^e à la 9 999^e). Puisque nous souhaitons prévoir une seule valeur pour chaque série, les cibles sont des vecteurs colonnes (par exemple, la forme de `y_train` est [7 000, 1]).

7.3.1 Indicateurs de référence

Avant d'utiliser des RNR, il vaut mieux disposer de quelques indicateurs de référence afin d'être certains que le fonctionnement de notre modèle est meilleur et non pire que les modèles de base. Par exemple, la méthode la plus simple consiste à prédire la dernière valeur dans chaque série. Il s'agit d'une *prévision simpliste*, mais elle est parfois difficile à réaliser correctement. Dans le cas suivant, nous obtenons une erreur quadratique moyenne d'environ 0,020 :

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Une autre approche simple se fonde sur un réseau intégralement connecté. Puisqu'il attend pour chaque entrée une liste plate de caractéristiques, nous devons ajouter une couche `Flatten`. Un simple modèle de régression linéaire permettra d'obtenir une prédiction qui soit une combinaison linéaire des valeurs des séries temporales :

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

Si nous compilons ce modèle en utilisant la perte MSE et l'optimiseur Adam par défaut, l'appliquons au jeu d'entraînement pendant vingt époques et l'évaluons sur le jeu de validation, nous obtenons une MSE d'environ 0,004. Le résultat est bien meilleur que celui de l'approche simpliste !

7.3.2 Implémenter un RNR simple

Voyons si nous pouvons faire mieux avec un RNR simple :

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Voilà le RNR vraiment le plus simple que vous puissiez construire. Il comprend une seule couche, avec un seul neurone, comme nous l'avons vu à la figure 7.1. Il est inutile de préciser la longueur des séries d'entrée (contrairement au modèle précédent), car un réseau de neurones récurrents est capable de traiter un nombre quelconque d'étapes temporelles (voilà pourquoi nous fixons la première dimension d'entrée à `None`). Par défaut, la couche `SimpleRNN` applique la tangente hyperbolique comme fonction d'activation. Elle opère exactement comme nous l'avons vu précédemment : l'état initial $h_{(init)}$ est fixé à 0, puis il est passé à un seul neurone récurrent, accompagné de la valeur de la première étape temporelle, $x_{(0)}$.

Le neurone calcule une somme pondérée de ces valeurs et applique la fonction d'activation tangente hyperbolique au résultat. Nous obtenons alors la première sortie, y_0 . Dans un RNR, cette sortie est également le nouvel état h_0 . Il est passé au même neurone récurrent, avec la valeur d'entrée suivante $x_{(1)}$. Le processus se répète jusqu'à la dernière étape temporelle. La couche produit alors la dernière valeur, y_{49} . Tout cela est réalisé simultanément pour chaque série chronologique.



Dans Keras, les couches récurrentes retournent par défaut uniquement la sortie finale. Pour qu'elles retournent une sortie par étape temporelle, vous devez indiquer `return_sequences=True`.

Si vous compilez, préparez et évaluez ce modèle (comme précédemment, l'entraînement se fait sur vingt époques avec Adam), vous constaterez que sa MSE arrive à 0,014 seulement. C'est mieux que l'approche simpliste, mais nous n'avons pas battu le modèle linéaire simple. Notez que pour chaque neurone, un modèle linéaire comprend un paramètre par entrée et par étape temporelle, plus un terme constant (dans le modèle linéaire simple que nous avons utilisé, le nombre total de paramètres est égal à 51). À l'opposé, pour chaque neurone récurrent d'un RNR simple, nous avons un paramètre par entrée et par dimension d'état caché (dans un RNR simple, cela correspond simplement au nombre de neurones récurrents dans la couche), plus un terme constant. Dans notre exemple de base, cela représente un total de trois paramètres uniquement.

Tendance et saisonnalité

Il existe de nombreux autres modèles pour la prévision de séries chronologiques, par exemple les modèles de *moyenne mobile pondérée* ou les modèles ARIMA (*auto-regressive integrated moving average*). Certains d'entre eux vous obligent à retirer tout d'abord la tendance et la saisonnalité.

Par exemple, si vous étudiez le nombre d'utilisateurs actifs sur votre site web et s'il croît de 10 % chaque mois, vous devez tout d'abord retirer cette tendance dans les séries temporelles. Après l'entraînement du modèle, lorsqu'il commence à effectuer des prédictions, vous devez la réintégrer pour obtenir les prévisions finales.

De manière comparable, si vous tentez de prédire la quantité de lotions solaires vendues chaque mois, vous observerez probablement une saisonnalité importante. Puisqu'elles se vendent bien pendant l'été, ce même schéma va se répéter tous les ans. Vous devez donc retirer cette saisonnalité des séries chronologiques, par exemple en calculant la différence entre la valeur à chaque étape temporelle et celle de l'année précédente (cette technique est appelée *déférenciation*). De nouveau, après l'entraînement du modèle, lorsqu'il commence ses prévisions, vous devez remettre la saisonnalité pour obtenir les prévisions finales.

Avec les RNR, tout cela est en général inutile. Mais, dans certains cas, les performances peuvent s'en trouver améliorées, car le modèle n'aura pas à apprendre la tendance et la saisonnalité.

Apparemment, notre RNR était trop simple pour arriver à de bonnes performances. Voyons ce que cela donne en ajoutant d'autres couches récurrentes !

7.3.3 RNR profonds

Il est assez fréquent d'empiler plusieurs couches de cellules de mémoire (voir la figure 7.7), afin d'obtenir un *RNR profond*.

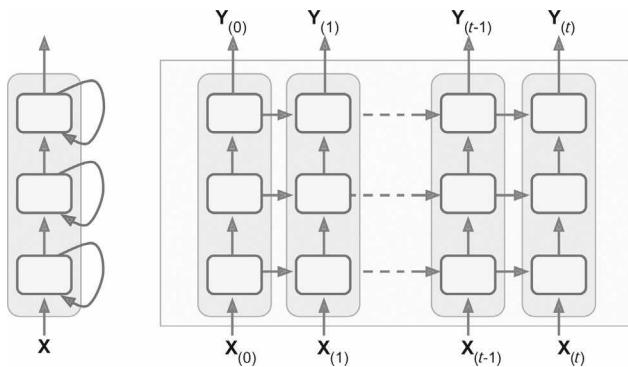


Figure 7.7 – RNR profond (à gauche), déplié dans le temps (à droite)

L'implémentation d'un RNR profond avec tf.keras est plutôt simple : il suffit d'empiler des couches récurrentes. Dans l'exemple suivant, nous utilisons trois couches

SimpleRNN (mais nous pouvons ajouter n'importe quel autre type de couche récurrente, comme une couche LSTM ou une couche GRU; nous le verrons plus loin):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```



N'oubliez pas de préciser `return_sequences=True` pour toutes les couches récurrentes (à l'exception de la dernière, si seule la dernière sortie vous intéresse). Dans le cas contraire, elles produiront un tableau à deux dimensions (contenant uniquement la sortie de la dernière étape temporelle) à la place d'un tableau à trois dimensions (contenant les sorties de toutes les étapes temporelles), et la couche récurrente suivante manifestera son mécontentement car vous ne lui fournissez pas des séquences dans le format 3D qu'elle attend.

Si vous compilez, préparez et évaluez ce modèle, vous constaterez que sa MSE arrive à 0,003. Finalement, nous avons réussi à battre le modèle linéaire !

Vous noterez que la dernière couche n'est pas idéale : puisque nous voulons prévoir une série chronologique univariée, et donc avoir une seule valeur de sortie par étape temporelle, elle ne doit avoir qu'une seule unité. Mais, cette seule unité signifie que l'état caché est un seul nombre. C'est vraiment peu et probablement pas très utile ; il est fort probable que le RNR utilisera principalement les états cachés des autres couches récurrentes pour reporter d'une étape temporelle à l'autre toutes les informations dont il a besoin et qu'il n'utilisera pas l'état caché de la dernière couche.

Par ailleurs, puisqu'une couche SimpleRNN se sert par défaut de la fonction d'activation tanh, les valeurs prédites doivent se situer dans la plage -1 à 1. Et si vous souhaitez utiliser une autre fonction d'activation ? Pour ces deux raisons, il peut être plus intéressant de remplacer la couche de sortie par une couche Dense : son exécution sera légèrement plus rapide, la précision sera quasi identique, et nous pourrons choisir la fonction d'activation de sortie qui nous convient. Si vous effectuez ce changement, n'oubliez pas de retirer `return_sequences=True` de la deuxième (à présent la dernière) couche récurrente :

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

Si vous entraînez ce modèle, vous constaterez qu'il converge plus rapidement et que ses performances sont aussi bonnes. Mais, en bonus, vous pouvez modifier la fonction d'activation de sortie si nécessaire.

7.3.4 Prévision de plusieurs étapes temporelles à l'avance

Jusqu'à présent, nous avons uniquement prédit la valeur à l'étape temporelle suivante, mais nous aurions pu tout aussi bien prédire la valeur plusieurs étapes à l'avance en

modifiant les cibles de façon appropriée (par exemple, pour effectuer la prédiction dix étapes à l'avance, il suffit de changer les cibles de sorte qu'elles aient la valeur pour dix étapes à l'avance plutôt qu'une seule). Mais comment pouvons-nous prédire les dix prochaines valeurs ?

La première solution consiste à utiliser le modèle déjà entraîné, à lui faire prédire la valeur suivante, à ajouter cette valeur aux entrées (faire comme si cette valeur prédite s'était déjà produite), et à utiliser à nouveau le modèle pour prédire la valeur d'après, et ainsi de suite :

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

Vous l'aurez deviné, la prédiction de l'étape suivante sera généralement plus précise que celles des étapes temporelles ultérieures, car les erreurs peuvent se cumuler (voir la figure 7.8). Si vous évaluez cette approche sur le jeu de validation, vous obtiendrez une MSE d'environ 0,029. Le résultat est plus élevé qu'avec les modèles précédents, mais la tâche est beaucoup plus difficile et la comparaison n'a pas vraiment de sens. Il est plus intéressant de comparer cette performance avec les prédictions simplistes (prévoir simplement que la série temporelle restera constante pendant dix étapes temporelles) ou avec un modèle linéaire simple. L'approche simpliste est minable (sa MSE est d'environ 0,223), mais le modèle linéaire obtient une MSE d'environ 0,0188. C'est beaucoup mieux qu'avec notre RNR qui prévoit le futur une étape à la fois, sans compter que l'entraînement et l'exécution sont beaucoup plus rapides. Néanmoins, pour effectuer des prévisions quelques étapes temporelles à l'avance, sur des tâches plus complexes, cette approche peut parfaitement fonctionner.

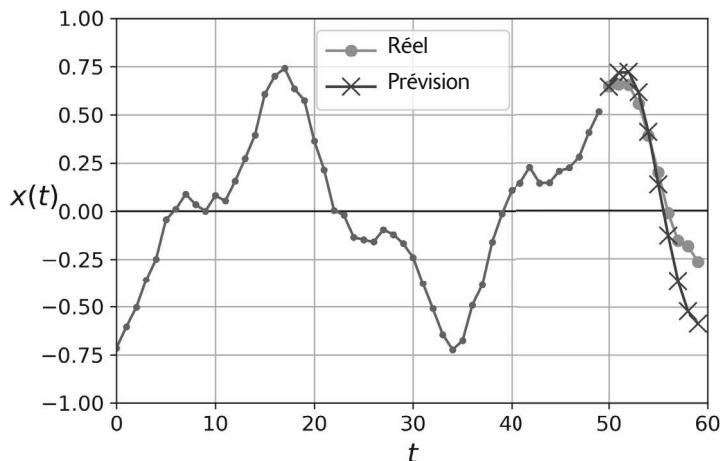


Figure 7.8 – Prévisions dix étapes à l'avance, une étape à la fois

La deuxième solution est d'entraîner un RNR pour qu'il prédise les dix prochaines valeurs en une fois. Nous pouvons toujours utiliser un modèle série-vers-vecteur, mais il produira dix valeurs à la place d'une seule. Nous devons commencer par modifier les cibles pour qu'elles soient des vecteurs contenant les dix prochaines valeurs :

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Puis, il suffit d'ajouter une couche de sortie avec dix unités au lieu d'une :

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

Après avoir entraîné ce modèle, vous pouvez prédire très facilement les dix prochaines valeurs en une fois :

```
Y_pred = model.predict(X_new)
```

Ce modèle donne de bons résultats : la MSE pour les dix étapes suivantes est d'environ 0,008. C'est beaucoup mieux que le modèle linéaire. Mais nous pouvons faire mieux encore : au lieu d'entraîner le modèle à prévoir les dix valeurs suivantes uniquement lors de la toute dernière étape temporelle, nous pouvons l'entraîner pour la prévision des dix valeurs suivantes à chaque étape temporelle. Autrement dit, nous pouvons convertir ce RNR série-vers-vecteur en un RNR série-vers-série. Grâce à cette technique, la perte contiendra un terme pour la sortie du RNR non pas uniquement lors de la dernière étape temporelle mais lors de chaque étape temporelle. Autrement dit, le nombre de gradients d'erreur circulant dans le modèle sera plus élevé et ce ne sera pas uniquement en suivant le temps ; ils découleront également de la sortie de chaque étape temporelle. L'entraînement en sera ainsi stabilisé et accéléré.

Pour être plus clair, à l'étape temporelle 0, le modèle produira un vecteur contenant les prévisions pour les étapes 1 à 10, puis, à l'étape temporelle 1, le modèle prévoira les étapes temporelles 2 à 11, et ainsi de suite. Par conséquent, chaque cible doit être une série de même longueur que celle d'entrée, en contenant un vecteur à dix dimensions à chaque étape. Préparons ces séries cibles :

```
Y = np.empty((10000, n_steps, 10)) # Chaque cible est une suite de vecteurs
# à dix dimensions
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```



Vous pourriez être surpris que les cibles contiennent des valeurs qui apparaissent dans les entrées (la similitude entre X_train et Y_train est importante). N'est-ce pas tricher ? Non, absolument pas : à chaque étape temporelle, le modèle connaît uniquement les étapes temporelles passées et ne regarde donc pas en avant. Il s'agit d'un modèle *causal*.

Pour transformer le modèle en une version série-vers-série, nous devons préciser `return_sequences=True` dans toutes les couches récurrentes (même la dernière) et nous devons appliquer la couche de sortie `Dense` à chaque étape temporelle. Dans ce but, Keras fournit la couche `TimeDistributed`: elle enveloppe n'importe quelle couche (par exemple, une couche `Dense`) et l'applique à chaque étape temporelle de sa série d'entrée. Elle procède de façon efficace, en modifiant les entrées de sorte que chaque étape temporelle soit traitée comme une instance distincte (autrement dit, elle change la forme des entrées de `[taille de lot, étapes temporelles, dimensions d'entrée]` à `[taille de lot × étapes temporelles, dimensions d'entrée]`; dans notre exemple, le nombre de dimensions d'entrée est égal à 20, car la couche `SimpleRNN` précédente comprend vingt unités), puis elle exécute la couche `Dense`, et, pour finir, remet les sorties sous forme de séries (autrement dit, elle change la forme des sorties de `[taille de lot × étapes temporelles, dimensions d'entrée]` à `[taille de lot, étapes temporelles, dimensions d'entrée]`; dans notre exemple, le nombre de dimensions de sortie est égal à 10, car la couche `Dense` comprend dix unités)¹⁷⁸. Voici le modèle actualisé:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

La couche `Dense` accepte en entrée des séries (et même des entrées ayant des dimensions supérieures). Elle les traite comme `TimeDistributed(Dense(...))`, ce qui signifie qu'elle est appliquée uniquement à la dernière dimension d'entrée (indépendamment sur toutes les étapes). Nous pouvons donc remplacer la dernière couche par `Dense(10)`. Cependant, par souci de clarté, nous continuerons à utiliser `TimeDistributed(Dense(10))`, car cela montre bien que la couche `Dense` est appliquée indépendamment à chaque étape temporelle et que le modèle produit non pas un seul vecteur mais une série.

Toutes les sorties sont nécessaires au cours de l'entraînement, mais seule celle de la dernière étape temporelle est utile aux prévisions et à l'évaluation. Par conséquent, même si nous exploitons la MSE sur toutes les sorties pour l'entraînement, nous utiliserons un indicateur personnalisé pour l'évaluation afin de calculer la MSE uniquement sur la sortie de la dernière étape temporelle :

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

La MSE de validation obtenue est d'environ 0,006, c'est-à-dire une amélioration de 25 % par rapport au modèle précédent. Vous pouvez associer cette approche à la première : prédire uniquement les dix prochaines valeurs en utilisant ce RNR, puis concaténer ces valeurs à la série chronologique d'entrée, et utiliser à nouveau le modèle pour prédire les dix valeurs suivantes, en répétant cette procédure autant de

¹⁷⁸. Notez que les couches `TimeDistributed(Dense(n))` et `Conv1D(n, filter_size=1)` sont équivalentes.

fois que nécessaire. Avec cette approche, vous pouvez générer des séries de longueur quelconque. Elle ne sera peut-être pas très précise pour les prédictions à long terme, mais elle fera un bon travail si votre objectif est de générer un morceau de musique ou un texte original, comme nous le verrons au chapitre 8.



Lors de la prévision de séries chronologiques, il est souvent intéressant de disposer de barres d'erreur avec les prédictions. Pour cela, la technique MC Dropout, introduite au chapitre 3, est efficace: ajoutez une couche MC Dropout dans chaque cellule de mémoire, en supprimant une partie des entrées et des états cachés. Après l'entraînement, pour les prévisions d'une nouvelle série chronologique, utilisez plusieurs fois le modèle et calculez la moyenne et l'écart-type des prédictions lors de chaque étape temporelle.

Des RNR simples peuvent se révéler plutôt bons pour les prévisions de séries chronologiques ou le traitement d'autres sortes de séries, mais leurs performances sont moins probantes sur des séries chronologiques ou des séquences longues. Voyons pourquoi et voyons ce que nous pouvons faire.

7.4 TRAITER LES LONGUES SÉRIES

Pour entraîner un RNR sur des séries longues, nous devons l'exécuter sur de nombreuses étapes temporelles, faisant du RNR déroulé un réseau très profond. À l'instar de n'importe quel réseau de neurones profond, il peut être confronté au problème d'instabilité des gradients décrit au chapitre 3: l'entraînement peut durer indéfiniment ou devenir instable. Par ailleurs, lorsqu'un RNR traite une longue série, il en oublie progressivement les premières entrées. Nous allons voir comment résoudre ces deux problèmes, en commençant par l'instabilité des gradients.

7.4.1 Combattre le problème d'instabilité des gradients

Nous avons décrit plusieurs solutions pour alléger le problème d'instabilité des gradients dans les réseaux profonds et nombre d'entre elles peuvent être employées avec les RNR: une bonne initialisation des paramètres, des optimiseurs plus rapides, le dropout, etc. En revanche, les fonctions d'activation non saturantes (par exemple, ReLU) nous aideront peu dans ce cas; elles peuvent même empirer l'instabilité du RNR pendant l'entraînement.

Pour quelles raisons? Supposons que la descente de gradient actualise les poids de telle manière que les sorties augmentent légèrement lors de la première étape temporelle. Puisque les mêmes poids sont employés à chaque étape temporelle, les sorties de la deuxième étape peuvent également augmenter légèrement, puis celles de la troisième, et ainsi de suite jusqu'à ce que les sorties explosent – une fonction d'activation non saturante n'empêche pas ce comportement. Pour réduire ce risque, vous pouvez choisir un taux d'apprentissage plus faible ou simplement employer une fonction d'activation saturante comme la tangente hyperbolique (voilà pourquoi elle est activée par défaut). De manière comparable, les gradients eux-mêmes peuvent exploser. Si vous remarquez que l'entraînement est instable, surveillez la taille des

gradients (par exemple avec TensorBoard) et optez éventuellement pour l'écrêtage des gradients.

Par ailleurs, la normalisation par lots ne sera pas aussi efficace dans les RNR que dans les réseaux non bouclés profonds. En réalité, vous ne pouvez pas l'employer entre les étapes temporelles, mais uniquement entre les couches récurrentes. Pour être plus précis, il est techniquement possible d'ajouter une couche BN à une cellule de mémoire (comme nous le verrons bientôt) pour qu'elle soit appliquée à chaque étape temporelle (à la fois sur les entrées de cette étape temporelle et sur l'état caché de l'étape précédente). Mais, la même couche BN sera utilisée à chaque étape temporelle, avec les mêmes paramètres, quels que soient la mise à l'échelle et le décalage réels des entrées et de l'état caché.

En pratique, cela ne donne pas de bons résultats, comme l'ont démontré César Laurent *et al.* dans un article¹⁷⁹ publié en 2015. Les auteurs ont constaté que la normalisation par lots apportait un léger bénéfice uniquement si elle était appliquée non pas aux états cachés mais aux entrées. Autrement dit, elle avait un petit intérêt lorsqu'elle était appliquée entre les couches récurrentes (c'est-à-dire verticalement dans la figure 7.7), mais aucun à l'intérieur des couches récurrentes (c'est-à-dire horizontalement). Dans Keras, cette solution peut être mise en place simplement en ajoutant une couche `BatchNormalization` avant chaque couche récurrente, mais n'en attendez pas trop d'elle.

Il existe une autre forme de normalisation mieux adaptée aux RNR : la *normalisation par couches*. Cette idée a été émise par Jimmy Lei Ba *et al.* dans un article¹⁸⁰ publié en 2016. Elle ressemble énormément à la normalisation par lots, mais elle se fait non plus suivant les lots mais suivant les caractéristiques. L'intérêt est que le calcul des statistiques requises peut se faire à la volée, à chaque étape temporelle, indépendamment pour chaque instance. En conséquence, elle se comporte de la même manière pendant l'entraînement et les tests (contrairement à la normalisation par lots), et elle n'a pas besoin des moyennes mobiles exponentielles pour estimer les statistiques de caractéristiques sur toutes les instances du jeu d'entraînement. À l'instar de la normalisation par lots, la normalisation par couche apprend un paramètre de mise à l'échelle et un paramètre de décalage pour chaque entrée. Dans un RNR, elle se place généralement juste après la combinaison linéaire des entrées et des états cachés.

Utilisons `tf.keras` pour mettre en place une normalisation par couche à l'intérieure d'une cellule de mémoire simple. Pour cela, nous devons définir une cellule de mémoire personnalisée. Elle est comparable à une couche normale, excepté que sa méthode `call()` attend deux arguments : les entrées `inputs` de l'étape temporelle courante et les états cachés `states` de l'étape temporelle précédente. Notez que l'argument `states` est une liste contenant un ou plusieurs tenseurs. Dans le cas d'une cellule de RNR simple, il contient un seul tenseur égal aux sorties de l'étape temporelle précédente, mais d'autres cellules pourraient avoir des tenseurs à plusieurs

179. César Laurent *et al.*, « Batch Normalized Recurrent Neural Networks », *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016), 2657-2661 : <https://homl.info/rnnbn>.

180. Jimmy Lei Ba *et al.*, « Layer Normalization » (2016) : <https://homl.info/layernorm>.

états (par exemple, un `LSTMCell` possède un état à long terme et un état à court terme, comme nous le verrons plus loin).

Une cellule doit également avoir les attributs `state_size` et `output_size`. Dans un RNR simple, tous deux sont simplement égaux au nombre d'unités. Le code suivant implémente une cellule de mémoire personnalisée qui se comporte comme un `SimpleRNNCell`, hormis la normalisation par couche appliquée à chaque étape temporelle :

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Le code ne pose pas vraiment de difficultés¹⁸¹. Notre classe `LNSimpleRNNCell` dérive de la classe `keras.layers.Layer`, comme n'importe quelle couche personnalisée. Le constructeur prend en paramètres le nombre d'unités et la fonction d'activation souhaitée. Il fixe la valeur des attributs `state_size` et `output_size`, puis crée un `SimpleRNNCell` sans fonction d'activation (car nous voulons appliquer la normalisation par couche après l'opération linéaire, mais avant la fonction d'activation). Ensuite, il crée la couche `LayerNormalization` et finit par récupérer la fonction d'activation souhaitée. La méthode `call()` commence par appliquer la cellule RNR simple, qui calcule une combinaison linéaire des entrées actuelles et des états cachés précédents, et retourne deux fois le résultat (dans un `SimpleRNNCell`, les sorties sont simplement égales aux états cachés : autrement dit, `new_states[0]` est égal à `outputs` et nous pouvons ignorer `new_states` dans le reste de la méthode `call()`). Puis, la méthode `call()` applique la normalisation par couches, suivie de la fonction d'activation. Pour finir, elle retourne deux fois les sorties (une fois en tant que sorties et une fois en tant que nouveaux états cachés).

Pour profiter de cette cellule personnalisée, nous devons simplement créer une couche `keras.layers.RNN` en lui passant une instance de la cellule :

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                    input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

¹⁸¹. Il aurait été plus simple de dériver de `SimpleRNNCell` afin de ne pas avoir à créer un `SimpleRNNCell` interne ni à gérer les attributs `state_size` et `output_size`, mais l'objectif ici est de montrer comment créer une cellule personnalisée à partir de zéro.

De manière comparable, vous pouvez créer une cellule personnalisée pour appliquer un dropout entre chaque étape temporelle. Mais il existe une solution plus simple : toutes les couches récurrentes (à l'exception de `keras.layers.RNN`) et toutes les cellules fournies par Keras disposent des hyperparamètres `dropout` et `recurrent_dropout`. Le premier définit le taux d'extinction appliqué aux entrées (à chaque étape temporelle), le second, celui destiné aux états cachés (également à chaque étape temporelle). Il est donc inutile de créer une cellule personnalisée pour appliquer le dropout à chaque étape temporelle dans un RNR.

À l'aide de ces techniques, vous pouvez alléger le problème d'instabilité des gradients et entraîner un RNR beaucoup plus efficacement. Voyons à présent comment traiter le problème de mémoire à court terme.

7.4.2 Combattre le problème de mémoire à court terme

En raison des transformations appliquées aux données lors de la traversée d'un RNR, certaines informations sont perdues après chaque étape temporelle. Au bout d'un certain temps, l'état du RNR ne contient quasi plus de traces des premières entrées. Ce comportement peut être rédhibitoire. Imaginons que Dory¹⁸² essaie de traduire une longue phrase ; lorsqu'elle a terminé sa lecture, elle n'a plus d'informations sur son début. Pour résoudre ce problème, plusieurs types de cellules avec une mémoire à long terme ont été imaginés. Leur efficacité a été telle que les cellules de base ne sont quasi plus employées. Commençons par étudier la plus populaire de ces cellules, la cellule LSTM.

Cellules LSTM

La cellule de *longue mémoire à court terme* (LSTM, Long Short-Term Memory) a été proposée¹⁸³ en 1997 par Sepp Hochreiter et Jürgen Schmidhuber. Elle a été progressivement améliorée au fil des ans par plusieurs chercheurs, comme Alex Graves¹⁸⁴, Haşim Sak¹⁸⁵ et Wojciech Zaremba¹⁸⁶. Si l'on considère la cellule LSTM comme une boîte noire, on peut s'en servir presque comme une cellule de base, mais avec de bien meilleures performances. L'entraînement convergera plus rapidement et détectera les dépendances à long terme présentes dans les données. Dans Keras, il suffit de remplacer la couche `SimpleRNN` par une couche LSTM :

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

182. Un personnage des films d'animation *Le monde de Nemo* et *Le monde de Dory* qui souffre de pertes de mémoire à court terme.

183. Sepp Hochreiter et Jürgen Schmidhuber, « Long Short-Term Memory », *Neural Computation*, 9, n° 8 (1997), 1735-1780 : <https://homl.info/93>.

184. <https://homl.info/graves>

185. Haşim Sak *et al.*, « Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition » (2014) : <https://homl.info/94>.

186. Wojciech Zaremba *et al.*, « Recurrent Neural Network Regularization » (2014) : <https://homl.info/95>.

Une autre solution consiste à utiliser la couche générique `keras.layers.RNN`, en lui passant `LSTMCell` en argument :

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                    input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Puisque l'implémentation de la couche LSTM a été optimisée pour une exécution sur les GPU (voir le chapitre 11), il est préférable de l'employer (la couche RNN est surtout utile lors de la définition de cellules personnalisées, comme nous l'avons fait précédemment).

Comment une cellule LSTM fonctionne-t-elle ? Son architecture est illustrée par la figure 7.9.

Si l'on ignore le contenu de la boîte noire, la cellule LSTM ressemble fortement à une cellule normale, à l'exception de son état qui est divisé en deux vecteurs : $\mathbf{h}_{(t)}$ et $\mathbf{c}_{(t)}$ (« c » pour « cellule »). $\mathbf{h}_{(t)}$ peut être vu comme l'état à court terme et $\mathbf{c}_{(t)}$ comme l'état à long terme.

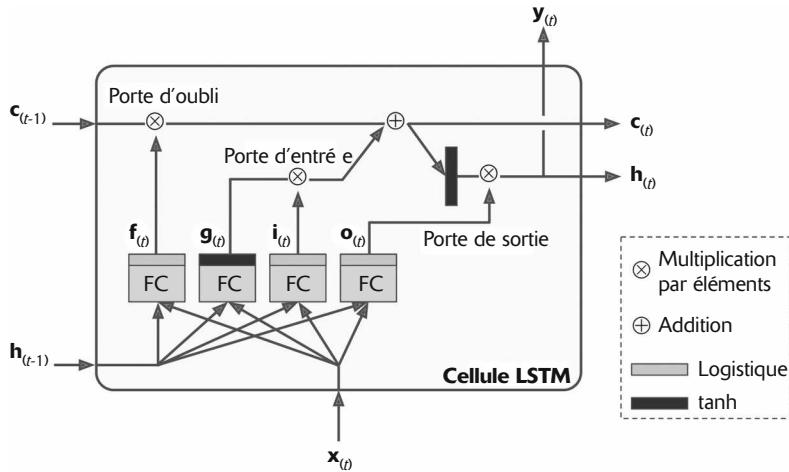


Figure 7.9 – Cellule LSTM (FC: couche intégralement connectée)

Ouvrons la boîte ! Voici l'idée centrale : le réseau peut apprendre ce qu'il faut stocker dans l'état à long terme, ce qui doit être oublié et ce qu'il faut y lire. Le parcours de l'état à long terme $\mathbf{c}_{(t-1)}$ au travers du réseau va de la gauche vers la droite. Il passe tout d'abord par une *porte d'oubli* (*forget gate*), qui abandonne certaines informations, puis en ajoute de nouvelles via l'opération d'addition (les informations ajoutées sont sélectionnées par une *porte d'entrée* [*input gate*]), et le résultat $\mathbf{c}_{(t)}$ est envoyé directement, sans autre transformation. Par conséquent, à chaque étape temporelle, des informations sont retirées et d'autres sont ajoutées. Par ailleurs, après l'opération

d'addition, l'état à long terme est copié et soumis à la fonction tanh, dont le résultat est filtré par la *porte de sortie* (*output gate*). On obtient alors l'état à court terme $\mathbf{h}_{(t)}$, qui est égal à la sortie de la cellule pour l'étape temporelle $\mathbf{y}_{(t)}$. Voyons d'où proviennent les nouvelles informations et comment fonctionnent les portes.

Premièrement, le vecteur d'entrée courant $\mathbf{x}_{(t)}$ et l'état à court terme précédent $\mathbf{h}_{(t-1)}$ sont fournis à quatre couches intégralement connectées, ayant toutes un objectif différent :

- La couche principale génère $\mathbf{g}_{(t)}$: elle joue le rôle habituel d'analyse des entrées courantes $\mathbf{x}_{(t)}$ et de l'état précédent (à court terme) $\mathbf{h}_{(t-1)}$. Une cellule de base comprend uniquement cette couche et sa sortie est transmise directement à $\mathbf{y}_{(t)}$ et à $\mathbf{h}_{(t)}$. En revanche, dans une cellule LSTM, la sortie de cette couche ne se fait pas directement mais ses parties les plus importantes sont stockées dans l'état à long terme (le reste est abandonné).
- Les trois autres couches sont des *contrôleurs de porte*. Puisqu'elles utilisent la fonction d'activation logistique, leurs sorties sont dans la plage 0 à 1. Celles-ci étant passées à des opérations de multiplication par éléments, une valeur 0 ferme la porte, tandis qu'une valeur 1 l'ouvre. Plus précisément :
 - La *porte d'oubli* (contrôlée par $\mathbf{f}_{(t)}$) décide des parties de l'état à long terme qui doivent être effacées.
 - La *porte d'entrée* (contrôlée par $\mathbf{i}_{(t)}$) choisit les parties de $\mathbf{g}_{(t)}$ qui doivent être ajoutées à l'état à long terme.
 - La *porte de sortie* (contrôlée par $\mathbf{o}_{(t)}$) sélectionne les parties de l'état à long terme qui doivent être lues et produites lors de cette étape temporelle, à la fois dans $\mathbf{h}_{(t)}$ et dans $\mathbf{y}_{(t)}$.

En résumé, une cellule LSTM peut apprendre à reconnaître une entrée importante (le rôle de la porte d'entrée), la stocker dans l'état à long terme, la conserver aussi longtemps que nécessaire (le rôle de la porte d'oubli) et l'extraire lorsqu'elle est requise. Cela explique pourquoi elles réussissent très bien à identifier des motifs à long terme dans des séries chronologiques, des textes longs, des enregistrements audio, etc.

L'équation 7.3 récapitule le calcul de l'état à long terme d'une cellule, de son état à court terme et de sa sortie à chaque étape temporelle, pour une seule instance (les équations pour un mini-lot complet sont très similaires).

Équation 7.3 – Calculs LSTM

$$\begin{aligned}
 i_{(t)} &= \sigma\left(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i\right) \\
 f_{(t)} &= \sigma\left(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f\right) \\
 o_{(t)} &= \sigma\left(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o\right) \\
 g_{(t)} &= \tanh\left(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g\right) \\
 c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
 y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})
 \end{aligned}$$

Dans cette équation :

- W_{xi} , W_{xf} , W_{xo} et W_{xg} sont les matrices de poids de chacune des quatre couches pour leurs connexions au vecteur d'entrée $x_{(t)}$.
- W_{hi} , W_{hf} , W_{ho} et W_{hg} sont les matrices de poids de chacune des quatre couches pour leurs connexions à l'état à court terme précédent $h_{(t-1)}$.
- b_i , b_f , b_o et b_g sont les termes constants pour chacune des quatre couches. TensorFlow initialise b_f à un vecteur non pas de 0 mais de 1. Cela évite que tout soit oublié dès le début de l'entraînement.

Connexions judas

Dans une cellule LSTM de base, les contrôleurs de porte peuvent uniquement examiner l'entrée $x_{(t)}$ et l'état à court terme précédent $h_{(t-1)}$. Il pourrait être intéressant de leur donner un contexte plus large en leur permettant également de jeter un œil sur l'état à long terme. Cette idée a été émise en 2000 par Felix Gers et Jürgen Schmidhuber¹⁸⁷. Ils ont proposé une variante de LSTM avec des connexions supplémentaires appelées *connexions judas* (*peephole connections*). L'état à long terme précédent $c_{(t-1)}$ est ajouté en tant qu'entrée des contrôleurs des portes d'oubli et d'entrée, et l'état à long terme courant $c_{(t)}$ est ajouté en tant qu'entrée du contrôleur de la porte de sortie. Les performances s'en trouvent souvent améliorées, mais pas toujours, et aucun critère précis ne permet de savoir à l'avance si ce sera le cas. Vous devrez essayer cette technique avec votre tâche spécifique pour le déterminer.

Dans Keras, la couche LSTM se fonde sur la cellule `keras.layers.LSTMCell`, qui ne prend pas en charge les connexions judas. En revanche, c'est le cas de la cellule expérimentale `tf.keras.experimental.PeepholeLSTMCell`. Il vous suffit donc de créer une couche `keras.layers.RNN` et de passer un `PeepholeLSTMCell` à son constructeur.

Il existe de nombreuses autres variantes de la cellule LSTM, dont l'une est particulièrement répandue et que nous allons examiner à présent : la cellule GRU.

¹⁸⁷ F. A. Gers et J. Schmidhuber, « Recurrent Nets That Time and Count », *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000), 189-194 : <https://homl.info/96>.

Cellule GRU

La cellule d'*unité récurrente à porte* (GRU, Gated Recurrent Unit), illustrée à la figure 7.10, a été proposée¹⁸⁸ en 2014 par Kyunghyun Cho *et al.* Dans leur article, ils décrivent également le réseau encodeur-décodeur mentionné précédemment.

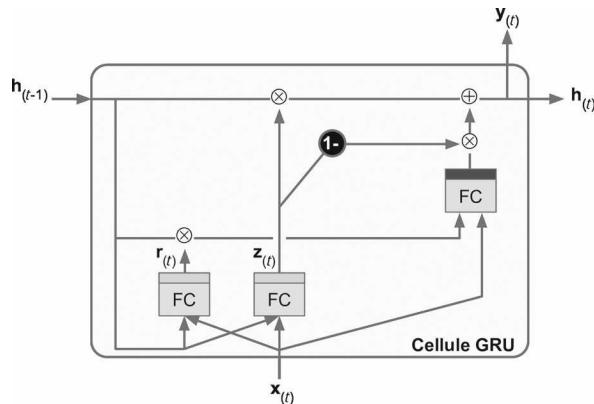


Figure 7.10 – Cellule GRU (FC: couche intégralement connectée)

La cellule GRU est une version simplifiée de la cellule LSTM. Puisque ses performances semblent tout aussi bonnes¹⁸⁹, sa popularité ne fait que croître. Voici les principales simplifications :

- Les deux vecteurs d'état sont fusionnés en un seul vecteur $\mathbf{h}_{(t)}$.
- Un seul contrôleur de porte $\mathbf{z}_{(t)}$ s'occupe des portes d'oubli et d'entrée. S'il produit un 1, la porte d'oubli est ouverte ($= 1$) et la porte d'entrée est fermée ($1 - 1 = 0$). S'il produit un 0, la logique inverse s'applique. Autrement dit, dès qu'une information doit être stockée, son emplacement cible est tout d'abord effacé. Il s'agit en réalité d'une variante répandue de la cellule LSTM en soi.
- La porte de sortie a disparu : le vecteur d'état complet est sorti à chaque étape temporelle. Toutefois, un nouveau contrôleur de porte $\mathbf{r}_{(t)}$ décide des parties de l'état précédent qui seront présentées à la couche principale ($\mathbf{g}_{(t)}$).

L'équation 7.4 résume le calcul de l'état de la cellule à chaque étape temporelle pour une seule instance.

188. Kyunghyun Cho *et al.*, « Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation », *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014), 1724-1734 : <https://homl.info/97>.

189. Un article publié en 2015 par Klaus Greff *et al.* (« LSTM: A Search Space Odyssey ») montre que toutes les variantes de LSTM ont des performances quasi équivalentes ; il est disponible à l'adresse <https://homl.info/98>.

Équation 7.4 – Calculs GRU

$$\begin{aligned} z_{(t)} &= \sigma\left(W_{xz}^T x_{(t)} + W_{hz}^T h_{(t-1)} + b_z\right) \\ r_{(t)} &= \sigma\left(W_{xr}^T x_{(t)} + W_{hr}^T h_{(t-1)} + b_r\right) \\ g_{(t)} &= \tanh\left(W_{xg}^T x_{(t)} + W_{hg}^T \left(r_{(t)} \otimes h_{(t-1)}\right) + b_g\right) \\ h_{(t)} &= z_{(t)} \otimes h_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)} \end{aligned}$$

Keras fournit une couche `keras.layers.GRU` (fondée sur la cellule de mémoire `keras.layers.GRUCell`) ; pour l'utiliser, il suffit de remplacer `SimpleRNN` ou `LSTM` par `GRU`.

Les cellules LSTM et GRU ont énormément contribué au succès des RNR. Toutefois, bien qu'elles permettent de traiter des séries plus longues que les RNR simples, leur mémoire à court terme reste assez limitée et il leur est difficile d'apprendre des motifs à long terme dans des séries de 100 étapes temporelles ou plus, comme les échantillons audio, les longues séries chronologiques ou les longues phrases. Une solution consiste à raccourcir la série d'entrée, par exemple en utilisant des couches de convolution à une dimension.

Utiliser des couches de convolution à une dimension pour traiter des séries

Au chapitre 6, nous avons vu qu'une couche de convolution à deux dimensions travaille en déplaçant plusieurs noyaux (ou filtres) assez petits sur une image, en produisant plusieurs cartes de caractéristiques (une par noyau) à deux dimensions. De façon comparable, une couche de convolution à une dimension fait glisser plusieurs noyaux sur une série, en produisant une carte de caractéristiques à une dimension par noyau. Chaque noyau va apprendre à détecter un seul très court motif séquentiel (pas plus long que la taille du noyau). Avec dix noyaux, la couche de sortie sera constituée de dix séries à une dimension (toutes de la même longueur) ; vous pouvez également voir cette sortie comme une seule séquence de dimension 10.

Autrement dit, vous pouvez construire un réseau de neurones qui mélange les couches récurrentes et les couches de convolution à une dimension (ou même les couches de pooling à une dimension). En utilisant une couche de convolution à une dimension avec un pas de 1 et un remplissage "same", la série de sortie aura la même longueur que la série d'entrée. En revanche, avec un remplissage "valid" ou un pas supérieur à 1, la série de sortie sera plus courte que la série d'entrée ; n'oubliez pas d'ajuster les cibles en conséquence.

Par exemple, le modèle suivant est identique au précédent, excepté qu'il débute par une couche de convolution à une dimension qui sous-échantillonne la série d'entrée avec un facteur 2, en utilisant un pas de 2. Puisque la taille du noyau est plus importante que le pas, toutes les entrées serviront au calcul de la sortie de la couche et le modèle peut donc apprendre à conserver les informations utiles, ne retirant que les détails non pertinents. En raccourcissant les séries, la couche de convolution peut aider les couches GRU à détecter des motifs plus longs. Notez que vous devez

également couper les trois premières étapes temporelles dans les cibles (la taille du noyau étant égale à 4, la première entrée de la couche de convolution sera fondée sur les étapes temporelles d'entrée 0 à 3) et sous-échantillonner les cibles d'un facteur 2 :

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                      validation_data=(X_valid, Y_valid[:, 3::2]))
```

Si vous entraînez et évaluez ce modèle, vous constaterez qu'il est le meilleur de tous ceux vus jusqu'à présent. L'aide de la couche de convolution est bien réelle. Il est même possible d'utiliser uniquement des couches de convolution à une dimension et de retirer totalement les couches récurrentes !

WaveNet

Dans un article¹⁹⁰ de 2016, Aaron van den Oord et d'autres chercheurs de DeepMind ont présenté une architecture nommée *WaveNet*. Ils ont empilé des couches de convolution à une dimension, en doublant le taux de dilatation (la distance de séparation entre les entrées d'un neurone) à chaque nouvelle couche : la première couche de convolution reçoit uniquement deux étapes temporelles à la fois), tandis que la suivante en voit quatre (son champ récepteur est long de quatre étapes temporelles), celle d'après en voit huit, et ainsi de suite (voir la figure 7.11). Ainsi, les couches inférieures apprennent des motifs à court terme, tandis que les couches supérieures apprennent des motifs à long terme. Grâce au taux de dilatation qui double, le réseau peut traiter très efficacement des séries extrêmement longues.

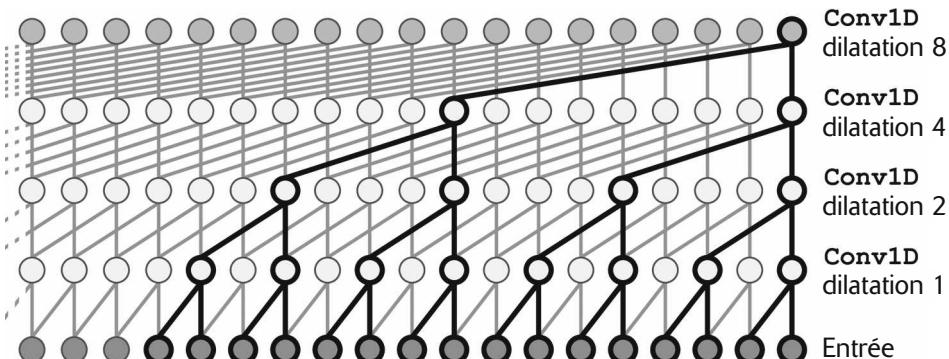


Figure 7.11 – Architecture de WaveNet

190. Aaron van den Oord *et al.*, « WaveNet: A Generative Model for Raw Audio », (2016) : <https://homl.info/wavenet>.

Dans leur article, les auteurs ont empilé dix couches de convolution avec des taux de dilatation égaux à 1, 2, 4, 8, ..., 256, 512, puis un autre groupe de dix couches identiques (toujours avec les mêmes taux de dilatation), et encore un autre groupe identique. Pour justifier cette architecture, ils ont souligné qu'une seule pile de dix couches de convolution avec ces taux de dilatation fonctionnera comme une couche de convolution super efficace avec un noyau de taille 1 024 (en étant plus rapide et plus puissante, et en demandant beaucoup moins de paramètres). Voilà pourquoi ils ont empilé trois de ces blocs. Avant chaque couche, ils ont également appliqué un remplissage des séries d'entrée avec un nombre de zéros égal au taux de dilatation, cela pour que les séries conservent la même longueur tout au long du réseau. Voici comment implémenter un WaveNet simple pour traiter les mêmes séries que précédemment¹⁹¹:

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                 activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                      validation_data=(X_valid, Y_valid))
```

Ce modèle Sequential commence par une couche d'entrée explicite (c'est plus simple que d'essayer de fixer `input_shape` uniquement sur la première couche), puis ajoute une couche de convolution à une dimension avec un remplissage "causal". Nous sommes ainsi certains que la couche de convolution ne regarde pas dans le futur lors de ses prédictions (cela équivaut à remplir les entrées avec la bonne quantité de zéros à gauche et à utiliser le remplissage "valid"). Nous ajoutons ensuite des paires de couches similaires en utilisant des taux de dilatation croissant: 1, 2, 4, 8, et de nouveau 1, 2, 4, 8. Nous terminons par la couche de sortie: une couche de convolution avec dix filtres de taille 1 et sans aucune fonction d'activation. Grâce aux couches de remplissage, chaque couche de convolution produit une série de la même longueur que les séries d'entrée, et les cibles utilisées pendant l'entraînement peuvent être des séries complètes; nous n'avons pas besoin de les couper ni de les sous-échantillonner.

Les deux derniers modèles permettent d'obtenir les meilleures prévisions pour nos séries chronologiques ! Dans l'article sur WaveNet, les auteurs sont arrivés à des performances de pointe sur différentes tâches audio (d'où le nom de l'architecture), y compris dans le domaine de la dictée vocale, en produisant des voies incroyablement réalistes dans différentes langues. Ils ont également utilisé ce modèle pour générer de la musique, un échantillon audio à la fois. Cet exploit est d'autant plus impressionnant si vous réalisez qu'une seule seconde de son contient des dizaines de milliers

¹⁹¹. Le WaveNet complet inclut quelques techniques supplémentaires, comme les connexions de saut que l'on trouve dans un ResNet, et les *unités d'activation à porte* semblables à celles qui existent dans une cellule GRU. Vous trouverez plus de détails dans le notebook (voir « `15_processing_sequences_using_rnns_and_cnns.ipynb` » sur <https://github.com/ageron/handson-ml2>).

d'étapes temporelles – même les LSTM et les GRU ne sont pas capables de traiter de si longues séries.

Au chapitre 8, nous allons continuer à explorer les RNR et nous verrons comment ils permettent d'aborder différentes tâches de TALN.

7.5 EXERCICES

1. Donnez quelques applications d'un RNR série-vers-série. Que pouvez-vous proposer pour un RNR série-vers-vecteur ? Et pour un RNR vecteur-vers-série ?
2. Combien de dimensions doivent avoir les entrées d'une couche de RNR ? Que représente chaque dimension ? Qu'en est-il des sorties ?
3. Si vous souhaitez construire un RNR série-vers-série profond, pour quelles couches du RNR devez-vous préciser `return_sequences=True` ? Et dans le cas d'un RNR série-vers-vecteur ?
4. Supposons que vous disposez quotidiennement d'une série chronologique univariée et que vous souhaitez prévoir les sept jours suivants. Quelle architecture de RNR devez-vous employer ?
5. Quelles sont les principales difficultés de l'entraînement des RNR ? Comment pouvez-vous les résoudre ?
6. Esquissez l'architecture d'une cellule LSTM.
7. Pourquoi voudriez-vous utiliser des couches de convolution à une dimension dans un RNR ?
8. Quelle architecture de réseau de neurones est adaptée à la classification de vidéos ?
9. Entraînez un modèle de classification sur le jeu de données SketchRNN (disponible dans TensorFlow Datasets).
10. Téléchargez le jeu de données Bach chorales (<https://homl.info/bach>) et extrayez son contenu. Il est constitué de 382 chants choraux composés par Jean-Sébastien Bach. Chaque chant comporte entre 100 et 640 étapes temporelles, chacune contenant quatre entiers, correspondant chacun à l'indice d'une note sur un piano (à l'exception de la valeur 0 qui indique qu'aucune note n'est jouée). Entraînez un modèle, récurrent, convolutif ou les deux, capable de prédire l'étape temporelle suivante (quatre notes), à partir d'une série d'étapes temporelles provenant d'un chant. Utilisez ensuite ce modèle pour générer une composition dans le style de Bach, une note à la fois. Pour cela, vous pouvez fournir au modèle le début d'un chant et lui demander de prédire l'étape temporelle suivante, puis ajouter cette étape à la série d'entrée et demander au modèle de donner la note suivante, et ainsi de suite. Pensez à regarder le modèle Coconet de Google (<https://homl.info/coconet>) qui a servi à produire un joli doodle Google en hommage à Bach.

Les solutions de ces exercices sont données à l'annexe A.

8

Traitements automatiques du langage naturel avec les RNR et les attentions

Lorsque Alan Turing a imaginé son fameux test de Turing¹⁹² en 1950, son objectif était d'évaluer la capacité d'une machine à égaler l'intelligence humaine. Il aurait pu tester diverses capacités, comme reconnaître des chats dans des images, jouer aux échecs, composer de la musique ou sortir d'un labyrinthe, mais il a choisi une tâche linguistique. Plus précisément, il a conçu un *chatbot* capable de tromper son interlocuteur en lui faisant croire qu'il discutait avec un être humain¹⁹³.

Ce test présente des faiblesses : un jeu de règles codées peut tromper des personnes naïves ou crédules (par exemple, la machine peut donner des réponses prédéfinies en réponse à certains mots-clés, prétendre plaisanter ou être ivre pour expliquer des réponses bizarres, ou écarter les questions difficiles en répondant par d'autres questions) et de nombreux aspects de l'intelligence humaine sont complètement ignorés (par exemple, la capacité à interpréter une communication non verbale, comme des expressions faciales, ou à apprendre une tâche manuelle). Quoi qu'il en soit, le test met en évidence le fait que la maîtrise du langage est sans doute la plus grande capacité cognitive de l'*Homo sapiens*. Sommes-nous en mesure de construire une machine capable de lire et d'écrire en langage naturel ?

Pour les applications dans le domaine du langage naturel, l'approche la plus répandue se fonde sur les réseaux de neurones récurrents. Nous allons donc poursuivre

192. Alan Turing, « Computing Machinery and Intelligence », *Mind*, 49 (1950), 433-460 : <https://homl.info/turingtest>.

193. Bien entendu, le terme *chatbot* est arrivé beaucoup plus tard. Turing a appelé son test le *jeu d'imitation* : la machine A et l'homme B discutent avec un interrogateur humain C au travers de messages textuels ; l'interrogateur pose des questions afin de déterminer qui est la machine (A ou B). La machine réussit le test si elle parvient à tromper l'interrogateur, tandis que l'homme B doit essayer de l'aider.

notre exploration des RNR (introduits au chapitre 7), en commençant par un RNR à caractères entraîné à prédire le caractère suivant dans une phrase. Cela nous permettra de générer du texte original et nous en profiterons pour expliquer comment construire un dataset TensorFlow sur une très longue séquence. Nous emploierons tout d'abord un *RNR sans état* (qui apprend à partir de portions aléatoires d'un texte à chaque itération, sans aucune information sur le reste du texte), puis nous construirons un *RNR avec état* (qui conserve l'état caché entre des itérations d'entraînement et poursuit la lecture là où il s'était arrêté, en étant capable d'apprendre des motifs plus longs). Ensuite, nous élaborerons un RNR ayant la capacité d'analyser des opinions (par exemple, lire la critique d'un film et en extraire l'avis de l'évaluateur), en traitant cette fois-ci les phrases comme des séries non pas de caractères mais de mots. Puis nous montrerons comment utiliser les RNR pour mettre en place une architecture d'encodeur-décodeur capable de réaliser une *traduction automatique neuronale* (NMT, *Neural Machine Translation*). Pour cela, nous profiterons de l'API seq2seq fournie par le projet TensorFlow Addons.

Dans la deuxième partie de ce chapitre, nous étudierons les *mécanismes d'attention*. Ces composants d'un réseau de neurones apprennent à sélectionner les parties des entrées sur lesquelles le reste du modèle doit se focaliser à chaque étape temporelle. Nous verrons tout d'abord comment les attentions permettent d'améliorer les performances d'une architecture encodeur-décodeur à base de RNR, puis nous retirerons totalement les RNR pour présenter une architecture fondée exclusivement sur les attentions et appelée *Transformer*. Pour finir, nous décrirons certaines des avancées les plus importantes survenues en 2018 et 2019 dans le domaine du TALN, y compris des modèles de langage incroyablement puissants comme GPT-2 et BERT, tous deux basés sur Transformer.

Commençons par un modèle simple et amusant capable d'écrire à la manière de Shakespeare (ou presque).

8.1 GÉNÉRER UN TEXTE SHAKESPEARIEN À L'AIDE D'UN RNR À CARACTÈRES

Dans un fameux billet de blog publié en 2015 (<https://hml.info/charnn>) et intitulé « The Unreasonable Effectiveness of Recurrent Neural Networks », Andrej Karpathy a montré comment entraîner un RNR pour prédire le caractère suivant dans une phrase. Ce Char-RNN peut ensuite servir à générer le texte d'un roman, un caractère à la fois. Voici un petit exemple d'un texte produit par un modèle de Char-RNN après qu'il a été entraîné sur l'ensemble de l'œuvre de Shakespeare :

PANDARUS:

*Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.*

Ce n'est pas véritablement un chef-d'œuvre, mais il n'en est pas moins étonnant de constater la capacité du modèle à apprendre des mots, une grammaire, une ponctuation correcte et d'autres aspects linguistiques, simplement en apprenant à prédire le caractère suivant dans une phrase. Voyons comment construire un Char-RNN pas à pas, en commençant par la création du jeu de données.

8.1.1 Créer le jeu de données d'entraînement

Tout d'abord, nous récupérons l'intégralité de l'œuvre de Shakespeare, en utilisant la fonction pratique de Keras `get_file()` et en téléchargeant les données à partir du projet Char-RNN d'Andrej Karpathy (<https://github.com/karpathy/char-rnn>) :

```
shakespeare_url = "https://homl.info/shakespeare" # raccourci d'URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Nous devons ensuite encoder chaque caractère sous forme d'un entier. Une solution consiste à créer une couche de prétraitement personnalisée, comme nous l'avons fait au chapitre 5. Mais, dans le cas présent, il est plus simple d'utiliser la classe `Tokenizer` de Keras. Nous devons tout d'abord adapter un `tokenizer` au texte : il recherchera tous les caractères employés dans le texte et associera chacun d'eux à des identifiants de caractères allant de 1 au nombre de caractères différents reconnus (puisque'il ne commence pas à zéro, nous pourrons nous servir de cette valeur comme masque) :

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

Nous fixons `char_level` à `True` pour réaliser un encodage non pas au niveau du mot (comportement par défaut), mais au niveau du caractère. Notez que ce `tokenizer` convertit par défaut le texte en minuscules (pour éviter cette opération, précisez `lower=False`). À présent, le convertisseur peut encoder une phrase (ou une liste de phrases) en une liste d'identifiants de caractères, et inversement. Il nous indique également le nombre de caractères différents identifiés, ainsi que le nombre total de caractères dans le texte :

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # nombre de caractères différents
>>> dataset_size = tokenizer.document_count # nombre total de caractères
```

Encodons l'intégralité du texte afin que chaque caractère soit représenté par son identifiant (nous soustrayons 1 pour obtenir des identifiants dans la plage 0 à 38, plutôt que 1 à 39) :

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Avant d'aller plus loin, nous devons séparer le dataset en un jeu d'entraînement, un jeu de validation et un jeu de test. Puisqu'il n'est pas possible de simplement mélanger tous les caractères du texte, comment pouvons-nous fractionner un dataset séquentiel ?

8.1.2 Fractionner un dataset séquentiel

Il est très important d'éviter tout chevauchement entre le jeu d'entraînement, le jeu de validation et le jeu de test. Par exemple, nous pouvons réservier les premiers 90 % du texte au jeu d'entraînement, les 5 % suivants au jeu de validation et les derniers 5 % au jeu de test. Il est également préférable de laisser un vide entre ces jeux afin d'éviter qu'un paragraphe ne se trouve à cheval sur deux d'entre eux.

Lorsqu'on manipule des séries chronologiques, le découpage se fait en général selon le temps. Par exemple, les années 2000 à 2012 sont attribuées au jeu d'entraînement, les années 2013 à 2015 au jeu de validation, et les années 2016 à 2018 au jeu de test. Mais il est parfois possible de découper selon d'autres dimensions afin d'obtenir une période de temps plus longue pour l'entraînement. Par exemple, si vous disposez de données sur la santé financière de 10 000 entreprises, de 2000 à 2018, vous pourriez découper ces données en fonction des entreprises. Cependant, il est probable que la corrélation entre plusieurs de ces entreprises soit importante (par exemple, tout un secteur économique peut monter ou ralentir conjointement). Si des entreprises sont corrélées dans le jeu d'entraînement et le jeu de test, l'utilité de ce dernier sera moindre car sa mesure de l'erreur de généralisation sera biaisée de façon optimiste.

Par conséquent, il est souvent plus sûr de réaliser le découpage selon le temps – mais cela suppose implicitement que les motifs appris par le RNR dans le passé (c'est-à-dire dans le jeu d'entraînement) existeront toujours dans le futur. Autrement dit, nous supposons que la série chronologique est *stationnaire* (au sens large)¹⁹⁴. Cette hypothèse est raisonnable pour de nombreuses séries chronologiques (par exemple dans le cas des réactions chimiques, car les lois de la chimie ne changent pas tous les jours), mais beaucoup moins pour d'autres (par exemple, les marchés financiers sont connus pour être inconstants, car des schémas disparaissent aussitôt que des traders les repèrent et commencent à les exploiter). Pour être certain que la série chronologique est suffisamment stationnaire, vous pouvez afficher les erreurs du modèle sur le jeu de validation en fonction du temps. Si le modèle se comporte mieux sur la première partie du jeu de validation que sur la dernière, alors la stationnarité de la série chronologique n'est probablement pas suffisante et vous devrez sans doute entraîner le modèle sur une période de temps plus courte.

En résumé, le découpage d'une série chronologique en un jeu d'entraînement, un jeu de validation et un jeu de test n'est pas une opération triviale et la manière de procéder dépend fortement de la tâche à accomplir.

Revenons à Shakespeare ! Nous prenons les premiers 90 % du texte pour le jeu d'entraînement (en gardant le reste pour le jeu de validation et le jeu de test) et créons un `tf.data.Dataset` qui retournera un par un chaque caractère de ce jeu :

194. Par définition, la moyenne, la variance et les *autocorrelations* (c'est-à-dire les corrélations entre les valeurs de cette série chronologique séparées par un intervalle donné) d'une série chronologique stationnaire ne changent pas au fil du temps. C'est plutôt restrictif, en excluant, par exemple, les séries temporelles comportant des tendances ou des motifs cycliques. Les RNR sont plus tolérants à cette contrainte, car ils peuvent apprendre des tendances et des motifs cycliques.

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

8.1.3 Découper le dataset séquentiel en plusieurs fenêtres

Pour le moment, le jeu d'entraînement est constitué d'une seule suite de plusieurs millions de caractères. Il ne peut donc pas être utilisé directement pour entraîner le réseau de neurones. En effet, le RNR serait équivalent à un réseau profond de plus d'un million de couches et n'aurait qu'une seule (très longue) instance pour s'entraîner. À la place, nous allons utiliser la méthode `window()` du dataset pour convertir cette longue séquence de caractères en portions de texte plus petites. Chaque instance dans le dataset correspondra à une sous-chaîne relativement courte du texte complet, et le RNR sera déroulé uniquement sur la longueur de ces sous-chaînes. Cette technique se nomme *rétropropagation tronquée dans le temps* (*truncated backpropagation through time*). Invoquons la méthode `window()` de façon à créer un dataset de petites fenêtres de texte :

```
n_steps = 100
window_length = n_steps + 1 # cible = entrée décalée d'un caractère en tête
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



Vous pouvez essayer d'ajuster `n_steps` : il est plus facile d'entraîner un RNR sur des séquences d'entrée plus courtes, mais, évidemment, le RNR ne pourra pas apprendre des motifs plus longs que `n_steps`. Il ne faut donc pas le choisir trop petit.

Par défaut, la méthode `window()` crée des fenêtres sans intersection. Mais, puisque nous voulons un jeu d'entraînement le plus grand possible, nous indiquons `shift=1` afin que la première fenêtre contienne les caractères 0 à 100, la deuxième, les caractères 1 à 101, et ainsi de suite. Pour être certains que toutes les fenêtres comprennent exactement 101 caractères (ce qui nous permettra de créer des lots sans avoir à utiliser le remplissage), nous fixons `drop_remainder` à `True` (dans le cas contraire, les cent dernières fenêtres contiendront 100 caractères, 99 caractères, etc., jusqu'à 1 caractère).

La méthode `window()` crée un dataset contenant des fenêtres, chacune étant elle-même représentée sous forme d'un dataset. Il s'agit d'un *dataset imbriqué*, comparable à une liste de listes. Sa structure se révélera utile si vous voulez transformer chaque fenêtre en appelant sa méthode de dataset (par exemple, pour les mélanger ou les mettre en lots). Cependant, le dataset imbriqué ne peut pas être employé directement pour l'entraînement, car notre modèle attend en entrée non pas des datasets mais des tenseurs. Nous appelons donc la méthode `flat_map()`, qui convertit un dataset imbriqué en un *dataset plat* (qui ne contient aucun autre dataset).

Par exemple, supposons que `{1, 2, 3}` représente un dataset qui contient la suite de tenseurs `1, 2` et `3`. Si vous aplatissez le dataset imbriqué `[[1, 2], [3, 4, 5, 6]]`, vous obtenez en retour le dataset plat `[1, 2, 3, 4, 5, 6]`. Par ailleurs, la méthode `flat_map()` prend en argument une fonction qui vous permet de transformer chaque

dataset du dataset imbriqué avant l’aplatissement. Par exemple, si vous passez la fonction `lambda ds: ds.batch(2)` à `flat_map()`, elle transformera le dataset imbriqué `{[1, 2], [3, 4, 5, 6]}` en un dataset plat `[[1, 2], [3, 4], [5, 6]]`, c’est-à-dire un dataset de tenseurs de taille 2. Nous pouvons maintenant aplatis notre dataset :

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Vous noterez que nous appelons `batch(window_length)` sur chaque fenêtre. Puisque toutes les fenêtres ont exactement la même longueur, nous obtenons un seul tenseur pour chacune d’elles. Le dataset contient à présent des fenêtres consécutives de 101 caractères chacune. Puisque la descente de gradient est plus efficace lorsque les instances du jeu d’entraînement sont indépendantes et distribuées de façon identique¹⁹⁵, nous devons mélanger ces fenêtres. Ensuite, nous pouvons les mettre en lots et séparer les entrées (les 100 premiers caractères) de la cible (le dernier caractère) :

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

La figure 8.1 résume les étapes de préparation du dataset que nous venons de décrire (avec des fenêtres de taille 11 plutôt que 101, et des lots de taille 3 plutôt que 32).



Figure 8.1 – Préparation d’un dataset de fenêtres mélangées

Nous en avons discuté au chapitre 5, les caractéristiques de catégorie en entrée doivent généralement être encodées, le plus souvent sous forme de vecteurs *one-hot* ou de plongements. Dans cet exemple, nous encoderons chaque caractère à l’aide d’un vecteur *one-hot* car le nombre de caractères différents est assez faible (seulement 39) :

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Pour terminer, nous ajoutons simplement la prélecture :

```
dataset = dataset.prefetch(1)
```

195. Voir le chapitre 1.

Et voilà ! La préparation du dataset était la partie la plus complexe. Créons à présent le modèle.

8.1.4 Construire et entraîner le modèle Char-RNN

Pour prédire le caractère suivant en fonction des 100 caractères précédents, nous pouvons utiliser un RNR constitué de deux couches GRU de 128 unités chacune et un dropout de 20 % sur les entrées (`dropout`) et sur les états cachés (`recurrent_dropout`). Si nécessaire, nous pourrons ajuster ces hyperparamètres ultérieurement. La couche de sortie est une couche Dense distribuée temporellement, comme nous l'avons vu au chapitre 7. Cette fois-ci, elle doit avoir 39 unités (`max_id`), car le texte est constitué de 39 caractères différents et nous voulons sortir une probabilité pour chaque caractère possible (à chaque étape temporelle). Puisque la somme des probabilités de sortie à chaque étape temporelle doit être égale à 1, nous appliquons la fonction softmax aux sorties de la couche Dense. Nous pouvons ensuite compiler ce modèle, en utilisant la perte "`sparse_categorical_crossentropy`" et un optimiseur Adam. Enfin, nous sommes prêts à l'entraîner sur plusieurs époques (en fonction de votre matériel, l'entraînement peut prendre jusqu'à plusieurs heures) :

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                    dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                    dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

8.1.5 Utiliser le modèle Char-RNN

Nous disposons désormais d'un modèle capable de prédire le caractère suivant dans un texte rédigé par Shakespeare. Avant de pouvoir lui fournir un texte, nous devons prétraiter celui-ci comme nous l'avons fait précédemment. Pour cela, créons une petite fonction :

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Utilisons le modèle pour prédire la lettre suivante dans une phrase :

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1re phrase, dernier
                                                # caractère
'u'
```

Parfait ! Le modèle a deviné juste. Voyons comment l'utiliser pour générer un nouveau texte.

8.1.6 Créer un faux texte shakespearien

Pour générer un nouveau texte en utilisant le modèle Char-RNN, nous pouvons lui passer un texte, lui faire prédire la lettre suivante la plus probable, ajouter celle-ci à la fin du texte, lui donner le texte étendu pour qu'il devine la lettre suivante, et ainsi de suite. En pratique, cela produit généralement les mêmes mots répétés à l'infini. À la place, nous pouvons, avec la fonction `tf.random.categorical()` de TensorFlow, sélectionner aléatoirement le caractère suivant, avec une probabilité égale à la probabilité estimée. Nous obtiendrons ainsi un texte plus diversifié et intéressant. La fonction `categorical()` échantillonne des indices de classe aléatoires, en donnant les probabilités logarithmiques des classes (*logits*).

Pour disposer d'un plus grand contrôle sur la diversité du texte généré, nous pouvons diviser les logits par une valeur appelée *température*, ajustable selon nos besoins. Une température proche de zéro favorisera les caractères ayant une probabilité élevée, tandis qu'une température très élevée donnera une probabilité égale à tous les caractères. La fonction `next_char()` se fonde sur cette approche pour sélectionner le caractère suivant qui sera concaténé au texte d'entrée :

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Nous pouvons ensuite écrire une petite fonction qui appellera `next_char()` de façon répétée afin d'obtenir le caractère suivant et l'ajouter au texte donné :

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

Nous sommes prêts à générer du texte ! Essayons différentes températures :

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

Apparemment, notre modèle de Shakespeare travaille mieux à des températures proches de 1. Pour générer un texte plus convaincant, vous pouvez essayer d'ajouter d'autres couches GRU et d'augmenter le nombre de neurones par couche, de prolonger l'entraînement, et d'ajouter une régularisation (par exemple, vous pouvez fixer `recurrent_dropout` à 0.3 dans les couches GRU). Le modèle est actuellement incapable d'apprendre des motifs plus longs que `n_steps`, c'est-à-dire 100 caractères. Vous pouvez essayer d'agrandir cette fenêtre, mais l'entraînement s'en trouvera

plus compliqué et même les cellules LSTM et GRU ne sont pas capables de traiter des séries très longues. Une autre solution consiste à utiliser un RNR avec état.

8.1.7 RNR avec état

Jusqu'à présent, nous n'avons utilisé que des RNR *sans état*. À chaque itération d'entraînement, le modèle démarre avec un état caché rempli de zéros, il actualise ensuite cet état lors de chaque étape temporelle, et, après la dernière, il le détruit, car il n'est plus utile. Et si nous demandions au RNR de conserver cet état final après le traitement d'un lot d'entraînement et de l'utiliser comme état initial du lot d'entraînement suivant ? En procédant ainsi, le modèle peut apprendre des motifs à long terme même si la propagation se fait sur des séries courtes. Dans ce cas, il s'agit d'un RNR *avec état*. Voyons comment en construire un.

Tout d'abord, notez qu'un RNR avec état n'a de sens que si chaque série d'entrée dans un lot débute exactement là où la série correspondante dans le lot précédent s'était arrêtée. Pour construire un RNR avec état, la première chose à faire est donc d'utiliser des séries d'entrée séquentielles sans intersection (à la place des séries mélangées et se chevauchant que nous avons utilisées pour entraîner des RNR sans état). Lors de la création du Dataset, nous devons indiquer `shift=n_steps` (à la place de `shift=1`) dans l'appel à la méthode `window()`. Et, bien sûr, nous retirons l'invocation de la méthode `shuffle()`. Malheureusement, la mise en lots est plus compliquée lors de la préparation d'un RNR avec état que d'un RNR sans état. Si nous appelons `batch(32)`, trente-deux fenêtres consécutives seraient alors placées dans le même lot, et le lot suivant ne poursuivrait pas chacune de ces fenêtres là où elles se sont arrêtées. Le premier lot contiendrait les fenêtres 1 à 32, et le deuxième lot, les fenêtres 33 à 64. Par conséquent, si nous prenons, par exemple, la première fenêtre de chaque lot (c'est-à-dire les fenêtres 1 et 33), il est évident qu'elles ne sont pas consécutives. La solution la plus simple à ce problème consiste à utiliser des « lots » d'une seule fenêtre :

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

La figure 8.2 résume les premières étapes.

La mise en lots est plus compliquée, mais pas impossible. Par exemple, nous pouvons découper le texte de Shakespeare en trente-deux textes de longueur égale, créer un dataset de séries d'entrée consécutives pour chacun d'eux, et finalement utiliser `tf.data.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` pour créer des lots consécutifs appropriés. La $n^{\text{ème}}$ séquence d'entrée dans un lot commence exactement là où la $n^{\text{ème}}$ série d'entrée s'est arrêtée dans le lot précédent (le code complet est disponible dans le notebook¹⁹⁶).

196. Voir « 16_nlp_with_rnns_and_attention.ipynb » sur <https://github.com/ageron/handson-ml2>.

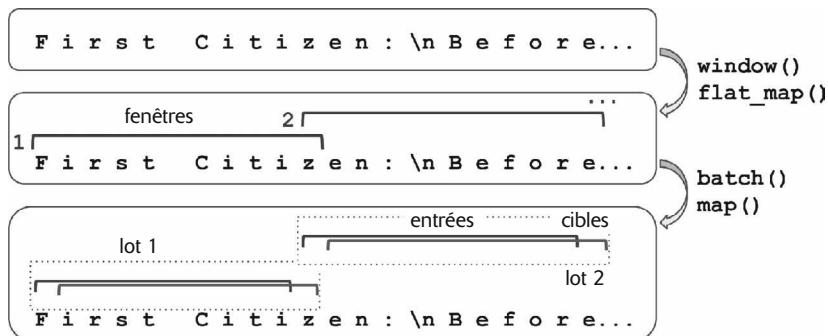


Figure 8.2 – Préparation d'un dataset de fragments de séries consécutifs pour un RNR avec état

Créons à présent le RNR avec état. Tout d'abord, nous devons indiquer `stateful=True` lors de la création de chaque couche récurrente. Ensuite, le RNR avec état doit connaître la taille d'un lot (puisque'il conservera un état pour chaque série d'entrée dans le lot) et nous devons donc préciser l'argument `batch_input_shape` dans la première couche. Notez qu'il est inutile d'indiquer la seconde dimension puisque la longueur des entrées est quelconque :

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                    dropout=0.2, recurrent_dropout=0.2,
                    batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                    dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

Au terme de chaque époque, nous devons réinitialiser les états avant de revenir au début du texte. Pour cela, nous utilisons un petit rappel :

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

Nous pouvons à présent compiler et ajuster le modèle (pour un plus grand nombre d'époques, car chaque époque est plus courte que précédemment et chaque lot ne comprend qu'une seule instance) :

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```



Après l'entraînement de ce modèle, il ne pourra être employé que pour effectuer des prédictions sur des lots dont la taille est identique à celle des lots utilisés pendant l'entraînement. Pour éviter cette contrainte, créez un modèle *sans état* identique et copiez le poids du modèle avec état dans ce nouveau modèle.

À présent que nous avons construit un modèle de niveau caractère, il est temps d'examiner des modèles de niveau mot et de nous attaquer à une tâche de traitement du langage naturel plus répandue, l'*analyse de sentiments* (ou *analyse d'opinions*). Nous en profiterons pour apprendre à traiter les séries de longueur variable en utilisant les masques.

8.2 ANALYSE DE SENTIMENTS

Si MNIST est le « hello world » de la vision par ordinateur, alors le jeu de données IMDb reviews est celui du traitement automatique du langage naturel. Il contient 50 000 critiques de films en anglais (25 000 pour l'entraînement, 25 000 pour les tests) extraites de la réputée base de données cinématographiques d'Internet (Internet Movie Database, <https://imdb.com/>), avec une cible binaire simple pour chaque critique indiquant si elle est négative (0) ou positive (1). À l'instar de MNIST, le jeu de données IMDb reviews est devenu populaire pour d'excellentes raisons : il est suffisamment simple pour être exploité sur un ordinateur portable en un temps raisonnable, mais suffisamment difficile pour être amusant et gratifiant. Keras dispose d'une fonction pour le charger :

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Où se trouvent donc les critiques cinématographiques ? Comme vous pouvez le constater, le dataset a déjà été prétraité. `X_train` contient une liste de critiques, chacune représentée sous forme d'un tableau NumPy d'entiers, chaque entier représentant un mot. Toute la ponctuation a été retirée et les mots ont été convertis en minuscules, séparés par des espaces, et indexés en fonction de leur fréquence (les valeurs faibles correspondent aux mots fréquents). Les entiers 0, 1 et 2 ont une signification particulière. Ils représentent le caractère de remplissage, le caractère *début de séquence* (*start-of-sequence*) et les mots inconnus. Pour visualiser une critique, vous pouvez la décoder de la manière suivante :

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
...     id_to_word[id_] = token
...
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

Dans un projet réel, ce serait à vous de prétraiter le texte. Pour cela, vous pouvez employer la classe `Tokenizer` utilisée précédemment, mais en indiquant cette fois-ci `char_level=False` (la configuration par défaut). Lors de l'encodage des mots, cette classe retire un grand nombre de caractères, notamment la majorité des signes de ponctuation, les sauts de ligne et les tabulations (son comportement peut être modifié à l'aide de l'argument `filters`). Le plus important est qu'elle utilise les espaces pour identifier les frontières de mots. Si cela convient pour les textes en anglais et autres langues qui séparent les mots par des espaces, ce n'est pas le cas de tous les écrits. Le chinois ne met pas d'espaces entre les mots, le vietnamien emploie

des espaces même à l'intérieur des mots et les langues comme l'allemand attachent souvent plusieurs mots ensemble, sans espace. Même en anglais, les espaces ne sont pas toujours la meilleure façon de distinguer les mots, par exemple « San Francisco » ou « #ILoveDeepLearning ».

Heureusement, il existe de meilleures solutions. L'article¹⁹⁷ publié en 2018 par Taku Kudo propose une technique d'apprentissage non supervisé permettant de transformer un texte en tokens, et inversement. L'analyse se fait au niveau des sous-mots, de manière indépendante de la langue, en traitant l'espace comme n'importe quel autre caractère. Grâce à cette approche, le modèle peut raisonnablement deviner ce que signifie un mot inconnu. Par exemple, s'il n'a pas rencontré le terme « smartest » pendant l'entraînement, il a pu apprendre le mot « smart » et apprendre également que le suffixe « est » signifie « le meilleur ». Il peut donc déduire le sens de « smartest ». Le projet SentencePiece de Google (<https://github.com/google/sentencepiece>) fournit une implémentation open source de cette technique. Elle est décrite dans un article¹⁹⁸ rédigé par Taku Kudo et John Richardson.

Une autre solution avait été proposée dans un article¹⁹⁹ antérieur par Rico Sennrich *et al.* Elle explorait d'autres façons de créer des encodages de sous-mots (par exemple, en utilisant un *encodage par paire d'octets*). Enfin et surtout, l'équipe TensorFlow a proposé en juin 2019 la bibliothèque TF.Text (<https://homl.info/tftext>), qui implémente différentes stratégies de conversion en tokens, notamment WordPiece²⁰⁰ (une variante de l'encodage par paire d'octets).

Si vous souhaitez déployer votre modèle sur un appareil mobile ou un navigateur web, sans avoir à écrire à chaque fois une fonction de prétraitement différente, vous pouvez effectuer ce prétraitement à la volée en utilisant uniquement des opérations TensorFlow afin qu'il soit inclus dans le modèle lui-même. Voyons comment procéder. Tout d'abord, chargeons le jeu de données original IMDb reviews, sous forme d'un texte (chaînes de caractères d'octets) en utilisant TensorFlow Datasets (voir le chapitre 5) :

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Écrivons ensuite la fonction de prétraitement :

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\\\s*/?>", b" ")
```

197. Taku Kudo, « Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates » (2018) : <https://homl.info/subword>.

198. Taku Kudo et John Richardson, « SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing » (2018) : <https://homl.info/sentencepiece>.

199. Rico Sennrich *et al.*, « Neural Machine Translation of Rare Words with Subword Units », *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics 1* (2016), 1715-1725 : <https://homl.info/rarewords>.

200. Yonghui Wu *et al.*, « Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation » (2016) : <https://homl.info/wordpiece>.

```
X_batch = tf.strings.regex_replace(X_batch, b"^[^a-zA-Z]", b" ")
X_batch = tf.strings.split(X_batch)
return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

Elle commence par tronquer les critiques, en ne conservant que leurs 300 premiers caractères. Cela permettra d'accélérer l'entraînement et aura peu d'impact sur les performances, car vous pouvez généralement savoir si une critique est positive ou non en lisant uniquement les une ou deux premières phrases. Ensuite, elle se sert d'*expressions régulières* pour remplacer les balises
 par des espaces, ainsi que tous les caractères autres que des lettres et des apostrophes. Par exemple, la phrase "Well, I can't
" va devenir "Well I can't". Enfin, `preprocess()` divise les critiques en fonction des espaces. Le tenseur irrégulier obtenu est converti en tenseur dense, avec un remplissage des critiques à l'aide du caractère "<pad>" de sorte qu'elles soient toutes de la même longueur.

Ensuite, nous devons construire le vocabulaire. Pour cela, nous parcourons une fois l'intégralité du jeu d'entraînement, en appliquant notre fonction `preprocess()` et en utilisant un `Counter` pour dénombrer les occurrences de chaque mot :

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Affichons les trois mots les plus fréquents :

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Parfait ! Puisque notre modèle n'a probablement pas besoin de connaître tous les mots du dictionnaire pour obtenir de bonnes performances, nous allons tronquer le vocabulaire en ne conservant que les 10 000 mots les plus utilisés :

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Nous devons à présent ajouter une étape de prétraitement pour remplacer chaque mot par son identifiant (c'est-à-dire son indice dans le vocabulaire). De la même manière qu'au chapitre 5, nous créons une table de correspondance, avec 1 000 emplacements hors vocabulaire :

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

Nous pouvons ensuite utiliser cette table pour rechercher les identifiants de quelques mots :

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,   12,   11, 10054]])>
```

Notez que les mots «this», «movie» et «was» ont été trouvés dans la table et que leur identifiant est donc inférieur à 10 000, tandis que le mot «faaaaaantastic»

en était absent et qu'il a donc été associé à l'un des emplacements hors vocabulaire, avec un identifiant supérieur ou égal à 10 000.



TF Transform (voir le chapitre 5) propose des fonctions utiles à la prise en charge de tels vocabulaires. Par exemple, la fonction `tft.compute_and_apply_vocabulary()` parcourt le jeu de données afin de trouver tous les mots différents et de construire le vocabulaire, puis génère les opérations TF nécessaires à l'encodage de chaque mot à l'aide de ce vocabulaire.

Nous sommes maintenant prêts à créer le jeu d'entraînement final. Nous mettons les critiques sous forme de lots, les convertissons en courtes suites de mots à l'aide de la fonction `preprocess()`, encodons ces mots à l'aide d'une fonction `encode_words()` simple qui utilise la table que nous venons de construire, et effectuons la prélecture du lot suivant :

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

Enfin, nous créons et entraînons le modèle :

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

La première couche est une couche `Embedding` qui convertira les identifiants de mots en plongements (voir le chapitre 5). La matrice des plongements doit avoir une ligne par identifiant de mots (`vocab_size + num_oov_buckets`) et une colonne par dimension de plongement (dans cet exemple, nous utilisons 128 dimensions, mais cet hyperparamètre peut être ajusté). Alors que les entrées du modèle seront des tenseurs à deux dimensions de forme [taille de lot, étapes temporelles], la sortie de la couche `Embedding` sera un tenseur à trois dimensions de forme [taille de lot, étapes temporelles, taille de plongement].

Le reste du modèle ne pose pas de véritables difficultés. Il est constitué de deux couches `GRU`, la seconde retournant uniquement la sortie de la dernière étape temporelle. La couche de sortie comprend un seul neurone qui utilise la fonction d'activation sigmoïde pour produire la probabilité estimée que la critique exprime un avis positif sur le film. Nous compilons ensuite le modèle assez simplement et l'appliquons au jeu de données préparé précédemment, sur quelques époques.

8.2.1 Masquage

En l'état, le modèle devra apprendre que les caractères de remplissage doivent être ignorés. Mais cela, nous le savons déjà ! Alors, pourquoi ne pas indiquer au modèle d'ignorer ces tokens afin qu'il puisse se focaliser sur les données qui sont réellement importantes ? C'est possible et la solution est triviale : il suffit d'ajouter `mask_zero=True` au moment de la création de la couche `Embedding`. Cela signifie que les caractères de remplissage (ceux dont l'identifiant vaut 0)²⁰¹ seront ignorés par toutes les couches en aval. Voilà tout !

La couche `Embedding` crée un *tenseur de masque* égal à `K.not_equal(inputs, 0)` (où `K = keras.backend`). Il s'agit d'un tenseur booléen ayant la même forme que les entrées et valant `False` partout où les identifiants de mots sont égaux à 0, sinon `True`. Ce tenseur de masque est ensuite propagé automatiquement par le modèle à toutes les couches suivantes, tant que la dimension temporelle est conservée. Dans cet exemple, les deux couches `GRU` vont recevoir automatiquement ce masque, mais, puisque la seconde ne retourne pas des suites (elle retourne uniquement la sortie de la dernière étape temporelle), le masque ne sera pas transmis à la couche `Dense`. Chaque couche fait un usage différent du masque, mais, en général, elles ignorent simplement les étapes temporelles masquées (autrement dit, celles pour lesquelles le masque vaut `False`). Par exemple, lorsqu'une couche récurrente rencontre une étape temporelle masquée, elle copie simplement la sortie de l'étape temporelle précédente. Si le masque se propage tout le long jusqu'à la sortie (dans les modèles qui produisent des séries, ce qui n'est pas le cas dans cet exemple), il sera également appliqué aux pertes. Les étapes temporelles masquées ne vont donc pas contribuer à la perte (leur perte sera égale à 0).



Les couches `LSTM` et `GRU` ont une implémentation optimisée pour les processeurs graphiques, fondée sur la bibliothèque cuDNN de Nvidia. Mais cette implémentation ne prend pas en charge le masquage. Si votre modèle utilise un masque, elles vont revenir à l'implémentation par défaut (beaucoup plus lente). Notez que l'implémentation optimisée vous oblige également à utiliser la valeur par défaut de plusieurs hyperparamètres : `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias` et `reset_after`.

Toutes les couches qui reçoivent le masque doivent prendre en charge le masquage (sinon une exception est levée). Cela concerne toutes les couches récurrentes, ainsi que la couche `TimeDistributed`, plus quelques autres. Toute couche qui prend en charge le masquage doit avoir un attribut `supports_masking` fixé à `True`. Si vous souhaitez implémenter votre propre couche personnalisée avec prise en charge du masquage, vous devez ajouter un argument `mask` à la méthode `call()` (et, évidemment, faire en sorte que cette méthode utilise le masque d'une manière ou d'une autre). Par ailleurs, vous devez indiquer `self.supports_masking = True`

²⁰¹ Leur identifiant est égal à 0 uniquement parce qu'ils sont les « mots » les plus fréquents dans le jeu de données. Il serait probablement plus judicieux de faire en sorte que ces caractères de remplissage soient toujours encodés sous forme d'un 0, même s'ils ne sont pas les plus fréquents.

dans le constructeur. Si votre couche ne commence pas par une couche Embedding, vous avez la possibilité d'utiliser à la place la couche keras.layers.Masking. Elle fixe le masque à K.any(K.not_equal(inputs, 0), axis=-1), signifiant par là que les étapes temporelles dont la dernière dimension est remplie de zéros seront masquées dans les couches suivantes (de nouveau, tant que la dimension temporelle existe).

Les couches de masquage et la propagation automatique du masque fonctionnent bien pour les modèles Sequential simples. Ce ne sera pas toujours le cas pour des modèles plus complexes, par exemple lorsque vous devez associer des couches Conv1D et des couches récurrentes. Dans de telles configurations, vous devrez calculer explicitement le masque et le passer aux couches appropriées, en utilisant l'API Functional ou l'API Subclassing. Par exemple, le modèle suivant est identique au précédent, excepté qu'il se fonde sur l'API Functional et gère le masquage manuellement :

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

Après un entraînement sur quelques époques, le modèle parvient à juger à peu près correctement si une critique est positive ou non. À l'aide d'un rappel TensorBoard(), vous pouvez visualiser l'apprentissage des plongements dans TensorBoard. Le regroupement progressif des mots comme « awesome » et « amazing » d'un côté de l'espace des plongements et celui des mots comme « awful » et « terrible » de l'autre côté est un spectacle fascinant. Certains mots ne sont pas aussi positifs que vous pourriez le croire (tout au moins avec ce modèle). C'est par exemple le cas de l'adjectif « good », probablement parce que de nombreuses critiques négatives contiennent l'expression « not good ».

Il est assez incroyable que le modèle soit en mesure d'apprendre des plongements de mots utiles uniquement à partir de 25 000 critiques cinématographiques. Imaginez la qualité des plongements s'il disposait de critiques par milliards ! Ce n'est malheureusement pas le cas, mais nous pouvons peut-être réutiliser les plongements de mots obtenus sur d'autres ensembles de textes plus vastes (par exemple, des articles Wikipédia), même s'ils ne correspondent pas à des critiques de films. Après tout, le sens du mot « amazing » ne change pas, quel que soit le contexte dans lequel il est employé. Par ailleurs, les plongements pourraient peut-être se révéler utiles à l'analyse de sentiments même s'ils ont été entraînés pour une autre tâche. Puisque les mots « awesome » et « amazing » ont un sens comparable, ils vont probablement se regrouper de la même manière dans l'espace des plongements, quelle que soit la tâche (par exemple, prédire le mot suivant dans une phrase).

Si tous les termes positifs et tous les termes négatifs forment des groupes, ils seront utiles pour l'analyse d'opinion. Par conséquent, au lieu d'utiliser autant de paramètres

pour apprendre des plongements de mots, voyons si nous ne pouvons pas simplement réutiliser des plongements préentraînés.

8.2.2 Réutiliser des plongements préentraînés

Le projet TensorFlow Hub facilite la réutilisation de composants de modèles préentraînés dans nos propres modèles. Ces composants de modèles sont appelés *modules*. Allez simplement sur le dépôt TF Hub (<https://tfhub.dev>), recherchez celui dont vous avez besoin, puis copiez l'exemple de code dans votre projet. Le module sera téléchargé automatiquement, avec ses poids préentraînés, et inclus dans votre modèle. Facile !

Par exemple, utilisons le module de plongement de phrases `nnlm-en-dim50`, version 1, dans notre modèle d'analyse de sentiments :

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                  dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])
```

La couche `hub.KerasLayer` télécharge le module à partir de l'URL indiquée. Ce module particulier est un *encodeur de phrases* : il prend des chaînes de caractères en entrée et encode chacune d'elles sous forme d'un seul vecteur (dans ce cas, un vecteur à 50 dimensions). De façon interne, il analyse la chaîne (en séparant les mots selon les espaces) et intègre chaque mot en utilisant une matrice de plongements ayant été préentraînés sur un vaste corpus : Google News 7B (contenant sept milliards de mots !). Il calcule ensuite la moyenne de tous les plongements de mots, le résultat étant le plongement de phrase²⁰². Nous pouvons ensuite ajouter deux couches `Dense` simples pour créer un bon modèle d'analyse de sentiments. Par défaut, un `hub.KerasLayer` n'est pas entraînable, mais vous pouvez préciser `trainable=True` lors de sa création pour changer cela et l'ajuster à votre tâche.



Tous les modules de TF Hub ne prenant pas en charge TensorFlow 2, vérifiez que votre choix porte sur un module compatible.

Ensuite, il nous faut simplement charger le jeu de données IMDb reviews – son traitement est inutile (excepté pour la mise en lots et la prélecture) – et entraîner directement le modèle :

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

²⁰². Plus précisément, le plongement de phrase est égal à la moyenne des plongements de mots multipliée par la racine carrée du nombre de mots dans la phrase. Cela compense le fait que la moyenne de n vecteurs baisse lorsque n augmente.

```
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

Notez que la dernière partie de l'URL du module TF Hub indique que nous voulons la version 1. Ce système de version permet d'être certain que la sortie d'une nouvelle version d'un module ne remet pas en cause notre modèle. Si vous saisissez cette URL dans un navigateur web, vous obtenez la documentation du module. Par défaut, TF Hub place les fichiers téléchargés dans un cache dans le répertoire temporaire du système local. Si vous préférez les récupérer dans un répertoire plus permanent afin d'éviter un nouveau téléchargement après chaque nettoyage du système, indiquez ce répertoire dans la variable d'environnement `TFHUB_CACHE_DIR` (par exemple, `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

À ce stade, nous avons examiné les séries chronologiques, la génération de textes avec Char-RNN et l'analyse de sentiments avec des modèles RNR au niveau du mot. Nous avons également entraîné nos propres plongements de mots et réutilisé des plongements préentraînés. Nous allons aborder à présent une autre tâche importante du traitement automatique du langage naturel: la *traduction automatique neuronale* (NMT, *Neural Machine Translation*). Nous utiliserons tout d'abord un modèle encodeur-décodeur pur, puis nous l'améliorons avec des mécanismes d'attention, pour finir par présenter l'extraordinaire architecture Transformer.

8.3 RÉSEAU ENCODEUR-DÉCODEUR POUR LA TRADUCTION AUTOMATIQUE NEURONALE

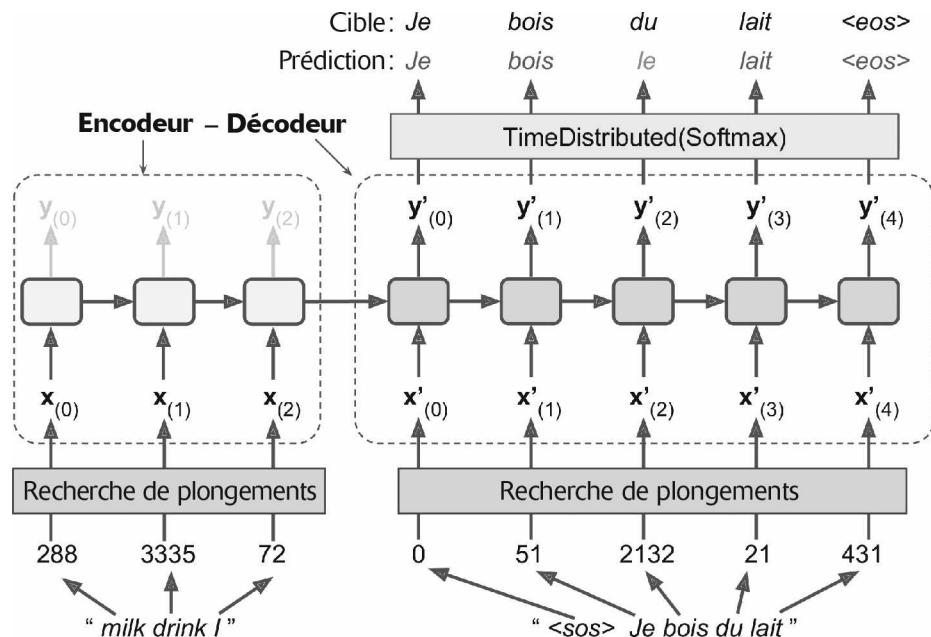
Étudions un modèle de traduction automatique neuronale simple²⁰³ qui va traduire en français des phrases en anglais (voir la figure 8.3).

Les phrases anglaises sont fournies à l'encodeur, et le décodeur produit leur traduction française. Les traductions françaises sont également utilisées comme entrées du décodeur, mais en étant décalées d'une étape en arrière. Autrement dit, le décodeur reçoit en entrée le mot qu'il aurait dû produire lors de l'étape précédente (quel que soit ce qu'il génère en réalité). Pour le premier mot, il reçoit le marqueur qui représente le début de la phrase (SOS, *start-of-sequence*). Le décodeur est censé marquer la fin de la phrase par un jeton de fin de série (EOS, *end-of-sequence*).

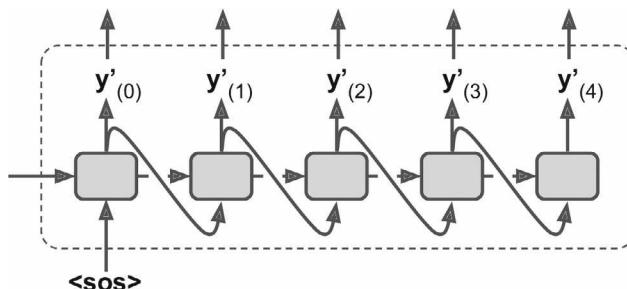
Les phrases anglaises sont inversées avant d'entrer dans l'encodeur. Par exemple, « I drink milk » est transformé en « milk drink I ». De cette manière, le début de la phrase anglaise sera fourni en dernier à l'encodeur, ce qui est intéressant car c'est en général ce que le décodeur doit traduire en premier.

Chaque mot est initialement représenté par son identifiant (par exemple, 288 pour le mot « milk »). Ensuite, une couche `embedding` permet d'obtenir le plongement du mot. Ce sont ces plongements de mots qui sont en réalité fournis à l'encodeur et au décodeur.

^{203.} Ilya Sutskever *et al.*, « Sequence to Sequence Learning with Neural Networks » (2014) : <https://homl.info/103>.

**Figure 8.3** – Modèle simple pour la traduction automatique

À chaque étape, le décodeur génère un score pour chaque mot dans le vocabulaire de sortie (c'est-à-dire, le français), puis la couche softmax convertit ces scores en probabilités. Par exemple, lors de la première étape, le mot « Je » peut avoir une probabilité de 20 %, le mot « Tu », une probabilité de 1 %, et ainsi de suite. Le mot ayant la probabilité la plus élevée est envoyé en sortie. Puisque cela ressemble fortement à une tâche de classification ordinaire, on peut entraîner le modèle en utilisant la perte "sparse_categorical_crossentropy", comme dans le modèle Char-RNN.

**Figure 8.4** – Lors de l'inférence, transmission en entrée du mot de la sortie précédente

Au moment de l'inférence (après l'entraînement), vous ne disposez d'aucune phrase cible à donner au décodeur. À la place, fournissez-lui simplement le mot qu'il a sorti lors de l'étape précédente, comme le montre la figure 8.4 (il faut pour cela une recherche de plongement, qui n'est pas montrée sur la figure).

Voilà pour la vision d'ensemble. Il reste cependant quelques détails à examiner avant de pouvoir implémenter ce modèle :

- Nous avons supposé jusqu'à présent que toutes les séries d'entrée (pour l'encodeur et le décodeur) étaient de longueur constante. Mais, il est évident que la longueur des phrases peut varier. Puisque les tenseurs réguliers ont des formes figées, ils ne peuvent contenir que des phrases de même longueur. Nous l'avons vu précédemment, vous pouvez employer le masquage pour gérer cette situation. Cependant, si les phrases ont des tailles très différentes, vous ne pourrez pas vous contenter de les couper comme nous l'avons fait pour l'analyse de sentiments (car nous voulons non pas des traductions partielles mais complètes). À la place, les phrases sont regroupées en paquets de longueurs comparables (par exemple, un paquet pour les phrases de 1 à 6 mots, un autre pour celles de 7 à 12 mots, etc.), et les phrases les plus courtes sont complétées à l'aide d'un jeton de remplissage spécial (voir la fonction `tf.data.experimental.bucket_by_sequence_length()` pour cela). Par exemple, « I drink milk » devient « <pad><pad><pad> milk drink I ».
- Puisque nous voulons ignorer toute sortie après le jeton EOS, ceux-ci ne doivent pas contribuer à la perte (ils doivent être masqués). Par exemple, si le modèle produit « Je bois du lait <eos> oui », la perte correspondant au dernier mot doit être ignorée.
- Lorsque le vocabulaire de sortie est vaste (ce qui est le cas ici), la génération d'une probabilité pour chaque mot possible sera terriblement lente. Si le vocabulaire cible comprend, par exemple, 50 000 mots français, le décodeur doit sortir des vecteurs à 50 000 dimensions. L'application de la fonction softmax à un vecteur aussi grand va demander énormément de calcul. Pour éviter cela, une solution consiste à examiner uniquement les logits produits par le modèle pour le mot correct et pour un échantillon aléatoire de mots incorrects, puis à calculer une approximation de la perte fondée sur ces logits. Cette technique softmax échantillonnée (*sampled softmax*) a été présentée en 2015 par Sébastien Jean *et al.*²⁰⁴ Dans TensorFlow, vous pouvez utiliser pour cela la fonction `tf.nn.sampled_softmax_loss()` pendant l'entraînement et choisir la fonction softmax normale pour les inférences (la fonction softmax échantillonnée ne peut pas être utilisée pour les inférences car elle a besoin de connaître la cible).

204. Sébastien Jean *et al.*, «On Using Very Large Target Vocabulary for Neural Machine Translation», *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing 1* (2015), 1-10: <https://homl.info/104>.

Le projet TensorFlow Addons propose de nombreux outils série-vers-série qui vous permettront de construire facilement des encodeurs-décodeurs prêts à passer en production. Nous avons installé cette librairie au chapitre 1 lors de la création de l'environnement tf2. Malheureusement, à l'heure de l'écriture de ces lignes, la librairie tensorflow-addons n'est pas encore disponible pour Windows (mais elle devrait l'être sous peu). À titre d'exemple, le code suivant crée un modèle encodeur-décodeur de base semblable à celui présenté à la figure 8.3 :

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                     outputs=[Y_proba])
```

Une grande partie de ce code n'a pas besoin d'explication, mais quelques points méritent notre attention. Premièrement, nous indiquons `return_state=True` lors de la création de la couche LSTM afin que nous puissions obtenir son état caché final et le transmettre au décodeur. Puisque nous utilisons une cellule LSTM, elle retourne en réalité deux états cachés (à court terme et à long terme). `TrainingSampler` fait partie des nombreux échantillonnages disponibles dans TensorFlow Addons. Leur rôle est de dire au décodeur, à chaque étape, ce qu'il doit annoncer comme étant la sortie précédente. Au cours de l'inférence, il s'agira du plongement du token qui a été réellement produit. Pendant l'entraînement, il s'agira du plongement du token cible précédent ; voilà pourquoi nous avons utilisé le `TrainingSampler`.

En pratique, il est souvent préférable de commencer l'entraînement avec le plongement de la cible de l'étape temporelle précédente et de passer progressivement à l'utilisation du plongement du token réel produit lors de l'étape précédente.

Cette idée a été décrite dans un article²⁰⁵ publié en 2015 par Samy Bengio *et al.* Le `ScheduledEmbeddingTrainingSampler` choisira aléatoirement entre la cible et la sortie réelle, avec une probabilité que vous pouvez modifier progressivement pendant l’entraînement.

8.3.1 RNR bidirectionnels

Lors de chaque étape temporelle, une couche récurrente normale examine uniquement les entrées passées et présentes avant de produire sa sortie. Autrement dit, elle est « causale » et ne peut donc pas regarder dans le futur. Si ce type de RNR a un sens pour les prévisions de séries chronologiques, pour de nombreuses tâches de TALN, comme la traduction automatique neuronale, il est souvent préférable d’examiner les mots suivants avant d’encoder un mot donné.

Prenons, par exemple, les phrases « la Reine du Royaume-Uni », « la reine de cœur » et « la reine des abeilles ». Pour encoder correctement le mot « reine », il faut examiner ce qui vient ensuite. Pour mettre en place un tel fonctionnement, on exécute deux couches récurrentes sur les mêmes entrées, l’une lisant les mots de gauche à droite, l’autre les lisant de droite à gauche. Il suffit ensuite de combiner leur sortie à chaque étape temporelle, en général en les concaténant. Nous avons là une *couche récurrente bidirectionnelle* (voir la figure 8.5).

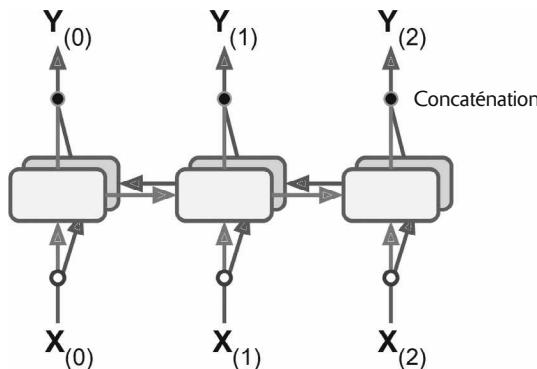


Figure 8.5 – Une couche récurrente bidirectionnelle

Pour implémenter une couche récurrente bidirectionnelle avec Keras, placez une couche récurrente dans une couche `keras.layers.Bidirectional`. Par exemple, le code suivant crée une couche GRU bidirectionnelle :

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```

²⁰⁵. Samy Bengio *et al.*, « Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks » (2015) : <https://homl.info/scheduledsampling>.



La couche Bidirectional crée un clone de la couche GRU (mais inversée), exécute les deux couches et concatène leur sortie. Bien que la couche GRU possède dix unités, la couche Bidirectional produira donc en sortie vingt valeurs par étape temporelle.

8.3.2 Recherche en faisceau

Supposons que vous entraîniez un modèle encodeur-décodeur et l'utilisiez pour traduire en anglais la phrase « Comment vas-tu ? ». Vous espérez qu'il produira la traduction appropriée (« How are you ? »), mais vous obtenez malheureusement « How will you ? ». En examinant le jeu d'entraînement, vous remarquez de nombreuses phrases comme « Comment vas-tu jouer ? », qui se traduit en « How will you play ? ». Pour le modèle, il n'était donc pas absurde de générer « How will » après avoir rencontré « Comment vas ». Dans ce cas c'était une erreur et le modèle n'a pas pu pas revenir en arrière pour la corriger. Il a donc essayé de compléter la phrase du mieux qu'il pouvait. En produisant précipitamment le mot le plus probable à chaque étape, nous n'arrivons pas à une traduction optimale.

Comment pouvons-nous donner au modèle la possibilité de revenir en arrière et de corriger ses erreurs ? L'une des solutions les plus répandues se fonde sur une recherche en faisceau. Elle consiste à garder une short list des k phrases les plus prometteuses (disons les trois principales) et, à chaque étape du décodeur, d'essayer de les étendre d'un mot, en gardant uniquement les k phrases les plus probables. Le paramètre k est la *largeur du faisceau*.

Par exemple, reprenons la traduction de la phrase « Comment vas-tu ? » en utilisant la recherche en faisceau avec une largeur de faisceau égal à 3. À la première étape du décodeur, le modèle génère une probabilité estimée pour chaque mot possible. Supposons que les trois premiers mots de la liste obtenue soient « How » (probabilité estimée de 75 %), « What » (3 %) et « You » (1 %). Ils correspondent à la short list à ce stade. Ensuite, nous créons trois copies de notre modèle et les utilisons pour rechercher le mot suivant pour chaque phrase. Chaque modèle va sortir une probabilité estimée pour chaque mot du vocabulaire. Le premier tentera de trouver le mot suivant dans la phrase « How », et produira peut-être une probabilité de 36 % pour le mot « will », 32 % pour le mot « are », 16 % pour le mot « do », et ainsi de suite. Notez qu'il s'agit en réalité de probabilités *conditionnelles*, pour le cas où la phrase commence par « How ». Le deuxième modèle tentera de compléter la phrase « What » ; il peut générer une probabilité conditionnelle de 50 % pour le mot « are », et ainsi de suite. Si l'on suppose un vocabulaire de 10 000 mots, chaque modèle produira 10 000 probabilités.

Ensuite, nous calculons les probabilités de chacune des 30 000 phrases de deux mots que ces modèles ont envisagées ($3 \times 10 000$). Pour cela, nous multiplions la probabilité conditionnelle estimée de chaque mot par la probabilité estimée de la phrase obtenue. Par exemple, puisque la probabilité estimée de la phrase « How » était de 75 %, tandis que la probabilité conditionnelle estimée du mot « will » (en supposant que le premier mot est « How ») était de 36 %, la probabilité estimée de la phrase « How will » est de $75\% \times 36\% = 27\%$. Après avoir calculé les probabilités

des 30 000 phrases de deux mots, nous conservons uniquement les trois meilleures. Peut-être qu'elles commenceront toutes par le mot « How » : « How will » (27 %), « How are » (24 %) et « How do » (12 %). Pour le moment, la gagnante est la phrase « How will », mais « How are » n'a pas été éliminée.

Nous reprenons la même procédure : nous utilisons trois modèles pour prédire le prochain mot dans chacune des trois phrases retenues et calculons les probabilités des 30 000 phrases de trois mots envisagées. Peut-être que les trois premières phrases sont à présent « How are you » (10 %), « How do you » (8 %) et « How will you » (2 %). À l'étape suivante, nous pourrions obtenir « How do you do » (7 %), « How are you <eos> » (6 %) et « How are you doing » (3 %). Vous noterez que « How will » a été éliminé et que nous disposons désormais de trois traductions tout à fait acceptables. Nous avons amélioré les performances de notre modèle encodeur-décodeur sans entraînement supplémentaire, simplement en élargissant son utilisation.

L'implémentation de la recherche en faisceau est assez simple avec TensorFlow Addons :

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

Nous créons tout d'abord un `BeamSearchDecoder`, qui enveloppe tous les clones du décodeur (dix dans ce cas). Puis, nous créons une copie de l'état final de l'encodeur pour chaque clone et passons ces états au décodeur, avec les jetons de début et de fin.

Avec cette mise en œuvre, vous pouvez obtenir de bonnes traductions lorsque les phrases sont relativement courtes (en particulier si vous utilisez des plongements de mots préentraînés). Malheureusement, ce modèle se révélera plutôt mauvais pour la traduction de longues phrases. Une fois encore, le problème vient de la mémoire à court terme limitée des RNR. La solution à ce problème vient des *mécanismes d'attention*, une innovation qui a changé la donne.

8.4 MÉCANISMES D'ATTENTION

Examinons le chemin du mot « milk » vers sa traduction en « lait » sur la figure 8.3 : il est plutôt long ! Cela signifie qu'une représentation de ce mot (ainsi que celles de tous les autres mots) doit être transportée pendant plusieurs étapes avant qu'elle ne soit réellement utilisée. Pouvons-nous raccourcir ce chemin ?

Là était l'idée centrale d'un article²⁰⁶ révolutionnaire publié en 2014 par Dzmitry Bahdanau *et al.* Les auteurs ont présenté une technique qui permet au décodeur de

206. Dzmitry Bahdanau *et al.*, « Neural Machine Translation by Jointly Learning to Align and Translate » (2014) : <https://homl.info/attention>.

se focaliser sur des mots pertinents (tels qu'encodés par l'encodeur) lors de chaque étape temporelle. Par exemple, à l'étape temporelle où le décodeur doit produire le mot « lait », il focalisera son attention sur le mot « milk ». Autrement dit, le chemin depuis le mot d'entrée vers sa traduction est à présent beaucoup plus court et les limitations en mémoire à court terme des RNR ont donc moins d'impact. Les mécanismes d'attention ont révolutionné la traduction automatique neuronale (et le TALN en général), en permettant une amélioration significative de l'état de l'art, notamment pour les longues phrases (plus de trente mots)²⁰⁷.

La figure 8.6 illustre l'architecture de ce modèle (légèrement simplifiée, comme nous le verrons). En partie gauche se trouvent l'encodeur et le décodeur. Au lieu d'envoyer au décodeur uniquement l'état caché final de l'encodeur (cette transmission est toujours présente, mais elle n'est pas représentée sur la figure), nous lui envoyons à présent toutes les sorties. À chaque étape temporelle, la cellule de mémoire du décodeur calcule une somme pondérée de toutes les sorties de l'encodeur. Cela permet de déterminer les mots sur lesquels il va se concentrer lors de cette étape.

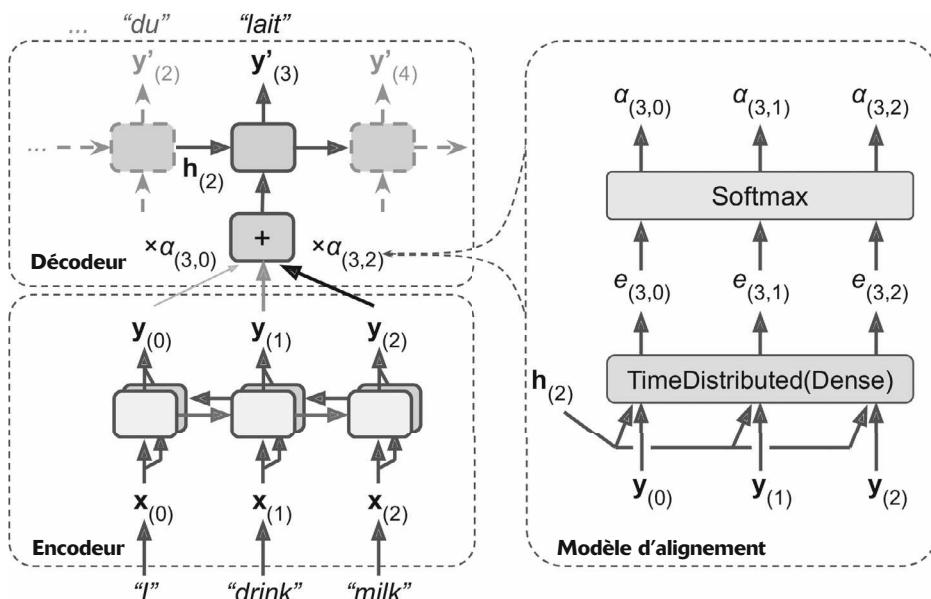


Figure 8.6 – Traduction automatique neuronale à l'aide d'un réseau encodeur-décodeur avec un modèle d'attention

Le poids $\alpha_{(t,i)}$ est celui de la $i^{\text{ème}}$ sortie de l'encodeur pour l'étape temporelle t du décodeur. Par exemple, si le poids $\alpha_{(3,2)}$ est supérieur aux poids $\alpha_{(3,0)}$ et $\alpha_{(3,1)}$, alors le

207. L'indicateur le plus utilisé en NMT est le score BLEU (*BiLingual Evaluation Understudy*). Il compare chaque traduction produite par le modèle à plusieurs bonnes traductions produites par des humains. Il compte le nombre de n -grammes (séquences de n mots) qui apparaissent dans toutes les traductions cibles et ajuste le score pour prendre en compte la fréquence des n -grammes produits dans les traductions cibles.

décodeur prêtera une attention plus élevée au mot numéro deux (« milk ») qu’aux deux autres mots, tout au moins lors de cette étape temporelle. Les autres parties du décodeur fonctionnent comme précédemment. À chaque étape temporelle, la cellule de mémoire reçoit les entrées dont nous venons de parler, plus l’état caché de l’étape temporelle précédente et (même si cela n’est pas représenté sur la figure) le mot cible de l’étape temporelle précédente (ou, pour les inférences, la sortie de l’étape temporelle précédente).

Mais d'où proviennent ces poids $\alpha_{(t,i)}$? Assez simplement, ils sont générés par un type de petits réseaux de neurones appelé *modèle d'alignement* (ou *couche d'attention*), qui est entraîné conjointement aux autres parties du modèle encodeur-décodeur. Ce modèle d'alignement est représenté en partie droite de la figure 8.6. Il commence par une couche Dense distribuée temporellement²⁰⁸ avec un seul neurone, qui reçoit en entrée toutes les sorties de l'encodeur, concaténées à l'état caché précédent du décodeur (par exemple, $h_{(2)}$). Cette couche produit un score (ou énergie) pour chaque sortie de l'encodeur (par exemple, $e_{(3,2)}$). Il mesure l'alignement de chaque sortie avec l'état caché précédent du décodeur. Enfin, tous les scores passent par une couche softmax afin d'obtenir un poids final pour chaque sortie de l'encodeur (par exemple, $\alpha_{(3,2)}$).

Pour une étape temporelle donnée du décodeur, la somme de tous les poids est égale à 1 (puisque la couche softmax n'est pas distribuée temporellement). Ce mécanisme d'attention spécifique est appelé *attention de Bahdanau* (du nom du premier auteur de l'article). Puisqu'il concatène la sortie de l'encodeur avec l'état caché précédent du décodeur, il est parfois appelé *attention par concaténation* (ou *attention additive*).



Si la phrase d'entrée contient n mots et si l'on suppose que la phrase de sortie est presque aussi longue, alors ce modèle doit calculer n^2 poids. Cette complexité de calcul quadratique n'est pas un problème, car même les plus longues phrases ne sont pas constituées de milliers de mots.

Un autre mécanisme d'attention commun a été proposé peu après, dans un article²⁰⁹ de Minh-Thang Luong *et al.* publié en 2015. Puisque l'objectif du mécanisme d'attention est de mesurer la similitude entre l'une des sorties de l'encodeur et l'état caché précédent du décodeur, les auteurs ont simplement proposé de calculer le *produit scalaire*²¹⁰ de ces deux vecteurs, car il s'agit souvent d'un bon indicateur de similitude et les équipements modernes peuvent le calculer très rapidement. Pour que cette opération soit possible, les deux vecteurs doivent avoir la même dimension. Cette technique est appelée *attention de Luong* (de nouveau, d'après le nom du premier auteur de l'article) ou, parfois, *attention multiplicativa*. Le produit scalaire donne

208. Rappelons qu'une couche Dense distribuée temporellement équivaut à une couche Dense normale que vous appliquez indépendamment à chaque étape temporelle (beaucoup plus rapidement).

209. Minh-Thang Luong *et al.*, « Effective Approaches to Attention-Based Neural Machine Translation », *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015), 1412-1421 : <https://homl.info/luongattention>.

210. Voir le chapitre 1.

une note et toutes les notes (pour une étape temporelle donnée du décodeur) passent par une couche softmax de façon à calculer les poids finaux, comme pour l'attention de Bahdanau. Une autre simplification proposée consiste à utiliser l'état caché du décodeur de l'étape temporelle courante plutôt que celui de l'étape temporelle précédente (c'est-à-dire $\mathbf{h}_{(t)}$ à la place de $\mathbf{h}_{(t-1)}$), puis d'employer la sortie du mécanisme d'attention (notée $\tilde{\mathbf{h}}_{(t)}$ pour calculer directement les prédictions du décodeur (au lieu de l'utiliser pour calculer l'état caché courant du décodeur).

Les auteurs ont également proposé une variante du mécanisme du produit scalaire dans lequel les sorties du décodeur subissent tout d'abord une transformation linéaire (c'est-à-dire une couche Dense distribuée temporellement sans terme constant) avant d'effectuer les produits scalaires. Il s'agit d'un produit scalaire « général ». Ils ont comparé les deux méthodes de calcul du produit scalaire au mécanisme d'attention par concaténation (en ajoutant un vecteur de paramètres de mise à l'échelle \mathbf{v}) et ont observé que les variantes du produit scalaire obtenaient de meilleures performances que l'attention par concaténation. C'est pourquoi cette dernière méthode est moins employée aujourd'hui. Les opérations de ces trois mécanismes d'attention sont résumées dans l'équation 8.1.

Équation 8.1 – Mécanismes d'attention

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

avec $\alpha_{(t,i)} = \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})}$

et $e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^T \mathbf{y}_{(i)} & \text{scalaire} \\ \mathbf{h}_{(t)}^T \mathbf{W} \mathbf{y}_{(i)} & \text{général} \\ \mathbf{v}^T \tanh(\mathbf{W}[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{concaténation} \end{cases}$

Voici comment ajouter une attention de Luong à un modèle encodeur-décodeur avec TensorFlow Addons :

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

Nous enveloppons simplement la cellule du décodeur dans un Attention Wrapper et fournissons le mécanisme d'attention souhaité (attention de Luong dans ce cas).

8.4.1 Attention visuelle

Les mécanismes d'attention sont à présent employés dans diverses situations. Outre la NMT, l'une de leurs premières applications a été la génération de légendes d'images

en utilisant l'*attention visuelle*²¹¹. Un réseau de neurones convolutif commence par traiter l'image et produit des cartes de caractéristiques. Ensuite, un RNR décodeur doté d'un mécanisme d'attention génère la légende, un mot à la fois. À chaque étape temporelle du décodeur (chaque mot), celui-ci utilise le modèle d'attention pour se focaliser uniquement sur la partie pertinente de l'image. Par exemple, à la figure 8.7, le modèle a généré la légende «une femme lance un frisbee dans un parc» et vous pouvez voir la partie de l'image d'entrée sur laquelle le décodeur focalise son attention avant qu'il ne produise le mot «frisbee». Il est clair que l'essentiel de son attention est porté sur le frisbee.

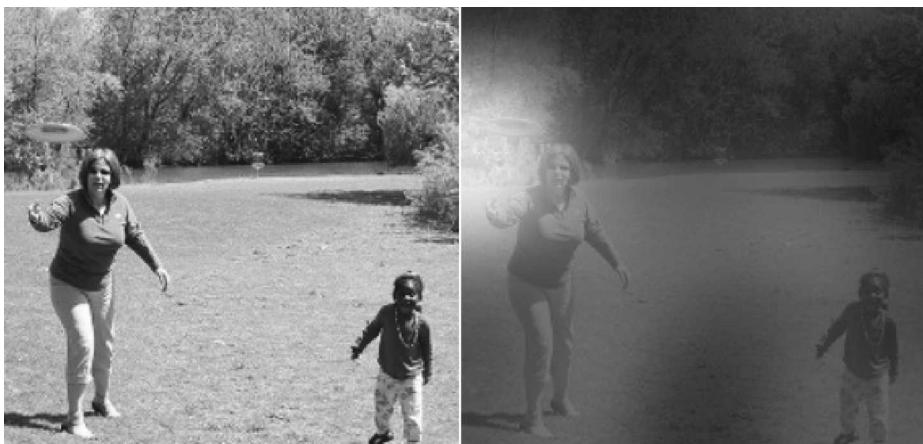


Figure 8.7 – Attention visuelle: une image d'entrée (à gauche) et la zone de focalisation du modèle avant de produire le mot «frisbee» (à droite)²¹²

Explicabilité

Les mécanismes d'attention ont pour avantage supplémentaire de faciliter la compréhension de l'origine des raisons de la sortie d'un modèle. C'est ce que l'on nomme *explicabilité*. Cela peut se révéler très utile lorsque le modèle fait une erreur.

Par exemple, si la légende produite pour l'image d'un chien marchant dans la neige est «un loup marchant dans la neige», vous pouvez revenir en arrière et vérifier ce sur quoi le modèle s'est focalisé lorsqu'il a généré le mot «loup». Vous constaterez peut-être qu'il prêtait attention non seulement au chien mais également à la neige, essayant de trouver une explication possible: le modèle avait peut-être appris à distinguer les chiens des loups en vérifiant si l'environnement est enneigé. Pour corriger ce problème, il suffit d'entraîner le modèle avec d'autres images de loups sans neige et

211. Kelvin Xu *et al.*, «Show, Attend and Tell: Neural Image Caption Generation with Visual Attention», *Proceedings of the 32nd International Conference on Machine Learning* (2015), 2048-2057: <https://homl.info/visualattention>.

212. Il s'agit d'un extrait de la figure 3 de l'article, reproduit avec l'aimable autorisation des auteurs.

de chiens avec de la neige. Cet exemple est tiré de l'excellent article²¹³ publié en 2016 par Marco Tulio Ribeiro *et al.*, dans lequel les auteurs utilisent une autre approche pour l'explicabilité : apprendre un modèle interprétable localement autour d'une prédition d'un classificateur.

Dans certaines applications, l'explicabilité n'est pas qu'un simple outil de débogage d'un modèle. Il peut s'agir d'une exigence légale (pensez à un système qui décide s'il doit vous accorder un prêt).

Les mécanismes d'attention sont si puissants que vous pouvez construire des modèles de pointe en utilisant uniquement ces mécanismes.

8.4.2 L'attention et rien d'autre : l'architecture Transformer

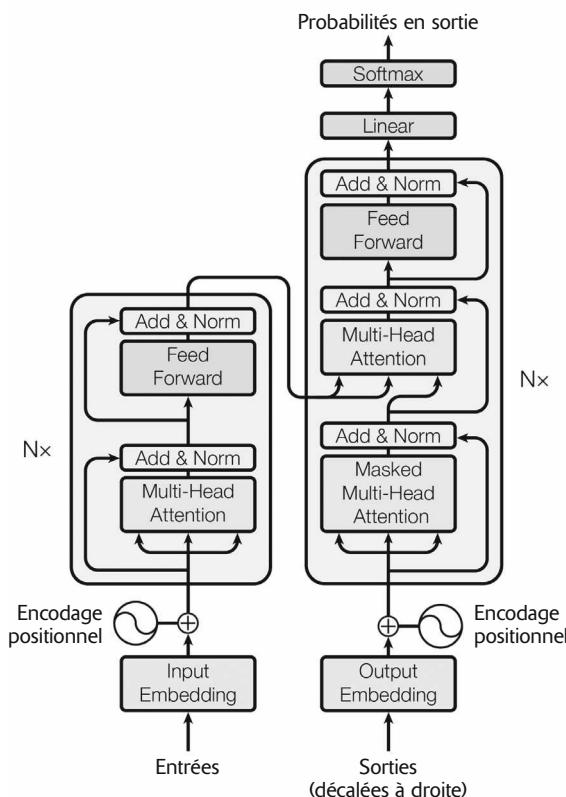


Figure 8.8 – L'architecture Transformer²¹⁴

213. Marco Tulio Ribeiro *et al.*, « ‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier », *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), 1135-1144 : <https://hml.info/explainclass>.

214. Il s'agit de la figure 1 de l'article, reproduite avec l'aimable autorisation des auteurs.

Dans un article²¹⁵ révolutionnaire publié en 2017, une équipe de chercheurs chez Google a suggéré que l'attention suffit. Ils ont réussi à créer une architecture, nommée *Transformer*, qui a énormément amélioré l'état de l'art dans la NMT. Elle n'utilise aucune couche récurrente ou convolutive²¹⁶, uniquement des mécanismes d'attention (plus des couches de plongements, des couches denses, des couches de normalisation et quelques autres éléments). Cerise sur le gâteau, elle est également beaucoup plus rapide à entraîner et plus facile à paralléliser. Ils ont donc réussi à l'entraîner en un temps et pour un coût beaucoup plus faibles que les modèles de pointe précédents.

L'architecture *Transformer* est représentée à la figure 8.8.

Examinons les différents éléments de cette architecture :

- La partie gauche correspond à l'encodeur. Comme précédemment, il prend en entrée un lot de phrases représentées sous forme de suites d'identifiants de mots (la forme de l'entrée est *[taille de lot, longueur maximale d'une phrase d'entrée]*) et encode chaque mot sous forme d'une représentation à 512 dimensions (la forme de la sortie de l'encodeur est donc *[taille de lot, longueur maximale d'une phrase d'entrée, 512]*). Notez que la partie supérieure de l'encodeur est empilée N fois (dans l'article, $N = 6$).
- La partie droite correspond au décodeur. Pendant l'entraînement, il prend en entrée la phrase cible (également représentée sous forme d'une suite d'identifiants de mots), décalée d'une étape temporelle vers la droite (autrement dit, un jeton début de séquence est inséré en tête). Il reçoit également les sorties de l'encodeur (c'est-à-dire les flèches issues du côté gauche). Notez que la partie supérieure du décodeur est empilée N fois et que les sorties finales de la pile de l'encodeur sont fournies au décodeur à chacun de ces N niveaux. Comme précédemment, le décodeur produit une probabilité pour chaque mot suivant possible, à chaque étape temporelle (sa sortie est de forme *[taille de lot, longueur maximale d'une phrase d'entrée, longueur du vocabulaire]*).
- Au cours d'une inférence, puisque le décodeur ne peut pas recevoir de cible, nous lui fournissons les mots de sortie précédents (commençant par un jeton début de séquence). Le modèle doit donc être appelé de façon répétée, en prédisant un mot supplémentaire à chaque tour (qui est fourni au décodeur lors du tour suivant, jusqu'à la sortie du jeton fin de séquence).
- En y regardant de plus près, vous pouvez voir que la plupart des composants vous sont déjà familiers. Nous trouvons deux couches de plongements, $5 \times N$ connexions de sauts, chacune suivie d'une couche de normalisation, $2 \times N$ modules «Feed Forward» constitués de deux couches denses chacun (la première utilise la fonction d'activation ReLU, la seconde en est dépourvue), et, pour finir, une couche dense de sortie qui utilise la fonction d'activation

215. Ashish Vaswani *et al.*, «Attention Is All You Need», *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), 6000-6010: <https://homl.info/transformer>.

216. Puisque *Transformer* utilise des couches Dense distribuées temporellement, vous pouvez défendre le fait que cette architecture utilise des couches convolutives à une dimension avec un noyau de taille 1.

softmax. Puisque toutes ces couches sont distribuées temporellement, chaque mot est traité indépendamment des autres. Mais comment est-il possible de traduire une phrase en examinant uniquement un mot à la fois ? C'est là que les nouveaux composants entrent en scène :

- La couche d'attention à têtes multiples (*Multi-Head Attention*) de l'encodeur encode la relation de chaque mot avec chaque autre mot de la même phrase, en prêtant une attention plus élevée aux mots les plus pertinents. Par exemple, la sortie de cette couche pour le mot « Queen » dans la phrase « They welcomed the Queen of the United Kingdom » dépendra de tous les mots de cette phrase, mais l'attention portée sera probablement plus importante sur les mots « United » et « Kingdom » que sur les mots « They » ou « welcomed ». Ce mécanisme d'attention est appelé *auto-attention* (la phrase prête attention à elle-même). Nous détaillerons son fonctionnement plus loin. La couche *Masked Multi-Head Attention* du décodeur procède de la même manière, mais chaque mot ne peut être associé qu'aux mots qui le précèdent. Enfin, c'est dans la couche *Multi-Head Attention* supérieure du décodeur que celui-ci prête attention aux mots de la phrase d'entrée. Par exemple, le décodeur fera probablement très attention au mot « Queen » dans la phase d'entrée lorsqu'il cherchera à sortir sa traduction.
- Les *plongements positionnels* sont simplement des vecteurs denses (semblables aux plongements de mots) qui représentent la position d'un mot dans la phrase. Le $i^{\text{ème}}$ plongement positionnel est ajouté au plongement de mot du $i^{\text{ème}}$ mot dans chaque phrase. Cela permet modèle d'accéder à la position de chaque mot, une information indispensable car les couches *Multi-Head Attention* ne prennent pas en compte l'ordre ou la position des mots ; elles examinent uniquement leurs relations. Puisque toutes les autres couches sont distribuées temporellement, elles n'ont aucun moyen de connaître la position de chaque mot (relative ou absolue). Pourtant, les positions relatives ou absolues d'un mot sont importantes et nous devons trouver le moyen de fournir cette information au Transformer. Les plongements positionnels sont une bonne manière de procéder.

Examinons de plus près ces deux nouveaux composants de l'architecture Transformer, en commençant par les plongements positionnels.

Plongements positionnels

Un plongement positionnel (*positional embedding*) est un vecteur dense qui encode la position d'un mot à l'intérieur d'une phrase : le $i^{\text{ème}}$ plongement positionnel est simplement ajouté au plongement de mot du $i^{\text{ème}}$ mot dans la phrase. Ces plongements positionnels peuvent être appris par le modèle, mais, dans leur article, les auteurs ont préféré employer des plongements positionnels figés, définis à l'aide des fonctions sinus et cosinus de différentes fréquences. La matrice des plongements positionnels \mathbf{P} est définie par l'équation 8.2 et représentée en partie inférieure de la figure 8.9 (transposée), où $P_{p,i}$ est le $i^{\text{ème}}$ composant du plongement pour le mot qui se trouve au $p^{\text{ième}}$ emplacement dans la phrase.

Équation 8.2 – Plongements positionnels sinus/cosinus

$$P_{p,2i} = \sin \frac{p}{10000^{\frac{2i}{d}}}$$

$$P_{p,2i+1} = \cos \frac{p}{10000^{\frac{2i}{d}}}$$

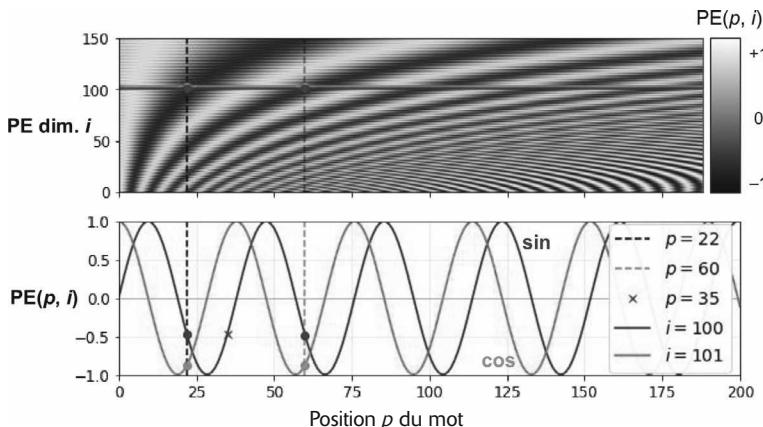


Figure 8.9 – Matrice des plongements positionnels sinus/cosinus (transposée, en haut) avec une focalisation sur deux valeurs de i (en bas)

Cette solution permet d'obtenir les mêmes performances qu'avec des plongements positionnels appris, mais, puisqu'elle peut s'étendre à des phrases arbitrairement longues, elle a été préférée. Après que les changements positionnels ont été ajoutés aux plongements de mots, le reste du modèle a accès à la position absolue de chaque mot dans la phrase, car il existe un plongement positionnel unique pour chaque emplacement (par exemple, le plongement positionnel du mot situé au 22^e emplacement dans la phrase est représenté par la ligne pointillée verticale en partie inférieure gauche de la figure 8.9 et vous pouvez constater qu'il est unique en cette position).

Par ailleurs, en raison des fonctions oscillantes (sinus et cosinus), le modèle est également capable d'apprendre des positions relatives. Par exemple, des mots séparés par 38 mots (comme aux emplacements $p = 22$ et $p = 60$) conservent leur valeur de plongement positionnel dans les dimensions de plongement $i = 100$ et $i = 101$ (voir la figure 8.9). Voilà pourquoi nous avons besoin du sinus et du cosinus pour chaque fréquence: si nous avions utilisé uniquement le sinus (la courbe à $i = 100$), le modèle n'aurait pas su distinguer les emplacements $p = 25$ et $p = 35$ (signalés par une croix).

TensorFlow ne propose aucune couche `PositionalEmbedding`, mais sa création n'a rien de compliqué. Pour une question d'efficacité, nous précalculons la matrice des plongements positionnels dans le constructeur (nous devons donc

connaître la longueur maximale d'une phrase, `max_steps`, ainsi que le nombre de dimensions pour chaque représentation d'un mot `max_dims`). Ensuite, la méthode `call()` adapte cette matrice de plongements à la taille des entrées et l'ajoute aux entrées. Puisque nous avons ajouté une première dimension supplémentaire de taille 1 lors de la création de la matrice des plongements positionnels, les règles de diffusion garantiront que la matrice est ajoutée à chaque phrase d'entrée :

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims doit être pair
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**2 * i / max_dims).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**2 * i / max_dims).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Puis nous créons les premières couches du Transformer :

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Nous allons à présent entrer au cœur du modèle Transformer : la couche Multi-Head Attention.

Attention à têtes multiples

Pour comprendre le fonctionnement d'une couche Multi-Head Attention, nous devons tout d'abord expliquer la couche *Scaled Dot-Product Attention*, sur laquelle elle se fonde. Supposons que l'encodeur ait analysé la phrase d'entrée « They played chess » et qu'il ait réussi à comprendre que le mot « They » correspond au sujet et que le mot « played » est le verbe. Il a donc encodé ces informations dans les représentations de ces mots. Supposons à présent que le décodeur ait déjà traduit le sujet et qu'il pense qu'il doit ensuite traduire le verbe. Pour cela, il doit récupérer le verbe dans la phase d'entrée. Cela équivaut à une recherche dans un dictionnaire : c'est comme si l'encodeur avait créé un dictionnaire {« subject » : « They », « verb » : « played »...} et que le décodeur souhaitait récupérer la valeur qui correspond à la clé « verb ». Cependant, le modèle ne dispose d'aucun jeton discret pour représenter les clés (comme « subject » ou « verb »). Puisqu'il possède des représentations vectorielles de ces concepts (apprises au cours de l'entraînement), la clé qu'il utilisera pour la recherche (appelée *requête*) ne correspondra pas exactement à une clé du dictionnaire.

La solution consiste à calculer une mesure de similitude entre la requête et chaque clé du dictionnaire, puis d'utiliser la fonction softmax pour convertir ces scores de similitude en poids dont la somme totale est égale à 1. Si la clé qui représente le verbe est de loin la plus semblable à la requête, alors le poids de cette clé sera proche de 1. Ensuite, le modèle peut calculer une somme pondérée des valeurs correspondantes et, si le poids de la clé « verb » est proche de 1, alors la somme pondérée sera proche de la représentation du mot « played ». En résumé, vous pouvez voir l'intégralité de ce processus comme une recherche différentielle dans un dictionnaire. La mesure de similitude utilisée par Transformer n'est que le produit scalaire, à l'instar de l'attention de Luong. En réalité, l'équation est identique à celle de l'attention de Luong, à l'exception du facteur de mise à l'échelle. Elle est donnée dans l'équation 86.3, sous forme vectorisée.

Équation 8.3 – Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{clés}}}}\right)\mathbf{V}$$

Dans cette équation :

- \mathbf{Q} est une matrice qui contient une ligne par requête. Sa forme est $[n_{\text{requêtes}}, d_{\text{clés}}]$, où $n_{\text{requêtes}}$ est le nombre de requêtes, et $d_{\text{clés}}$, le nombre de dimensions de chaque requête et chaque clé.
- \mathbf{K} est une matrice qui contient une ligne par clé. Sa forme est $[n_{\text{clés}}, d_{\text{clés}}]$, où $n_{\text{clés}}$ est le nombre de clés et de valeurs.
- \mathbf{V} est une matrice qui contient une ligne par valeur. Sa forme est $[n_{\text{clés}}, d_{\text{valeurs}}]$, où d_{valeurs} est le nombre de chaque valeur.
- La forme de $\mathbf{Q}\mathbf{K}^T$ est $[n_{\text{requêtes}}, n_{\text{clés}}]$: elle contient un score de similitude pour chaque couple requête/clé. La sortie de la fonction softmax a la même forme, mais la somme de toutes les lignes est égale à 1. La forme de la sortie finale est $[n_{\text{requêtes}}, d_{\text{valeurs}}]$: nous avons une ligne par requête, où chaque ligne représente le résultat de la requête (une somme pondérée des valeurs).
- Le facteur de mise à l'échelle réduit les scores de similitude afin d'éviter la saturation de la fonction softmax, qui pourrait conduire à des gradients minuscules.
- Il est possible de masquer certains couples clés/valeur en ajoutant une très grande valeur négative au score de similitude correspondant, juste avant d'appliquer la fonction softmax. Cela sera utile dans la couche Masked Multi-Head Attention.

Dans l'encodeur, cette équation est appliquée à chaque phrase d'entrée du lot, en prenant \mathbf{Q} , \mathbf{K} et \mathbf{V} toutes égales à la liste des mots de la phrase d'entrée (chaque mot de la phrase sera donc comparé à chaque mot de la même phrase, y compris lui-même). De façon similaire, dans la couche d'attention masquée du décodeur, l'équation sera appliquée à chaque phrase cible du lot, avec \mathbf{Q} , \mathbf{K} et \mathbf{V} toutes égales à la liste des mots de la phrase cible, mais, cette fois-ci, en utilisant un masque afin d'éviter qu'un mot ne soit comparé aux mots qui suivent (au moment de l'inférence, le décodeur

n'aura accès qu'aux mots déjà produits, non aux mots futurs, et l'entraînement doit donc masquer les jetons de sortie futurs). Dans la couche d'attention supérieure du décodeur, les clés **K** et les valeurs **V** sont simplement la liste des encodages de mots produits par l'encodeur, tandis que les requêtes **Q** sont la liste des encodages de mots produits par le décodeur.

La couche `keras.layers.Attention` implémente cette Scaled Dot-Product Attention, en appliquant efficacement l'équation 8.3 à de multiples phrases d'un lot. Ses entrées sont comme **Q**, **K** et **V**, à l'exception d'une dimension de lot supplémentaire (la première).



Dans TensorFlow, si **A** et **B** sont des tenseurs ayant plus de deux dimensions – par exemple, de forme [2, 3, 4, 5] et [2, 3, 5, 6], respectivement –, alors `tf.matmul(A, B)` traitera ces tenseurs comme des tableaux 2x3 où chaque cellule contient une matrice et multipliera les matrices correspondantes : la matrice de la ligne *i* et de la colonne *j* de **A** sera multipliée par la matrice de la ligne *i* et de la colonne *j* de **B**. Le produit d'une matrice 4x5 par une matrice 5x6 étant une matrice 4x6, `tf.matmul(A, B)` retournera alors un tableau de forme [2, 3, 4, 6].

Si nous ignorons les connexions de saut, les couches de normalisation de couche, les blocs Feed Forward et le fait qu'il s'agit non pas d'une vraie Multi-Head Attention mais d'une Scaled Dot-Product Attention, le reste du modèle Transformer peut être implémenté de la manière suivante :

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True) ([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True) ([Z, Z])
    Z = keras.layers.Attention(use_scale=True) ([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax")) (Z)
```

L'argument `use_scale=True` crée un paramètre supplémentaire qui permet à la couche d'apprendre à réduire correctement les scores de similitude. C'est là une petite différence par rapport au modèle Transformer, qui réduit toujours les scores de similitude du même facteur ($\sqrt{d_{\text{clés}}}$). L'argument `causal=True` indiqué lors de la création de la seconde couche d'attention garantit que chaque jeton de sortie accompagne non pas les futurs jetons mais uniquement les jetons de sorties précédents.

Il est temps à présent d'examiner la dernière pièce du puzzle : qu'est-ce qu'une couche Multi-Head Attention ? Son architecture est représentée à la figure 8.10.

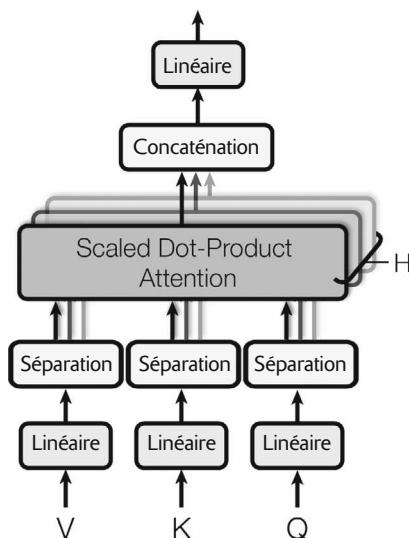


Figure 8.10 – Architecture d'une couche Multi-Head Attention²¹⁷

Vous le constatez, il s'agit simplement d'un groupe de couches Scaled Dot-Product Attention, chacune précédée d'une transformation linéaire des valeurs, des clés et des requêtes (autrement dit, une couche Dense distribuée temporellement sans fonction d'activation). Toutes les sorties sont simplement concaténées et le résultat passe par une dernière transformation linéaire (de nouveau distribuée temporellement). Pour quelles raisons ? Quelle est l'idée sous-jacente à cette architecture ? Prenons le mot « They » dans le contexte de la phrase précédente « They played chess ». L'encodeur a été suffisamment intelligent pour encoder le fait qu'il s'agit d'un verbe. Mais la représentation du mot comprend également son emplacement dans le texte, grâce aux encodages positionnels, et elle inclut probablement d'autres informations utiles à sa traduction, comme le fait qu'il est conjugué au passé.

En résumé, la représentation d'un mot encode de nombreuses caractéristiques différentes du mot. Si nous utilisons uniquement une seule couche Scaled Dot-Product Attention, nous sommes incapables de demander toutes ces caractéristiques en une fois. Voilà pourquoi la couche Multi-Head Attention applique différentes transformations linéaires aux valeurs, clés et requêtes : cela permet au modèle d'appliquer différentes projections de la représentation du mot dans différents sous-espaces, chacun se focalisant sur un sous-ensemble des caractéristiques du mot. Peut-être que l'une des couches linéaires projette la représentation du mot dans un sous-espace où ne reste plus que l'information indiquant que ce mot est un verbe, qu'une autre extraîtra le fait qu'il est conjugué au passé, etc. Ensuite, les couches Scaled Dot-Product Attention implémentent la phase de recherche, puis, finalement, tous les résultats sont concaténés et projetés de nouveau dans l'espace d'origine.

217. Il s'agit de la partie droite de la figure 2 de l'article, reproduite avec l'aimable autorisation des auteurs.

Au moment de l'écriture de ces lignes, aucune classe Transformer ou MultiHeadAttention n'est disponible pour TensorFlow 2. Vous pouvez toutefois vous tourner vers l'excellent tutoriel TensorFlow de construction d'un modèle Transformer pour la compréhension du langage (<https://homl.info/transformertuto>). Par ailleurs, l'équipe TF Hub est en train de porter plusieurs modules Transformer sur TensorFlow 2 ; ils devraient être disponibles sous peu. En attendant, j'espère avoir montré que vous pouvez facilement implémenter un Transformer par vous-même, et c'est indiscutablement un très bon exercice !

8.5 INNOVATIONS RÉCENTES DANS LES MODÈLES DE LANGAGE

L'année 2018 a été appelée le « moment ImageNet pour le TALN » : les avancées ont été stupéfiantes, avec des LSTM de plus en plus larges et des architectures à base de Transformer entraînées sur des jeux de données immenses. Je vous conseille fortement de lire les articles suivants, tous publiés en 2018 :

- L'article ELMo²¹⁸ rédigé par Matthew Peters présente les plongements pour les modèles de langage (ELMo, *Embeddings from Language Models*) : il s'agit de plongements de mots contextualisés appris à partir des états internes d'un modèle de langage bidirectionnel profond. Par exemple, le mot « reine » n'aura pas le même plongement dans « Reine du Royaume-Uni » et dans « reine des abeilles ».
- L'article ULMFiT²¹⁹ écrit par Jeremy Howard et Sebastian Ruder démontre l'efficacité des préentraînements non supervisés pour les tâches de TALN. Les auteurs ont entraîné un modèle de langage LSTM en utilisant un apprentissage autosupervisé (c'est-à-dire en générant automatiquement les étiquettes à partir des données) sur un très volumineux corpus de textes, pour l'affiner ensuite sur différentes tâches. Leur modèle a largement dépassé l'état de l'art sur six tâches de classification de texte (en réduisant le taux d'erreur de 18 à 24 % dans la plupart des cas). Ils ont également montré qu'en peaufinant le modèle préentraîné sur uniquement 100 exemples étiquetés, ils pouvaient arriver aux mêmes performances qu'un modèle entraîné à partir de zéro sur 10 000 exemples.
- L'article GPT²²⁰ publié par Alec Radford et d'autres chercheurs d'OpenAI démontre également l'efficacité du préentraînement non supervisé, mais cette fois-ci en utilisant une architecture de type Transformer. Les auteurs ont préentraîné une architecture vaste mais relativement simple constituée d'une

218. Matthew Peters *et al.*, « Deep Contextualized Word Representations », *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 1 (2018), 2227-2237 : <https://homl.info/elmo>.

219. Jeremy Howard et Sebastian Ruder, « Universal Language Model Fine-Tuning for Text Classification », *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 1 (2018), 328-339 : <https://homl.info/ulmfit>.

220. Alec Radford *et al.*, « Improving Language Understanding by Generative Pre-Training » (2018) : <https://homl.info/gpt>.

pile de douze modules Transformer (en utilisant uniquement des couches Masked Multi-Head Attention) sur un grand jeu de données, de nouveau avec un apprentissage autosupervisé. Ils l'ont ensuite peaufiné sur différentes tâches linguistiques, en appliquant uniquement des adaptations mineures pour chaque tâche. Les tâches étaient plutôt variées. Ils ont choisi la classification de texte, l'*implication* (si une phrase A implique une phrase B)²²¹, les similitudes (par exemple, « Nous avons beau temps aujourd’hui » est très similaire à « C'est ensoleillé ») et la réponse aux questions (à partir de quelques paragraphes de texte établissant un contexte, le modèle doit répondre à des questions à choix multiple).

Quelques mois après, en février 2019, Alec Radford, Jeffrey Wu et d'autres chercheurs d'OpenAI ont publié l'article GPT-2²²², dans lequel ils proposent une architecture comparable, mais encore plus vaste (avec plus de 1,5 milliard de paramètres!). Ils ont montré qu'elle permettait d'atteindre de bonnes performances sur de nombreuses tâches sans aucun réglage fin. Cette approche se nomme *Zero-Shot Learning* (ZSL). Une version plus petite du modèle GPT-2 (avec « seulement » 117 millions de paramètres) est disponible à l'adresse <https://github.com/openai/gpt-2>, avec ses poids préentraînés.

- L'article BERT²²³ de Jacob Devlin et d'autres chercheurs de Google démontre également l'efficacité du préentraînement autosupervisé sur un vaste corpus, en utilisant une architecture comparable à GPT mais avec des couches Multi-Head Attention non masquées (comme l'encodeur de Transformer). Cela signifie que le modèle est naturellement bidirectionnel, d'où le B de BERT (*Bidirectional Encoder Representations from Transformers*). Le plus important est que les auteurs ont proposé deux tâches de préentraînement qui expliquent en grande partie la force du modèle :
 - *Masked Language Model (MLM)*

Chaque mot d'une phrase a une probabilité de 15 % d'être masqué et le modèle est entraîné à prédire les mots masqués. Par exemple, si la phrase d'origine est « Elle s'est amusée à la fête d'anniversaire », alors le modèle peut recevoir « Elle <masque> amusée à la <masque> d'anniversaire » et doit prédire les mots « s'est » et « fête » (les autres sorties seront ignorées). Pour être plus précis, chaque mot sélectionné a 80 % de chances d'être masqué, 10 % de chances d'être remplacé par un mot aléatoire (pour réduire la divergence entre le préentraînement et le peaufinage, car le modèle ne verra

221. Par exemple, la phrase « Jeanne s'est beaucoup amusée à la fête d'anniversaire de son amie » implique « Jeanne a apprécié la fête » mais est contredite par « Tout le monde a détesté la fête » et n'a aucun rapport avec « La Terre est plate ».

222. Alec Radford et al., « Language Models Are Unsupervised Multitask Learners » (2019) : <https://hml.info/gpt2>.

223. Jacob Devlin et al., « BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding », *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 1 (2019) : <https://hml.info/bert>.

pas les jetons <masque> pendant le peaufinage), et 10 % de chances d'être écarté (pour attirer le modèle vers la réponse correcte).

– Next Sentence Prediction (NSP)

Le modèle est entraîné à prédire si deux phrases sont consécutives ou non. Par exemple, il doit prédire que les phrases « Le chien dort » et « Il ronfle bruyamment » sont consécutives, contrairement aux phrases « Le chien dort » et « La Terre tourne autour du Soleil ». Il s'agit d'une tâche difficile qui améliore énormément les performances du modèle lorsqu'il est peaufiné sur des tâches telles que la réponse à des questions ou l'implication.

Vous le constatez, les principales innovations en 2018 et 2019 ont concerné la meilleure tokenisation des sous-mots, en passant des LSTM aux Transformer, et en préentraînant des modèles de langage universels avec un apprentissage autosupervisé, puis en les peaufinant avec très peu de changements architecturaux (voire aucun). Les choses bougent rapidement ; personne ne peut dire quelles architectures prédomineront l'année prochaine. Aujourd'hui, il s'agit clairement des Transformer, mais demain pourrait voir émerger les CNN (lire, par exemple, l'article²²⁴ publié en 2018 par Maha Elbayad *et al.*, dans lequel les chercheurs utilisent des couches de convolution masquées à deux dimensions pour des tâches série-vers-série). Il pourrait également s'agir des RNR, s'ils font un retour surprise (lire, par exemple, l'article²²⁵ publié en 2018 par Shuai Li *et al.*, qui montre que, en rendant les neurones indépendants les uns des autres dans une couche de RNR donnée, il est possible d'entraîner des RNR beaucoup plus profonds capables d'apprendre des séries beaucoup plus longues).

Dans le prochain chapitre, nous verrons comment apprendre des représentations profondes de manière non supervisée en utilisant des autoencodeurs et nous utiliserons des réseaux antagonistes génératifs (GAN) pour produire, entre autres, des images !

8.6 EXERCICES

1. Citez les avantages et les inconvénients de l'utilisation d'un RNR avec état par rapport à celle d'un RNR sans état.
2. Pourquoi utiliserait-on des RNR de type encodeur-décodeur plutôt que des RNR purement série-vers-série pour la traduction automatique ?
3. Comment prendriez-vous en charge des séries d'entrée de longueur variable ? Qu'en est-il des séries de sortie de longueur variable ?
4. Qu'est-ce que la recherche en faisceau et pourquoi voudriez-vous l'utiliser ? Donnez un outil qui permet de la mettre en œuvre.

224. Maha Elbayad *et al.*, « Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction » (2018) : <https://homl.info/pervasiveattention>.

225. Shuai Li *et al.*, « Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN », *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), 5457-5466 : <https://homl.info/indrnn>.

5. Qu'est-ce qu'un mécanisme d'attention ? En quoi peut-il aider ?
6. Quelle est la couche la plus importante dans l'architecture Transformer ? Précisez son objectif.
7. De quoi avez-vous besoin pour utiliser une fonction softmax échantillonnée ?
8. Dans leur article sur les LSTM, Hochreiter et Schmidhuber ont utilisé des grammaires de Reber embarquées (<https://homl.info/93>). Il s'agit de grammaires artificielles qui produisent des chaînes de caractères comme « BPBTSXXVPSEPE ». Jenny Orr propose une bonne introduction sur ce sujet (<https://homl.info/108>). Choisissez une grammaire de Reber embarquée spécifique (comme celle représentée sur la page de Jenny Orr), puis entraînez un RNR pour qu'il détermine si une chaîne respecte ou non cette grammaire. Vous devrez tout d'abord écrire une fonction capable de générer un lot d'entraînement contenant environ 50 % de chaînes qui respectent la grammaire et 50 % qui ne la respectent pas.
9. Entraînez un modèle encodeur-décodeur capable de convertir une chaîne de caractères représentant une date d'un format en un autre (par exemple, de « 22 avril 2019 » à « 2019-04-22 »).
10. Lisez le tutoriel TensorFlow Neural Machine Translation with Attention (<https://homl.info/nmttuto>).
11. Utilisez l'un des modèles de langage récents (par exemple, BERT) pour générer un texte shakespearien plus convaincant.

Les solutions de ces exercices sont données à l'annexe A.

9

Apprentissage de représentations et apprentissage génératif avec des autoencodeurs et des GAN

Les autoencodeurs sont des réseaux de neurones artificiels capables d'apprendre des représentations denses des données d'entrée, appelées *représentations latentes* ou *codages*, sans aucune supervision (autrement dit, le jeu d'entraînement est dépourvu d'étiquettes). Ces codages ont généralement une dimensionnalité plus faible que les données d'entrée, d'où l'utilité des autoencodeurs dans la réduction de dimensionnalité²²⁶, notamment pour les applications de vision. Les autoencodeurs sont également des détecteurs de caractéristiques puissants et ils peuvent être utilisés pour le préentraînement non supervisé de réseaux de neurones profonds (comme nous l'avons indiqué au chapitre 3). Enfin, certains autoencodeurs sont des *modèles génératifs* capables de produire aléatoirement de nouvelles données qui ressemblent énormément aux données d'entraînement. Par exemple, en entraînant un tel autoencodeur sur des photos de visages, il sera capable de générer de nouveaux visages. Toutefois, les images obtenues sont souvent floues et pas toujours très réalistes.

À l'opposé, les visages générés par les réseaux antagonistes génératifs (GAN, *Generative Adversarial Network*) sont aujourd'hui si convaincants qu'il est difficile

226. Voir le chapitre 8 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

de croire que les personnes représentées sont virtuelles. Vous pouvez en juger par vous-même en allant sur le site d'adresse <https://thispersondoesnotexist.com/>. Il présente des visages produits par une architecture GAN récente nommée StyleGAN. (Vous pouvez également visiter <https://thisrentaldoesnotexist.com/> pour visualiser quelques chambres Airbnb générées.) Les GAN sont à présent très utilisés pour la super-résolution (augmentation de la résolution d'une image), la colorisation (<https://github.com/jantic/DeOldify>), la retouche élaborée d'images (par exemple, remplacer les éléments indésirables dans une photo par des arrière-plans réalistes), convertir une simple esquisse en une image photoréaliste, prédire les trames suivantes dans une vidéo, augmenter un jeu de données (pour entraîner d'autres modèles), générer d'autres types de données (par exemple du texte, de l'audio ou des séries chronologiques), identifier les faiblesses d'autres modèles pour les renforcer, etc.

Les autoencodeurs et les GAN sont des réseaux non supervisés. Ils apprennent des représentations denses, peuvent être utilisés en tant que modèles génératifs et ont des applications comparables. En revanche, ils fonctionnent de façons très différentes :

- Les autoencodeurs opèrent en apprenant simplement à copier leurs entrées vers leurs sorties. Cela peut sembler une tâche triviale mais nous verrons qu'elle peut se révéler assez difficile lorsque l'on fixe des contraintes au réseau. Par exemple, on peut limiter la taille des représentations latentes ou ajouter du bruit aux entrées et entraîner le réseau pour qu'il rétablisse les entrées d'origine. Ces contraintes évitent que l'autoencodeur se contente de recopier les entrées directement vers les sorties et l'obligeant à découvrir des méthodes efficaces de représentation des données. En résumé, les codages sont des sous-produits issus des tentatives de l'autoencodeur d'apprendre la fonction identité sous certaines contraintes.
- Les GAN sont constitués de deux réseaux de neurones : un *générateur* tente de produire des données qui ressemblent aux données d'entraînement, et un *discriminateur* essaie de différencier les données réelles et les données factices. Cette architecture est très originale dans le monde du Deep Learning, car, pendant l'entraînement, le générateur et le discriminateur sont mis en concurrence. Le générateur est souvent comparé au criminel qui tente de produire de faux billets réalistes, tandis que le discriminateur est le policier qui essaie de faire la différence entre les vrais et les faux billets. L'*entraînement antagoniste* (l'entraînement de réseaux de neurones en concurrence) est considéré comme l'une des idées les plus importantes de ces dernières années. En 2016, Yann LeCun a même dit qu'il s'agissait de « l'idée la plus intéressante émise ces dix dernières années dans le domaine du Machine Learning ».

Dans ce chapitre, nous commencerons par détailler le fonctionnement des autoencodeurs et leur utilisation pour la réduction de la dimensionnalité, l'extraction de caractéristiques, le préentraînement non supervisé ou en tant que modèles génératifs. Cela nous conduira naturellement aux GAN. Nous construirons tout d'abord un GAN simple permettant de générer de fausses images, mais nous verrons que son entraînement est souvent assez difficile. Nous expliquerons les principales difficultés

de l'entraînement antagoniste, ainsi que quelques techniques majeures pour les contourner. Commençons par les autoencodeurs !

9.1 REPRÉSENTATIONS EFFICACES DES DONNÉES

Parmi les séries de nombres suivantes, laquelle trouvez-vous la plus facile à mémoriser ?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

Puisque la première série est plus courte, on pourrait penser qu'il est plus facile de s'en souvenir. Cependant, en examinant attentivement la seconde, on remarque qu'il s'agit simplement d'une liste de nombres pairs allant de 50 à 14. Dès lors que ce motif a été identifié, la seconde série devient beaucoup plus facile à mémoriser que la première, car il suffit de se souvenir du motif (c'est-à-dire des nombres pairs par ordre décroissant) et des premier et dernier nombres (c'est-à-dire 50 et 14). Si nous avions la capacité de mémoriser rapidement et facilement de très longues séries, nous n'aurions pas besoin de nous préoccuper de l'existence d'un motif dans la seconde suite et nous pourrions simplement apprendre chaque nombre par cœur. C'est cette difficulté de mémorisation des longues suites qui donne tout son intérêt à la reconnaissance de motifs. Cela permet également de comprendre pourquoi contraindre l'entraînement d'un autoencodeur le pousse à découvrir et à exploiter les motifs présents dans les données.

La relation entre mémoire, perception et correspondance de motifs a été étudiée²²⁷ par William Chase et Herbert Simon dès le début des années 1970. Ils ont observé que les grands joueurs d'échecs étaient capables de mémoriser la position de toutes les pièces en regardant l'échiquier pendant cinq secondes seulement ; un défi que la plupart des gens trouveraient irréalisable. Cependant, ce n'était le cas que si les pièces se trouvaient dans des configurations réalistes (des parties réelles) et non lorsqu'elles étaient placées aléatoirement. Les joueurs d'échecs professionnels n'ont pas une meilleure mémoire que vous et moi, ils reconnaissent simplement des motifs de placement plus facilement en raison de leur expérience de ce jeu. Ils sont ainsi capables de stocker les informations plus efficacement.

À l'instar des joueurs d'échecs dans cette expérience sur la mémoire, un autoencodeur examine les entrées, les convertit en une représentation latente efficace et produit en sortie quelque chose qui (on l'espère) ressemble énormément à l'entrée. Un autoencodeur est toujours constitué de deux parties : un *encodeur* (ou *réseau de reconnaissance*) qui convertit les entrées en une représentation latente, suivi d'un *décodeur* (ou *réseau de génération*) qui convertit la représentation interne en sorties (voir la figure 9.1).

²²⁷ William G. Chase et Herbert A. Simon, « Perception in Chess », *Cognitive Psychology*, 4, n° 1 (1973), 55-81 : <https://homl.info/111>.

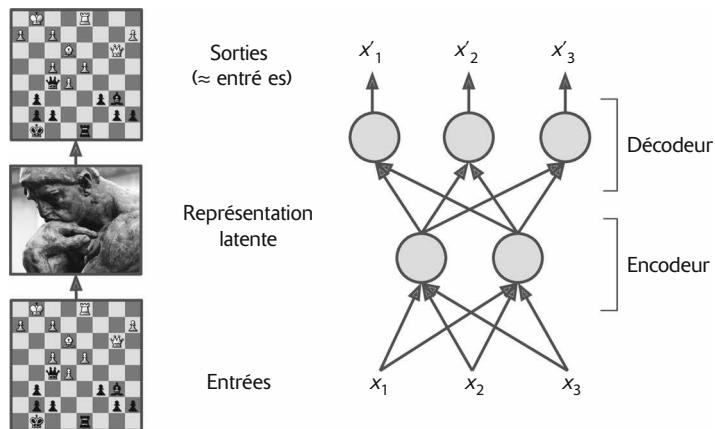


Figure 9.1 – Expérience de mémorisation dans les échecs (à gauche) et autoencodeur simple (à droite)

Un autoencodeur possède généralement la même architecture qu'un perceptron multicouche (PMC; voir le chapitre 2), mais le nombre de neurones de la couche de sortie doit être égal au nombre d'entrées. Dans l'exemple illustré, nous avons une seule couche cachée constituée de deux neurones (l'encodeur) et une couche de sortie constituée de trois neurones (le décodeur). Les sorties sont souvent appelées *reconstructions*, car l'autoencodeur tente de reconstruire les entrées, et la fonction de coût inclut une *perte de reconstruction* qui pénalise le modèle lorsque les reconstructions diffèrent des entrées.

Puisque la dimensionnalité de la représentation interne est inférieure à celle des données d'entrée (deux dimensions à la place de trois), on qualifie l'autoencodeur de *sous-complet*. Un autoencodeur sous-complet ne peut pas copier simplement ses entrées dans les codages, mais il doit pourtant trouver une façon de produire en sortie une copie de ses entrées. Il est obligé d'apprendre les caractéristiques les plus importantes des données d'entrée (et d'ignorer les moins importantes).

Voyons comment implémenter un autoencodeur sous-complet très simple pour la réduction de dimensionnalité.

9.2 ACP AVEC UN AUTOENCODEUR LINÉAIRE SOUS-COMPLET

Si l'autoencodeur utilise uniquement des activations linéaires et si la fonction de coût est l'erreur quadratique moyenne (MSE, Mean Squared Error), alors on peut montrer qu'il réalise une *analyse en composantes principales* (ACP)²²⁸.

228. Voir le chapitre 8 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

Le code suivant construit un autoencodeur linéaire simple pour effectuer une ACP sur un jeu de données à trois dimensions, en le projetant sur deux dimensions :

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

Ce code n'est pas très différent de tous les PMC que nous avons construits dans les chapitres précédents. Quelques remarques cependant :

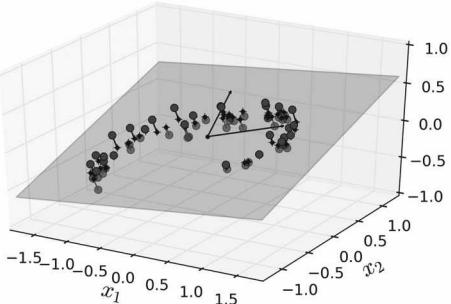
- Nous avons organisé l'autoencodeur en deux sous-composants : l'encodeur et le décodeur. Tous deux sont des modèles Sequential normaux, chacun avec une seule couche Dense. L'autoencodeur est un modèle Sequential qui contient l'encodeur suivi du décodeur (rappelez-vous qu'un modèle peut être utilisé en tant que couche dans un autre modèle).
- Le nombre de sorties de l'autoencodeur est égal au nombre d'entrées (c'est-à-dire trois).
- Pour effectuer une ACP simple, nous n'utilisons pas de fonction d'activation (autrement dit, tous les neurones sont linéaires) et la fonction de coût est la MSE. Nous verrons des autoencodeurs plus complexes ultérieurement.

Entraînons à présent le modèle sur un jeu de données généré simple à trois dimensions et utilisons-le pour encoder ce même jeu de données (c'est-à-dire en faire une projection sur deux dimensions) :

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Notez que le même jeu de données, X_{train} , est utilisé pour les entrées et pour les cibles. La figure 9.2 montre le jeu de données original à trois dimensions (à gauche) et la sortie de la couche cachée de l'autoencodeur (c'est-à-dire la couche de codage, à droite). Vous le constatez, l'autoencodeur a trouvé le meilleur plan à deux dimensions sur lequel projeter les données, tout en conservant autant de différence que possible dans les données (à l'instar de l'ACP).

Jeu de données d'origine en 3D



Projection 2D avec une variance maximale

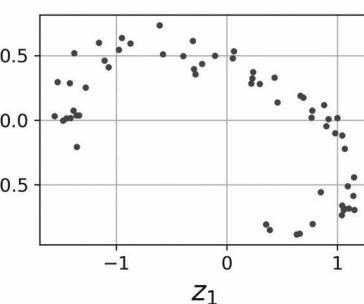


Figure 9.2 – ACP réalisée par un autoencodeur linéaire sous-complet



Vous pouvez voir les autoencodeurs comme une forme d'apprentissage autosupervisé (autrement dit, utilisant une technique d'apprentissage supervisé avec des étiquettes générées automatiquement, dans ce cas simplement égales aux entrées).

9.3 AUTOENCODEURS EMPILÉS

À l'instar des autres réseaux de neurones que nous avons présentés, les autoencodeurs peuvent comprendre plusieurs couches cachées. Dans ce cas, il s'agit d'*autoencodeurs empilés* (ou *autoencodeurs profonds*). En ajoutant des couches, l'autoencodeur devient capable d'apprendre des codages plus complexes. Il faut cependant faire attention à ne pas le rendre trop puissant. Imaginons un autoencodeur si puissant qu'il apprendrait à faire correspondre chaque entrée à un seul nombre arbitraire (et le décodeur apprendrait la correspondance inverse). Dans ce cas, l'autoencodeur reconstruirait parfaitement les données d'entraînement, mais il n'aurait appris aucune représentation utile des données (et il est peu probable que sa généralisation aux nouvelles instances serait bonne).

L'architecture d'un autoencodeur empilé est le plus souvent symétrique par rapport à la couche cachée centrale (la couche de codage). Plus simplement, il ressemble à un sandwich. Par exemple, un autoencodeur pour la classification MNIST²²⁹ pourrait avoir 784 entrées, une couche cachée de 100 neurones, une couche cachée centrale de 30 neurones, une autre couche cachée de 100 neurones, et une couche de sortie de 784 neurones. Un tel autoencodeur est représenté à la figure 9.3.

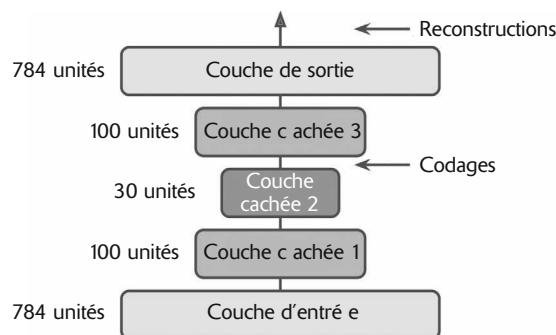


Figure 9.3 – Autoencodeur empilé

9.3.1 Implémenter un autoencodeur empilé avec Keras

La mise en œuvre d'un autoencodeur empilé est comparable à celle d'un PMC classique. On peut notamment appliquer les techniques pour l'entraînement des réseaux

229. Voir le chapitre 3 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

profonds décrites au chapitre 3. Par exemple, le code suivant construit un auto-encodeur empilé pour Fashion MNIST (chargé et normalisé comme au chapitre 2), en utilisant la fonction d'activation SELU :

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                    optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                          validation_data=[X_valid, X_valid])
```

Examinons ce code :

- Comme précédemment, nous divisons le modèle de l'autoencodeur en deux sous-modèles : l'encodeur et le décodeur.
- L'encodeur prend en entrée des images en niveaux de gris de 28×28 pixels, les aplatis afin de représenter chacune sous forme d'un vecteur de taille 784, puis passe les vecteurs obtenus au travers de deux couches Dense de taille décroissante (100 puis 30 unités), toutes deux utilisant la fonction d'activation SELU (vous pouvez également ajouter une initialisation normale de LeCun mais, le réseau n'étant pas très profond, la différence ne sera pas notable). Pour chaque image d'entrée, l'encodeur produit un vecteur de taille 30.
- Le décodeur prend les codages de taille 30 (générés par l'encodeur) et les passe au travers de deux couches Dense de taille croissante (100 puis 784 unités). Il remet les vecteurs finaux sous forme de tableaux 28×28 afin que ses sorties aient la même forme que les entrées de l'encodeur.
- Pour la compilation de l'autoencodeur empilé, nous remplaçons l'erreur quadratique moyenne par la perte d'entropie croisée binaire. Nous considérons la tâche de reconstruction comme un problème de classification binaire multi-étiquette : chaque intensité de pixels représente la probabilité que le pixel doit être noir. En abordant le problème de cette façon (plutôt qu'un problème de régression), le modèle a tendance à converger plus rapidement²³⁰.
- Enfin, nous entraînons le modèle en utilisant X_train à la fois pour les entrées et les cibles (de façon similaire, nous utilisons X_valid pour les entrées et les cibles de validation).

²³⁰. Vous pourriez être tenté d'utiliser l'indicateur de précision, mais il ne sera pas très efficace car il attend des étiquettes égales à 0 ou à 1 pour chaque pixel. Vous pouvez facilement contourner ce problème en créant un indicateur personnalisé qui calcule la précision après avoir arrondi la cible et les prédictions à 0 ou 1.

9.3.2 Visualiser les reconstructions

Pour s'assurer que l'entraînement d'un autoencodeur est correct, une solution consiste à comparer les entrées et les sorties : les différences doivent concerner des détails peu importants. Affichons quelques images du jeu de validation et leur reconstruction :

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

La figure 9.4 montre les images résultantes.



Figure 9.4 – Les images originales (en haut) et leur reconstruction (en bas)

Les reconstructions sont reconnaissables, mais la perte est un peu trop importante. Nous pourrions entraîner le modèle plus longtemps, rendre l'encodeur et le décodeur plus profonds, ou rendre les codages plus grands. Mais si le réseau devient trop puissant, il réalisera des reconstructions parfaites sans découvrir de motifs utiles dans les données. Pour le moment, conservons ce modèle.

9.3.3 Visualiser le jeu de données Fashion MNIST

À présent que nous disposons d'un autoencodeur empilé entraîné, nous pouvons nous en servir pour réduire la dimensionnalité du jeu de données. Dans le cadre de la visualisation, les résultats ne sont pas aussi bons que ceux obtenus avec d'autres algorithmes de réduction de la dimensionnalité²³¹, mais les autoencodeurs ont le grand

231. Comme ceux décrits au chapitre 8 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

avantage de pouvoir traiter des jeux de données volumineux, avec de nombreuses instances et caractéristiques. Une stratégie consiste donc à utiliser un autoencodeur pour réduire la dimensionnalité jusqu'à un niveau raisonnable, puis à employer un autre algorithme de réduction de la dimensionnalité pour la visualisation. Mettons en place cette stratégie pour visualiser le jeu de données Fashion MNIST. Nous commençons par utiliser l'encodeur de notre autoencodeur empilé de façon à abaisser la dimensionnalité jusqu'à 30, puis nous nous servons de l'implémentation Scikit-Learn de l'algorithme t-SNE pour la réduire jusqu'à 2 en vue d'un affichage :

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Nous pouvons alors tracer le jeu de données :

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

La figure 9.5 montre le diagramme de dispersion obtenu (décoré de quelques images). L'algorithme t-SNE a identifié plusieurs groupes, qui correspondent raisonnablement aux classes (chacune est représentée par une couleur différente).



Figure 9.5 – Visualisation de Fashion MNIST
à l'aide d'un autoencodeur suivi de l'algorithme t-SNE

Les autoencodeurs peuvent donc être employés pour la réduction de la dimensionnalité. Parmi leurs autres applications, le préentraînement non supervisé figure en bonne place.

9.3.4 Préentraînement non supervisé avec des autoencodeurs empilés

Comme nous l'avons vu au chapitre 3, si l'on s'attaque à une tâche supervisée complexe sans disposer d'un grand nombre de données d'entraînement étiquetées, une solution consiste à trouver un réseau de neurones qui effectue une tâche comparable et à réutiliser ses couches inférieures. Cela permet d'entraîner un modèle très performant avec peu de données d'entraînement, car votre réseau de neurones n'aura pas à apprendre toutes les caractéristiques de bas niveau. Il réutilisera simplement les détecteurs de caractéristiques appris par le réseau existant.

De manière comparable, si vous disposez d'un vaste jeu de données, mais dont la plupart ne sont pas étiquetées, vous pouvez commencer par entraîner un autoencodeur empilé avec toutes les données, réutiliser les couches inférieures pour créer un réseau de neurones dédié à votre tâche réelle et entraîner celui-ci à l'aide des données étiquetées. Par exemple, la figure 9.6 montre comment exploiter un autoencodeur empilé afin d'effectuer un préentraînement non supervisé pour un réseau de neurones de classification. Pour l'entraînement du classificateur, si vous avez vraiment peu de données d'entraînement étiquetées, vous pouvez figer les couches préentraînées (au moins les plus basses).

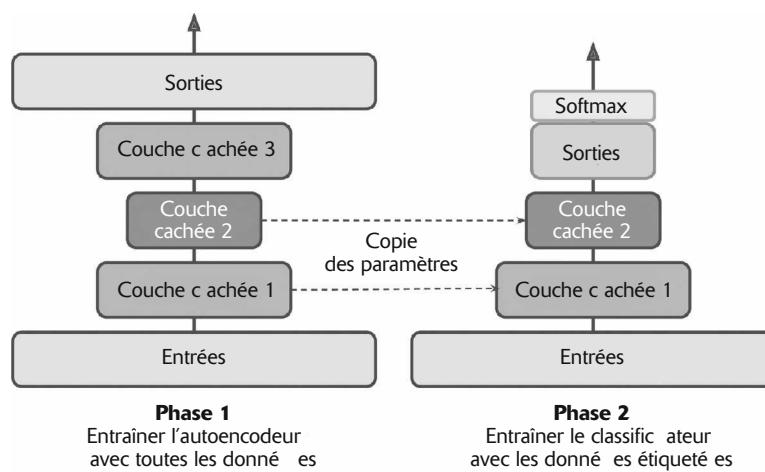


Figure 9.6 – Préentraînement non supervisé avec des autoencodeurs (FC: couche intégralement connectée)



Ce cas est en réalité assez fréquent, car la construction d'un vaste jeu de données non étiquetées est souvent peu coûteuse (par exemple, un simple script est en mesure de télécharger des millions d'images à partir d'Internet), alors que leur étiquetage fiable ne peut être réalisé que par des humains (par exemple, classer des images comme mignonnes ou non). Cette procédure étant longue et coûteuse, il est assez fréquent de n'avoir que quelques centaines d'instances étiquetées.

L'implémentation n'a rien de particulier. Il suffit d'entraîner un autoencodeur avec toutes les données d'entraînement (étiquetées et non étiquetées), puis de réutiliser les couches de l'encodeur pour créer un nouveau réseau de neurones (voir les exercices à la fin de ce chapitre).

Examinons à présent quelques techniques d'entraînement des autoencodeurs empilées.

9.3.5 Lier des poids

Lorsqu'un autoencodeur est parfaitement symétrique, comme celui que nous venons de construire, une technique répandue consiste à *lier* les poids des couches du décodeur aux poids des couches de l'encodeur. Cela permet de diviser par deux le nombre de poids dans le modèle, accélérant ainsi l'entraînement et limitant les risques de surajustement. Plus précisément, si l'autoencodeur comporte N couches (sans compter la couche d'entrée) et si \mathbf{W}_L représente les poids des connexions de la $L^{\text{ème}}$ couche (par exemple, la couche 1 est la première couche cachée, la couche $N/2$ est la couche de codage, et la couche N est la couche de sortie), alors les poids de la couche du décodeur peuvent être simplement définis par $\mathbf{W}_{N-L+1} = \mathbf{W}_L^T$ (avec $L = 1, 2, \dots, N/2$).

Pour lier des poids entre des couches à l'aide de Keras, nous définissons une couche personnalisée :

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

Cette couche personnalisée opère à la manière d'une couche `Dense` normale, mais elle utilise les poids d'une autre couche `Dense`, transposés (préciser `transpose_b=True` équivaut à transposer le second argument, mais l'approche choisie est plus efficace car la transposition est effectuée à la volée dans l'opération `matmul()`). Cependant, elle utilise son propre vecteur de termes constants. Nous pouvons ensuite construire un nouvel autoencodeur empilé, semblable au précédent, mais dont les couches `Dense` du décodeur sont liées aux couches `Dense` de l'encodeur :

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")
tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])
```

```

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])

```

Ce modèle permet d'obtenir une erreur de reconstruction légèrement plus faible que le modèle précédent, avec un nombre de paramètres presque divisé par deux.

9.3.6 Entrainer un autoencodeur à la fois

Au lieu d'entraîner l'intégralité de l'autoencodeur empilé en un seul coup, comme nous venons de le faire, il est souvent plus rapide d'entraîner un autoencodeur peu profond à la fois, puis de les empiler tous de façon à obtenir un seul autoencodeur empilé (d'où le nom) ; voir la figure 9.7. Cette technique est peu employée aujourd'hui, mais vous risquez de la rencontrer dans des articles qui parlent d'« entraînement glouton par couche » (*greedy layer-wise training*). Il est donc intéressant de savoir ce que cela signifie.

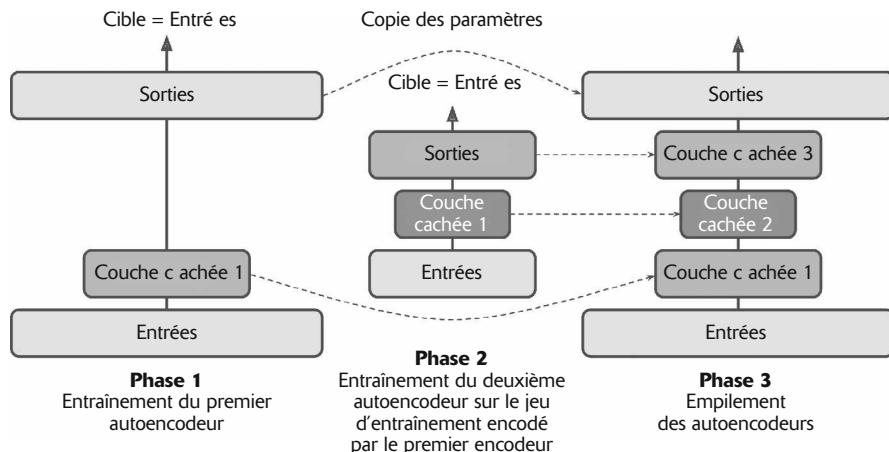


Figure 9.7 – Entraînement d'un autoencodeur à la fois

Au cours de la phase 1 de l'entraînement, le premier autoencodeur apprend à reconstruire les entrées. Puis nous encodons l'intégralité du jeu d'entraînement en utilisant ce premier autoencodeur, ce qui nous donne un nouveau jeu d'entraînement (compressé). Ensuite, au cours de la phase 2, nous entraînons un deuxième autoencodeur sur ce nouveau jeu de données. Pour finir, nous construisons un gros sandwich avec tous ces autoencodeurs (voir la figure 9.7), en commençant par empiler les couches cachées de chaque autoencodeur, puis les couches de sortie en ordre inverse. Nous obtenons alors l'autoencodeur empilé final (une implémentation est donnée dans la section « Training One Autoencoder at a Time » du notebook²³²).

232. Voir « 17_autoencoders_and_gans.ipynb » sur <https://github.com/ageron/handson-ml2>.

En procédant ainsi, nous pouvons facilement entraîner un plus grand nombre d'autoencodeurs, pour arriver à un autoencodeur empilé très profond.

Comme nous l'avons expliqué précédemment, le regain d'intérêt pour le Deep Learning est notamment dû à une découverte de Geoffrey Hinton *et al.* en 2006 (<https://homl.info/136>) : les réseaux de neurones profonds peuvent être préentraînés de façon non supervisée, en utilisant cette approche gloutonne par couche. Ils ont utilisé pour cela des machines de Boltzmann restreintes (RBM ; voir l'annexe C) mais, en 2007, Yoshua Bengio *et al.* ont montré²³³ que les autoencodeurs fonctionnaient également très bien. Pendant plusieurs années, il s'agissait de la seule méthode efficace d'entraîner des réseaux profonds, jusqu'à ce que nombre des techniques décrites au chapitre 3 rendent possible l'entraînement d'un réseau profond en une fois.

Les autoencodeurs ne sont pas limités aux réseaux denses. Vous pouvez également construire des autoencodeurs convolutifs, ou même des autoencodeurs récurrents. Voyons cela.

9.4 AUTOENCODEURS CONVOLUTIFS

Si vous manipulez des images, les autoencodeurs décrits jusqu'à présent ne fonctionneront pas très bien (sauf si les images sont très petites). Comme nous l'avons vu au chapitre 6, les réseaux de neurones convolutifs conviennent mieux au traitement des images que les réseaux denses. Si vous souhaitez construire un autoencodeur pour des images (par exemple, pour un préentraînement non supervisé ou une réduction de dimensionnalité), vous devez construire un autoencodeur convolutif²³⁴. L'encodeur est un CNN normal constitué de couches de convolution et de couches de pooling. Il réduit la dimensionnalité spatiale des entrées (c'est-à-dire la hauteur et la largeur), tout en augmentant la profondeur (c'est-à-dire le nombre de cartes de caractéristiques). Le décodeur doit réaliser l'inverse (augmenter la taille de l'image et réduire sa profondeur aux dimensions d'origine) et, pour cela, vous pouvez employer des couches de convolution transposée (vous pouvez également combiner des couches de suréchantillonnage à des couches de convolution).

Voici un autoencodeur convolutif simple pour Fashion MNIST :

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
```

233. Yoshua Bengio *et al.*, « Greedy Layer-Wise Training of Deep Networks », *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006), 153-160 : <https://homl.info/112>.

234. Jonathan Masci *et al.*, « Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction », *Proceedings of the 21st International Conference on Artificial Neural Networks*, 1 (2011), 52-59 : <https://homl.info/convae>.

```

conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2,
                                padding="valid",
                                activation="selu",
                                input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                                activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                                activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])

```

9.5 AUTOENCODEURS RÉCURRENTS

Si vous devez construire un autoencodeur pour des séquences, comme des séries chronologiques ou du texte (par exemple, pour un entraînement non supervisé ou une réduction de dimensionnalité), alors les réseaux de neurones récurrents (voir le chapitre 7) peuvent être mieux adaptés que les réseaux denses. La construction d'un *autoencodeur récurrent* est simple. L'encodeur est généralement un RNR série-vers-vecteur qui compresse la séquence d'entrée en un seul vecteur. Le décodeur est un RNR vecteur-vers-série qui effectue l'opération inverse :

```

recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])

```

Cet autoencodeur récurrent est capable de traiter des séries de n'importe quelle longueur, avec 28 dimensions par étape temporelle. Cela signifie qu'il peut s'occuper des images Fashion MNIST en considérant chacune comme une suite de lignes : à chaque étape temporelle, le RNR traitera une seule ligne de 28 pixels. Bien évidemment, vous pouvez employer un autoencodeur récurrent pour tout type de série. Notez que la première couche du décodeur est une couche `RepeatVector` afin que son vecteur d'entrée soit transmis au décodeur à chaque étape temporelle.

Revenons un petit peu en arrière. Nous avons vu différentes sortes d'autoencodeurs (de base, empilés, convolutifs et récurrents) et comment les entraîner (en une fois ou couche par couche). Nous avons également examiné deux applications : la visualisation de données et le préentraînement non supervisé.

Jusqu'à présent, pour forcer l'autoencodeur à apprendre des caractéristiques intéressantes, nous avons limité la taille de la couche de codage, le rendant sous-complet. Mais il est possible d'utiliser de nombreuses autres sortes de contraintes, y compris

autoriser la couche de codage à être aussi vaste que les entrées, voire plus vaste, donnant un *autoencodeur sur-complet*. Voyons certaines de ces approches.

9.6 AUTOENCODEURS DÉBRUITEURS

Pour obliger l'autoencodeur à apprendre des caractéristiques utiles, une autre approche consiste à rajouter du bruit sur ses entrées et à l'entraîner pour qu'il retrouve les entrées d'origine, sans le bruit. Cette idée date des années 1980 (par exemple, elle est mentionnée en 1987 dans la thèse de doctorat de Yann LeCun). Dans leur article²³⁵ publié en 2008, Pascal Vincent *et al.* ont montré que les autoencodeurs peuvent également servir à extraire des caractéristiques. Et, dans un article²³⁶ de 2010, Vincent *et al.* ont présenté les *autoencodeurs débruiteurs empilés*.

Le bruit peut être un bruit blanc gaussien ajouté aux entrées ou un blocage aléatoire des entrées, comme dans la technique du dropout (voir le chapitre 3). La figure 9.8 illustre ces deux possibilités.

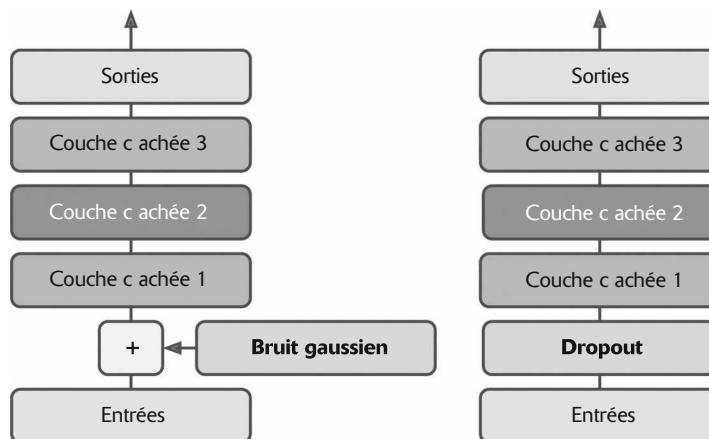


Figure 9.8 – Autoencodeur débruiteur, avec un bruit gaussien (à gauche) ou le dropout (à droite)

L'implémentation d'un autoencodeur débruiteur est simple. Il s'agit d'un autoencodeur empilé normal auquel on ajoute une couche Dropout appliquée aux entrées de l'encodeur (vous pouvez également utiliser une couche GaussianNoise). Rappelons que la couche Dropout est active uniquement pendant l'entraînement (il en va de même pour la couche GaussianNoise) :

235. Pascal Vincent *et al.*, « Extracting and Composing Robust Features with Denoising Autoencoders », *Proceedings of the 25th International Conference on Machine Learning* (2008), 1096-1103 : <https://homl.info/113>.

236. Pascal Vincent *et al.*, « Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion », *Journal of Machine Learning Research*, 11 (2010), 3371-3408 : <https://homl.info/114>.

```

dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])

```

La figure 9.9 montre quelques images avec du bruit (la moitié des pixels ont été désactivés) et les images reconstruites par l'autoencodeur débruiteur avec dropout. Vous remarquerez que l'autoencodeur a été capable de deviner des détails qui ne font pas partie de l'entrée, comme le col de la chemise blanche (ligne du bas, quatrième image). Ainsi, les autoencodeurs débruteurs peuvent non seulement être utilisés pour la visualisation de données et le préentraînement non supervisé, comme les autres autoencodeurs décrits jusqu'à présent, mais ils peuvent également être exploités assez simplement et efficacement pour retirer du bruit dans des images.

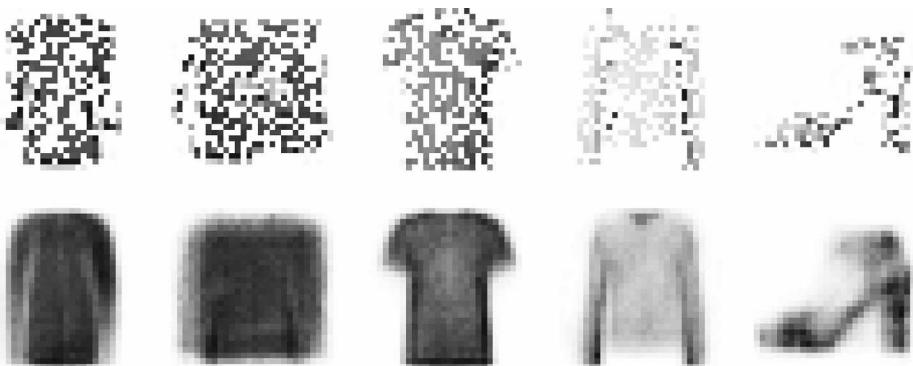


Figure 9.9 – Des images comportant du bruit (en haut) et leur reconstruction (en bas)

9.7 AUTOENCODEURS ÉPARS

La *dispersion* est un autre type de contrainte qui conduit souvent à une bonne extraction de caractéristiques. En ajoutant le terme approprié à la fonction de coût, l'autoencodeur est poussé à réduire le nombre de neurones actifs dans la couche de codage. Par exemple, il peut être incité à n'avoir en moyenne que 5 % de neurones hautement actifs. Il est ainsi obligé de représenter chaque entrée comme une combinaison d'un petit nombre d'activations. En conséquence, chaque neurone de la couche de codage finit généralement par représenter une caractéristique utile (si l'on ne pouvait prononcer que quelques mots chaque mois, on ne choisirait que les plus importants).

Une approche simple consiste à utiliser la fonction d'activation sigmoïde dans la couche de codage (pour contraindre les codages à des valeurs situées entre 0 et 1), à utiliser une couche de codage large (par exemple, avec 300 unités) et à ajouter une régularisation ℓ_1 aux activations de la couche de codage (le décodeur n'est qu'un décodeur normal) :

```
sparse_11_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_11_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_11_ae = keras.models.Sequential([sparse_11_encoder, sparse_11_decoder])
```

Cette couche ActivityRegularization se contente de retourner ses entrées, mais elle ajoute une perte d'entraînement égale à la somme des valeurs absolues de ses entrées (elle n'agit que pendant l'entraînement). De manière équivalente, vous pouvez retirer la couche ActivityRegularization et fixer activity_regularizer à keras.regularizers.l1(1e-3) dans la couche précédente. Cette pénalité encouragera le réseau de neurones à produire des codages proches de 0, mais, puisqu'il sera également pénalisé s'il ne parvient pas à reconstruire correctement les entrées, il devra sortir au moins quelques valeurs différentes de zéro. En utilisant la norme ℓ_1 plutôt que la norme ℓ_2 , nous incitons le réseau de neurones à préserver les codages les plus importants tout en éliminant ceux qui ne sont pas indispensables pour l'image d'entrée (plutôt que de réduire simplement tous les codages).

Une autre approche, qui conduit souvent à de meilleurs résultats, consiste à mesurer la dispersion actuelle de la couche de codage lors de chaque itération d'entraînement et à pénaliser le modèle lorsque la dispersion mesurée diffère d'une dispersion cible. Pour cela, on calcule l'activation moyenne de chaque neurone de la couche de codage, sur l'intégralité du lot d'entraînement. La taille du lot ne doit pas être trop faible pour que la moyenne puisse être suffisamment précise.

Après avoir obtenu l'activation moyenne par neurone, on pénalise ceux qui sont trop actifs, ou pas assez, en ajoutant à la fonction de coût une *perte de dispersion (sparsity loss)*. Par exemple, si l'on détermine que l'activation moyenne d'un neurone est de 0,3 alors que la dispersion cible est de 0,1, il faut le pénaliser pour réduire son degré d'activité. Une solution serait d'ajouter simplement l'erreur quadratique $(0,3 - 0,1)^2$ à la fonction de coût, mais, en pratique, une meilleure approche consiste à utiliser la divergence de Kullback-Leibler (décrise brièvement au chapitre 1). En effet, elle a des gradients beaucoup plus forts que l'erreur quadratique moyenne, comme le montre la figure 9.10.

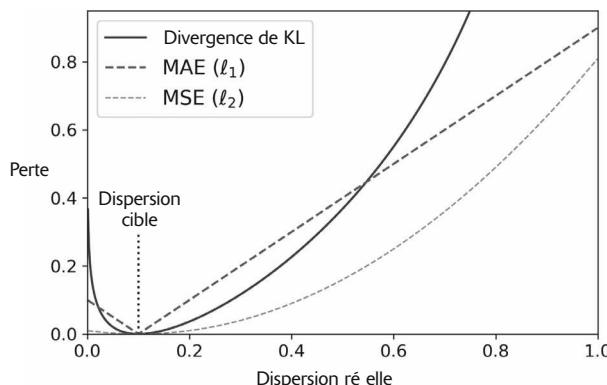


Figure 9.10 – Perte de dispersion

Étant donné deux distributions de probabilité discrètes P et Q , la divergence de KL entre ces distributions, notée $D_{\text{KL}}(P \mid\mid Q)$, peut être calculée à l'aide de l'équation 9.1.

Équation 9.1 – Divergence de Kullback-Leibler

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

Dans notre cas, nous voulons mesurer la divergence entre la probabilité cible p qu'un neurone de la couche de codage s'activera et la probabilité réelle q (c'est-à-dire l'activation moyenne sur le lot d'entraînement). Nous pouvons simplifier la divergence de KL et obtenir l'équation 9.2.

Équation 9.2 – Divergence de KL entre la dispersion cible p et la dispersion réelle q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Après avoir calculé la perte de dispersion pour chaque neurone de la couche de codage, il suffit d'additionner ces pertes et d'ajouter le résultat à la fonction de coût. Pour contrôler l'importance relative de la perte de dispersion et de la perte de reconstruction, on multiplie la première par un hyperparamètre de poids de dispersion. Si ce poids est trop élevé, le modèle restera proche de la dispersion cible, mais il risquera de ne pas reconstruire correctement les entrées et donc d'être inutile. À l'inverse, s'il est trop faible, le modèle ignorera en grande partie l'objectif de dispersion et n'apprendra aucune caractéristique intéressante.

Nous disposons à présent des éléments nécessaires à l'implémentation d'un autoencodeur épars fondé sur la divergence de KL. Commençons par créer un régularisateur personnalisé de façon à appliquer une régularisation par divergence de KL :

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
```

```

def __init__(self, weight, target=0.1):
    self.weight = weight
    self.target = target
def __call__(self, inputs):
    mean_activities = K.mean(inputs, axis=0)
    return self.weight * (
        kl_divergence(self.target, mean_activities) +
        kl_divergence(1. - self.target, 1. - mean_activities))

```

Nous pouvons maintenant construire l'autoencodeur épars, en utilisant le KLDivergence Regularizer pour les activations de la couche de codage :

```

kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])

```

Au terme de l'entraînement de cet autoencodeur épars sur Fashion MNIST, les activations des neurones dans la couche de codage sont pour la plupart proches de 0 (environ 70 % de toutes les activations sont inférieures à 0,1) et tous les neurones ont une activation moyenne autour de 0,1 (près de 90 % des neurones ont une activation moyenne entre 0,1 et 0,2). La figure 9.11 illustre tout cela.

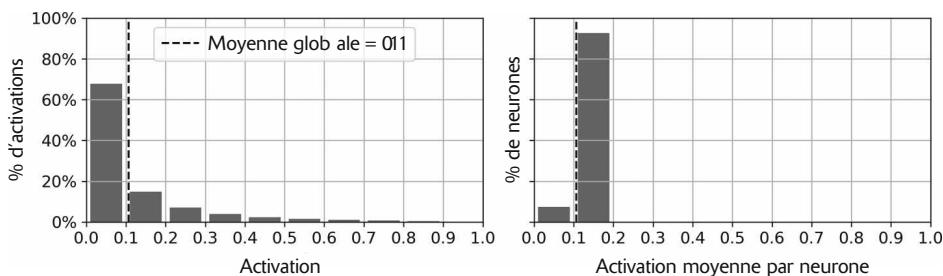


Figure 9.11 – Distribution de toutes les activations dans la couche de codage (à gauche) et distribution de l'activation moyenne par neurone (à droite)

9.8 AUTOENCODEURS VARIATIONNELS

Une autre catégorie importante d'autoencodeurs a été proposée en 2014 par Diederik Kingma et Max Welling et elle est rapidement devenue l'un des types d'autoencodeurs les plus populaires : les *autoencodeurs variationnels*²³⁷.

Ils sont relativement différents de tous les autres autoencodeurs décrits jusqu'à présent, notamment sur les points suivants :

- Il s'agit d'*autoencodeurs probabilistes*, ce qui signifie que leurs sorties sont partiellement déterminées par le hasard, même après l'entraînement (à l'opposé des autoencodeurs débruiteurs, qui n'utilisent le hasard que pendant l'entraînement).
- Plus important encore, il s'agit d'*autoencodeurs génératifs*, c'est-à-dire capables de générer de nouvelles instances qui semblent provenir du jeu d'entraînement.

Ces deux propriétés les rendent comparables aux RBM, mais ils sont plus faciles à entraîner et le processus d'échantillonnage est plus rapide (avec les RBM, il faut attendre que le réseau se stabilise dans un « équilibre thermique » avant de pouvoir échantillonner une nouvelle instance). Comme leur nom le suggère, les autoencodeurs variationnels effectuent une inférence bayésienne variationnelle²³⁸, une manière efficace de réaliser une inférence bayésienne approchée.

Étudions leur fonctionnement. La figure 9.12 (en partie gauche) montre un autoencodeur variationnel. On peut reconnaître la structure de base de tous les autoencodeurs, avec un encodeur suivi d'un décodeur (dans cet exemple, ils ont chacun deux couches cachées), mais on constate également un petit changement. Au lieu de produire directement un codage pour une entrée donnée, l'encodeur produit un codage moyen μ et un écart-type σ . Le codage réel est ensuite échantillonné aléatoirement avec une loi normale de moyenne μ et d'écart-type σ . Ensuite, le décodeur décode normalement le codage échantillonné. La partie droite de la figure montre une instance d'entraînement qui passe par cet autoencodeur. Tout d'abord, l'encodeur produit μ et σ , puis un codage est pris aléatoirement (remarquons qu'il ne se trouve pas exactement à μ). Enfin, ce codage est décodé et la sortie finale ressemble à l'instance d'entraînement.

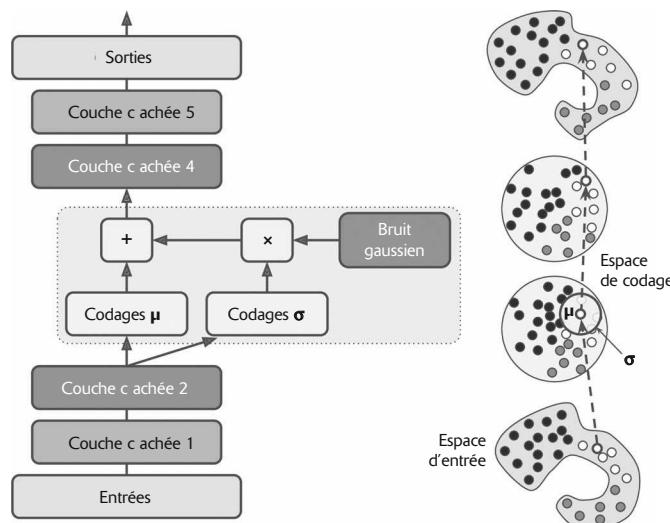


Figure 9.12 – Autoencodeur variationnel (à gauche) traitant une instance (à droite)

238. Voir le chapitre 9 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

Vous le voyez sur la figure, même si les entrées peuvent avoir une distribution très complexe, un autoencodeur variationnel a tendance à produire des codages qui semblent avoir été échantillonnés à partir d'une distribution normale simple²³⁹. Au cours de l'entraînement, la fonction de coût (voir ci-après) pousse les codages à migrer progressivement vers l'espace de codage (également appelé *espace latent*) pour occuper une région semblable à un nuage de points gaussien. En conséquence, après l'entraînement d'un autoencodeur variationnel, vous pouvez très facilement générer une nouvelle instance : il suffit de prendre un codage aléatoirement à partir de la distribution normale et de le décoder.

Passons à présent à la fonction de coût, qui comprend deux parties. La première est la perte de reconstruction habituelle, qui pousse l'autoencodeur à reproduire ses entrées (on peut opter pour l'entropie croisée, comme expliqué précédemment). La seconde est la *perte latente*, qui pousse l'autoencodeur à produire des codages semblant avoir été échantillonnés à partir d'une simple distribution normale, pour laquelle on utilisera la divergence de KL entre la distribution cible (la loi normale) et la distribution réelle des codages. Les aspects mathématiques sont légèrement plus complexes qu'avec les autoencodeurs épars, notamment en raison du bruit gaussien, qui limite la quantité d'informations pouvant être transmises à la couche de codage (poussant ainsi l'autoencodeur à apprendre des caractéristiques utiles). Heureusement, les équations se simplifient et la perte latente peut être calculée assez simplement avec l'équation 9.3²⁴⁰ :

Équation 9.3 – Perte latente d'un autoencodeur variationnel

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K [1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2]$$

Dans cette équation, \mathcal{L} correspond à la perte latente, n est la dimensionnalité des codages, et μ_i et σ_i sont la moyenne et l'écart-type du $i^{\text{ème}}$ composant des codages. Les vecteurs $\boldsymbol{\mu}$ et $\boldsymbol{\sigma}$ (qui contiennent tous les μ_i et σ_i) sont produits par l'encodeur, comme le montre la figure 9.12 (partie gauche).

Une variante répandue de l'architecture de l'autoencodeur variationnel consiste à entraîner l'encodeur pour produire $\gamma = \log(\sigma^2)$ à la place de σ . La perte latente peut alors être calculée conformément à l'équation 9.4. Cette approche est numériquement plus stable et accélère l'entraînement.

Équation 9.4 – Perte latente d'un autoencodeur variationnel, réécrite en utilisant $\gamma = \log(\sigma^2)$

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K [1 + \gamma_i - \exp(\gamma_i) - \mu_i^2]$$

Commençons à construire un autoencodeur variationnel pour Fashion MNIST (comme représenté à la figure 9.12, mais en utilisant la variante γ). Tout d'abord,

239. Les autoencodeurs variationnels sont en réalité plus généraux ; les codages ne se limitent pas à une distribution normale.

240. Tous les détails mathématiques se trouvent dans l'article d'origine sur les autoencodeurs variationnels et dans le superbe tutoriel écrit en 2016 par Carl Doersch (<https://homl.info/116>).

nous créons une couche personnalisée pour échantillonner les codages à partir de μ et γ :

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

Cette couche `Sampling` prend deux entrées: `mean` (μ) et `log_var` (γ). Elle utilise la fonction `K.random_normal()` pour échantillonner un vecteur aléatoire (de la même forme que γ) à partir de la distribution normale, avec la moyenne 0 et l'écart-type 1. Elle multiplie ensuite ce vecteur par $\exp(\gamma/2)$ (qui est égal à σ , comme vous pouvez le vérifier), et termine en ajoutant μ et en retournant le résultat. La couche échantillonne un vecteur de codages à partir de la distribution normale avec la moyenne μ et l'écart-type σ .

Nous pouvons à présent créer l'encodeur, en utilisant l'API Functional, car le modèle n'est pas intégralement séquentiel:

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z) # μ
codings_log_var = keras.layers.Dense(codings_size)(z) # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Notez que les couches `Dense` qui produisent `codings_mean` (μ) et `codings_log_var` (γ) ont les mêmes entrées (c'est-à-dire les sorties de la deuxième couche `Dense`). Nous pouvons ensuite passer `codings_mean` et `codings_log_var` à la couche `Sampling`. Enfin, le modèle `variational_encoder` possède trois sorties, pour le cas où vous souhaiteriez inspecter les valeurs de `codings_mean` et de `codings_log_var`. Nous utiliserons uniquement la dernière sortie (`codings`). Construisons à présent le décodeur:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

Dans ce cas, nous aurions pu utiliser l'API Sequential à la place de l'API Functional, car le décodeur n'est qu'une simple pile de couches, virtuellement identique à bon nombre d'autres décodeurs construits jusqu'à présent. Nous terminons par la mise en place du modèle d'autoencodeur variationnel:

```
_ , codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])
```

Nous ignorons les deux premières sorties du décodeur (nous souhaitons simplement fournir les codages au décodeur). Enfin, nous devons ajouter la perte latente et la perte de reconstruction :

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Nous commençons par appliquer l'équation 9.4 de façon à calculer la perte latente pour chaque instance du lot (l'addition se fait sur le dernier axe). Puis nous calculons la perte moyenne sur toutes les instances du lot et divisons le résultat par 784 afin de lui donner l'échelle appropriée par rapport à la perte de reconstruction. La perte de reconstruction de l'autoencodeur variationnel est supposée correspondre à la somme des erreurs de reconstruction des pixels. Mais, lorsque Keras calcule la perte "binary_crossentropy", il calcule la moyenne non pas sur la somme mais sur l'ensemble des 784 pixels. La perte de reconstruction est donc 784 fois plus petite que celle dont nous avons besoin. Nous pourrions définir une perte personnalisée pour calculer la somme plutôt que la moyenne, mais il est plus simple de diviser la perte latente par 784 (la perte finale sera 784 fois plus petite qu'elle ne le devrait, mais cela signifie simplement que nous devons utiliser un taux d'apprentissage plus important).

Notez que nous utilisons l'optimiseur RMSprop, qui convient parfaitement dans ce cas. Pour finir, nous pouvons entraîner l'autoencodeur !

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

9.8.1 Générer des images Fashion MNIST

Utilisons cet autoencodeur variationnel pour générer des images qui ressemblent à des articles de mode. Il suffit de prendre aléatoirement des codages selon une loi normale et de les décoder.

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

La figure 9.13 montre les 12 images obtenues.



Figure 9.13 – Images Fashion MNIST générées par l'autoencodeur variationnel

Même si elles sont un peu trop floues, la plupart de ces images restent plutôt convaincantes. Pour les autres, le résultat n'est pas extraordinaire, mais ne soyez pas trop dur avec l'autoencodeur, car il n'a eu que quelques minutes pour apprendre ! Avec des réglages plus fins et un entraînement plus long, ces images seraient bien meilleures.

Grâce aux autoencodeurs variationnels, il est possible d'effectuer une *interpolation sémantique*. Au lieu d'interpoler deux images au niveau du pixel (ce qui équivaudrait à une superposition des deux images), nous pouvons effectuer l'interpolation au niveau des codages. Nous commençons par soumettre les deux images à l'encodeur, puis nous interpolons les deux codages obtenus. Nous terminons en décodant les codages interpolés de façon à obtenir l'image finale. Elle ressemblera à une image Fashion MNIST, mais correspondra à un intermédiaire entre les deux images corrigées. Dans l'exemple de code suivant, nous prenons les douze codages que nous venons de générer, nous les organisons en une grille 3×4 , et nous utilisons la fonction `tf.image.resize()` de TensorFlow pour redimensionner cette grille au format 5×7 . Par défaut, la fonction `resize()` effectue une interpolation bilinéaire. Par conséquent, toutes les lignes et colonnes supplémentaires contiendront des codages interpolés. Nous utilisons ensuite le décodeur pour produire les images :

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

La figure 9.14 montre les images obtenues. Les images d'origine sont encadrées, les autres sont le résultat de l'interpolation sémantique entre les images voisines. Notez, par exemple, que la chaussure située en quatrième ligne et cinquième colonne est une bonne interpolation des deux modèles qui se trouvent au-dessus et en dessous.

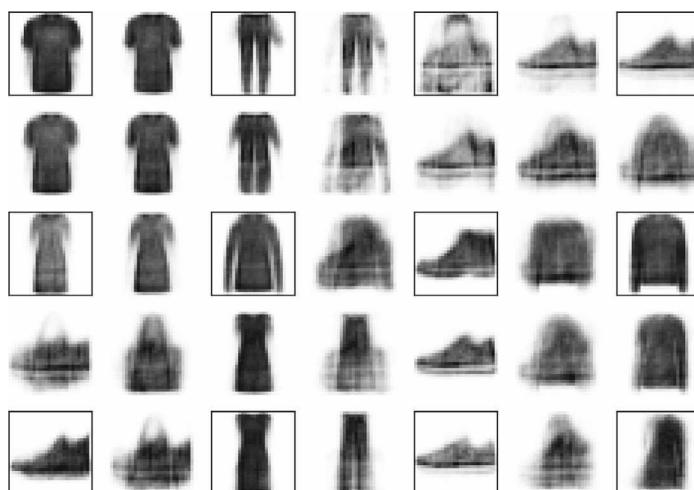


Figure 9.14 – Interpolation sémantique

Les autoencodeurs variationnels ont été assez populaires pendant plusieurs années, mais les GAN ont fini par prendre la tête, notamment parce qu'ils sont capables de

générer des images beaucoup plus nettes et réalistes. Focalisons-nous à présent sur les GAN.

9.9 RÉSEAUX ANTAGONISTES GÉNÉRATIFS (GAN)

Les réseaux antagonistes génératifs ont été proposés en 2014 dans un article²⁴¹ rédigé par Ian Goodfellow *et al.* Même si l'idée a presque instantanément intéressé les chercheurs, il aura fallu quelques années pour surmonter certaines des difficultés de l'entraînement des GAN. À l'instar de nombreuses grandes idées, elle semble simple après coup : faire en sorte que des réseaux de neurones soient mis en concurrence les uns contre les autres en espérant que ce duel les pousse à exceller. Comme le montre la figure 9.15, un GAN est constitué de deux réseaux de neurones :

- *Un générateur*

Il prend en entrée une distribution aléatoire (typiquement gaussienne) et produit en sortie des données – généralement une image. Vous pouvez considérer les entrées aléatoires comme les représentations latentes (c'est-à-dire les codages) de l'image qui doit être générée. Vous le comprenez, le générateur joue un rôle comparable au décodeur dans un autoencodeur variationnel et peut être employé de la même manière pour générer de nouvelles images (il suffit de lui fournir un bruit gaussien pour qu'il produise une toute nouvelle image). En revanche, son entraînement est très différent, comme nous le verrons plus loin.

- *Un discriminateur*

Il prend en entrée soit une image factice provenant du générateur, soit une image réelle provenant du jeu d'entraînement, et doit deviner si cette image d'entrée est fausse ou réelle.

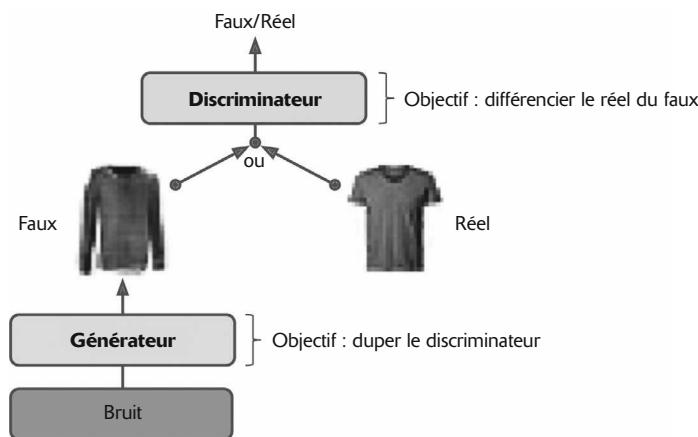


Figure 9.15 – Réseau antagoniste génératif

241. Ian Goodfellow *et al.*, « Generative Adversarial Nets », *Proceedings of the 27th International Conference on Neural Information Processing Systems*, 2 (2014), 2672-2680 : <https://homl.info/gan>.

Pendant l'entraînement, le générateur et le discriminateur ont des objectifs opposés. Le discriminateur tente de distinguer les images factices des images réelles, tandis que le générateur tente de produire des images suffisamment réelles pour tromper le discriminateur. Puisque le GAN est constitué de deux réseaux aux objectifs différents, il ne peut pas être entraîné à la manière d'un réseau de neurones normal. Chaque itération d'entraînement comprend deux phases :

- Au cours de la première phase, nous entraînons le discriminateur. Un lot d'images réelles est échantillonné à partir du jeu entraînement, auquel est ajouté un nombre égal d'images factices produites par le générateur. Les étiquettes sont fixées à 0 pour les images factices et à 1 pour les images réelles. Le discriminateur est entraîné sur ce lot étiqueté pendant une étape, en utilisant une perte d'entropie croisée binaire. Il est important de noter que, au cours de cette étape, la rétropropagation optimise uniquement les poids du discriminateur.
- Au cours de la seconde phase, nous entraînons le générateur. Nous l'utilisons tout d'abord pour produire un autre lot d'images factices et, une fois encore, le discriminateur doit dire si les images sont fausses ou réelles. Cette fois-ci, nous n'ajoutons aucune image réelle au lot et toutes les étiquettes sont fixées à 1 (réelles). Autrement dit, nous voulons que le générateur produise des images que le discriminateur considérera (à tort) réelles ! Il est indispensable que les poids du discriminateur soient figés au cours de cette étape, de sorte que la rétropropagation affecte uniquement les poids du générateur.



Le générateur ne voit jamais d'images réelles et pourtant il apprend progressivement à produire de fausses images convaincantes ! Il reçoit uniquement les gradients qui reviennent du discriminateur. Heureusement, plus le discriminateur est bon, plus les informations sur les images réelles contenues dans ces gradients de seconde main sont pertinentes. Le générateur peut donc réaliser des progrès significatifs.

Construisons un GAN simple pour Fashion MNIST.

Nous devons tout d'abord construire le générateur et le discriminateur. Le générateur est comparable au décodeur d'un autoencodeur, tandis que le discriminateur est un classificateur binaire normal (il prend en entrée une image et se termine par une couche Dense qui contient une seule unité et utilise la fonction d'activation sigmoïde). Pour la seconde phase de chaque itération d'entraînement, nous avons également besoin du modèle GAN complet constitué du générateur suivi du discriminateur :

```

codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

```

```

discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])

```

Nous devons à présent compiler ces modèles. Puisque le discriminateur est un classificateur binaire, nous pouvons naturellement utiliser la perte d'entropie croisée binaire. Le générateur n'étant entraîné qu'au travers du modèle gan, il est donc inutile de le compiler. Le modèle gan est également un classificateur binaire et peut donc utiliser la perte d'entropie croisée binaire. Le discriminateur ne devant pas être entraîné au cours de la seconde phase, nous le rendons non entraînable avant de compiler le modèle gan :

```

discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")

```



Keras tient compte de l'attribut `trainable` uniquement au moment de la compilation d'un modèle. Par conséquent, après avoir exécuté ce code, le `discriminator` est entraînable si nous invoquons sa méthode `fit()` ou sa méthode `train_on_batch()` (que nous utiliserons), tandis qu'il n'est *pas* entraînable lorsque nous invoquons ces méthodes sur le modèle `gan`.

Puisque la boucle d'entraînement est inhabituelle, nous ne pouvons pas utiliser la méthode `fit()` normale. À la place, nous écrivons une boucle d'entraînement personnalisée. Pour cela, nous devons tout d'abord créer un `Dataset` de façon à parcourir les images :

```

batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)

```

Nous sommes alors prêts à écrire la boucle d'entraînement, que nous plaçons dans une fonction `train_gan()` :

```

def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # Phase 1 - entraînement du discriminateur
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # Phase 2 - entraînement du générateur
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)

    train_gan(gan, dataset, batch_size, codings_size)

```

Comme nous l'avons décrit précédemment, vous pouvez constater les deux phases de chaque itération :

- Dans la phase 1, nous fournissons au générateur un bruit gaussien afin qu'il produise des images factices et nous complétons ce lot d'images par autant d'images réelles. Les cibles y_1 sont fixées à 0 pour les fausses images et à 1 pour les images réelles. Ensuite, nous entraînons le discriminateur sur ce lot. Notez que nous fixons à `True` l'attribut `trainable` du discriminateur. Cela permet simplement d'éviter que Keras n'affiche un avertissement lorsqu'il remarque que `trainable` vaut à présent `False` alors qu'il valait `True` à la compilation du modèle (ou inversement).
- Dans la phase 2, nous fournissons au GAN du bruit gaussien. Son générateur commence à produire des images factices, puis le discriminateur tente de deviner si des images sont fausses ou réelles. Puisque nous voulons que le discriminateur pense que les images fausses sont réelles, nous fixons les cibles y_2 à 1. Notez que l'attribut `trainable` est mis à `False`, de nouveau pour éviter un avertissement.

Voilà tout ! Si vous affichez les images générées (voir la figure 9.16), vous verrez qu'au bout de la première époque, elles commencent déjà à ressembler à des images Fashion MNIST (mais avec beaucoup de bruit).



Figure 9.16 – Images générées par le GAN après une époque d'entraînement

Malheureusement, les images ne vont jamais réellement s'améliorer et vous pourrez même trouver des époques où le GAN semble oublier ce qu'il a appris. Quelle en est la raison ? En réalité, l'entraînement d'un GAN peut se révéler un véritable défi. Voyons pourquoi.

9.9.1 Les difficultés de l'entraînement des GAN

Pendant l'entraînement, le générateur et le discriminateur tentent constamment de se montrer plus malin que l'autre, dans un jeu à somme nulle. Alors que l'entraînement progresse, le jeu peut arriver dans un état que les théoriciens du jeu appellent *équilibre de Nash* (du nom du mathématicien John Nash). Cela se produit lorsque aucun joueur n'a intérêt à changer sa stratégie, en supposant que les autres joueurs ne changent pas la leur.

Par exemple, un équilibre de Nash est atteint lorsque tout le monde conduit du côté droit de la route : aucun conducteur n'a intérêt à être le seul à changer de côté. Bien entendu, il existe un second équilibre de Nash possible : lorsque tout le monde conduit du côté *gauche* de la route. En fonction de l'état initial et de la dynamique, l'équilibre atteint peut être l'un ou l'autre. Dans cet exemple, il n'existe qu'une seule stratégie optimale lorsqu'un équilibre est atteint (c'est-à-dire rouler du même côté que tout le monde), mais un équilibre de Nash peut impliquer de multiples stratégies concurrentes (par exemple, un prédateur chasse sa proie, la proie tente de s'échapper et aucun d'eux n'a intérêt à changer de stratégie).

En quoi cela concerne-t-il les GAN ? Les auteurs de l'article ont démontré qu'un GAN ne peut atteindre qu'un seul équilibre de Nash : lorsque le générateur produit des images parfaitement réalistes et que le discriminateur est alors obligé de deviner (50 % réelles, 50 % factices). Cette conclusion est très encourageante, car il semblerait qu'il suffise donc d'entraîner le GAN pendant un temps suffisamment long pour qu'il finisse par atteindre cet équilibre, nous donnant le générateur parfait. Malheureusement, ce n'est pas si simple. Rien ne garantit que cet équilibre sera un jour atteint.

La plus grande difficulté se nomme *mode collapse* : il s'agit du moment où les sorties du générateur deviennent progressivement de moins en moins variées. Comment cela peut-il se produire ? Supposons que le générateur parvienne de mieux en mieux à produire des chaussures convaincantes, plus que toute autre classe. Il trompera un peu mieux le discriminateur avec les chaussures, ce qui l'encouragera à générer encore plus d'images de chaussures. Progressivement, il oubliera comment produire d'autres images. Dans le même temps, les seules images factices que verra le discriminateur représenteront des chaussures, et il oubliera progressivement comment identifier les fausses images d'autres classes. À terme, lorsque le discriminateur réussira à différencier les fausses chaussures des vraies, le générateur sera obligé de passer à une autre classe. Il pourra alors devenir performant sur les chemises, par exemple, oubliant tout des chaussures, et le discriminateur fera de même. Le GAN pourrait ainsi passer par plusieurs classes, sans jamais devenir bon dans aucune d'entre elles.

Par ailleurs, puisque le générateur et le discriminateur luttent constamment l'un contre l'autre, leurs paramètres peuvent finir par osciller et devenir instables. L'entraînement peut débuter correctement, puis diverger soudainement sans cause apparente, en raison de ces instabilités. Et, puisque de nombreux facteurs affectent ces mouvements complexes, les GAN sont très sensibles aux divers paramètres. Vous devrez peut-être consacrer beaucoup d'efforts à les ajuster.

Ces problèmes ont largement occupé les chercheurs depuis 2014. De nombreux articles ont été écrits sur ce sujet, certains proposant de nouvelles fonctions de coût²⁴² (même si un article²⁴³ publié en 2018 par des chercheurs de Google remet en question leur efficacité) ou des techniques permettant de stabiliser l'entraînement ou d'éviter le problème de *mode collapse*. Par exemple, une technique répandue nommée *rejeu d'expériences (experience replay)* consiste à stocker dans un tampon de rejeu les images produites par le générateur à chaque itération (les images plus anciennes sont retirées au fur et à mesure) et à entraîner le discriminateur en employant des images réelles et des images factices extraites de ce tampon (plutôt que seulement des images factices produites par le générateur actuel). Cela réduit les risques que le discriminateur surajuste les dernières sorties du générateur.

Une autre technique fréquente se nomme *discrimination par mini-lots (mini-batch discrimination)*. Elle mesure la similitude des images sur le lot et fournit cette information au discriminateur, qui peut alors facilement rejeter tout un lot d'images factices qui manquent de diversité. Cela encourage le générateur à produire une grande variété d'images, réduisant les risques de *mode collapse*. D'autres articles proposent simplement des architectures spécifiques qui montrent de bonnes performances.

En résumé, ce domaine de recherche est encore très actif et la dynamique des GAN n'est pas encore parfaitement comprise. Néanmoins, les avancées existent et certains résultats sont véritablement époustouflants ! Examinons à présent certaines des architectures les plus abouties, en commençant par les GAN convolutifs profonds, qui représentaient encore l'état de l'art il y a quelques années. Ensuite, nous étudierons deux architectures plus récentes (et plus complexes).

9.9.2 GAN convolutifs profonds

Dans l'article original de 2014, les GAN se fondaient sur des couches de convolution mais ne tentaient de générer que de petites images. Peu après, de nombreux chercheurs ont essayé de construire des GAN fondés sur des réseaux convolutifs plus profonds afin de produire des images de plus grande taille. Cela s'est révélé délicat car l'entraînement était très instable. Cependant, fin 2015, Alec Radford *et al.* ont fini par réussir, après avoir mené de nombreuses expériences avec diverses architectures et différents hyperparamètres. Ils ont nommé leur architecture *GAN convolutif profond (DCGAN, Deep Convolutional GAN)*²⁴⁴. Voici leurs principales propositions pour mettre en place des GAN convolutifs stables :

- Remplacer les couches de pooling par des convolutions à pas (dans le discriminateur) et par des convolutions transposées (dans le générateur).

242. Le projet GitHub mené par Hwalsuk Lee (<https://homl.info/ganloss>) propose une bonne comparaison des principales pertes des GAN.

243. Mario Lucic *et al.*, « Are GANs Created Equal? A Large-Scale Study », *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018), 698-707 : <https://homl.info/gansequall>.

244. Alec Radford *et al.*, « Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks » (2015) : <https://homl.info/dcgan>.

- Utiliser une normalisation par lots dans le générateur et dans le discriminateur, excepté dans la couche de sortie du générateur et dans la couche d'entrée du discriminateur.
- Retirer les couches cachées intégralement connectées pour les architectures plus profondes.
- Dans le générateur, choisir l'activation ReLU pour toutes les couches excepté la couche de sortie, qui doit opter pour la fonction tanh.
- Dans le discriminateur, choisir l'activation Leaky ReLU pour toutes les couches.

Ces recommandations se révéleront pertinentes dans de nombreux cas, mais pas tous. Vous devrez continuer à faire des essais avec différents hyperparamètres (en réalité, le simple fait de changer le germe aléatoire et d'entraîner le même modèle peut parfois fonctionner). Voici, par exemple, un petit DCGAN qui donne des résultats plutôt bons avec Fashion MNIST :

```

codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                               activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                               activation="tanh")
])
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                      activation=keras.layers.LeakyReLU(0.2),
                      input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                      activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])

```

Le générateur prend des codages de taille 100, les projette en 6272 dimensions ($= 7 \times 7 \times 128$), et reforme le résultat pour obtenir un tenseur $7 \times 7 \times 128$. Celui-ci subit une normalisation par lots, puis est transmis à une couche de convolution transposée avec un pas de 2. Elle le suréchantillonne de 7×7 à 14×14 et réduit sa profondeur de 128 à 64. Le résultat subit de nouveau une normalisation par lots, puis est transmis à une autre couche de convolution transposée avec un pas de 2. Elle le suréchantillonne de 14×14 à 28×28 et réduit sa profondeur de 64 à 1. Puisque cette couche utilise la fonction d'activation tanh, les sorties sont dans la plage -1 à 1 . C'est pourquoi, avant d'entraîner le GAN, nous devons mettre à l'échelle le jeu d'entraînement de sorte qu'il se trouve dans cette plage. Nous devons également en changer la forme pour ajouter la dimension du canal :

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # Changement de forme et
# d'échelle
```

Le discriminateur ressemble fortement à un CNN de classification binaire, excepté que les couches de pooling maximum servant à sous-échantillonner l'image sont remplacées par des convolutions à pas (`strides=2`). Notez également que nous utilisons la fonction d'activation Leaky ReLU.

Globalement, nous avons respecté les recommandations DCGAN, à l'exception du remplacement, dans le discriminateur, des couches BatchNormalization par des couches Dropout (sinon, dans ce cas, l'entraînement était instable), et, dans le générateur, de ReLU par SELU. N'hésitez pas à modifier légèrement cette architecture. Vous constaterez combien elle est sensible aux hyperparamètres (en particulier les taux d'apprentissage relatifs des deux réseaux).

Enfin, pour construire le jeu de données, puis compiler et entraîner ce modèle, nous réutilisons le code précédent. Après 50 époques d'entraînement, le générateur produit des images semblables à celles illustrés à la figure 9.17. Il n'est toujours pas parfait, mais ces images sont plutôt convaincantes.



Figure 9.17 – Images générées par le DCGAN au bout de 50 époques d'entraînement

Si vous étendez cette architecture et l'entraînez sur un jeu de données de visages volumineux, vous pourrez obtenir des images plutôt réalistes. Comme vous pouvez le voir à la figure 9.18, les GAN sont capables d'apprendre des représentations latentes assez significatives. De nombreuses images ont été générées et neuf d'entre elles ont été sélectionnées manuellement (en haut à gauche) : trois représentent des hommes portant des lunettes, trois autres, des hommes sans lunettes, et trois autres encore, des femmes sans lunettes. Pour chacune de ces catégories, nous avons effectué une moyenne sur les codages utilisés pour générer les images, et une image a été générée à partir des codages moyens résultant (en bas à gauche). Autrement dit, chacune des trois images de la partie inférieure gauche représente la moyenne des trois images qui

se trouvent au-dessus. Il s'agit non pas d'une simple moyenne calculée au niveau du pixel (cela donnerait trois visages superposés), mais d'une moyenne calculée dans l'espace latent. Les images représentent donc des visages normaux.

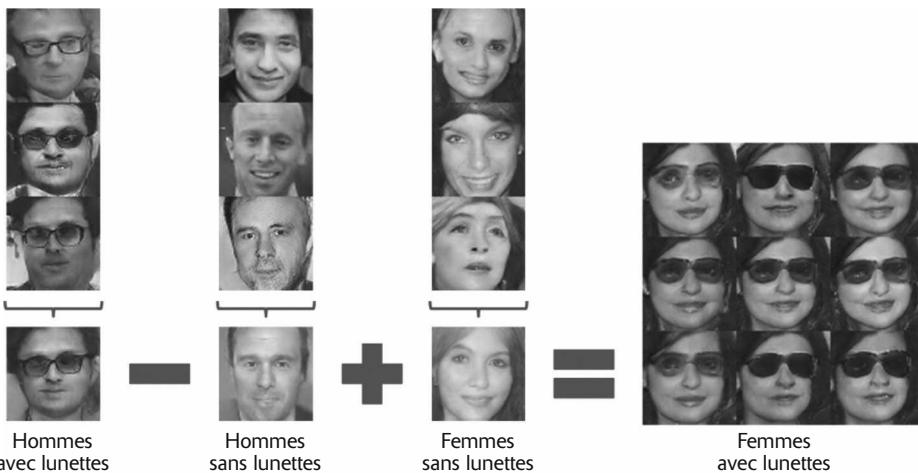


Figure 9.18 – Arithmétique vectorielle pour des concepts visuels
(une partie de la figure 7 tirée de l'article DCGAN)²⁴⁵

Si vous effectuez le calcul hommes avec lunettes, moins hommes sans lunettes, plus femmes sans lunettes – où chaque terme correspond à l'un des codages moyens – et si vous générez l'image qui correspond à ce codage, vous obtenez celle placée au centre de la grille 3×3 de visages sur la droite : une femme avec des lunettes ! Les huit autres images ont été générées à partir du même vecteur, auquel un léger bruit a été ajouté. Cela permet d'illustrer les capacités d'interpolation sémantique des DCGAN. Avec l'arithmétique de visages, nous entrons dans le monde de la science-fiction !



Si vous ajoutez chaque classe d'image comme entrée supplémentaire au générateur et au discriminateur, ils apprendront tous deux l'aspect de chaque classe. Vous serez ainsi capable de contrôler la classe de chaque image produite par le générateur. Il s'agit d'un *GAN conditionnel* (*CGAN, Conditional GAN*)²⁴⁶.

Toutefois, les DCGAN sont loin d'être parfaits. Par exemple, si vous essayez de générer de très grandes images avec des DCGAN, vous obtenez souvent des caractéristiques locales convaincantes mais des incohérences globales (comme une chemise avec une manche plus longue que l'autre). Quelles sont les solutions à ce problème ?

245. Reproduite avec l'aimable autorisation des auteurs.

246. Mehdi Mirza et Simon Osindero, « Conditional Generative Adversarial Nets » (2014) : <https://homl.info/cgan>.

9.9.3 Croissance progressive des GAN

Une technique importante a été proposée dans un article²⁴⁷ publié en 2018 par Tero Karras *et al.*, chercheurs chez Nvidia. Les auteurs ont proposé de générer de petites images au début de l'entraînement, puis d'ajouter progressivement des couches de convolution au générateur et au discriminateur afin de produire des images de plus en plus grandes (4×4 , 8×8 , 16×16 , ..., 512×512 , 1024×1024). Cette approche ressemble à un entraînement glouton par couche d'autoencodeurs empilés. La couche supplémentaire est ajoutée à la fin du générateur et au début du discriminateur, les couches précédemment entraînées restant entraînables.

Par exemple, lors de l'augmentation des sorties du générateur de 4×4 à 8×8 (voir la figure 9.19), une couche de suréchantillonnage (utilisant un filtrage plus proche voisin) est ajoutée à la couche de convolution existante. Elle produit donc des cartes de caractéristiques 8×8 , qui sont ensuite transmises à la nouvelle couche de convolution (puisque elle utilise le remplissage "same" et des pas de 1, ses sorties sont également 8×8). Celle-ci est suivie d'une nouvelle couche de convolution de sortie. Il s'agit d'une couche de convolution normale avec un noyau de taille 1 qui réduit la sortie pour obtenir le nombre de canaux de couleur souhaité (par exemple, trois). Pour ne pas remettre en question les poids entraînés de la première couche de convolution lorsque la nouvelle est ajoutée, la sortie finale est une somme pondérée de la couche de sortie d'origine (qui produit à présent des cartes de caractéristiques 8×8) et de la nouvelle couche de sortie. Le poids des nouvelles sorties est α , tandis que celui des sorties d'origine est $1 - \alpha$, et α augmente lentement de 0 à 1.

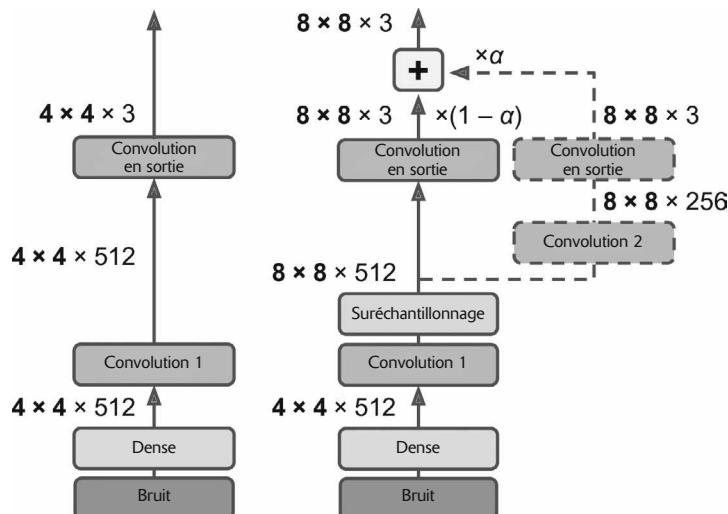


Figure 9.19 – Croissance progressive d'un GAN: le générateur du GAN produit des images en couleur 4×4 (à gauche) et il est étendu pour produire des images 8×8 (à droite)

247. Tero Karras *et al.*, « Progressive Growing of GANs for Improved Quality, Stability, and Variation », *Proceedings of the International Conference on Learning Representations* (2018) : <https://homl.info/progan>.

Autrement dit, les nouvelles couches de convolution (représentées par des lignes pointillées sur la figure 9.19) apparaissent progressivement, tandis que la couche de sortie d'origine disparaît progressivement. Une technique d'apparition/disparition semblable est utilisée lorsqu'une nouvelle couche de convolution est ajoutée au discriminateur (suivie d'une couche de pooling moyen pour le sous-échantillonnage).

L'article propose plusieurs autres techniques dont l'objectif est d'augmenter la diversité des sorties (pour éviter tout *mode collapse*) et de rendre l'entraînement plus stable :

- *Couche d'écart-type par mini-lot*

Elle est ajoutée près de la fin du discriminateur. Pour chaque emplacement dans les entrées, elle calcule l'écart-type sur tous les canaux et toutes les instances du lot ($S = \text{tf.math.reduce_std(inputs, axis=[0, -1])}$). La moyenne de ces écarts-types sur tous les points est ensuite calculée pour obtenir une valeur unique ($v = \text{tf.reduce_mean}(S)$). Enfin, une carte de caractéristiques supplémentaire est ajoutée à chaque instance du lot et remplie à l'aide de la valeur calculée ($\text{tf.concat}([\text{inputs}, \text{tf.fill}([\text{batch_size, height, width, 1}], v)], axis=-1)$). En quoi cela peut-il aider ? Si le générateur produit des images présentant peu de diversité, les cartes de caractéristiques du discriminateur auront un écart-type faible. Grâce à cette couche, le discriminateur aura accès à cette information et il sera moins trompé par un générateur peu imaginatif. Celui-ci sera donc encouragé à produire des sorties plus variées, réduisant le risque de *mode collapse*.

- *Taux d'apprentissage égalisé*

Tous les poids sont initialisés non pas à l'aide d'une initialisation de He mais d'une simple loi normale de moyenne 0 et d'écart-type 1. Cependant, au moment de l'exécution (autrement dit, chaque fois que la couche est exécutée), les poids sont réduits du même facteur que dans l'initialisation de He : ils sont divisés par $\sqrt{\frac{2}{n_{\text{entrées}}}}$, où $n_{\text{entrées}}$ est le nombre d'entrées de la couche. L'article

a montré que cette technique améliore de façon importante les performances du GAN lorsqu'un optimiseur de gradients adaptatif, comme RMSProp, Adam ou autre, est utilisé. Bien entendu, ces optimiseurs normalisent les mises à jour de gradients par leur écart-type estimé (voir le chapitre 3). Par conséquent, il faudra plus de temps pour entraîner des paramètres dont la plage dynamique²⁴⁸ est plus étendue, tandis que les paramètres dont la plage dynamique est faible pourraient être actualisés trop rapidement, conduisant à des instabilités.

En redimensionnant les poids au sein du modèle lui-même plutôt qu'au moment de l'initialisation, cette approche garantit que la plage dynamique reste la même pour tous les paramètres, tout au long de l'entraînement. Ils seront donc appris à la même vitesse. L'entraînement s'en trouve ainsi à la fois accéléré et stabilisé.

248. La plage dynamique d'une variable est le rapport entre la valeur la plus élevée et la valeur la plus faible qu'elle peut avoir.

- *Couche de normalisation par pixel*

Elle est ajoutée dans le générateur après chaque couche de convolution. Elle normalise chaque activation en fonction de toutes les activations dans la même image et au même emplacement, mais sur tous les canaux (en divisant par la racine carrée de l'activation quadratique moyenne). Dans un code TensorFlow, cela se traduit par `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (le terme de lissage `1e-8` est indispensable pour éviter la division par zéro). Cette technique évite les explosions des activations dues à une compétition excessive entre le générateur et le discriminateur.

La combinaison de toutes ces techniques a permis aux auteurs de générer des images de visages en haute définition extrêmement convaincantes (<https://homl.info/progandemo>). Mais qu'entendons-nous précisément par « convaincantes » ? L'évaluation est l'un des plus grands défis des GAN. S'il est possible d'évaluer automatiquement la variété des images générées, juger de leur qualité est une tâche beaucoup plus ardue et subjective. Une technique consiste à utiliser des évaluateurs humains, mais elle est coûteuse et chronophage. Les auteurs ont donc proposé de mesurer la similarité de structure d'image locale entre les images générées et les images entraînées, en prenant en compte chaque échelle. Cette idée les a conduits à une autre innovation révolutionnaire : les StyleGAN.

9.9.4 StyleGAN

L'état de l'art de la génération d'images en haute résolution a de nouveau fait un bond en avant grâce à la même équipe de chercheurs chez Nvidia. Dans un article²⁴⁹ publié en 2018, les auteurs ont présenté l'architecture StyleGAN. Ils ont utilisé des techniques de *transfert de style* dans le générateur pour s'assurer que les images générées présentent la même structure locale que les images d'entraînement, à chaque échelle, améliorant énormément la qualité des images produites. Le discriminateur et la fonction de perte n'ont pas été modifiés, seul le générateur l'a été. Un StyleGAN est constitué de deux réseaux (voir la figure 9.20) :

- *Réseau de correspondance*

Un PMC de huit couches associe les représentations latentes z (c'est-à-dire les codages) à un vecteur w . Ce vecteur passe ensuite par plusieurs *transformations affines* (c'est-à-dire des couches Dense sans fonction d'activation, représentées par les carrés « A » dans la figure 9.20). Nous obtenons alors plusieurs vecteurs, qui contrôlent le style de l'image générée à différents niveaux, allant de la texture fine (par exemple, la couleur des cheveux) à des caractéristiques de haut niveau (par exemple, adulte ou enfant). En résumé, le réseau de correspondance associe les codages à plusieurs vecteurs de styles.

249. Tero Karras *et al.*, « A Style-Based Generator Architecture for Generative Adversarial Networks » (2018) : <https://homl.info/stylegan>.

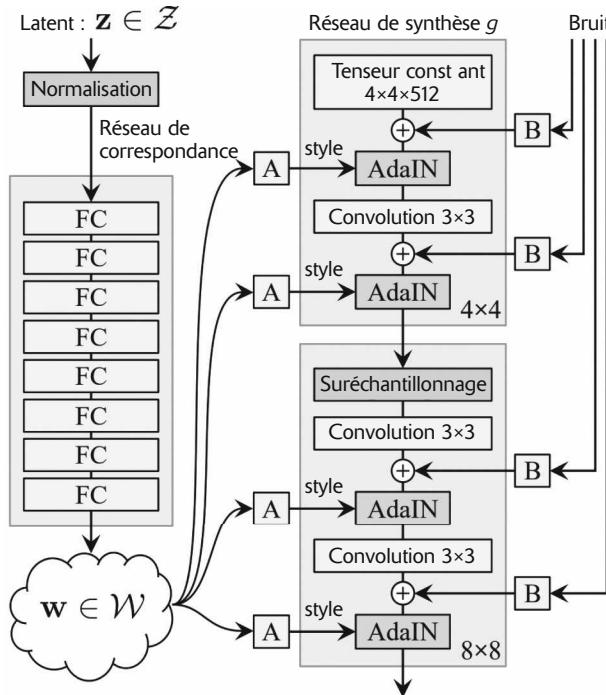


Figure 9.20 – Architecture du générateur d'un StyleGAN
(partie de la figure 1 de l'article StyleGAN)²⁵⁰; (FC: couche intégralement connectée)

- *Réseau de synthèse*

Il est responsable de la génération des images. Il dispose d'une entrée apprise constante (plus précisément, cette entrée sera constante *après* l'entraînement, mais, *pendant* l'entraînement, elle reste ajustée par la rétropropagation). Comme précédemment, il traite cette entrée à l'aide de plusieurs couches de convolution et de suréchantillonnage, mais avec deux changements. Premièrement, du bruit est ajouté à l'entrée et à toutes les sorties des couches de convolution (avant la fonction d'activation). Deuxièmement, chaque couche de bruit est suivie d'une couche de *normalisation d'instance adaptative* (AdaIn, *Adaptive Instance Normalization*). Elle normalise indépendamment chaque carte de caractéristiques (en soustrayant la moyenne de la carte de caractéristiques et en divisant par son écart-type), puis utilise le vecteur de styles pour déterminer l'échelle et le décalage de chaque carte de caractéristiques (le vecteur de styles contient une échelle et un terme constant pour chaque carte de caractéristiques).

L'idée d'ajouter du bruit indépendamment des codages est très importante. Certaines parties d'une image sont assez aléatoires, comme l'emplacement exact de chaque tache de rousseur ou des cheveux. Dans les GAN précédents, ce côté

aléatoire devait provenir soit des codages, soit d'un bruit pseudo-aléatoire produit par générateur lui-même. S'il était issu des codages, le générateur devait dédier une partie significative de la puissance de représentation des codages au stockage du bruit; un vrai gaspillage. De plus, le bruit devait passer par tout le réseau pour atteindre les couches finales du générateur; une contrainte plutôt inutile qui ralentissait probablement l'entraînement. Enfin, certains artefacts visuels pouvaient apparaître car le même bruit était utilisé à différents niveaux. Si, à la place, le générateur tente de produire son propre bruit pseudo-aléatoire, celui-ci risque de ne pas être très convaincant, conduisant à encore plus d'artefacts visuels. Sans oublier qu'une partie des poids du générateur doit être réservée à la production d'un bruit pseudo-aléatoire, ce qui ressemble de nouveau à du gaspillage. En ajoutant des entrées de bruit supplémentaires, tous ces problèmes sont évités; le GAN est capable d'utiliser le bruit fourni pour ajouter la bonne quantité de hasard à chaque partie de l'image.

Le bruit ajouté est différent pour chaque niveau. Chaque entrée de bruit est constituée d'une seule carte de caractéristiques remplie d'un bruit gaussien, qui est diffusé à toutes les cartes de caractéristiques (du niveau donné) et mis à l'échelle à l'aide des facteurs de redimensionnement par caractéristique appris (les carrés « B » dans la figure 9.20) avant d'être ajouté.

Pour finir, un StyleGAN utilise une technique de *régularisation de mélange* (*mixing regularization*), ou *mélange de style* (*style mixing*), dans laquelle un pourcentage des images générées est produit en utilisant deux codages différents. Plus précisément, les codages c_1 et c_2 passent par le réseau de correspondance, avec deux vecteurs de styles w_1 et w_2 . Ensuite, le réseau de synthèse génère une image à partir des styles w_1 pour les premiers niveaux, et les styles w_2 pour les niveaux restants. Le niveau de transition est choisi aléatoirement. Cela évite que le réseau ne suppose une corrélation entre des styles de niveaux adjacents, ce qui encourage la localité dans le GAN, autrement dit que chaque vecteur de styles affecte uniquement un nombre limité de traits dans l'image générée.

Il existe une telle variété de GAN qu'il faudrait un livre entier pour les décrire tous. Nous espérons que cette introduction vous a apporté les idées principales et, plus important, le souhait d'en savoir plus. Si vous avez du mal avec un concept mathématique, vous trouverez probablement des billets de blog pour vous aider à mieux le comprendre. Ensuite, implémentez votre propre GAN et ne soyez pas découragé si son apprentissage est au départ problématique. Malheureusement, ce comportement est normal et il faut un peu de patience avant d'obtenir des résultats, mais ils en valent la peine. Si vous rencontrez un problème avec un détail d'implémentation, vous trouverez un grand nombre d'implémentations avec Keras ou TensorFlow. En réalité, si vous souhaitez uniquement obtenir des résultats impressionnantes très rapidement, vous pouvez vous contenter d'utiliser un modèle préentraîné (il existe des modèles StyleGAN préentraînés pour Keras).

Dans le chapitre suivant, nous aborderons un domaine du Deep Learning totalement différent: l'apprentissage par renforcement profond.

9.10 EXERCICES

1. Quelles sont les principales tâches dans lesquelles les autoencodeurs sont employés ?
2. Supposons que vous souhaitiez entraîner un classificateur et que vous disposiez d'un grand nombre de données d'entraînement non étiquetées, mais seulement de quelques milliers d'instances étiquetées. En quoi les autoencodeurs peuvent-ils vous aider ? Comment procéderiez-vous ?
3. Si un autoencodeur reconstruit parfaitement les entrées, est-il nécessairement un bon autoencodeur ? Comment pouvez-vous évaluer les performances d'un autoencodeur ?
4. Que sont les autoencodeurs sous-complets et sur-complets ? Quel est le principal risque d'un autoencodeur excessivement sous-complet ? Et celui d'un autoencodeur sur-complet ?
5. Comment liez-vous des poids dans un autoencodeur empilé ? Quel en est l'intérêt ?
6. Qu'est-ce qu'un modèle génératif ? Nommez un type d'autoencodeur génératif.
7. Qu'est-ce qu'un GAN ? Nommez quelques tâches dans lesquelles les GAN peuvent briller.
8. Quelles sont les principales difficultés de l'entraînement des GAN ?
9. Préentraînement d'un classificateur d'images avec un autoencodeur débruiteur. Vous pouvez utiliser MNIST (option la plus simple) ou n'importe quel autre jeu d'images plus complexes, comme CIFAR10 (<https://homl.info/122>), si vous souhaitez augmenter la difficulté. Quel que soit le jeu de données choisi, voici les étapes à suivre :
 - Divisez les données en un jeu d'entraînement et un jeu de test. Entraînez un autoencodeur débruiteur profond sur l'intégralité du jeu d'entraînement.
 - Vérifiez que les images sont correctement reconstruites. Affichez les images qui produisent la plus grande activation de chaque neurone dans la couche de codage.
 - Construisez un réseau de neurones profond de classification, en réutilisant les couches inférieures de l'autoencodeur. Entraînez-le avec uniquement 500 images du jeu d'entraînement. Ses performances sont-elles meilleures avec ou sans préentraînement ?
10. Entraînez un autoencodeur variationnel sur le jeu d'images de votre choix et faites-lui générer des images. Vous pouvez également essayer de trouver un jeu de données non étiquetées qui vous intéresse et vérifier que l'autoencodeur est capable de générer de nouveaux échantillons.
11. Entraînez un DCGAN sur le jeu de données d'images de votre choix et servez-vous-en pour générer des images. Ajoutez un rejet d'expériences et voyez si les performances sont meilleures. Convertissez-le en un GAN conditionnel dans lequel vous pouvez contrôler la classe générée.

Les solutions de ces exercices sont données à l'annexe A.

10

Apprentissage par renforcement

L'apprentissage par renforcement (RL, Reinforcement Learning) est aujourd'hui l'un des domaines les plus passionnants du Machine Learning, mais il est également l'un des plus anciens. Datant des années 1950, il a trouvé de nombreuses applications intéressantes au fil des années²⁵¹, notamment dans le jeu (par exemple, TD-Gammon, un jeu de backgammon) et dans la commande de machines, mais il a rarement fait la une des journaux.

L'année 2013 a connu une révolution, lorsque des chercheurs de la start-up anglaise DeepMind ont présenté un système capable d'apprendre à jouer à n'importe quel jeu Atari²⁵², allant jusqu'à battre les humains²⁵³ dans la plupart d'entre eux, en utilisant uniquement les pixels bruts en entrée et sans connaissances préalables des règles du jeu²⁵⁴. Cela n'a été que le premier d'une série d'exploits stupéfiants, avec pour point culminant la victoire de leur système AlphaGo en mars 2016 sur Lee Sedol, un joueur de go légendaire, et, en mai 2017, sur Ke Jie, le champion du monde. Aucun programme n'avait jamais été en mesure de battre un maître de ce jeu, encore moins le champion du monde. Aujourd'hui, le domaine de l'apprentissage par renforcement fourmille de nouvelles idées, avec une grande diversité d'applications. DeepMind a été rachetée par Google en 2014 pour plus de 500 millions de dollars.

251. Pour de plus amples informations, consultez l'ouvrage *Reinforcement Learning: An Introduction* (MIT Press), de Richard Sutton et Andrew Barto (<https://homl.info/l26>).

252. Volodymyr Mnih *et al.*, « Playing Atari with Deep Reinforcement Learning » (2013): <https://homl.info/dqn>.

253. Volodymyr Mnih *et al.*, « Human-Level Control Through Deep Reinforcement Learning », *Nature*, 518 (2015), 529-533 : <https://homl.info/dqn2>.

254. Des vidéos montrant le système de DeepMind qui apprend à jouer à *Space Invaders*, *Breakout* et d'autres sont disponibles à l'adresse <https://homl.info/dqn3>.

Comment les chercheurs de DeepMind sont-ils arrivés à ce résultat ? Avec le recul, cela semble plutôt simple. Ils ont appliqué la puissance de l'apprentissage profond au domaine de l'apprentissage par renforcement et cela a fonctionné au-delà de leurs espérances.

Dans ce chapitre, nous commencerons par expliquer ce qu'est l'apprentissage par renforcement et ses applications de préférence. Nous présenterons ensuite deux des techniques les plus importantes de l'apprentissage par renforcement profond, les *gradients de politique* et les DQN (*Deep Q-Networks*), et nous expliquerons les *processus de décision markoviens* (MDP, *Markov decision processes*). Nous emploierons ces techniques pour entraîner des modèles permettant de garder un bâton en équilibre sur un chariot en déplacement. Puis nous décrirons la bibliothèque TF-Agents, qui se fonde sur des algorithmes modernes simplifiant la construction de systèmes RL performants. Nous l'utiliserons pour entraîner un agent à jouer à *Breakout*, le fameux jeu d'Atari. Nous terminerons ce chapitre par un passage en revue des dernières avancées dans le domaine.

10.1 APPRENDRE À OPTIMISER LES RÉCOMPENSES

Dans l'apprentissage par renforcement, un *agent* logiciel procède à des *observations* et réalise des *actions* au sein d'un *environnement*. En retour, il reçoit des *récompenses*. Son objectif est d'apprendre à agir de façon à maximiser les récompenses espérées sur le long terme. En faisant un peu d'anthropomorphisme, on peut voir une récompense positive comme un plaisir, et une récompense négative comme une douleur (le terme « récompense » est quelque peu malheureux dans ce cas). Autrement dit, l'*agent* opère dans l'environnement et apprend par tâtonnements à augmenter son plaisir et à diminuer sa douleur.

Cette formulation plutôt générale peut s'appliquer à une grande diversité de tâches. En voici quelques exemples (voir la figure 10.1) :

- L'*agent* peut être le programme qui contrôle un robot. Dans ce cas, l'environnement est le monde réel, l'*agent* observe l'environnement au travers de *capteurs*, comme des caméras et des capteurs tactiles, et ses actions consistent à envoyer des signaux pour activer les moteurs. Il peut être programmé de façon à recevoir des récompenses positives lorsqu'il approche de la destination visée, et des récompenses négatives lorsqu'il perd du temps ou va dans la mauvaise direction.
- L'*agent* peut être le programme qui contrôle Ms. Pac-Man. Dans ce cas, l'environnement est une simulation du jeu Atari, les actions sont les neuf positions possibles du joystick (en haut à gauche, en bas, au centre, etc.), les observations sont les captures d'écran et les récompenses sont simplement les points obtenus au jeu.
- De façon comparable, l'*agent* peut être le programme qui joue à un jeu de plateau, comme le jeu de go.
- L'*agent* ne contrôle pas obligatoirement le déplacement d'un objet physique (ou virtuel). Il peut s'agir, par exemple, d'un thermostat intelligent qui reçoit

des récompenses positives dès qu'il permet d'atteindre la température visée en économisant de l'énergie, et des récompenses négatives lorsque la température doit être ajustée manuellement. L'agent doit donc apprendre à anticiper les besoins des personnes.

- L'agent peut observer les prix des actions et décider du nombre à acheter ou à vendre chaque seconde. Les récompenses dépendent évidemment des gains et des pertes.

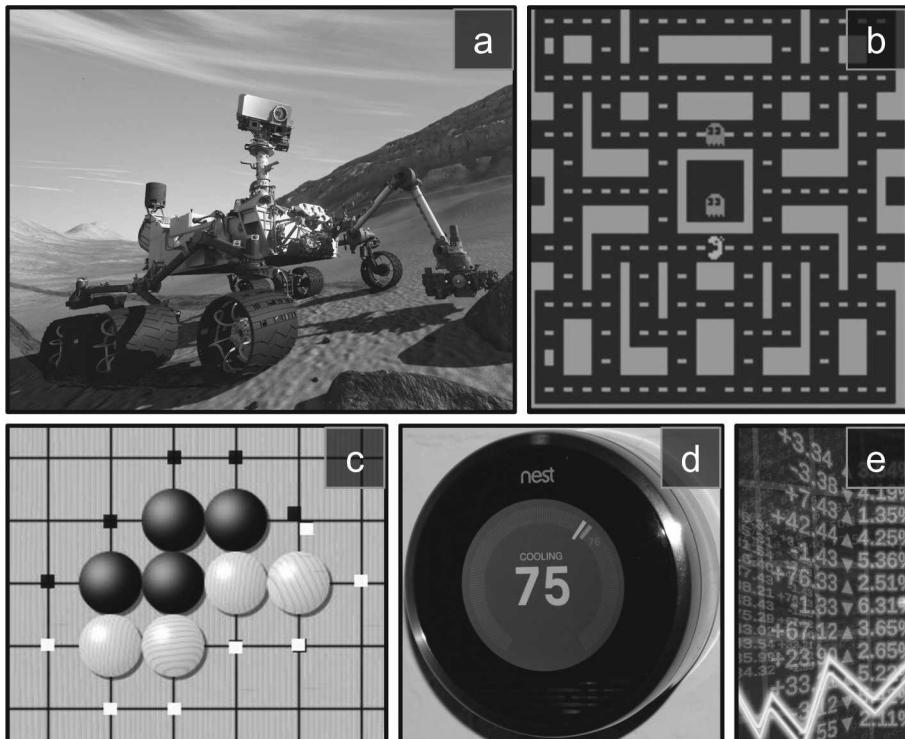


Figure 10.1 – Exemples d'apprentissage par renforcement : (a) robot, (b) Ms. Pac-Man, (c) joueur de Go, (d) thermostat, (e) courtier automatique²⁵⁵

Parfois, les récompenses positives n'existeront pas. Par exemple, l'agent peut se déplacer dans un labyrinthe, en recevant une récompense négative à chaque pas. Il vaut donc mieux pour lui trouver la sortie aussi rapidement que possible ! Il existe de nombreux autres exemples de tâches pour lesquelles l'apprentissage par renforcement convient parfaitement, comme les voitures autonomes, les systèmes de

255. L'image (a) provient de la NASA (domaine public). (b) est une capture d'écran du jeu Ms. Pac-Man d'Atari (l'auteur pense que son utilisation dans ce chapitre est acceptable). (c) et (d) proviennent de Wikipédia. (c) a été créée par l'utilisateur Stevertigo et a été publiée sous licence Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>). (e) a été reproduite à partir de Pixabay, publiée sous licence Creative Commons CC0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

recommandation, le placement de publicités dans les pages web ou encore le contrôle de la zone d'une image sur laquelle un système de classification d'images doit focaliser son attention.

10.2 RECHERCHE DE POLITIQUE

L'algorithme que l'agent logiciel utilise pour déterminer ses actions est appelé sa *politique* (*policy*). La politique peut être un réseau de neurones qui prend en entrée des observations et produit en sortie l'action à réaliser (voir la figure 10.2).

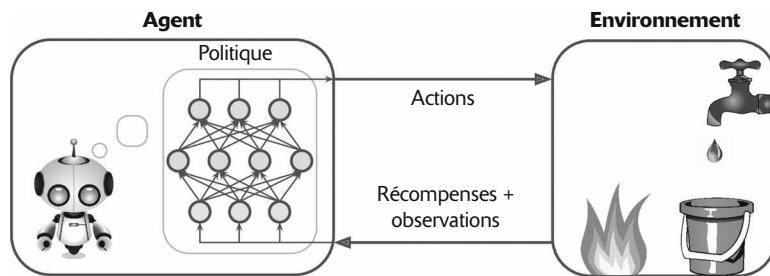


Figure 10.2 – Apprentissage par renforcement qui utilise un réseau de neurones comme politique

La politique peut être tout algorithme imaginable et n'est pas nécessairement déterministe. En réalité, dans certains cas, il n'a même pas besoin d'observer son environnement ! Prenons un robot aspirateur dont la récompense est le volume de poussière ramassée en 30 minutes. Sa politique pourrait être d'avancer à chaque seconde avec une probabilité p ou de tourner aléatoirement vers la gauche ou la droite avec une probabilité $1 - p$. L'angle de rotation aurait une valeur aléatoire située entre $-r$ et $+r$. Puisque cette politique présente un caractère aléatoire, il s'agit d'une *politique stochastique*. Le robot aura une trajectoire erratique de sorte qu'il finira par arriver à tout endroit qu'il peut atteindre et il ramassera toute la poussière. La question est : quelle quantité de poussière va-t-il ramasser en 30 minutes ?

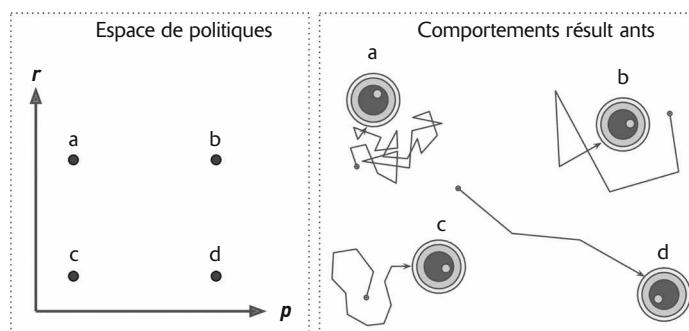


Figure 10.3 – Quatre points dans un espace de politiques (à gauche) et comportement correspondant de l'agent (à droite)

Comment pouvons-nous entraîner un tel robot ? Nous n'avons que deux *paramètres de politique* à ajuster : la probabilité p et la plage de l'angle r . L'algorithme d'apprentissage pourrait consister à essayer de nombreuses valeurs différentes pour ces paramètres et à retenir la combinaison qui affiche les meilleures performances (voir la figure 10.3). Voilà un exemple de *recherche de politique*, dans ce cas fondée sur une approche par force brute. Cependant, lorsque l'*espace de politiques* est trop vaste (ce qui est fréquent), une telle recherche du jeu de paramètres appropriés revient à rechercher une aiguille dans une gigantesque botte de foin.

Une autre manière d'explorer l'espace de politiques consiste à utiliser des *algorithmes génétiques*. Par exemple, on peut créer aléatoirement une première génération de 100 politiques et les essayer, puis « tuer » les 80 plus mauvaises²⁵⁶ et laisser les 20 survivantes produire chacune quatre descendants. Un descendant n'est rien d'autre qu'une copie de son parent²⁵⁷ avec une variation aléatoire. Les politiques survivantes et leurs descendants forment la deuxième génération. On peut ainsi poursuivre cette itération sur les générations jusqu'à ce que l'évolution produise une politique appropriée²⁵⁸.

Une autre approche se fonde sur des techniques d'optimisation. Il s'agit d'évaluer les gradients des récompenses en lien avec les paramètres de politique, puis d'ajuster ces paramètres en suivant les gradients vers des récompenses plus élevées²⁵⁹. Cette approche est appelée *gradient de politique* et nous y reviendrons plus loin dans ce chapitre. Par exemple, dans le cas du robot aspirateur, on peut augmenter légèrement p et évaluer si cela augmente la quantité de poussière ramassée par le robot en 30 minutes. Dans l'affirmative, on augmente encore un peu p , sinon on le réduit. Nous implémenterons avec TensorFlow un algorithme de gradient de politique populaire, mais avant cela nous devons commencer par créer un environnement dans lequel opérera l'agent. C'est le moment de présenter OpenAI Gym.

10.3 INTRODUCTION À OPENAI GYM

Dans l'apprentissage par renforcement, l'entraînement d'un agent ne peut pas se faire sans un environnement de travail. Si l'on veut programmer un agent qui apprend à jouer à un jeu Atari, on a besoin d'un simulateur de jeu Atari. Si l'on veut programmer un robot marcheur, l'environnement est alors le monde réel et l'entraînement peut se faire directement dans cet environnement, mais cela a certaines limites : si le robot tombe d'une falaise, on ne peut pas simplement cliquer sur « annuler ». Il est également impossible d'accélérer le temps ; l'augmentation de la puissance de

256. Il est souvent préférable de laisser une petite chance de survie aux moins bons éléments afin de conserver une certaine diversité dans le « patrimoine génétique ».

257. S'il n'y a qu'un seul parent, il s'agit d'une *reproduction asexuée*. Avec deux parents ou plus, il s'agit d'une *reproduction sexuée*. Le génome d'un descendant (dans ce cas un jeu de paramètres de politique) est constitué de portions aléatoires des génomes de ses parents.

258. L'algorithme *NeuroEvolution of Augmenting Topologies* (NEAT) (<https://homl.info/neat>) est un exemple intéressant d'algorithme génétique utilisé pour l'apprentissage par renforcement.

259. Il s'agit d'une *montée de gradient*. Cela équivaut à une descente de gradient, mais dans le sens opposé : maximisation à la place de minimisation.

calcul ne fera pas avancer le robot plus vite. Par ailleurs, il est généralement trop coûteux d'entraîner 1 000 robots en parallèle. En résumé, puisque, dans le monde réel, l'entraînement est difficile et long, on a généralement besoin d'un *environnement simulé*, au moins pour initier l'entraînement. Vous pouvez, par exemple, employer des bibliothèques comme PyBullet (<https://pybullet.org/>) ou MuJoCo (<http://www.mujoco.org/>) pour des simulations physiques en trois dimensions.

OpenAI Gym²⁶⁰ (<https://gym.openai.com/>) est une librairie qui fournit divers environnements simulés (jeux Atari, jeux de plateau, simulations physiques en 2D et 3D, etc.). Elle permet d'entraîner des agents, de les comparer ou de développer de nouveaux algorithmes RL. Nous avons déjà installé Open Gym au chapitre 1, lors de la création de l'environnement `tf2`. Ouvrons à présent une console Python ou un notebook Jupyter et créons un environnement avec `make()` :

```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614,  0.04207708, -0.00180545])
```

Nous avons créé un environnement CartPole. Il s'agit d'une simulation 2D dans laquelle un chariot peut être déplacé vers la gauche ou la droite afin de garder en équilibre le bâton qui est posé dessus (voir la figure 10.4). La liste de tous les environnements disponibles est donnée par `gym.envs.registry.all()`. Après que l'environnement a été créé, il faut l'initialiser à l'aide de la méthode `reset()`. On obtient alors la première observation. Les observations dépendent du type de l'environnement. Dans le cas de CartPole, chaque observation est un tableau NumPy à une dimension qui contient quatre valeurs. Elles représentent la position horizontale du chariot (0.0 = centre), sa vitesse, l'angle du bâton (0.0 = vertical) et sa vitesse angulaire (une valeur positive représente le sens des aiguilles d'une montre).

À présent, affichons cet environnement en appelant sa méthode `render()` (voir la figure 10.4) :

```
>>> env.render()
True
```

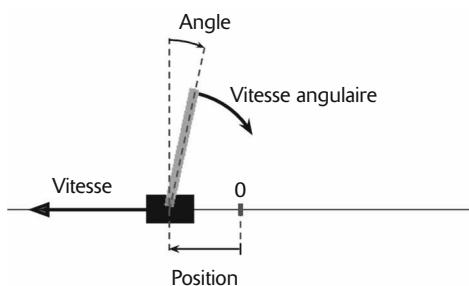


Figure 10.4 – L'environnement CartPole

260. OpenAI est une entreprise de recherche en intelligence artificielle à but non lucratif cofondée par Elon Musk. Son objectif annoncé est de promouvoir et de développer des intelligences artificielles conviviales qui bénéficieront à l'humanité (au lieu de l'exterminer).



Si vous utilisez un serveur sans écran (*headless*), comme une machine virtuelle dans le cloud, le rendu échouera. La seule manière d'éviter cela consiste à utiliser un serveur X factice, comme Xvfb ou Xdummy. Par exemple, vous pouvez installer Xvfb (`apt install xvfb sur Ubuntu ou Debian`) et démarrer Python à l'aide de la commande suivante: `xvfb-run -s "-screen 0 1400x900x24" python`. Vous pouvez également installer Xvfb et la bibliothèque Python `pyvirtualdisplay` (<https://hgomel.info/pyvd/>) (qui repose sur Xvfb) et exécuter, au début du programme, le code Python `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()`.

Si vous souhaitez que `render()` renvoie le rendu de l'image sous forme d'un tableau NumPy, vous pouvez indiquer `mode="rgb_array"` (bizarrement, ce mode affichera également l'environnement à l'écran):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # Hauteur, largeur, canaux (3 = rouge, vert, bleu)
(800, 1200, 3)
```

Demandons à l'environnement quelles sont les actions possibles:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` signifie que les actions possibles sont les entiers 0 et 1, qui représentent une accélération vers la gauche (0) ou vers la droite (1). D'autres environnements peuvent proposer plus d'actions discrètes ou d'autres types d'actions (par exemple, continues). Puisque le bâton penche vers la droite (`obs[2] > 0`), accélérerons le chariot vers la droite:

```
>>> action = 1 # Accélérer vers la droite
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

La méthode `step()` exécute l'action indiquée et retourne quatre valeurs:

- `obs`: il s'agit de la nouvelle observation. Le chariot se déplace à présent vers la droite (`obs[1]>0`). Le bâton est toujours incliné vers la droite (`obs[2]>0`), mais sa vitesse angulaire est devenue négative (`obs[3]<0`). Il va donc probablement pencher vers la gauche après l'étape suivante.
- `reward`: dans cet environnement, vous recevez la récompense 1.0 à chaque étape, quelle que soit votre action. L'objectif est donc de poursuivre l'exécution aussi longtemps que possible.
- `done`: cette valeur sera égale à `True` lorsque l'épisode sera terminé. Cela se produira lorsque l'inclinaison du bâton sera trop importante, sortira de l'écran ou après 200 étapes (dans ce dernier cas, vous aurez gagné). L'environnement doit alors être réinitialisé avant de pouvoir être de nouveau utilisé.

- `info`: ce dictionnaire spécifique à l'environnement peut fournir des informations supplémentaires qui pourront être utiles pour le débogage ou l'entraînement. Par exemple, dans certains jeux, elles peuvent indiquer le nombre de vies dont dispose l'agent.



Lorsque vous en avez terminé avec un environnement, vous devez invoquer sa méthode `close()` pour libérer les ressources qu'il occupait.

Implémentons une politique simple qui déclenche une accélération vers la gauche lorsque le bâton penche vers la gauche, et *vice versa*. Ensuite, exécutons cette politique pour voir quelle récompense moyenne elle permet d'obtenir après 500 épisodes :

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

Ce code se comprend de lui-même. Examinons le résultat :

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Même au bout de 500 essais, cette politique n'a pas réussi à garder le bâton vertical pendant plus de 68 étapes consécutives. Peu satisfaisant. Si vous observez la simulation dans les notebooks Jupyter²⁶¹ vous constaterez que le chariot oscille à gauche et à droite de plus en plus fortement jusqu'à ce que le bâton soit trop incliné. Voyons si un réseau de neurones ne pourrait pas arriver à une meilleure politique.

10.4 POLITIQUES PAR RÉSEAU DE NEURONES

Créons une politique par réseau de neurones. À l'instar de la politique codée précédemment, ce réseau de neurones prend une observation en entrée et produit en sortie l'action à exécuter. Plus précisément, il estime une probabilité pour chaque

261. Voir « 18_reinforcement_learning.ipynb » sur <https://github.com/ageron/handson-ml2>.

action et nous sélectionnons aléatoirement une action en fonction des probabilités estimées (voir la figure 10.5). Dans le cas de l'environnement CartPole, puisqu'il n'y a que deux actions possibles (gauche ou droite), nous n'avons besoin que d'un seul neurone de sortie. Il produira la probabilité p de l'action 0 (gauche), et celle de l'action 1 (droite) sera donc $1 - p$. Par exemple, s'il sort 0,7, nous choisirons une action 0 avec 70 % de probabilité et une action 1 avec 30 % de probabilité.

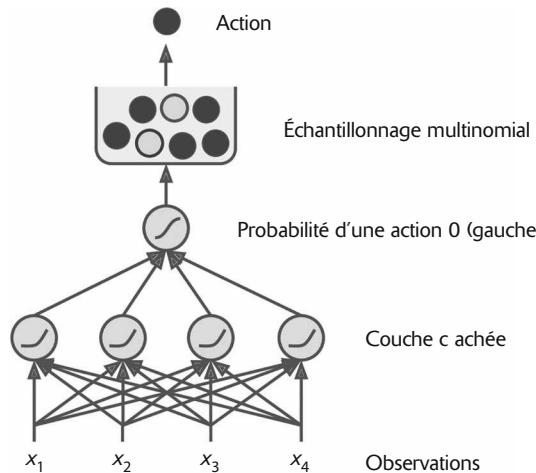


Figure 10.5 – Politique par réseau de neurones

Pourquoi sélectionner aléatoirement une action en fonction de la probabilité donnée par le réseau de neurones plutôt que prendre celle qui possède le score le plus élevé ? Cette approche permet à l'agent de trouver le bon équilibre entre *explorer* de nouvelles actions et *exploiter* les actions réputées bien fonctionner. Voici une analogie : supposons que vous alliez dans un restaurant pour la première fois et que, puisque tous les plats semblent aussi appétissants l'un que l'autre, vous en choisissiez un au hasard. S'il est effectivement bon, vous augmentez sa probabilité de le commander la prochaine fois, mais vous ne devez pas la passer à 100 % car cela vous empêcherait d'essayer d'autres plats alors que certains pourraient être meilleurs que votre choix initial.

Dans cet environnement particulier, les actions et les observations antérieures peuvent être ignorées, car chaque observation contient l'intégralité de l'état de l'environnement. S'il existait un état caché, vous devriez également prendre en compte les observations et les actions antérieures. Par exemple, si l'environnement ne donnait que la position du chariot, sans indiquer sa vitesse, vous auriez à considérer non seulement l'observation actuelle mais également l'observation précédente de façon à estimer sa vitesse actuelle. De même, lorsque les observations sont bruyantes, il faut généralement tenir compte de quelques observations antérieures afin d'estimer l'état courant le plus probable. Le problème de CartPole ne peut donc être plus simple ;

les observations sont dépourvues de bruit et contiennent l'intégralité de l'état de l'environnement.

Voici le code qui permet de construire cette politique par réseau de neurones avec tf.keras :

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4      # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])

```

Après les importations, nous employons un simple modèle Sequential pour définir le réseau de politique. Le nombre d'entrées correspond à la taille de l'espace des observations (qui, dans le cas de CartPole, est égale à 4), et nous avons uniquement cinq unités cachées car le problème n'est pas très complexe. Enfin, puisque nous voulons produire en sortie une seule probabilité (la probabilité d'aller à gauche), nous avons un seul neurone de sortie qui utilise la fonction d'activation sigmoïde. Si le nombre d'actions possibles était supérieur à deux, nous aurions un neurone de sortie par action et nous opterions à la place pour la fonction d'activation softmax.

Nous disposons à présent d'une politique par réseau de neurones qui prend des observations et produit des actions. Mais comment l'entraîner ?

10.5 ÉVALUER DES ACTIONS: LE PROBLÈME D'AFFECTATION DE CRÉDIT

Si nous connaissons la meilleure action à chaque étape, nous pourrions entraîner le réseau de neurones de façon habituelle, en minimisant l'entropie croisée entre la probabilité estimée et la probabilité visée. Il s'agirait simplement d'un apprentissage supervisé normal. Toutefois, dans l'apprentissage par renforcement, la seule aide que reçoit l'agent vient des récompenses, qui sont généralement rares et différenciées. Par exemple, si l'agent réussit à équilibrer le bâton pendant 100 étapes, comment peut-il savoir quelles actions parmi les 100 réalisées étaient bonnes, et lesquelles étaient mauvaises ? Il sait simplement que le bâton est tombé après la dernière action, mais bien sûr celle-ci n'est pas la seule responsable. Il s'agit du *problème d'affectation de crédit* (*credit assignment problem*) : lorsque l'agent obtient une récompense, il lui est difficile de déterminer quelles actions doivent en être créditées (ou blâmées). Comment un chien qui serait gratifié plusieurs heures après un bon comportement pourrait-il comprendre l'origine de la récompense ?

Pour résoudre ce problème, une stratégie classique consiste à évaluer une action en fonction de la somme de toutes les récompenses qui s'ensuivent, en appliquant généralement un *taux de rabais* (*discount rate*) γ (gamma) à chaque étape. Cette somme des récompenses avec rabais est appelée le *rendement* (*return*) de l'action. Par exemple

(voir la figure 10.6), si un agent décide d'aller à droite trois fois de suite et reçoit en récompense +10 après la première étape, 0 après la deuxième et enfin -50 après la troisième, et en supposant que l'on utilise un taux de rabais $\gamma = 0,8$, alors la première action obtient un rendement de $10 + \gamma \times 0 + \gamma^2 (-50) = -22$.

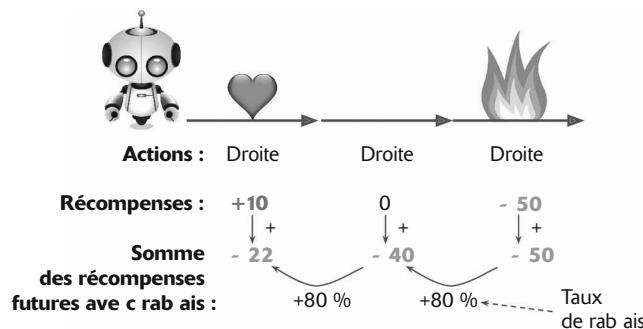


Figure 10.6 – Calcul du rendement d'une action : somme des récompenses futures avec rabais

Si le taux de rabais est proche de 0, les récompenses futures compteront peu en comparaison des récompenses immédiates. À l'inverse, si le taux de rabais est proche de 1, les récompenses arrivant très tardivement compteront presque autant que les récompenses immédiates. En général, le taux de rabais est fixé entre 0,9 et 0,99. Avec un taux égal à 0,95, les récompenses qui arrivent 13 étapes dans le futur comptent pour environ la moitié des récompenses immédiates (car $0,95^{13} \approx 0,5$). En revanche, avec un taux de rabais de 0,99, ce sont les récompenses arrivant 69 étapes dans le futur qui comptent pour moitié autant que les récompenses immédiates. Dans l'environnement CartPole, les actions ont plutôt des effets à court terme et le choix d'un taux de rabais à 0,95 semble raisonnable.

Bien entendu, une bonne action peut être suivie de plusieurs mauvaises actions qui provoquent la chute rapide du bâton. Dans ce cas, la bonne action reçoit un rendement faible (comme un bon acteur peut parfois jouer dans un très mauvais film). Cependant, si l'on joue un nombre suffisant de parties, les bonnes actions obtiendront en moyenne un meilleur rendement que les mauvaises. Nous souhaitons estimer la qualité moyenne d'une action en comparaison des autres actions possibles. Il s'agit du *bénéfice* de l'action. Pour cela, nous devons exécuter de nombreux épisodes et normaliser tous les rendements des actions (en soustrayant la moyenne et en divisant par l'écart-type). Suite à cela, on peut raisonnablement supposer que les actions ayant un bénéfice négatif étaient mauvaises, tandis que celles ayant un bénéfice positif étaient bonnes. Puisque nous avons à présent une solution pour évaluer chaque action, nous sommes prêts à entraîner notre premier agent en utilisant des gradients de politique.

10.6 GRADIENTS DE POLITIQUE

Comme nous l'avons indiqué précédemment, les algorithmes de gradients de politique (PG, *Policy Gradients*) optimisent les paramètres d'une politique en suivant les gradients vers les récompenses les plus élevées. L'une des classes d'algorithmes PG les plus répandus, appelés *algorithmes REINFORCE*, a été présentée²⁶² en 1992 par Ronald Williams. En voici une variante fréquente :

- Tout d'abord, on laisse la politique par réseau de neurones jouer plusieurs fois au jeu et, à chaque étape, on calcule les gradients qui augmenteraient la probabilité de l'action choisie, mais on ne les applique pas encore.
- Au bout de plusieurs épisodes, on calcule le bénéfice de chaque action (à l'aide de la méthode décrite au paragraphe précédent).
- Un bénéfice positif indique que l'action était bonne et l'on applique donc les gradients calculés précédemment pour que l'action ait davantage de chances d'être choisie dans le futur. En revanche, si son bénéfice est négatif, cela signifie que l'action était mauvaise et l'on applique donc les gradients opposés pour qu'elle devienne un peu moins probable dans le futur. La solution consiste simplement à multiplier chaque vecteur de gradient par le bénéfice de l'action correspondante.
- Enfin, on calcule la moyenne de tous les vecteurs de gradients obtenus et on l'utilise pour effectuer une étape de descente de gradient.

Implémentons cet algorithme avec tf.keras. Nous entraînerons le réseau de neurones construit plus haut de façon qu'il maintienne le bâton en équilibre sur le chariot. Tout d'abord, nous avons besoin d'une fonction qui joue une étape. Pour le moment, nous prétendrons que l'action réalisée, quelle qu'elle soit, est appropriée afin que nous puissions calculer la perte et ses gradients (les gradients seront simplement conservés pendant un certain temps et nous les modifierons ultérieurement en fonction de la qualité réelle de l'action) :

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
        grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```

Étudions cette fonction :

- Dans le bloc GradientTape (voir le chapitre 4), nous commençons par appeler le modèle en lui fournissant une seule observation (puisque le modèle attend un lot, nous modifions la forme de l'observation afin qu'elle devienne

262. Ronald J. Williams, « Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning », *Machine Learning*, 8 (1992), 229-256 : <https://homl.info/132>.

un lot d'une seule instance). Nous obtenons alors la probabilité d'aller à gauche.

- Ensuite, nous prenons un nombre aléatoire à virgule flottante entre 0 et 1, et nous le comparons à `left_proba`. `action` vaudra `False` avec une probabilité `left_proba`, ou `True` avec une probabilité `1 - left_proba`. Après avoir converti cette valeur booléenne en un entier, l'action sera égale à 0 (gauche) ou 1 (droite) avec les probabilités appropriées.
- Puis nous définissons la probabilité visée d'aller à gauche. Elle est égale à 1 moins l'action (convertie en un nombre à virgule flottante). Si l'action vaut 0 (gauche), alors la probabilité visée d'aller à gauche sera égale à 1. Si l'action vaut 1 (gauche), alors la probabilité visée sera égale à 0.
- Nous calculons alors la perte à l'aide de la fonction indiquée et nous utilisons l'enregistrement pour calculer les gradients de la perte en rapport avec les variables entraînables du modèle. À nouveau, ces gradients seront ajustés ultérieurement, avant de les appliquer, en fonction de la qualité réelle de l'action.
- Enfin, nous jouons l'action sélectionnée et retournons la nouvelle observation, la récompense, un indicateur de fin de l'épisode et, bien entendu, les gradients calculés.

Créons à présent une autre fonction qui s'appuiera sur la fonction `play_one_step()` pour jouer plusieurs épisodes, en retournant toutes les récompenses et tous les gradients pour chaque épisode et chaque étape :

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

Ce code retourne une liste de listes de récompenses (une liste de récompenses par épisode, contenant une récompense par étape), ainsi qu'une liste de listes de gradients (une liste de gradients par épisode, chacun contenant un tuple de gradients par étape, chaque tuple contenant un tenseur de gradient par variable entraînable).

L'algorithme utilisera la fonction `play_multiple_episodes()` pour jouer plusieurs fois au jeu (par exemple, dix fois), puis reviendra en arrière pour examiner toutes les récompenses, leur appliquer un rabais et les normaliser. Pour cela, nous

avons besoin de deux autres fonctions. La première calcule la somme des récompenses futures avec rabais à chaque étape. La seconde normalise toutes ces récompenses avec rabais (rendements) sur plusieurs épisodes, en soustrayant la moyenne et en divisant par l'écart-type.

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                               for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Vérifions que tout cela fonctionne :

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                     discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.07277777])]
```

L'appel à `discount_rewards()` retourne exactement ce que nous attendions (voir la figure 10.6). Vous pouvez vérifier que la fonction `discount_and_normalize_rewards()` retourne bien les bénéfices normalisés de l'action pour chaque action dans les deux épisodes. Notez que le premier épisode a été bien plus mauvais que le second, et que ses bénéfices normalisés sont donc tous négatifs. Toutes les actions du premier épisode seront considérées mauvaises et, inversement, toutes celles du second seront considérées bonnes.

Nous sommes presque prêts à exécuter l'algorithme ! Définissons à présent les hyperparamètres. Nous effectuerons 150 itérations d'entraînement, en jouant 10 épisodes par itération, et chaque épisode durera au moins 200 étapes. Nous utiliserons un facteur de rabais égal à 0,95 :

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

Nous avons également besoin d'un optimiseur et de la fonction de perte. Un optimiseur Adam normal avec un taux d'apprentissage de 0,01 fera l'affaire, et nous utiliserons la fonction de perte par entropie croisée binaire, car nous entraînons un classificateur binaire (il existe deux actions possibles : gauche et droite) :

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

Nous sommes prêts à construire et à exécuter la boucle d'entraînement !

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)

    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean([
            final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards in enumerate(all_final_rewards)
            for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Examinons ce code :

- À chaque itération d'entraînement, la boucle appelle la fonction `play_multiple_episodes()`, qui joue dix fois au jeu et retourne toutes les récompenses et tous les gradients de chaque épisode et étape.
- Ensuite, nous appelons `discount_and_normalize_rewards()` pour calculer le bénéfice normalisé de chaque action (que, dans le code, nous appelons `final_reward`). Cela nous indique après coup la qualité réelle de chaque action.
- Puis nous parcourons chaque variable d'entraînement et, pour chacune, nous calculons la moyenne pondérée des gradients sur tous les épisodes et toutes les étapes, pondérée par `final_reward`.
- Enfin, nous appliquons ces gradients moyens en utilisant l'optimiseur. Les variables entraînables du modèle seront ajustées et, espérons-le, la politique sera un peu meilleure.

Et voilà ! Ce code entraînera la politique par réseau de neurones et apprendra à équilibrer le bâton placé sur le chariot (vous pouvez l'essayer dans la section « Policy Gradients » du notebook Jupyter²⁶³). La récompense moyenne par épisode sera très proche de 200 (le maximum par défaut avec cet environnement). C'est gagné !



Les chercheurs tentent de trouver des algorithmes qui fonctionnent correctement même lorsque l'agent ne connaît initialement rien de son environnement. Cependant, hormis si vous rédigez un article, il est préférable de fournir à l'agent le maximum de connaissances préalables, car cela accélère énormément l'entraînement. Par exemple, puisque vous savez que le bâton doit être aussi vertical que possible, vous pouvez ajouter des récompenses négatives proportionnelles à l'angle du bâton. Ainsi, les récompenses se feront moins rares et l'entraînement s'en trouvera accéléré. De même, si vous disposez déjà d'une politique relativement convenable (par exemple, programmée à la main), vous pouvez entraîner le réseau de neurones afin de l'imiter, avant d'utiliser des gradients de politique pour l'améliorer.

263. Voir « 18_reinforcement_learning.ipynb » sur <https://github.com/ageron/handson-ml2>.

Le simple algorithme des gradients de politique que nous venons d'entraîner a résolu le problème CartPole, mais son adaptation à des tâches plus grandes et plus complexes n'est pas satisfaisante. Son efficacité d'échantillonnage est très mauvaise, ce qui signifie qu'il lui faut explorer le jeu pendant très longtemps avant de pouvoir vraiment progresser. Cela vient du fait qu'il doit exécuter plusieurs épisodes pour estimer le bénéfice de chaque action. Toutefois, il peut servir de base à des algorithmes plus puissants, comme des algorithmes *acteur-critique* (que nous verrons brièvement à la fin de ce chapitre).

Nous allons à présent examiner une autre famille d'algorithmes populaire. Alors que les algorithmes PG essaient d'optimiser directement la politique de façon à augmenter les récompenses, ces nouveaux algorithmes opèrent de façon plus indirecte. L'agent apprend à estimer le rendement attendu pour chaque état, ou pour chaque action dans chaque état, puis décide d'agir en fonction de ses connaissances. Pour comprendre ces algorithmes, nous devons commencer par présenter les *processus de décision markoviens* (MDP, *Markov decision processes*).

10.7 PROCESSUS DE DÉCISION MARKOVIENS

Au début du xx^e siècle, le mathématicien Andreï Markov a étudié les processus stochastiques sans mémoire, appelés *chaînes de Markov*. Un tel processus possède un nombre fixe d'états et évolue aléatoirement d'un état à l'autre à chaque étape. La probabilité pour qu'il passe de l'état s à l'état s' est fixée et dépend uniquement du couple (s, s') , non des états passés (le système n'a pas de mémoire).

La figure 10.7 illustre un exemple de chaînes de Markov avec quatre états. Supposons que le processus démarre dans l'état s_0 et que ses chances de rester dans cet état à l'étape suivante soient de 70 %. À terme, il finira bien par quitter cet état pour ne jamais y revenir, car aucun autre état ne pointe vers s_0 . S'il passe dans l'état s_1 , il ira probablement ensuite dans l'état s_2 (probabilité de 90 %), pour revenir immédiatement dans l'état s_1 (probabilité de 100 %). Il peut osciller un certain nombre de fois entre ces deux états mais finira par aller dans l'état s_3 , pour y rester indéfiniment (il s'agit d'un état terminal). Les chaînes de Markov peuvent avoir des dynamiques très différentes et sont très utilisées en thermodynamique, en chimie, en statistiques et bien d'autres domaines.

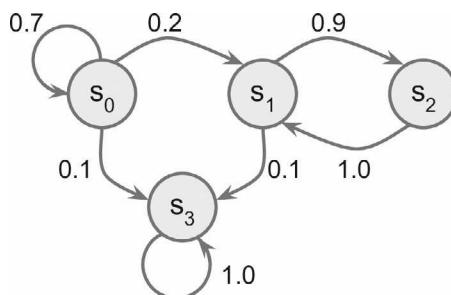


Figure 10.7 – Exemple d'une chaîne de Markov

Les processus de décision markoviens (MDP, *Markov decision processes*) ont été décrits²⁶⁴ pour la première fois dans les années 1950 par Richard Bellman. Ils ressemblent aux chaînes de Markov mais, à chaque étape, un agent peut choisir parmi plusieurs actions possibles et les probabilités des transitions dépendent de l'action choisie. Par ailleurs, certaines transitions entre états renvoient une récompense (positive ou négative) et l'objectif de l'agent est de trouver une politique qui maximise les récompenses au fil du temps.

Par exemple, le MDP représenté à la figure 10.8 possède trois états (représentés par des cercles) et jusqu'à trois actions discrètes possibles à chaque étape (représentées par des losanges). S'il démarre dans l'état s_0 , l'agent peut choisir entre les actions a_0 , a_1 et a_2 . S'il opte pour l'action a_1 , il reste dans l'état s_0 avec certitude et sans aucune récompense. Il peut ensuite décider d'y rester à jamais s'il le souhaite. En revanche, s'il choisit l'action a_0 , il a une probabilité de 70 % d'obtenir une récompense égale à +10 et de rester dans l'état s_0 . Il peut ensuite essayer de nouveau pour obtenir autant de récompenses que possible. Mais, à un moment donné, il finira bien par aller dans l'état s_1 , où il n'a que deux actions possibles : a_0 et a_2 . Il peut décider de ne pas bouger en choisissant en permanence l'action a_0 ou de passer dans l'état s_2 en recevant une récompense négative égale à -50. Dans l'état s_2 , il n'a pas d'autre choix que d'effectuer l'action a_1 , qui le ramènera très certainement dans l'état s_0 , gagnant au passage une récompense de +40.

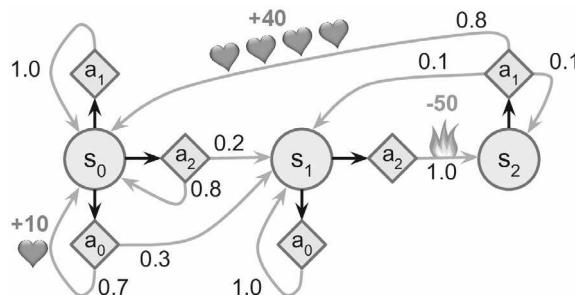


Figure 10.8 – Exemple de processus de décision markovien

Voilà pour le principe. En examinant ce MDP, pouvez-vous deviner quelle stratégie permettra d'obtenir la meilleure récompense au fil du temps ? Dans l'état s_0 , il est clair que l'action a_0 est la meilleure option, et que dans l'état s_2 l'agent n'a pas d'autre choix que de prendre l'action a_1 , mais, dans l'état s_1 , il n'est pas facile de déterminer si l'agent doit rester là (a_0) ou traverser les flammes (a_2).

Bellman a trouvé une façon d'estimer la valeur d'état optimale de tout état s , notée $V^*(s)$, qui correspond à la somme de toutes les récompenses futures avec rabais que l'agent peut espérer en moyenne après qu'il a atteint un état s , en supposant qu'il agisse de manière optimale. Il a montré que si l'agent opère de façon optimale, alors

264. Richard Bellman, « A Markovian Decision Process », *Journal of Mathematics and Mechanics*, 6, n° 5 (1957), 679-684 : <https://hsmi.info/133>.

l'équation d'optimalité de Bellman s'applique (voir l'équation 10.1). Cette équation récursive dit que si l'agent agit de façon optimale, alors la valeur optimale de l'état courant est égale à la récompense qu'il obtiendra en moyenne après avoir effectué une action optimale, plus la valeur optimale espérée pour tous les états suivants possibles vers lesquels cette action peut conduire.

Équation 10.1 – Équation d'optimalité de Bellman

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \text{ pour tout } s$$

Dans cette équation :

- $T(s, a, s')$ est la probabilité de transition de l'état s vers l'état s' si l'agent choisit l'action a . Par exemple, à la figure 10.8, $T(s_2, a_1, s_0) = 0,8$.
- $R(s, a, s')$ est la récompense obtenue par l'agent lorsqu'il passe de l'état s à l'état s' si l'agent choisit l'action a . Par exemple, à la figure 10.8, $R(s_2, a_1, s_0) = +40$.
- γ est le taux de rabais.

Cette équation conduit directement à un algorithme qui permet d'estimer précisément la valeur d'état optimale de chaque état possible. On commence par initialiser toutes les estimations des valeurs d'états à zéro et on les actualise progressivement en utilisant un algorithme d'itération sur la valeur (Value Iteration) (voir l'équation 10.2). Si l'on prend le temps nécessaire, ces estimations vont obligatoirement converger vers les valeurs d'état optimales, qui correspondent à la politique optimale.

Équation 10.2 – Algorithme d'itération sur la valeur

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \text{ pour tout } s$$

Dans cette équation, $V_k(s)$ est la valeur estimée de l'état s au cours de la $k^{\text{ème}}$ itération de l'algorithme.



Cet algorithme est un exemple de *programmation dynamique*, qui décompose un problème complexe en sous-problèmes traitables qui peuvent être résolus de façon itérative.

Connaître les valeurs d'état optimales est utile, en particulier pour évaluer une politique, mais cela ne dit pas explicitement à l'agent ce qu'il doit faire. Heureusement, Bellman a trouvé un algorithme comparable pour estimer les *valeurs état-action optimales*, généralement appelées *valeurs Q* (*Quality Values*). La valeur Q optimale du couple état-action (s, a) , notée $Q^*(s, a)$, est la somme des récompenses futures avec rabais que l'agent peut espérer en moyenne après avoir atteint l'état s et choisi l'action a , mais avant qu'il ne voie le résultat de cette action, en supposant qu'il agisse de façon optimale après cette action.

Voici comment il fonctionne. Une fois encore, on commence par initialiser toutes les estimations des valeurs Q à zéro, puis on les actualise à l'aide de l'algorithme d'itération sur la valeur Q (*Q-Value Iteration*) (voir l'équation 10.3).

Équation 10.3 – Algorithme d’itération sur la valeur Q

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a')] \text{ pour tout } (s, a)$$

Après avoir obtenu les valeurs Q optimales, il n’est pas difficile de définir la politique optimale, notée $\pi^*(s)$: lorsque l’agent se trouve dans l’état s , il doit choisir l’action qui possède la valeur Q la plus élevée pour cet état, c'est-à-dire $\pi^*(s) = \arg\max_a Q^*(s, a)$.

Appliquons cet algorithme au MDP représenté à la figure 10.8. Commençons par définir celui-ci :

```
transition_probabilities = [    # forme = [s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]]
rewards = [    # forme = [s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Par exemple, pour connaître la probabilité de transition de s_2 à s_0 après avoir réalisé l’action a_1 , nous examinons `transition_probabilities[2][1][0]` (qui vaut 0,8). De façon comparable, pour obtenir la récompense correspondante, nous examinons `rewards[2][1][0]` (qui vaut +40). Et, pour obtenir la liste des actions possibles en s_2 , nous consultons `possible_actions[2]` (dans ce cas, seule l’action a_1 est possible). Ensuite, nous devons initialiser toutes les valeurs Q à zéro (excepté pour les actions impossibles, pour lesquelles nous fixons les valeurs Q à $-\infty$) :

```
Q_values = np.full((3, 3), -np.inf) # -np.inf pour les actions impossibles
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # pour toutes les actions possibles
```

Exécutons à présent l’algorithme d’itération sur la valeur Q. Il applique l’équation 10.3 de façon répétitive, à toutes les valeurs Q, pour chaque état et chaque action possible :

```
gamma = 0.90 # taux de rabais

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

Voici les valeurs Q résultantes :

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])
```

Par exemple, lorsque l'agent se trouve dans l'état s_0 et choisit l'action a_1 , la somme attendue des récompenses futures avec rabais est environ égale à 17,0.

Pour chaque état, examinons l'action qui possède la plus haute valeur Q :

```
>>> np.argmax(Q_values, axis=1) # action optimale pour chaque état
array([0, 0, 1])
```

On obtient ainsi la politique optimale pour ce MDP, avec un taux de rabais 0,90 : dans l'état s_0 , choisir l'action a_0 , puis dans l'état s_1 , choisir l'action a_0 (rester sur place), et dans l'état s_2 , choisir l'action a_1 (la seule possible). Si l'on augmente le taux de rabais à 0,95, il est intéressant de constater que la politique optimale change : dans l'état s_1 , la meilleure action devient a_0 (traverser les flammes!). Ce résultat est sensé, car si l'on donne plus de valeur au futur qu'au présent, alors les perspectives des récompenses futures compensent les douleurs immédiates.

10.8 APPRENTISSAGE PAR DIFFÉRENCE TEMPORELLE

Les problèmes d'apprentissage par renforcement avec des actions discrètes peuvent souvent être modélisés à l'aide des processus de décision markoviens, mais l'agent n'a initialement aucune idée des probabilités des transitions (il ne connaît pas $T(s, a, s')$) et ne sait pas quelles seront les récompenses (il ne connaît pas $R(s, a, s')$). Il doit tester au moins une fois chaque état et chaque transition pour connaître les récompenses, et il doit le faire à plusieurs reprises s'il veut avoir une estimation raisonnable des probabilités des transitions.

L'algorithme d'apprentissage *par différence temporelle*, ou TD (*Temporal Difference*, *TD Learning*), est très proche de l'algorithme d'itération sur la valeur, mais il prend en compte le fait que l'agent n'a qu'une connaissance partielle du MDP. En général, on suppose que l'agent connaît initialement uniquement les états et les actions possibles, rien de plus. Il se sert d'une *politique d'exploration*, par exemple une politique purement aléatoire, pour explorer le MDP, et, au fur et à mesure de sa progression, l'algorithme d'apprentissage TD actualise les estimations des valeurs d'état en fonction des transitions et des récompenses observées (voir l'équation 10.4).

Équation 10.4 – Algorithme d'apprentissage TD

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

ou, de façon équivalente :

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

avec

$$\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

Dans cette équation :

- α est le taux d'apprentissage (par exemple, 0,01).
- $r + \gamma \cdot V_k(s')$ est appelé *cible TD*.
- $\delta_k(s, r, s')$ est appelé *erreur TD*.

Une façon plus concise d'écrire la première forme de cette équation se fonde sur la notation $a \xleftarrow{\alpha} b$, qui signifie $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. La première ligne de l'équation 10.4 peut donc être réécrite ainsi : $V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$.



L'apprentissage TD présente de nombreuses similitudes avec la descente de gradient stochastique, notamment son traitement d'un échantillon à la fois. À l'instar de cette descente de gradient, il ne peut réellement converger que si l'on réduit progressivement le taux d'apprentissage (sinon il oscillera en permanence autour des valeurs Q optimales).

Pour chaque état s , l'algorithme conserve simplement une moyenne mobile des récompenses immédiates reçues par l'agent en quittant cet état, plus les récompenses qu'il espère obtenir ultérieurement (en supposant qu'il agisse de façon optimale).

10.9 APPRENTISSAGE Q

De manière comparable, l'algorithme d'apprentissage Q (*Q-Learning*) correspond à l'algorithme d'itération sur la valeur Q adapté au cas où les probabilités des transitions et les récompenses sont initialement inconnues (voir l'équation 10.5). Il regarde un agent jouer (par exemple, aléatoirement) et améliore progressivement ses estimations des valeurs Q. Lorsqu'il dispose d'estimations de valeur Q précises (ou suffisamment proches), la politique optimale est de choisir l'action qui possède la valeur Q la plus élevée (autrement dit, la politique gloutonne).

Équation 10.5 – Algorithme d'apprentissage Q

$$Q(s, a) \xleftarrow{\alpha} r + \gamma \cdot \max_{a'} Q(s', a')$$

Pour chaque couple état-action (s, a) , cet algorithme garde une moyenne mobile des récompenses r reçues par l'agent lorsqu'il quitte l'état s avec l'action a , plus la somme des récompenses futures qu'il espère obtenir. Puisque la politique cible travaillera de manière optimale, pour estimer cette somme, nous prenons le maximum des estimations de la valeur Q pour l'état s' suivant.

Implémentons cet algorithme d'apprentissage Q. Tout d'abord, nous devons faire en sorte qu'un agent explore l'environnement. Pour cela, nous avons besoin d'une fonction (`step`) qui permet à l'agent d'exécuter une action et d'obtenir l'état et la récompense résultants :

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Passons à présent à la politique d'exploration de l'agent. Puisque l'espace des états est relativement réduit, une simple politique aléatoire suffira. Si nous exécutons

l'algorithme suffisamment longtemps, l'agent visitera chaque état à plusieurs reprises et essaiera également chaque action possible plusieurs fois :

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Ensuite, après avoir initialisé les valeurs Q comme précédemment, nous sommes prêts à exécuter l'algorithme d'apprentissage Q avec une décroissance du taux d'apprentissage (en utilisant la planification par exponentielle décrite au chapitre 3) :

```
alpha0 = 0.05 # taux d'apprentissage initial
decay = 0.005 # décroissance du taux d'apprentissage
gamma = 0.90 # taux de rabais
state = 0 # état initial

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

Cet algorithme convergera vers les valeurs Q optimales, mais il faudra de nombreuses itérations et, potentiellement, un assez grand nombre d'ajustements des hyperparamètres. Vous pouvez le voir à la figure 10.9, l'algorithme d'itération sur la valeur Q (à gauche) converge très rapidement, en moins de 20 itérations, tandis que la convergence de l'algorithme d'apprentissage Q (à droite) demande environ 8 000 itérations. Il est clair que ne pas connaître les probabilités de transition ou les récompenses complique énormément la recherche de la politique optimale !

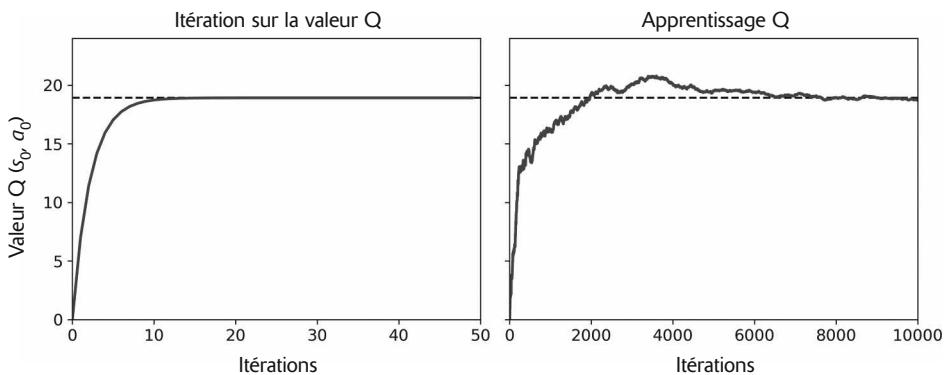


Figure 10.9 – L'algorithme d'itération sur la valeur Q (à gauche) et l'algorithme d'apprentissage Q (à droite)

L'algorithme d'apprentissage Q est un algorithme de *politique hors ligne* (*off-policy*) car la politique entraînée n'est pas nécessairement celle qui est exécutée. Dans le code précédent, la politique exécutée (la politique d'exploration) est totalement aléatoire,

tandis que la politique entraînée choisit toujours les actions ayant des valeurs Q les plus élevées. À l'inverse, l'algorithme des gradients de politique est un algorithme de *politique en ligne* (*on-policy*), car il explore le monde en utilisant la politique entraînée. Il est assez surprenant que l'apprentissage Q soit capable d'apprendre la politique optimale en regardant simplement un agent agir de façon aléatoire (c'est comme si l'on parvenait à apprendre à jouer au golf parfaitement en regardant simplement un singe ivre jouer). Peut-on faire mieux ?

10.9.1 Politiques d'exploration

L'apprentissage Q ne fonctionne que si la politique d'exploration fait suffisamment le tour du MDP. Même si une politique purement aléatoire finira nécessairement par visiter chaque état et chaque transition à de nombreuses reprises, elle risque de prendre un temps extrêmement long. Une meilleure approche consiste à employer la *politique ϵ -greedy* (ϵ est epsilon) : à chaque étape, elle agit de façon aléatoire avec la probabilité ϵ , ou de façon gourmande (*greedy*) avec la probabilité $1-\epsilon$ (c'est-à-dire en choisissant l'action ayant la valeur Q la plus élevée). En comparaison d'une politique totalement aléatoire, la politique ϵ -greedy présente l'avantage de passer de plus en plus de temps à explorer les parties intéressantes de l'environnement, au fur et à mesure que les estimations de la valeur Q s'améliorent, tout en passant un peu de temps dans les régions inconnues du MDP. Il est assez fréquent de débuter avec une valeur élevée pour ϵ (par exemple, 1,0) et ensuite de la réduire progressivement (par exemple, jusqu'à 0,05).

À la place d'une exploration au petit bonheur la chance, une autre approche consiste à encourager la politique d'exploration pour essayer des actions qu'elle a peu testées auparavant. Cela peut être mis en œuvre par l'intermédiaire d'un bonus ajouté aux estimations de la valeur Q (voir l'équation 10.6).

Équation 10.6 – Apprentissage Q avec une fonction d'exploration

$$Q(s,a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

Dans cette équation :

- $N(s', a')$ compte le nombre de fois où l'action a' a été choisie dans l'état s' .
- $f(Q, N)$ est une *fonction d'exploration*, par exemple $f(Q, N) = Q + \kappa/(1 + N)$, où κ est un hyperparamètre de curiosité qui mesure l'attrait de l'agent pour l'inconnu.

10.9.2 Apprentissage Q par approximation et apprentissage Q profond

L'apprentissage Q s'adapte mal aux MDP de grande, voire moyenne, taille, avec de nombreux états et actions. Envisageons, par exemple, l'utilisation de l'apprentissage Q pour entraîner un agent à jouer à Ms. Pac-Man (voir la figure 10.1). Il y a plus de 250 pastilles à manger, et comme à tout instant une pastille peut être présente ou absente (déjà mangée), le nombre d'états possibles est supérieur à $2^{150} \approx 10^{45}$. Si l'on ajoute toutes les combinaisons possibles d'emplacements des fantômes et de Ms.

Pac-Man, le nombre d'états possibles est largement supérieur au nombre d'atomes sur notre planète. Il est donc absolument impossible de conserver une estimation de chaque valeur Q.

La solution consiste à trouver une fonction $Q_\theta(s, a)$ qui permet d'obtenir une approximation de la valeur Q de n'importe quel couple état-action (s, a) en utilisant un nombre de paramètres raisonnable (donnés par le vecteur de paramètres Θ). Cette approche est appelée *apprentissage Q par approximation*. Pendant longtemps, la recommandation a été d'employer des combinaisons linéaires de caractéristiques forgées manuellement à partir de l'état (par exemple, la distance des fantômes les plus proches, leur direction, etc.) pour estimer les valeurs Q, mais, en 2013, DeepMind a montré que l'utilisation de réseaux de neurones profonds donne de bien meilleurs résultats (<https://homl.info/dqn>), en particulier avec des problèmes complexes. Cela permet également d'éviter le travail laborieux d'élaboration de bonnes caractéristiques. Un RNP utilisé pour l'estimation des valeurs Q est un *réseau Q profond* (DQN, *Deep Q-Network*), et l'utilisation d'un DQN pour l'apprentissage Q par approximation est appelée *apprentissage Q profond* (*Deep Q-Learning*).

Mais comment entraîner un DQN ? Pour le comprendre, considérons la valeur Q estimée par un DQN pour un couple état-action (s, a) . Grâce à Bellman, on sait qu'il faudrait que cette estimation soit aussi proche que possible de la récompense r que l'on observe après avoir joué l'action a dans l'état s , plus la somme des récompenses futures avec rabais que l'on peut espérer si l'on joue ensuite de façon optimale. Afin d'estimer cette valeur, on peut simplement utiliser le DQN sur l'état suivant s' , et pour toutes les actions suivantes possibles a' . On obtient ainsi une estimation de la valeur Q pour chaque action suivante possible. Il suffit alors de prendre la plus élevée (car on suppose qu'on jouera de façon optimale), de lui appliquer un rabais, et l'on obtient ainsi une estimation de la somme des récompenses futures avec rabais. En y ajoutant la récompense r , on obtient une valeur Q cible $y(s, a)$ pour le couple état-action (s, a) (voir l'équation 10.7).

Équation 10.7 – Valeur Q cible

$$Q_{\text{cible}}(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

Avec cette valeur Q cible, nous pouvons exécuter une étape d'entraînement à l'aide de tout algorithme de descente de gradient. Plus précisément, nous essayons en général de minimiser l'erreur quadratique entre la valeur Q estimée $Q(s, a)$ et la valeur Q cible (ou la perte de Hubber pour réduire la sensibilité de l'algorithme aux grandes erreurs). Voilà tout pour l'apprentissage Q profond de base ! Voyons comment l'implémenter pour résoudre l'environnement CartPole.

10.10 IMPLÉMENTER L'APPRENTISSAGE Q PROFOND

En premier lieu, il nous faut un réseau Q profond. En théorie, vous avez besoin d'un réseau de neurones qui prend en entrée un couple état-action et produit en sortie une valeur Q approchée. Mais, en pratique, il est beaucoup plus efficace d'utiliser un réseau de neurones qui prend en entrée un état et génère une valeur Q approchée

pour chaque action possible. Pour résoudre l'environnement CartPole, nous n'avons pas besoin d'un réseau très complexe ; deux couches cachées suffisent :

```
env = gym.make("CartPole-v0")
input_shape = [4]    # == env.observation_space.shape
n_outputs = 2      # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

Avec ce DQN, l'action choisie est celle dont la valeur Q prédictive est la plus grande. Pour nous assurer que l'agent explore l'environnement, nous utilisons une politique ϵ -greedy (autrement dit, nous choisissons une action aléatoire avec une probabilité ϵ) :

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Au lieu d'entraîner le DQN en fonction des dernières expériences uniquement, nous stockons toutes les expériences dans une *mémoire de rejet* (ou *tampon de rejet*) et nous en extrayons un lot aléatoire à chaque itération d'entraînement. Cela permet de réduire les corrélations entre les expériences d'un lot entraînement et facilite énormément cette étape. Pour cela, nous utilisons simplement une liste *deque* :

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```



Un *deque* est une liste chaînée dans laquelle chaque élément pointe sur le suivant et sur le précédent. L'insertion et la suppression des éléments sont très rapides, mais plus la liste *deque* est longue, plus l'accès aléatoire est lent. Si vous avez besoin d'une mémoire de rejet très grande, il est préférable d'employer un tampon circulaire ; une implémentation est donnée dans la section «*Deque vs Rotating List*» du notebook²⁶⁵.

Chaque expérience est constituée de cinq éléments : un état, l'action effectuée par l'agent, la récompense résultante, l'état suivant atteint et une valeur booléenne indiquant si l'épisode est à ce stade terminé (*done*). Nous avons besoin d'une petite fonction d'échantillonnage d'un lot aléatoire d'expériences à partir de la mémoire de rejet. Elle retourne cinq tableaux NumPy qui correspondent aux cinq éléments de l'expérience :

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
```

265. Voir «*18_reinforcement_learning.ipynb*» sur <https://github.com/ageron/handson-ml2>.

```

states, actions, rewards, next_states, dones = [
    np.array([experience[field_index] for experience in batch])
    for field_index in range(5)]
return states, actions, rewards, next_states, dones

```

Créons également une fonction qui réalise une seule étape en utilisant la politique ϵ -greedy, puis stocke l'expérience résultante dans la mémoire de rejeu:

```

def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info

```

Pour finir, écrivons une dernière fonction qui échantillonne un lot d'expériences à partir de la mémoire de rejeu et entraîne le DQN en réalisant une seule étape de descente de gradient sur ce lot:

```

batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
        (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Examinons ce code :

- Nous commençons par définir certains hyperparamètres et créons l'optimiseur et la fonction de perte.
- Puis nous créons la fonction `training_step()`. Elle commence par échantillonner un lot d'expériences, puis se sert du DQN pour prédire la valeur Q de chaque action possible dans l'état suivant de chaque expérience. Puisque nous supposons que l'agent va jouer de façon optimale, nous conservons uniquement la valeur maximale de chaque état suivant. Puis nous utilisons l'équation 10.7 pour calculer la valeur Q cible du couple état-action de chaque expérience.
- Nous voulons ensuite utiliser le DQN pour calculer la valeur Q de chaque couple état-action retenu. Toutefois, le DQN va produire non seulement les valeurs Q de l'action réellement choisie par l'agent mais également celles des autres actions possibles. Nous devons donc masquer toutes les valeurs Q inutiles. La fonction `tf.one_hot()` permet de convertir facilement un

tableau d'indices d'actions en un tel masque. Par exemple, si les trois premières expériences contiennent, respectivement, les actions 1, 1, 0, alors le masque commencera par $[[0, 1], [0, 1], [1, 0], \dots]$. Nous pouvons ensuite multiplier la sortie du DQN par ce masque afin d'annuler toutes les valeurs dont nous n'avons pas besoin. Puis nous effectuons une somme sur l'axe 1 pour nous débarrasser de tous les zéros, en ne conservant que les valeurs Q des couples état-action retenus. Nous obtenons alors le tenseur `Q_values`, qui contient une valeur Q prédictive pour chaque expérience du lot.

- Puis nous calculons la perte. Il s'agit de l'erreur quadratique moyenne entre les valeurs Q cibles et prédictives pour les couples état-action retenus.
- Enfin, nous effectuons une descente de gradient pour minimiser la perte vis-à-vis des variables entraînables du modèle.

Voilà pour la partie la plus difficile. L'entraînement du modèle est à présent un jeu d'enfants :

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

Nous exécutons 600 épisodes, chacun pour un maximum de 200 étapes. À chaque étape, nous commençons par calculer la valeur `epsilon` pour la politique ϵ -greedy. Elle passera linéairement de 1 à 0,01, en un peu moins de 500 épisodes. Ensuite, nous appelons la fonction `play_one_step()`, qui utilise la politique ϵ -greedy pour sélectionner une action, l'exécuter et enregistrer l'expérience dans la mémoire de rejeu. Si l'épisode est terminé, nous sortons de la boucle. Enfin, si nous avons dépassé le 50^e épisode, nous appelons la fonction `training_step()` pour entraîner le modèle sur un lot échantillonné à partir de la mémoire de rejeu. Les 50 épisodes sans entraînement donnent à la mémoire de rejeu le temps de se remplir (si nous n'attendons pas suffisamment longtemps, elle manquera de diversité). Nous venons d'implémenter l'algorithme d'apprentissage Q profond !

La figure 10.10 montre les récompenses totales obtenues par l'agent au cours de chaque épisode.

Vous le constatez, l'algorithme ne fait aucun progrès apparent pendant au moins 300 épisodes (en partie parce que la valeur de ϵ était très élevée au début), puis sa performance explose soudainement jusqu'à 200 (le maximum possible dans cet environnement). Voilà une bonne nouvelle : l'algorithme a parfaitement travaillé et beaucoup plus rapidement que l'algorithme par gradient de politique ! Mais attendez... au bout de quelques épisodes, il a oublié tout ce qu'il savait et sa performance est retombée sous 25 ! Il s'agit d'un *oubli catastrophique* et l'un des plus gros problèmes auxquels sont confrontés virtuellement tous les algorithmes RL.

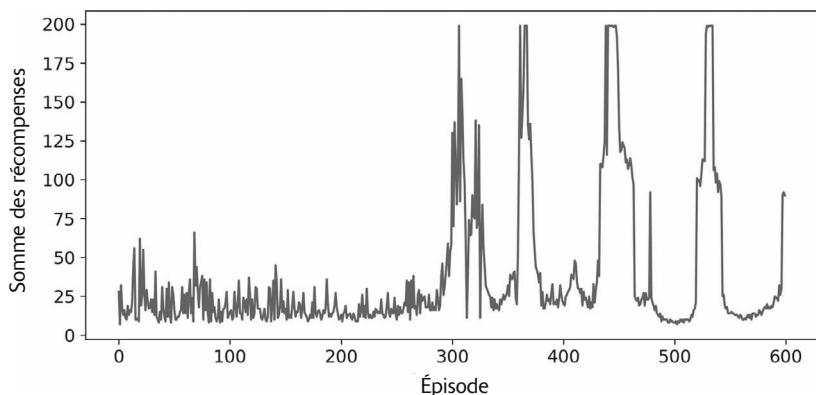


Figure 10.10 – Courbe de progression de l'algorithme d'apprentissage Q profond

Au fur et à mesure que l'agent explore l'environnement, il actualise sa politique, mais ce qu'il apprend dans une partie de l'environnement peut remettre en question ce qu'il a appris précédemment dans les autres parties de l'environnement. Les expériences présentent une certaine corrélation et l'environnement d'apprentissage change sans cesse – une situation peu idéale pour la descente de gradient ! Si vous augmentez la taille de la mémoire de rejet, l'algorithme sera moins sujet à ce problème. La réduction du taux d'apprentissage peut également aider. Mais, en vérité, l'apprentissage par renforcement est difficile. L'entraînement est souvent instable et vous devrez tester de nombreuses valeurs pour les hyperparamètres et les germes aléatoires avant de trouver une combinaison qui convienne parfaitement. Par exemple, en passant le nombre de neurones par couches de 32 à 30 ou 34, la performance ne dépasse jamais 100 (le DQN peut être plus stable avec une seule couche cachée à la place de deux).



L'apprentissage par renforcement est notoirement difficile, essentiellement en raison des instabilités de l'entraînement et de l'importante sensibilité au choix des valeurs des hyperparamètres et des germes aléatoires²⁶⁶. Comme le dit le chercheur Andrej Karpathy, «[l'apprentissage supervisé] souhaite travailler. [...] l'apprentissage par renforcement doit être forcé à travailler». Il vous faudra du temps, de la patience, de la persévérance et, peut-être aussi, un peu de chance. C'est l'une des principales raisons pour lesquelles l'apprentissage par renforcement n'est pas aussi largement adopté que l'apprentissage profond classique (par exemple, les réseaux convolutifs). Il existe cependant quelques applications réelles, en dehors d'AlphaGo et des jeux Atari. Par exemple, Google l'utilise pour optimiser les coûts de son datacenter et il est appliqué dans certaines applications robotiques, pour l'ajustement d'hyperparamètres et dans les systèmes de recommandations.

266. Un excellent billet publié en 2018 par Alex Irpan expose parfaitement les plus grandes difficultés et limites de l'apprentissage par renforcement : <https://homl.info/rhard>.

Nous n'avons pas affiché la perte, car il s'agit d'un piètre indicateur de la performance du modèle. La perte peut baisser alors que l'agent est mauvais. Par exemple, si l'agent reste bloqué dans une petite région de l'environnement et si le DQN commence à surajuster cette région. À l'inverse, la perte peut augmenter alors que l'agent travaille mieux. Par exemple, si le DQN avait sous-estimé les valeurs Q et s'il commence à améliorer ses prédictions, l'agent affichera de meilleures performances, obtiendra plus de récompense, mais la perte peut augmenter car le DQN fixe également les cibles, qui seront aussi plus grandes.

L'algorithme d'apprentissage Q profond de base que nous avons utilisé est trop instable pour apprendre à jouer aux jeux d'Atari. Comment ont donc procédé les chercheurs de DeepMind ? Ils ont simplement peaufiné l'algorithme.

10.11 VARIANTES DE L'APPRENTISSAGE Q PROFOND

Examinons quelques variantes de l'algorithme d'apprentissage Q profond qui permettent de stabiliser et d'accélérer l'entraînement.

10.11.1 Cibles de la valeur Q fixées

Dans l'algorithme d'apprentissage Q profond de base, le modèle est utilisé à la fois pour effectuer des prédictions et pour fixer ses propres cibles. Cela peut conduire à une situation analogue à un chien courant après sa queue. Cette boucle de rétroaction peut rendre le réseau instable : il peut diverger, osciller, se bloquer, etc. Pour résoudre ce problème, les chercheurs de DeepMind ont proposé, dans leur article de 2013, d'utiliser deux DQN au lieu d'un seul. Le premier est le *modèle en ligne*, qui apprend à chaque étape et sert à déplacer l'agent. Le second est le *modèle cible* utilisé uniquement pour définir les cibles. Le modèle cible n'est qu'un clone du modèle en ligne :

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Ensuite, dans la fonction `training_step()`, il suffit de changer une ligne pour utiliser le modèle cible à la place du modèle en ligne au moment du calcul des valeurs Q des états suivants :

```
next_Q_values = target.predict(next_states)
```

Enfin, dans la boucle d'entraînement, nous devons, à intervalles réguliers (par exemple, tous les 50 épisodes), copier les poids du modèle en ligne vers le modèle cible :

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Puisque le modèle cible est actualisé moins souvent que le modèle en ligne, les cibles de la valeur Q sont plus stables, la boucle de rétroaction mentionnée est atténuée et ses effets sont moins sévères. Cette approche a été l'une des principales contributions des chercheurs de DeepMind dans leur article de 2013, en permettant aux agents d'apprendre à jouer aux jeux Atari à partir des pixels bruts. Pour stabiliser l'entraînement, ils ont utilisé un taux d'apprentissage minuscule égal à 0,00025, ont

actualisé le modèle cible uniquement toutes les 10 000 étapes (à la place de 50 dans l'exemple de code précédent), et ont employé une très grande mémoire de rejeu d'un million d'expériences. Ils ont fait baisser epsilon très lentement, de 1 à 0,1 en un million d'étapes, et ils ont laissé l'algorithme s'exécuter pendant 50 millions d'étapes.

Plus loin dans ce chapitre, nous utiliserons la bibliothèque TF-Agents pour entraîner un agent DQN à jouer à *Breakout* en utilisant ces hyperparamètres. Mais, avant d'en arriver là, examinons une autre variante de DQN qui a réussi à battre la solution de référence.

10.11.2 DQN double

Dans un article²⁶⁷ publié en 2015, les chercheurs de DeepMind ont peaufiné leur algorithme DQN, en améliorant ses performances et en stabilisant quelque peu l'entraînement. Ils ont nommé cette variante *DQN double*. La modification s'est fondée sur l'observation suivante : le réseau cible est sujet à une surestimation des valeurs Q. Supposons que toutes les actions soient toutes aussi bonnes les unes que les autres. Les valeurs Q estimées par le modèle cible devraient être identiques, mais, puisqu'il s'agit d'approximations, certaines peuvent être légèrement supérieures à d'autres, uniquement par chance. Le modèle cible choisira toujours la valeur Q la plus élevée, qui pourra être légèrement supérieure à la valeur Q moyenne, conduisant probablement à une surestimation de la valeur Q réelle (un peu comme compter la hauteur de la plus haute vague aléatoire lors de la mesure de la profondeur d'une piscine).

Pour corriger cela, ils ont proposé d'utiliser le modèle en ligne à la place du modèle cible lors de la sélection des meilleures actions pour les prochains états, et d'utiliser le modèle cible uniquement pour estimer les valeurs Q pour ces meilleures actions. Voici la fonction `training_step()` revue :

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # La suite est identique
```

Quelques mois plus tard, une autre amélioration de l'algorithme DQN était proposée.

10.11.3 Rejeu d'expériences à priorités

Au lieu de prendre des expériences de façon *uniforme* dans la mémoire de rejeu, pourquoi ne pas échantillonner les expériences importantes plus fréquemment ?

267. Hado van Hasselt *et al.*, « Deep Reinforcement Learning with Double Q-Learning », *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015), 2094-2100 : <https://hml.info/doubledqn>.

Cette idée, nommée échantillonnage préférentiel (IS, *Importance Sampling*) ou *rejeu d'expériences à priorités* (PER, *Prioritized Experience Replay*), a été introduite dans un article²⁶⁸ de 2015 publié, de nouveau, par les chercheurs de DeepMind.

Plus précisément, des expériences sont considérées « importantes » si elles conduisent probablement à une rapide amélioration de l'apprentissage. Mais comment l'estimer ? Une approche satisfaisante consiste à mesurer l'ampleur de l'erreur TD $\delta = r + \gamma \cdot V(s') - V(s)$. Une erreur TD importante indique qu'une transition (s, r, s') est très surprenante et qu'il est donc probablement intéressant de l'apprendre²⁶⁹. Lorsqu'une expérience est enregistrée dans la mémoire de rejeu, sa priorité est fixée à une valeur très élevée, cela pour garantir qu'elle sera échantillonnée au moins une fois. Cependant, après qu'elle a été choisie (et chaque fois qu'elle est), l'erreur TD δ est calculée et la priorité de cette expérience est fixée à $p = |\delta|$ (plus une petite constante pour garantir que chaque expérience a une probabilité d'être choisie différente de zéro). La probabilité P d'échantillonner une expérience de priorité p est proportionnelle à p^ζ , où ζ est un hyperparamètre qui contrôle le niveau d'avidité de l'échantillonnage préférentiel. Lorsque $\zeta = 0$, nous voulons un échantillonnage uniforme, et lorsque $\zeta = 1$, nous voulons un échantillonnage préférentiel total. Dans l'article, les auteurs ont utilisé $\zeta = 0,6$, mais la valeur optimale dépendra de la tâche.

Il y a toutefois un petit souci. Puisque les échantillons vont privilégier les expériences importantes, nous devons compenser ce travers pendant l'entraînement en abaissant les poids des expériences en fonction de leur importance. Dans le cas contraire, le modèle surajusterait les expériences importantes. Plus clairement, nous voulons que les expériences importantes soient échantillonnées plus souvent, mais cela signifie également que nous devons leur donner un poids plus faible pendant l'entraînement. Pour cela, nous définissons le poids d'entraînement de chaque expérience à $w = (nP)^{-\beta}$, où n est le nombre d'expériences dans la mémoire de rejeu et β est un hyperparamètre qui contrôle le niveau de compensation du penchant de l'échantillonnage préférentiel (0 signifie aucun, tandis que 1 signifie totalement). Dans leur article, les auteurs ont choisi $\beta = 0,4$ au début de l'entraînement, pour l'augmenter linéairement jusqu'à 1 à la fin de l'entraînement. De nouveau, la valeur optimale dépendra de la tâche, mais, si vous augmentez l'un, vous devrez, en général, augmenter l'autre également.

Voyons à présent une dernière variante importante de l'algorithme DQN.

10.11.4 Duel de DQN

L'algorithme *duel de DQN* (DDQN, *Dueling DQN*, à ne pas confondre avec Double DQN, même si les deux techniques peuvent être facilement combinées) a été proposé dans un article²⁷⁰ publié en 2015 par, encore, d'autres chercheurs de DeepMind.

268. Tom Schaul et al., « Prioritized Experience Replay » (2015) : <https://homl.info/prioreplay>.

269. Il se pourrait également que les récompenses soient simplement bruyantes, auquel cas il existe de meilleures méthodes pour estimer l'importance d'une expérience (l'article donne quelques exemples).

270. Ziyu Wang et al., « Dueling Network Architectures for Deep Reinforcement Learning » (2015) : <https://homl.info/ddqn>.

Pour comprendre son fonctionnement, nous devons tout d'abord remarquer que la valeur Q d'un couple état-action (s, a) peut être exprimée sous la forme $Q(s, a) = V(s) + A(s, a)$, où $V(s)$ est la valeur de l'état s et $A(s, a)$ correspond à l'*avantage* de prendre l'action a dans l'état s , en comparaison de toutes les autres actions possibles dans cet état. Par ailleurs, la valeur d'un état est égale à la valeur Q de la meilleure action a^* pour cet état (puisque nous supposons que la politique optimale sélectionnera la meilleure action). Par conséquent, $V(s) = Q(s, a^*)$, ce qui implique que $A(s, a^*) = 0$. Dans un duel de DQN, le modèle estime à la fois la valeur de l'état et l'avantage de chaque action possible. Puisque la meilleure action doit avoir un avantage égal à zéro, le modèle soustrait l'avantage maximal prédict de tous les intérêts.

Voici un modèle de duel de DQN simple, implémenté avec l'API Functional :

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

La suite de l'algorithme est identique à l'algorithme précédent. En réalité, vous pouvez construire un duel de DQN double et le combiner avec le jeu d'expériences à priorités ! Plus généralement, de nombreuses techniques d'apprentissage par renforcement peuvent être associées, comme l'a démontré DeepMind dans un article²⁷¹ publié en 2017. Les auteurs ont combiné six techniques différentes dans un agent nommé *Rainbow*. Ils ont réussi à surpasser largement l'état de l'époque.

Malheureusement, l'implémentation de toutes ces techniques, leur débogage, leur peaufinage et, évidemment, l'entraînement des modèles peuvent nécessiter une énorme quantité de travail. Au lieu de réinventer la roue, il est souvent préférable de réutiliser des bibliothèques évolutives et éprouvées, comme TF-Agents.

10.12 LA BIBLIOTHÈQUE TF-AGENTS

TF-Agents (<https://github.com/tensorflow/agents>) est une bibliothèque d'apprentissage par renforcement qui se fonde sur TensorFlow. Développée par Google, elle est passée en open source en 2018. À l'instar d'OpenAI Gym, elle fournit de nombreux environnements clés en main (y compris des emballages pour tous les environnements OpenAI Gym), et elle prend en charge la bibliothèque PyBullet (pour les simulations physiques en trois dimensions), la bibliothèque DM Control de DeepMind (fondée sur MuJoCo, un moteur pour la physique) et la bibliothèque ML-Agents d'Unity (simulation de nombreux environnements 3D). Elle implémente également plusieurs algorithmes RL, notamment REINFORCE, DQN et DDQN, ainsi que

271. Matteo Hessel *et al.*, «Rainbow: Combining Improvements in Deep Reinforcement Learning» (2017), 3215-3222 : <https://hml.info/rainbow>.

divers composants RL, comme des mémoires de rejet et des indicateurs efficaces. Elle est rapide, évolutive, facile d'utilisation et personnalisable. Vous pouvez créer vos propres environnements et réseaux de neurones, et vous pouvez personnaliser pratiquement n'importe quel composant.

Dans cette section, nous utiliserons TF-Agents pour entraîner un agent à jouer à *Breakout*, le célèbre jeu d'Atari (voir la figure 10.11²⁷²), à l'aide de l'algorithme DQN (vous pouvez facilement basculer vers un autre algorithme si le vous souhaitez).

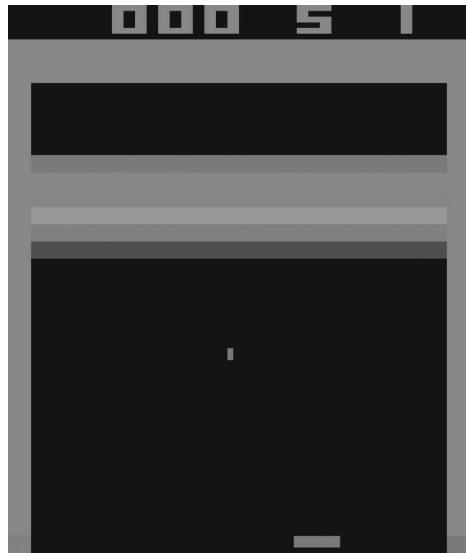


Figure 10.11 – Le célèbre jeu *Breakout*



La bibliothèque TF-Agents est encore relativement récente et bénéficie d'améliorations quotidiennes. L'API pourrait donc avoir légèrement évolué au moment où vous lirez ceci. Toutefois, les principes généraux devraient rester identiques, ainsi que la plupart du code. En cas de problème, j'actualiserais le notebook Jupyter en conséquence.

10.12.1 Installer les librairies TF-Agents et atari-py

Nous avons déjà installé la librairie TF-Agents au chapitre 1, lors de la création de l'environnement tf2. Nous avons également installé la librairie atari-py, qui est une interface Python à ALE (Arcade Learning Environment), un framework construit au-dessus de Stella, un émulateur d'Atari 2600.

272. Si vous ne connaissez pas ce jeu, son principe est simple : une balle rebondit et casse des briques lorsqu'elle les touche. Vous contrôlez une raquette située en bas de l'écran. La raquette peut se déplacer vers la gauche ou la droite. Vous devez casser chaque brique avec la balle, tout en l'empêchant de toucher le bas de l'écran.

Toutefois, au moment de l'écriture de ces lignes, la bibliothèque atari-py officielle ne fonctionne pas sous Windows. Si vous utilisez Windows, il vous faudra probablement installer une autre version de cette librairie :

```
$ conda activate tf2
$ set DL_URL=https://github.com/Kojoley/atari-py/releases
$ pip install --no-index -f %DL_URL% atari_py
```

10.12.2 Environnements TF-Agents

Si tout s'est bien déroulé, vous devriez être en mesure d'importer TF-Agents et de créer un environnement Breakout :

```
>>> from tf_agents.environments import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

Il s'agit uniquement d'un emballage autour d'un environnement OpenAI Gym, auquel vous pouvez accéder au travers de l'attribut `gym`:

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

Les environnements TF-Agents sont similaires aux environnements OpenAI Gym, hormis quelques différences. Tout d'abord, la méthode `reset()` retourne non pas une observation mais un objet `TimeStep` qui enveloppe l'observation, ainsi que quelques informations supplémentaires :

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

La méthode `step()` renvoie également un objet `TimeStep`:

```
>>> env.step(1)    # action FIRE
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

Les attributs `reward` et `observation` n'ont pas besoin d'explications et sont identiques à ceux d'OpenAI Gym (toutefois, `reward` est représenté par un tableau NumPy). L'attribut `step_type` est égal à 0 pour la première étape temporelle de l'épisode, à 1 pour les étapes intermédiaires, et à 2 pour la dernière. Vous pouvez appeler la méthode `is_last()` de l'étape temporelle pour savoir s'il s'agit de l'étape finale. Enfin, l'attribut `discount` indique le taux de rabais utilisé lors de cette étape temporelle. Dans cet exemple, sa valeur 1 indique une absence de rabais. Pour définir le taux de rabais, il suffit de fixer le paramètre `discount` au moment du chargement de l'environnement.



À tout moment, vous avez accès à l'étape temporelle courante de l'environnement en appelant sa méthode `current_time_step()`.

10.12.3 Spécifications d'environnement

Un environnement TF-Agents fournit les spécifications des observations, actions et étapes temporelles, y compris les formes, types de données et noms, sans oublier les valeurs minimales et maximales :

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                  minimum=[[0. 0. 0.], [0. 0. 0.], ...]],
                  maximum=[[255., 255., 255.], [255., 255., 255.], ...]])
>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                  minimum=0, maximum=3)
>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
         discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

Vous le constatez, les observations sont simplement des captures de l'écran Atari, représentées comme des tableaux NumPy ayant la forme [210, 160, 3]. Pour effectuer le rendu d'un environnement, `env.render(mode="human")` peut être appelé. Si vous souhaitez récupérer l'image sous forme d'un tableau NumPy, appelez simplement `env.render(mode="rgb_array")` (contrairement à OpenAI Gym, il s'agit du mode par défaut).

Quatre actions sont disponibles. Les environnements Atari de Gym disposent d'une méthode supplémentaire qui permet de savoir à quoi correspond chaque action :

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```



Les spécifications peuvent être des instances d'une classe de spécification, des listes imbriquées ou des dictionnaires de spécifications. Dans le cas d'une spécification imbriquée, l'objet spécifié doit alors correspondre à la structure imbriquée de la spécification. Par exemple, si la spécification d'une observation est `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, une observation valide serait alors `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`. Le package `tf.nest` fournit des outils pour prendre en charge de telles structures imbriquées (appelées *nests*).

Puisque les observations sont relativement volumineuses, nous allons les sous-échantillonner et les convertir en niveaux de gris. L'entraînement s'en trouvera accéléré et l'occupation mémoire sera moindre. Pour cela, nous pouvons utiliser un *wrapper d'environnement*.

10.12.4 Wrapper d'environnement et prétraitement Atari

Dans le package `tf_agents.environments.wrappers`, TF-Agents fournit plusieurs wrappers d'environnement. Leur rôle est d'envelopper un environnement, en lui transmettant chaque appel, mais en lui ajoutant également une fonctionnalité supplémentaire. Voici quelques-uns des wrappers disponibles :

- `ActionClipWrapper`

Limite les actions à la spécification d'action.

- `ActionDiscretizeWrapper`

Quantifie un espace d'actions continu en un espace d'actions discret. Par exemple, si l'espace d'actions de l'environnement d'origine est la plage continue allant de -1,0 à +1,0, alors que vous souhaitez employer un algorithme qui reconnaît uniquement les espaces d'actions discrets, comme un DQN, vous pouvez envelopper l'environnement avec `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)`. Le `discrete_env` obtenu aura alors un espace d'actions discret avec cinq actions possibles : 0, 1, 2, 3, 4. Ces actions correspondent aux actions -1,0, -0,5, 0,0, 0,5 et 1,0 dans l'environnement d'origine.

- `ActionRepeat`

Répète chaque action sur n étapes, tout en accumulant les récompenses. Dans de nombreux environnements, cette technique permet d'accélérer l'entraînement de façon significative.

- `RunStats`

Enregistre des statistiques sur l'environnement, comme le nombre d'étapes et le nombre d'épisodes.

- `TimeLimit`

Interrompt l'environnement s'il s'exécute plus longtemps que le nombre maximal d'étapes précisé.

- `VideoWrapper`

Enregistre une vidéo de l'environnement.

Pour créer un environnement enveloppé, il suffit de créer un wrapper, en passant l'environnement concerné au constructeur. Voilà tout ! Par exemple, le code suivant enveloppe notre environnement dans un wrapper `ActionRepeat` afin que chaque action soit répétée à quatre reprises :

```
from tf_agents.environments.wrappers import ActionRepeat
repeating_env = ActionRepeat(env, times=4)
```

Au travers du package `gym.wrappers`, OpenAI Gym propose lui-même des wrappers. Toutefois, ils sont conçus pour envelopper non pas des environnements TF-Agents mais des environnements Gym. Par conséquent, pour les utiliser, vous devez commencer par envelopper l'environnement Gym à l'aide d'un wrapper Gym, puis envelopper l'environnement résultant avec un wrapper TF-Agents. La fonction `suite_gym.wrap_env()` se chargera de tout cela à votre place, à

condition que vous lui passiez un environnement Gym et une liste de wrappers Gym et/ou une liste de wrappers TF-Agents. Vous pouvez également utiliser la fonction `suite_gym.load()`, qui créera l'environnement Gym et l'enveloppera pour vous, si vous lui donnez des wrappers. Chaque wrapper sera créé sans aucun argument. Si vous souhaitez fixer des arguments, vous devez passer un `lambda`. Par exemple, le code suivant crée un environnement Breakout qui s'exécutera pendant 10 000 étapes maximum au cours de chaque épisode, et chaque action sera répétée quatre fois :

```
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

Dans la plupart des articles qui utilisent des environnements Atari, des étapes de prétraitement standard sont appliquées. C'est pourquoi TF-Agents fournit un wrapper `AtariPreprocessing` pratique implémentant ces étapes. Voici la liste des prétraitements pris en charge :

- *Niveaux de gris et sous-échantillonnage*

Les observations sont converties en niveaux de gris et sous-échantillonnées (par défaut à 84×84 pixels).

- *Pooling maximum*

Un pooling maximum avec un filtre 1×1 est appliqué aux deux dernières trames du jeu. Cela permet de retirer le clignotement qui se produit dans certains jeux Atari en raison du nombre limité de sprites que l'Atari 2600 est capable d'afficher dans chaque trame.

- *Saut de trame*

L'agent ne peut voir que n trames du jeu (par défaut, $n = 4$) et ses actions sont répétées pour chaque trame, en collectant toutes les récompenses. Du point de vue de l'agent, cela permet d'accélérer le jeu, et l'entraînement s'en trouve également accéléré car les récompenses sont moins retardées.

- *Fin en cas de vie perdue*

Dans certains jeux, les récompenses se fondent uniquement sur le score et l'agent ne reçoit aucune pénalité immédiate s'il perd une vie. Une solution consiste à terminer immédiatement le jeu dès qu'une vie est perdue. Puisque l'intérêt de cette stratégie est sujet à discussion, elle est désactivée par défaut.

L'environnement Atari par défaut applique déjà un saut de trame aléatoire et un pooling maximum. Nous devons donc charger la variante de base sans saut appelée "`BreakoutNoFrameskip-v4`". Par ailleurs, puisqu'une seule trame du jeu `Breakout` ne suffit pas à connaître la direction et la vitesse de la balle, l'agent aura beaucoup de mal à jouer correctement (sauf s'il s'agit d'un agent RNR, qui conserve un état interne entre les étapes). Pour traiter ce point, une solution consiste à utiliser un wrapper d'environnement qui produira des observations constituées de plusieurs trames empilées le long de la dimension des canaux. Cette stratégie est mise

en œuvre par le wrapper `FrameStack4`, qui retourne des piles de quatre trames. Créons l'environnement Atari enveloppé !

```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000    # <=> 108k trames ALE car 1 étape = 4 trames
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

Le résultat de tout ce prétraitement est illustré à la figure 10.12. Vous pouvez constater que la résolution est inférieure, mais suffisante pour jouer. Par ailleurs, puisque des trames sont empilées le long de la dimension des canaux, le rouge représente la trame datant de trois étapes antérieures, le vert, celle de deux étapes antérieures, le bleu, la trame précédente, et le rose, la trame actuelle²⁷³. À partir de cette seule observation, l'agent peut déterminer que la balle se dirige vers l'angle inférieur gauche et qu'il doit continuer à déplacer la raquette vers la gauche (comme il l'a fait lors des étapes précédentes).

Enfin, nous pouvons envelopper l'environnement dans un `TFPyEnvironment`:

```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```



Figure 10.12 – Observation Breakout prétraitée

273. Puisque nous avons uniquement trois couleurs primaires, nous ne pouvons pas afficher simplement une image avec quatre canaux de couleur. C'est pourquoi nous avons combiné le dernier canal avec les trois premiers de façon à obtenir l'image RVB représentée sur la figure. Le rose est en réalité un mélange de bleu et de rouge, mais l'agent voit quatre canaux indépendants. La figure en couleur est visible dans le notebook (voir « 18_reinforcement_learning.ipynb » sur <https://github.com/ageron/handson-ml2>).

L'environnement devient ainsi utilisable dans un graphique TensorFlow (cette classe se fonde sur `tf.py_function()`, qui permet à un graphique d'appeler du code Python quelconque). Grâce à la classe `TFPyEnvironment`, TF-Agents prend en charge à la fois des environnements Python purs et des environnements fondés sur TensorFlow. Plus généralement, TF-Agents prend en charge et fournit des composants Python purs et des composants basés sur TensorFlow (agents, mémoires de rejet, indicateurs, etc.).

Nous disposons d'un bel environnement Breakout, avec tout le prétraitement approprié et une prise en charge de TensorFlow. Nous devons à présent créer l'agent DQN et les autres composants qui permettront de l'entraîner. Examinons l'architecture du système que nous allons construire.

10.12.5 Architecture d'entraînement

En général, un programme d'entraînement TF-Agents est divisé en deux parties qui s'exécutent en parallèle, comme l'illustre la figure 10.13. Sur la gauche, un *pilote* explore l'*environnement* en utilisant une *politique de collecte* afin de choisir des actions, et il collecte des *trajectoires* (c'est-à-dire des expériences), en les envoyant à un *observateur* qui les enregistre dans une mémoire de rejet. Sur la droite, un *agent* extrait des lots de trajectoires à partir de la mémoire de rejet et entraîne des *réseaux*, utilisés par la politique de collecte. En résumé, la partie gauche explore l'environnement et collecte des trajectoires, tandis que la partie droite apprend et met à jour la politique de collecte.

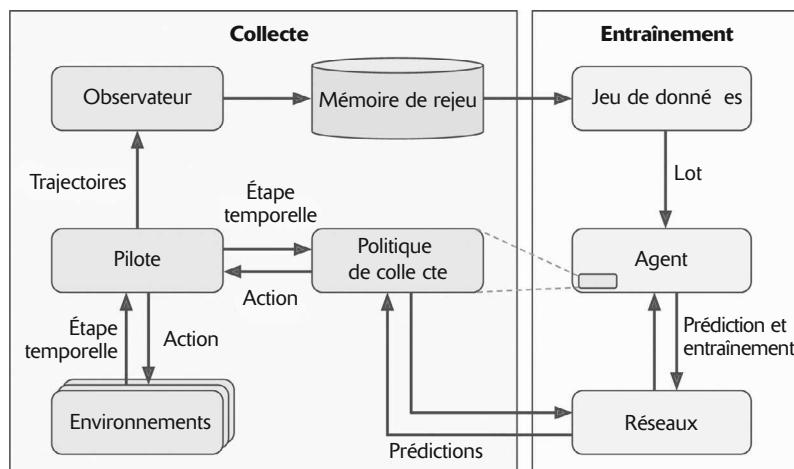


Figure 10.13 – Architecture d'entraînement type de TF-Agents

Cette figure soulève quelques questions, auxquelles nous allons tenter de répondre :

- Pourquoi y a-t-il plusieurs environnements ? Au lieu d'explorer un seul environnement, il est souvent préférable que le pilote explore plusieurs copies de l'environnement en parallèle, exploitant la puissance des coeurs du CPU,

gardant occupés les GPU dédiés à l’entraînement et fournissant à l’algorithme d’entraînement des trajectoires moins corrélées.

- Qu’est-ce qu’une *trajectoire*? Il s’agit d’une représentation concise de la transition d’une étape temporelle à la suivante, ou d’une série de transitions consécutives de l’étape temporelle n à l’étape temporelle $n + t$. Les trajectoires collectées par le pilote sont transmises à l’observateur, qui les enregistre dans une mémoire de rejet. Elles sont ensuite échantillonnées par l’agent et utilisées pour l’entraînement.
- Pourquoi avons-nous besoin d’un observateur? Le pilote ne peut-il pas enregistrer directement les trajectoires? Il le pourrait, évidemment, mais l’architecture perdrait en souplesse. Par exemple, et si vous ne souhaitez pas utiliser de mémoire de rejet? Et si vous souhaitez utiliser les trajectoires pour autre chose, comme calculer des indicateurs? En réalité, un observateur n’est qu’une fonction qui prend en argument une trajectoire. Vous pouvez utiliser un observateur pour placer les trajectoires dans une mémoire de rejet ou dans un fichier TFRecord (voir le chapitre 5), ou pour calculer des indicateurs, ou pour tout autre chose. Vous pouvez également passer plusieurs observateurs au pilote, à qui il diffusera les trajectoires.



Bien que cette architecture soit la plus courante, vous pouvez la personnaliser comme bon vous semble, et même remplacer certains composants par les vôtres. En réalité, sauf si vous travaillez sur de nouveaux algorithmes d’apprentissage par renforcement, vous voudrez probablement utiliser un environnement personnalisé pour votre tâche. Pour cela, vous devez simplement créer une classe personnalisée qui dérive de la classe `PyEnvironment` fournie par le package `tf_agents.environments.py_environment` et surcharger les méthodes appropriées, comme `action_spec()`, `observation_spec()`, `_reset()` et `_step()` (un exemple est donné dans la section «Creating a Custom TF-Agents Environment» du notebook²⁷⁴).

Nous allons à présent créer tous ces composants. Nous commencerons par le réseau Q profond, puis l’agent DQN (qui se chargera de créer la politique de collecte), puis la mémoire de rejet et l’observateur qui la remplira, puis quelques indicateurs d’entraînement, puis le pilote, et, pour finir, le dataset. Lorsque tous ces composants seront en place, nous remplirons la mémoire de rejet à l’aide de trajectoires initiales, puis nous lancerons la boucle d’entraînement principale. Tout d’abord, le réseau Q profond.

10.12.6 Créer le réseau Q profond

Dans le package `tf_agents.networks` et ses sous-packages, la bibliothèque TF-Agents fournit de nombreux réseaux. Nous choisissons la classe `tf_agents.networks.q_network.QNetwork`:

```
from tf_agents.networks.q_network import QNetwork
```

²⁷⁴. Voir «18_reinforcement_learning.ipynb» sur <https://github.com/ageron/handson-ml2>.

```

preprocessing_layer = keras.layers.Lambda(
    lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)

```

Ce `QNetwork` prend en entrée une observation et génère une valeur Q par action. Nous devons donc lui donner les spécifications des observations et des actions. Il commence par une couche de prétraitement : une simple couche `Lambda` qui transforme les observations en nombres à virgule flottante sur 32 bits et les normalise (les valeurs se situeront dans la plage 0,0 à 1,0). Les observations contiennent des octets non signés, qui occupent quatre fois moins d'espace que des flottants sur 32 bits. Voilà pourquoi nous n'avons pas converti les observations en flottants sur 32 bits plus tôt. Nous voulons économiser de la place dans la mémoire de jeu.

Ensuite, le réseau applique trois couches de convolution. La première possède 32 filtres 8×8 et utilise un pas de 4. La deuxième a 64 filtres 4×4 avec un pas de 2. La troisième comprend 64 filtres 3×3 avec un pas de 1. Pour finir, le réseau applique une couche dense de 512 unités, suivie d'une couche de sortie dense avec quatre unités, une par valeur Q (c'est-à-dire une par action). Toutes les couches de convolution et toutes les couches denses, à l'exception de la couche de sortie, utilisent par défaut la fonction d'activation ReLU (vous pouvez changer cela avec l'argument `activation_fn`). La couche de sortie n'applique aucune fonction d'activation.

Un `QNetwork` est constitué de deux parties : un réseau d'encodage qui traite les observations, suivi d'une couche de sortie dense qui produit une valeur Q par action. La classe `EncodingNetwork` de TF-Agents implémente une architecture de réseau de neurones que l'on retrouve dans différents agents (voir la figure 10.14).

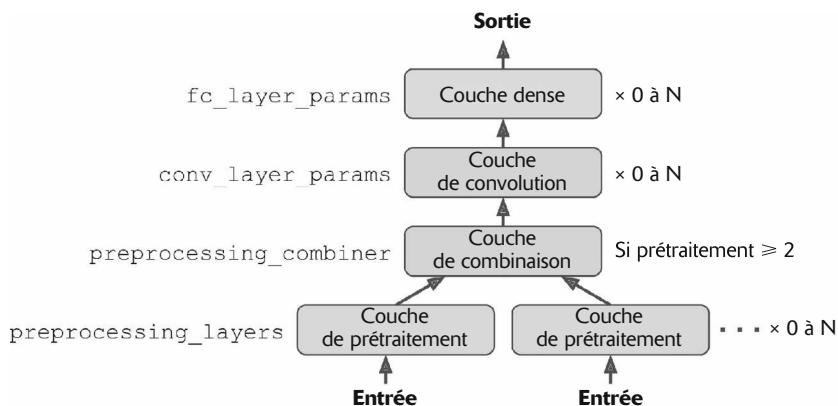


Figure 10.14 – Architecture d'un réseau d'encodage

Il peut avoir une entrée ou plus. Par exemple, si chaque observation comprend des données de capteurs et une image issue d'une caméra, vous aurez deux entrées. Chaque entrée peut nécessiter certaines étapes de prétraitement. Dans ce cas, vous pouvez indiquer une liste de couches Keras dans l'argument `preprocessing_layers`, avec une couche de prétraitement par entrée. Le réseau appliquera alors chaque couche à l'entrée correspondante (si une entrée requiert plusieurs couches de prétraitement, vous pouvez passer un modèle complet, puisqu'un modèle Keras peut toujours être utilisé en tant que couche). Si le nombre d'entrées est supérieur ou égal à deux, vous devez également passer une couche supplémentaire dans l'argument `preprocessing_combiner`, afin de réunir les sorties des couches de prétraitement en une seule sortie.

Le réseau d'encodage peut ensuite appliquer séquentiellement une liste de convolutions, mais vous devez préciser leurs paramètres dans l'argument `conv_layer_params`. Il suppose une liste constituée de trois tuples (un par couche de convolution) indiquant le nombre de filtres, la taille du noyau et le pas. Après ces couches de convolution, le réseau d'encodage peut appliquer une série de couches denses, si l'argument `fc_layer_params` a été précisé. Il s'agit d'une liste contenant le nombre de neurones de chaque couche dense. En option, vous pouvez également passer une liste de taux d'extinction (un par couche dense) à l'aide de l'argument `dropout_layer_params` de façon à appliquer un dropout après chaque couche dense. Le `QNetwork` prend la sortie de ce réseau d'encodage et la transmet à la couche de sortie dense (avec une unité par action).



La classe `QNetwork` est suffisamment flexible pour que vous puissiez construire de nombreuses architectures différentes, mais vous pouvez toujours développer votre propre classe de réseau si vous avez besoin d'une plus grande souplesse. Il suffit d'étendre la classe `tf_agents.networks.Network` et de l'implémenter comme une couche Keras personnalisée normale. La classe `tf_agents.networks.Network` dérive de la classe `keras.layers.Layer`, qui ajoute des fonctionnalités nécessaires à certains agents, comme la possibilité de créer des copies superficielles du réseau (autrement dit, copier l'architecture du réseau sans ses poids). Par exemple, `DQNAgent` utilise cette possibilité pour créer une copie du modèle en ligne.

Puisque nous disposons à présent du DQN, nous pouvons construire l'agent DQN.

10.12.7 Créer l'agent DON

La bibliothèque TF-Agents propose plusieurs types d'agents. Ils se trouvent dans le package `tf_agents.agents` et ses sous-packages. Nous optons pour la classe `tf_agents.agents.dqn.dqn_agent.DqnAgent`:

```

epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0,    # valeur initiale de  $\epsilon$ 
    decay_steps=250000 // update_period,    # <=> 1 000 000 trames ALE
    end_learning_rate=0.01)    # valeur finale de  $\epsilon$ 
agent = DqnAgent(tf_env.time_step_spec(),
                  tf_env.action_spec(),
                  q_network=q_net,
                  optimizer=optimizer,
                  target_update_period=2000,    # <=> 32 000 trames ALE
                  td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                  gamma=0.99,    # taux de rabais
                  train_step_counter=train_step,
                  epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()

```

Examinons ce code :

- Nous commençons par créer une variable qui comptera le nombre d'étapes d'entraînement.
- Puis nous construisons l'optimiseur, en utilisant les mêmes hyperparamètres que dans l'article de 2015 sur les DQN.
- Ensuite, nous créons un objet `PolynomialDecay` qui, étant donné l'étape d'entraînement courante, calculera la valeur ϵ pour la politique de collecte ϵ -greedy (il est normalement employé pour réduire le taux d'apprentissage, d'où les noms des arguments, mais il convient parfaitement à l'affaiblissement de n'importe quelle valeur). Elle ira de 1,0 à 0,01 (la valeur employée dans l'article de 2015) en un million de trames ALE. Cela correspond à 250 000 étapes, car nous utiliserons le saut de trame avec une période de 4. Par ailleurs, puisque nous entraînons l'agent toutes les quatre étapes (c'est-à-dire seize trames ALE), la décroissance de ϵ se fera en réalité sur 62 500 étapes d'entraînement.
- Nous construisons ensuite le `DQNAgent`, en lui passant l'étape temporelle et les spécifications de l'action, le `QNetwork` à entraîner, l'optimiseur, le nombre d'étapes d'entraînement entre les actualisations du modèle cible, la fonction de perte à employer, le taux de rabais, la variable `train_step` et une fonction qui retourne la valeur de ϵ (elle ne doit prendre aucun argument et c'est pourquoi nous avons besoin d'une couche `lambda` pour passer `train_step`).

Notez que la fonction de perte doit retourner non pas l'erreur moyenne mais une erreur par instance. Nous précisons donc `reduction="none"`.

- Pour finir, nous initialisons l'agent.

Passons maintenant à la mémoire de rejet et à l'observateur qui le remplira.

10.12.8 Créer la mémoire de rejet et l'observateur associé

Le package `tf_agents.replay_buffers` de la bibliothèque TF-Agents apporte différentes implémentations d'une mémoire de rejet. Certaines sont écrites en Python exclusivement (les noms des modules commencent par `py_`), tandis que d'autres sont développées à partir de TensorFlow (les noms des modules commencent

par `tf_`). Nous utilisons la classe `TFUniformReplayBuffer` du package `tf_agents.replay_buffers.tf_uniform_replay_buffer`. Elle fournit une implémentation efficace d'une mémoire de rejeu avec un échantillonnage uniforme²⁷⁵:

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

Examinons chacun de ces arguments :

- `data_spec`

La spécification des données qui seront enregistrées dans la mémoire de rejeu. L'agent DQN connaît la forme que prendront les données collectées et rend les spécifications des données disponibles au travers de l'attribut `collect_data_spec`. C'est donc ce que nous donnons à la mémoire de rejeu.

- `batch_size`

Le nombre de trajectoires qui seront ajoutées à chaque étape. Dans notre cas, il n'y en aura qu'une, car le pilote exécutera une action par étape et collectera une trajectoire. Si l'environnement était un *environnement par lots*, c'est-à-dire un environnement qui prend un lot d'actions à chaque étape et retourne un lot d'observations, alors le pilote devrait enregistrer un lot de trajectoires à chaque étape. Puisque nous utilisons une mémoire de rejeu TensorFlow, il doit connaître la taille des lots qu'il prendra en charge (pour construire le graphe de calcul). `ParallelPyEnvironment` (du package `tf_agents.environments.parallel_py_environment`) est un exemple d'environnement par lots. Il exécute plusieurs environnements en parallèle dans des processus séparés (ils peuvent être différents tant qu'ils ont les mêmes spécifications d'actions et d'observations) et, à chaque étape, prend un lot d'actions pour les exécuter dans les environnements (une action par environnement). Il retourne ensuite toutes les observations résultantes.

- `max_length`

La taille maximale de la mémoire de rejeu. Nous avons créé une grande mémoire de rejeu capable de stocker un million de trajectoires (comme c'était le cas dans l'article de 2015 sur les DQN). Cela nécessite une grande quantité de RAM.

²⁷⁵. Au moment de l'écriture de ces lignes, il n'existe aucune mémoire de rejeu avec expériences préférentielles, mais une implémentation open source devrait arriver sous peu.



Lorsque nous stockons deux trajectoires consécutives, elles contiennent deux observations consécutives de quatre trames chacune (puisque nous utilisons le wrapper `FrameStack4`). Malheureusement, trois des quatre trames de la deuxième observation sont redondantes (elles existent déjà dans la première). Autrement dit, nous utilisons environ quatre fois plus de mémoire que nécessaire. Pour éviter cela, vous pouvez opter à la place pour un `PyHashedReplayBuffer` venant du package `tf_agents.replay_buffers.py_hashed_replay_buffer`. Il permet de dé-dupliquer les données dans les trajectoires stockées le long du dernier axe des observations.

Nous pouvons à présent créer l'observateur qui placera les trajectoires dans la mémoire de rejet. Un observateur n'est rien d'autre qu'une fonction (ou un objet appellable) qui prend en argument une trajectoire. Nous pouvons donc utiliser directement la méthode `add_method()` (liée à l'objet `replay_buffer`) en tant qu'observateur :

```
replay_buffer_observer = replay_buffer.add_batch
```

Si vous voulez créer votre propre observateur, vous pouvez écrire n'importe quelle fonction qui prend un argument `trajectory`. Si elle doit gérer un état, vous pouvez développer une classe dotée d'une méthode `__call__(self, trajectory)`. Voici, par exemple, un observateur simple qui incrémente un compteur chaque fois qu'il est invoqué (excepté lorsque la trajectoire représente une frontière entre deux épisodes, qui ne compte donc pas comme une étape) et, toutes les 100 incrémentations, affiche la progression par rapport à un total donné (le retour chariot `\r` et `end=""` permettent de garder le compteur affiché sur la même ligne) :

```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Passons maintenant aux indicateurs d'entraînement.

10.12.9 Créer des indicateurs d'entraînement

Le package `tf_agents.metrics` de TF-Agents comprend plusieurs indicateurs RL, certains écrits en Python pur, d'autres fondés sur TensorFlow. Nous allons en créer quelques-uns de manière à dénombrer les épisodes et les étapes effectuées, et plus important encore, à calculer le rendement moyen par épisode et la longueur moyenne d'un épisode :

```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
```

```

    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]

```



Pour l'entraînement ou pour l'implémentation d'une politique, appliquer une réduction aux récompenses a du sens. En effet, cela permet de trouver un équilibre entre l'importance des récompenses immédiates et celle des récompenses futures. Toutefois, lorsqu'un épisode est terminé, nous pouvons évaluer sa qualité globale en effectuant l'addition des récompenses *sans rabais*. C'est pourquoi `AverageReturnMetric` calcule la somme des récompenses sans rabais pour chaque épisode et qu'il conserve la moyenne continue de ces sommes sur l'ensemble des épisodes rencontrés.

À tout moment, vous pouvez obtenir la valeur de chacun de ces indicateurs en invoquant sa méthode `result()` (par exemple, `train_metrics[0].result()`). Il est également possible de journaliser tous les indicateurs en appelant `log_metrics(train_metrics)` (cette fonction se trouve dans le package `tf_agents.eval.metric_utils`):

```

>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.get_logger().set_level(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0

```

Nous pouvons à présent créer le pilote de collecte.

10.12.10 Créez le pilote de collecte

Comme nous l'avons vu à la figure 10.13, un pilote est un objet qui explore un environnement conformément à une politique donnée, collecte des expériences et les diffuse à des observateurs. À chaque étape, voici ce qui se passe :

- Le pilote transmet l'étape temporelle courante à la politique de collecte, qui l'utilise pour choisir une action, qu'elle retourne dans un objet *étape d'action*.
- Le pilote passe ensuite l'action à l'environnement, qui retourne l'étape temporelle suivante.
- Enfin, le pilote crée un objet trajectoire pour représenter cette transition et le diffuse à tous les observateurs.

Certaines politiques, comme les politiques de RNR, gèrent un état. Elles choisissent une action en fonction de l'étape temporelle donnée et de leur propre état interne. Les politiques de ce type retournent leur propre état dans l'étape d'action, ainsi que l'action choisie. Le pilote repasse ensuite cet état à la politique pour l'étape temporelle suivante. Par ailleurs, le pilote enregistre l'état de la politique dans la trajectoire (dans le champ `policy_info`), qui finit donc pas arriver dans la mémoire

de rejeu. Ce fonctionnement est essentiel pour l'entraînement d'une politique avec état. Lorsque l'agent échantillonne une trajectoire, il doit replacer la politique dans l'état où elle se trouvait au moment de l'étape temporelle correspondante.

Nous l'avons également expliqué précédemment, l'environnement peut comprendre des lots, auquel cas le pilote passe à la politique une *étape temporelle par lots* (c'est-à-dire un objet d'étape temporelle qui contient un lot d'observations, un lot de types d'étapes, un lot de récompenses et un lot de rabais, tous ayant la même taille). Le pilote transmet également un lot des états de politique précédents. La politique retourne ensuite une *étape d'action par lots*, qui contient un lot d'actions et un lot d'états de politique. Enfin, le pilote crée une *trajectoire par lots* (c'est-à-dire une trajectoire contenant un lot de types d'étapes, un lot d'observations, un lot d'actions, un lot de récompenses et, plus généralement, un lot pour chaque attribut de trajectoire, tous les lots ayant la même taille).

Il existe deux principales classes de pilote: `DynamicStepDriver` et `DynamicEpisodeDriver`. La première collecte des expériences pour un nombre donné d'étapes, tandis que la seconde les collecte pour un nombre donné d'épisodes. Puisque nous voulons collecter des expériences pour quatre étapes de chaque itération d'entraînement (comme cela a été fait dans l'article de 2015 sur les DQN), nous créons un `DynamicStepDriver`:

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # Collecter 4 étapes pour chaque itération
                           # d'entraînement
```

Nous lui passons l'environnement de jeu, la politique de collecte de l'agent, une liste d'observateurs (y compris l'observateur de la mémoire de rejeu et les indicateurs d'entraînement), ainsi que le nombre d'étapes à exécuter (dans ce cas, quatre). Nous pourrions d'ores et déjà l'exécuter en invoquant sa méthode `run()`, mais il est préférable de remplir initialement la mémoire de rejeu à l'aide d'expériences collectées grâce à une politique purement aléatoire. Pour cela, nous pouvons utiliser la classe `RandomTFPolicy` et créer un second pilote qui exécutera cette politique pendant 20 000 étapes (ce qui équivaut aux 80 000 trames du simulateur, comme cela a été fait dans l'article de 2015). Notre observateur `ShowProgress` affichera la progression:

```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                         tf_env.action_spec())

init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80 000 trames ALE

final_time_step, final_policy_state = init_driver.run()
```

Nous sommes quasi prêts à exécuter la boucle d'entraînement ! Il nous reste un seul composant à créer, le dataset.

10.12.11 Créer le dataset

Pour échantillonner un lot de trajectoires à partir de la mémoire de rejet, il suffit d'invoquer sa méthode `get_next()`. Elle retourne le lot de trajectoires et un objet `BufferInfo` qui contient les identifiants des échantillons et leur probabilité d'échantillonnage (ces informations peuvent se révéler utiles, par exemple pour les algorithmes de PER). Le code suivant extrait un petit lot de deux trajectoires (sous-épisodes), contenant chacune trois étapes consécutives. Ces sous-épisodes sont illustrés à la figure 10.15 (chaque ligne contient trois étapes consécutives d'un épisode) :

```
>>> trajectories, buffer_info = replay_buffer.get_next()
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
 'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```



Figure 10.15 – Deux trajectoires contenant chacune trois étapes consécutives

L'objet `trajectories` est un tuple nommé, avec sept champs. Chaque champ contient un tenseur dont les deux premières dimensions sont 2 et 3 (puisque'il y

a deux trajectoires, chacune de trois étapes). Cela explique pourquoi la forme du champ *observation* est [2, 3, 84, 84, 4], c'est-à-dire deux trajectoires, chacune avec trois étapes, chaque étape d'observation faisant $84 \times 84 \times 4$. De façon comparable, la forme du tenseur *step_type* est [2, 3]. Dans cet exemple, les deux trajectoires contiennent trois étapes consécutives au milieu d'un épisode (types 1, 1, 1). Dans la seconde trajectoire, vous pouvez à peine voir la balle en partie inférieure gauche de la première observation, et elle disparaît dans les deux observations suivantes. L'agent est donc proche de perdre une vie, mais l'épisode ne va pas se terminer immédiatement car il lui reste encore plusieurs vies.

Chaque trajectoire est une représentation concise d'une série d'étapes temporelles consécutives et d'étapes d'action, conçue de façon à éviter la redondance. De quelle manière ? Comme le montre la figure 10.16, une transition n est constituée de l'étape temporelle n , de l'étape d'action n et de l'étape temporelle $n + 1$, tandis que la transition $n + 1$ comprend l'étape temporelle $n + 1$, l'étape d'action $n + 1$ et l'étape temporelle $n + 2$. Si nous nous contentons de stocker directement ces deux transitions dans la mémoire de rejet, l'étape temporelle $n + 1$ est dupliquée. Pour éviter cela, la $n^{\text{ème}}$ étape de trajectoire contient uniquement le type et l'observation de l'étape temporelle n (sans sa récompense et son rabais) et ne contient pas l'observation de l'étape temporelle $n + 1$ (bien qu'elle contienne une copie du type de l'étape temporelle suivante ; c'est la seule duplication).



Note : l'étape de trajectoire inclut également *next_step_type* $n = \text{step_type } n + 1$

Figure 10.16 – Trajectoires, transitions, étapes temporelles et étapes d'action

Par conséquent, si vous avez un lot de trajectoires, chacune comprenant $t + 1$ étapes (de l'étape temporelle n à l'étape temporelle $n + t$), il contient alors toutes les données de l'étape temporelle n à l'étape temporelle $n + t$, à l'exception de la récompense et du rabais à partir de l'étape temporelle n (mais contient la récompense et le rabais de l'étape temporelle $n + t + 1$). Cela représente t transitions (n à $n + 1$, $n + 1$ à $n + 2$, ..., $n + t - 1$ à $n + t$).

La fonction *to_transition()* du module *tf_agents.trajectories.trajectory* convertit une trajectoire par lots en une liste contenant un *time_*

step par lot, un action_step par lot et un next_time_step par lot. Notez que la deuxième dimension n'est plus 3 mais 2, car il y a t transitions entre $t + 1$ étapes temporelles (si cela vous semble confus, ne vous inquiétez pas, vous finirez par comprendre) :

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # 3 étapes temporelles = 2 transitions
```



Une trajectoire échantillonnée peut chevaucher deux épisodes (ou plus)! Dans ce cas, elle contiendra des *transitions limites*, autrement dit, des transitions avec step_type égal à 2 (fin) et next_step_type égal à 0 (début). Bien entendu, TF-Agents sait comment gérer de telles trajectoires (par exemple, en réinitialisant l'état de politique lorsqu'il rencontre une limite). La méthode `is_boundary()` de la trajectoire retourne un tenseur qui indique si chaque étape est une limite.

Pour notre boucle d'entraînement principale, nous allons non pas appeler la méthode `get_next()` mais utiliser un `tf.data.Dataset`. De cette manière, nous pourrons profiter des avantages de l'API Data (par exemple, parallélisme et prélecture). Pour cela, nous invoquons la méthode `as_dataset()` de la mémoire de rejeu :

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

Nous prenons des lots de 64 trajectoires à chaque étape d'entraînement (comme dans l'article de 2015), chacune de deux étapes (autrement dit, 2 étapes = 1 transition complète, y compris l'observation de l'étape suivante). Ce dataset traitera trois éléments en parallèle et effectuera une prélecture de trois lots.



Pour les algorithmes de politique en ligne, comme les gradients de politique, chaque expérience doit être échantillonnée une fois, utilisée pour l'entraînement, puis écartée. Dans ce cas, vous pouvez toujours utiliser une mémoire de rejeu, mais, à la place d'un Dataset, vous pouvez invoquer la méthode `gather_all()` du tampon à chaque itération d'entraînement de façon à obtenir un tenseur qui contient toutes les trajectoires enregistrées jusque-là, puis vous en servir pour effectuer une étape entraînement, et, enfin, vider la mémoire de rejeu en invoquant sa méthode `clear()`.

Tous les composants étant désormais en place, nous sommes prêts à entraîner le modèle !

10.12.12 Créer la boucle d'entraînement

Pour accélérer entraînement, nous convertissons les fonctions principales en fonctions TensorFlow. Pour cela, nous utilisons `tf_agents.utils.common.function()`, qui enveloppe `tf.function()` et ajoute des options expérimentales :

```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

Créons une petite fonction qui va exécuter la boucle d'entraînement principale pendant `n_iterations`:

```
def train_agent(n_iterations):
    time_step = None
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

Elle commence par demander à la politique de collecte son état initial (étant donné la taille de l'environnement qui, dans ce cas, vaut 1). Puisque la politique est sans état, nous obtenons un tuple vide (nous aurions donc pu écrire `policy_state = ()`). Nous créons ensuite un itérateur sur le dataset, puis nous démarrons la boucle d'entraînement. À chaque itération, nous invoquons la méthode `run()` du pilote, en lui passant l'étape temporelle courante (initialement `None`) et l'état actuel de la politique. Elle exécute la politique de collecte et récupère des expériences pour quatre étapes (comme configuré précédemment), en diffusant les trajectoires obtenues à la mémoire de rejeu, ainsi que les indicateurs. Ensuite, nous prenons un lot de trajectoires à partir du dataset et nous le passons à la méthode `train()` de l'agent. Elle retourne un objet `train_loss` qui peut varier en fonction du type d'agent. Puis, nous affichons le numéro d'itération et la perte d'entraînement. Toutes les 1 000 itérations, nous journalisons tous les indicateurs. Vous pouvez à présent juste appeler `train_agent()` en précisant le nombre d'itérations, et voir l'agent apprendre progressivement à jouer à *Breakout*!

```
train_agent(10000000)
```

Tout cela va demander une grande puissance de calcul et beaucoup de patience (il faudra des heures, voire des jours, en fonction du matériel), sans oublier que vous devrez peut-être exécuter l'algorithme à plusieurs reprises avec différents germes aléatoires pour obtenir de bons résultats. Mais, une fois l'entraînement terminé, l'agent sera devenu un surhomme (tout au moins pour *Breakout*). Vous pouvez également essayer d'entraîner cet agent DQN sur d'autres jeux Atari. Il sera capable d'acquérir des capacités surhumaines dans la plupart des jeux d'action, mais sera moins performant pour les jeux avec des sessions plus longues²⁷⁶.

276. La figure 3 de l'article publié en 2015 par DeepMind (<https://hml.info/dqn2>) compare les performances de cet algorithme sur différents jeux Atari.

10.13 QUELQUES ALGORITHMES RL POPULAIRES

Avant de terminer ce chapitre, passons brièvement en revue quelques algorithmes d'apprentissage par renforcement populaires :

- **Acteur-critique**

Il s'agit d'une famille d'algorithmes RL qui combinent les gradients de politique et les réseaux Q profonds. Un agent acteur-critique comprend deux réseaux de neurones : un réseau de politique et un DQN. Le DQN est entraîné de manière normale, en apprenant à partir des expériences de l'agent. Le réseau de politique apprend différemment (et beaucoup plus rapidement) d'un algorithme PG normal. Au lieu d'estimer la valeur de chaque action en parcourant plusieurs épisodes, puis en additionnant les récompenses à rabais futures pour chaque action, et, pour finir, en les normalisant, l'agent (acteur) se fonde sur les valeurs d'action estimées par le DQN (critique). C'est un peu comme un athlète (l'agent) qui apprend avec l'aide d'un coach (le DQN).

- **Acteur-critique asynchrone à avantages²⁷⁷ (A3C, Asynchronous Advantage Actor-Critic)**

Cette variante importante de l'acteur-critique a été présentée en 2016 par des chercheurs de DeepMind. De multiples agents apprennent en parallèle, en explorant différentes copies de l'environnement. À intervalles réguliers, mais de façon asynchrone, chaque agent envoie les actualisations de poids à un réseau maître, puis récupère les poids les plus récents auprès de ce réseau. Chaque agent contribue donc à l'amélioration du réseau maître et bénéficie des éléments appris par les autres agents. Par ailleurs, au lieu d'estimer les valeurs Q, le DQN estime l'avantage de chaque action, ce qui stabilise l'entraînement.

- **Acteur-critique à avantages²⁷⁸ (A2C, Advantage Actor-Critic)**

Cette variante de l'algorithme A3C retire l'asynchronisme. Puisque toutes les actualisations du modèle sont synchrones, les actualisations des gradients sont effectuées sur des lots plus grands. Cela permet au modèle de mieux profiter de la puissance du GPU.

- **Acteur-critique soft²⁷⁹ (SAC, Soft Actor-Critic)**

Il s'agit d'une variante de l'acteur-critique proposée en 2018 par Tuomas Haarnoja et d'autres chercheurs de l'université de Californie à Berkeley. Elle apprend non seulement les récompenses, mais maximise également l'entropie de ses actions. Autrement dit, elle tente d'être aussi imprévisible que possible

277. Volodymyr Mnih *et al.*, « Asynchronous Methods for Deep Reinforcement Learning », *Proceedings of the 33rd International Conference on Machine Learning* (2016), 1928-1937 : <https://homl.info/a3c>.

278. <https://homl.info/a2c>

279. Tuomas Haarnoja *et al.*, « Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor », *Proceedings of the 35th International Conference on Machine Learning* (2018), 1856-1865 : <https://homl.info/sac>.

tout en obtenant autant de récompenses que possible. Cela encourage l'agent à explorer l'environnement, accélérant ainsi l'entraînement, et réduit la probabilité qu'il exécute de façon répétée la même action lorsque le DQN produit des estimations imparfaites. Cet algorithme a démontré une efficacité étonnante (contrairement à tous les algorithmes antérieurs à SAC, qui apprennent très lentement). SAC est disponible dans TF-Agents.

- *Optimisation de politique proximale*²⁸⁰ (PPO, Proximal Policy Optimization)

Cet algorithme se fonde sur A2C mais coupe la fonction de perte de façon à éviter les actualisations excessivement importantes des poids (souvent sources d'instabilité de l'entraînement). PPO est une simplification de l'algorithme d'*optimisation de la politique de région de confiance*²⁸¹ (TRPO, Trust Region Policy Optimization), également proposé par John Schulman et d'autres chercheurs d'OpenAI. OpenAI a fait une en avril 2019 lorsque leur intelligence artificielle, appelée OpenAI Five et fondée sur l'algorithme PPO, a battu les champions du monde du jeu multijoueur Dota 2. PPO est également disponible dans TF-Agents.

- *Exploration fondée sur la curiosité*²⁸² (Curiosity-Based Exploration)

La rareté des récompenses est un problème récurrent de l'apprentissage par renforcement. L'apprentissage s'en trouve très lent et inefficace. Deepak Pathak et d'autres chercheurs de l'université de Californie à Berkeley ont proposé une manière très intéressante d'aborder ce problème : pourquoi ne pas ignorer les récompenses et rendre l'agent extrêmement curieux pour qu'il explore l'environnement ? Les récompenses ne viennent plus de l'environnement mais sont alors intrinsèques à l'agent. De façon comparable, la stimulation de la curiosité d'un enfant donnera probablement de meilleurs résultats qu'une simple récompense pour ses bonnes notes. Comment cette idée est-elle mise en œuvre ? L'agent tente en permanence de prédire le résultat de ses actions et recherche des situations dans lesquelles le résultat ne correspond pas à ses prédictions. Autrement dit, il veut être surpris. Si le résultat est prévisible (ennuyeux), il va ailleurs. Cependant, si le résultat est imprévisible et que l'agent remarque qu'il n'a aucun contrôle dessus, cela finit également par l'ennuyer. Grâce à la seule curiosité, les auteurs ont réussi à entraîner un agent dans de nombreux jeux vidéo. Même si l'agent n'est pas pénalisé lorsqu'il perd, le jeu recommence au début et cela finit par l'ennuyer. Il apprend donc à ne pas perdre.

280. John Schulman *et al.*, « Proximal Policy Optimization Algorithms » (2017) : <https://hml.info/ppo>.

281. John Schulman *et al.*, « Trust Region Policy Optimization », *Proceedings of the 32nd International Conference on Machine Learning* (2015), 1889-1897 : <https://hml.info/trpo>.

282. Deepak Pathak *et al.*, « Curiosity-Driven Exploration by Self-Supervised Prediction », *Proceedings of the 34th International Conference on Machine Learning* (2017), 2778-2787 : <https://hml.info/curiosity>.

Nous avons abordé de nombreux sujets dans ce chapitre : gradients de politique, chaînes de Markov, processus de décision markoviens, apprentissage Q, apprentissage Q par approximation et apprentissage Q profond, ainsi que ses principales variantes (cibles de la valeur Q fixées, DQN double, duel de DQN et rejeu avec expériences à priorités). Nous avons expliqué comment utiliser TF-Agents de façon à entraîner des agents à grande échelle, et, pour finir, nous avons décrit brièvement d'autres algorithmes répandus. L'apprentissage par renforcement est un domaine vaste et passionnant, avec de nouvelles idées et algorithmes surgissant quotidiennement. J'espère que ce chapitre a piqué votre curiosité : vous avez tout un monde à explorer !

10.14 EXERCICES

1. Comment définiriez-vous l'apprentissage par renforcement ? En quoi est-il différent d'un entraînement supervisé ou non supervisé classique ?
2. Imaginez trois applications possibles de l'apprentissage par renforcement que nous n'avons pas mentionnées dans ce chapitre. Pour chacune d'elles, décrivez l'environnement approprié, l'agent, les actions possibles et les récompenses.
3. Qu'est-ce que le taux de rabais ? La politique optimale change-t-elle si le taux de rabais est modifié ?
4. Comment pouvez-vous mesurer les performances d'un agent dans l'apprentissage par renforcement ?
5. Qu'est-ce que le problème d'affectation du crédit ? Quand survient-il ? Comment peut-il être réduit ?
6. Quel est l'intérêt d'utiliser une mémoire de rejeu ?
7. Qu'est-ce qu'un algorithme d'apprentissage par renforcement hors politique ?
8. Utilisez les gradients de politique pour résoudre l'environnement LunarLander-v2 d'OpenAI Gym.
Vous devrez installer les dépendances Box2D (`conda activate tf2`, puis `pip install gym[box2d]`).
9. En prenant l'un des algorithmes disponibles, utilisez TF-Agents pour entraîner un agent capable d'atteindre un niveau surhumain à SpaceInvaders-v4.
10. Si vous avez une centaine d'euros à dépenser, vous pouvez acheter un Raspberry Pi 3 et quelques composants robotiques bon marché, installer TensorFlow sur le Pi et partir à l'aventure ! Consultez, par exemple, le billet amusant publié par Lukas Biewald (<https://homl.info/2>) ou jetez un œil à GoPiGo ou à BrickPi. Commencez avec des objectifs simples, comme faire tourner le robot afin de trouver l'angle le plus lumineux (s'il est équipé d'un capteur lumineux) ou l'objet le plus proche (s'il est équipé d'un capteur à ultrasons) et de

le déplacer dans cette direction. Exploitez ensuite le Deep Learning. Par exemple, si le robot dispose d'une caméra, vous pouvez essayer d'implémenter un algorithme de détection d'objets afin qu'il repère les personnes et se dirige vers elles. Vous pouvez également essayer de mettre en place un apprentissage par renforcement de façon que l'agent apprenne lui-même comment utiliser ses moteurs pour atteindre cet objectif.

Amusez-vous !

Les solutions de ces exercices sont données à l'annexe A.

11

Entraînement et déploiement à grande échelle de modèles TensorFlow

Vous disposez d'un beau modèle qui réalise d'époustouflantes prédictions. Très bien, mais que pouvez-vous en faire ? Le mettre en production, évidemment ! Par exemple, vous pourriez tout simplement exécuter le modèle sur un lot de données, en écrivant éventuellement un script qui lance la procédure chaque nuit. Cependant, la mise en production est souvent beaucoup plus complexe. Il est possible que différentes parties de votre infrastructure aient besoin d'appliquer le modèle sur des données en temps réel, auquel cas vous voudrez probablement l'intégrer dans un service web. De cette manière, n'importe quelle partie de l'infrastructure peut interroger le modèle à tout moment en utilisant une simple API REST (ou tout autre protocole)²⁸³.

Mais, au bout d'un certain temps, il faudra certainement réentraîner le modèle sur des données récentes et mettre cette version actualisée en production. Vous devez donc assurer la gestion des versions du modèle, en proposant une transition en douceur d'une version à la suivante, avec la possibilité éventuelle de revenir au modèle précédent en cas de problème, voire exécuter plusieurs modèles différents en parallèle pour effectuer des tests A/B²⁸⁴. En cas de succès de votre produit, votre service va commencer à recevoir un grand nombre de *requêtes par seconde* (QPS, *queries per second*) et devra changer d'échelle pour supporter la charge. Nous le verrons dans ce chapitre, une bonne solution pour augmenter l'échelle de votre service consiste à utiliser TF Serving, soit sur votre propre infrastructure matérielle, soit au travers d'un service de cloud comme Google Cloud AI Platform. TF Serving se chargera de servir votre modèle de façon efficace, de traiter les changements de version en

283. Voir le chapitre 2 de l'ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

284. Un test A/B consiste à proposer deux versions différentes d'un produit à différents groupes d'utilisateurs afin de déterminer celle qui fonctionne le mieux et d'obtenir d'autres informations.

douceur, et de bien d'autres aspects. Si vous optez pour une plateforme de cloud, vous bénéficieriez également de nombreuses autres fonctionnalités, comme des outils de supervision puissants.

Par ailleurs, si la quantité de données d'entraînement est importante et si les modèles demandent des calculs intensifs, le temps d'entraînement risque d'être extrêmement long. Dans le cas où votre produit doit s'adapter rapidement à des changements, une durée d'entraînement trop longue risque d'être rédhibitoire (imaginez, par exemple, un système de recommandations d'informations qui promeut des nouvelles de la semaine précédente). Peut-être plus important encore, un entraînement trop long pourrait vous empêcher d'expérimenter de nouvelles idées. Dans le domaine du Machine Learning, comme dans bien d'autres, il est difficile de savoir à l'avance les idées qui fonctionneront. Vous devez donc en essayer autant que possible, aussi rapidement que possible. Une manière d'accélérer l'entraînement consiste à employer des accélérateurs matériels, comme des GPU ou des TPU. Pour aller encore plus vite, vous pouvez entraîner un modèle sur plusieurs machines, chacune équipée de multiples accélérateurs matériels. L'API Distribution Strategies de TensorFlow, simple et néanmoins puissante, facilite une telle mise en place.

Dans ce chapitre, nous verrons comment déployer des modèles, tout d'abord avec TF Serving, puis avec Google Cloud AI Platform. Nous expliquerons brièvement comment déployer des modèles sur des applications mobiles, des dispositifs embarqués et des applications web. Enfin, nous montrerons comment accélérer les calculs en utilisant des GPU et comment entraîner des modèles sur de multiples processeurs et serveurs à l'aide de l'API Distribution Strategies. Cela fait beaucoup de sujets à traiter, commençons tout de suite !

11.1 SERVIR UN MODÈLE TENSORFLOW

Après avoir entraîné un modèle TensorFlow, vous pouvez aisément l'utiliser dans n'importe quel code Python. S'il s'agit d'un modèle tf.keras, il suffit d'invoquer sa méthode `predict()` ! Mais, avec l'expansion de votre infrastructure, viendra le moment où il sera préférable d'intégrer le modèle dans un petit service, dont le seul rôle sera d'effectuer des prédictions, et de laisser le reste de l'infrastructure l'interroger (par exemple, au travers d'une API REST ou gRPC)²⁸⁵. Le modèle est ainsi découplé des autres parties de l'infrastructure, ce qui facilite le changement de version du modèle ou le dimensionnement du service en fonction des besoins (indépendamment du reste de l'infrastructure), la mise en place de tests A/B et l'homogénéité des versions du modèle pour tous les composants logiciels. Cela simplifie également les tests et les développements. Vous pouvez créer votre propre microservice en utilisant la technologie de votre choix (par exemple, la bibliothèque Flask), mais pourquoi réinventer la roue alors que vous pouvez bénéficier de TF Serving ?

285. Une API REST (ou RESTful) utilise des verbes HTTP, comme GET, POST, PUT et DELETE, et des entrées et des sorties au format JSON. Le protocole gRPC est plus complexe, mais aussi plus efficace. Les échanges de données se font avec Protocol Buffers (voir le chapitre 5).

11.1.1 Utiliser TensorFlow Serving

TF Serving est un serveur de modèles efficace et éprouvé, écrit en C++. Il est capable de supporter une charge élevée, de servir plusieurs versions de vos modèles, de surveiller un dépôt de modèles de façon à déployer automatiquement les dernières versions, etc. (voir la figure 11.1).

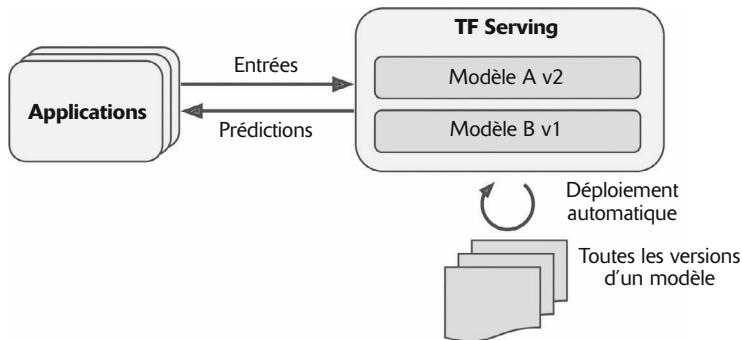


Figure 11.1 – TF Serving peut servir de multiples modèles et déployer automatiquement leur dernière version

Supposons que vous ayez entraîné un modèle MNIST avec tf.keras et que vous souhaitiez le déployer avec TF Serving. La première chose à faire est d'exporter ce modèle au format SavedModel de TensorFlow.

Exporter des SavedModel

TensorFlow fournit une fonction simple, `tf.saved_model.save()`, qui permet d'exporter des modèles au format SavedModel. Elle ne requiert que le modèle, son nom et son numéro de version. Elle enregistre son graphe de calcul et ses poids :

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit(...)

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Vous pouvez également invoquer la méthode `save()` du modèle (`model.save(model_path)`), à condition que l'extension du fichier soit autre que `.h5`. Le modèle sera alors enregistré non pas au format HD5F mais au format SavedModel.

En général, il est préférable d'inclure toutes les couches de prétraitement dans le modèle final qui sera exporté afin qu'il puisse ingérer les données sous leur forme naturelle après sa mise en production. Cela évite d'inclure une autre prise en charge du prétraitement dans l'application qui utilise le modèle. La mise à jour ultérieure des

étapes de prétraitement s'en trouve simplifiée, et les risques d'incohérence entre un modèle et les prétraitements dont il a besoin sont réduits.



Puisqu'un SavedModel comprend le graphe de calcul, il ne peut être utilisé qu'avec des modèles fondés exclusivement sur des opérations TensorFlow, excluant ainsi l'opération `tf.py_function()` (qui enveloppe du code Python quelconque). Cela exclut également les modèles tf.keras dynamiques (voir l'annexe E), car ceux-ci ne peuvent pas être convertis en graphes de calcul. Ils doivent donc être servis à l'aide d'autres outils (par exemple, Flask).

Un SavedModel représente une version du modèle. Il est enregistré sous forme d'un répertoire contenant un fichier `saved_model.pb`, qui définit le graphe de calcul (représenté sous forme d'un protobuf sérialisé), et un sous-répertoire `variables`, qui contient les valeurs des variables. Pour les modèles comprenant un grand nombre de poids, ces valeurs de variables peuvent être réparties dans plusieurs fichiers. Un SavedModel peut également inclure un sous-répertoire `assets` dans lequel se trouvent des données supplémentaires, comme des fichiers de vocabulaire, des noms de classes ou des instances d'exemples destinés au modèle. La structure du répertoire est la suivante (dans cet exemple, les informations complémentaires ne sont pas utilisées) :

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

Le chargement d'un SavedModel se fait avec la fonction `tf.saved_model.load()`. Cependant, l'objet retourné est non pas un modèle Keras mais une représentation du SavedModel, avec son graphe de calcul et ses valeurs de variables. Vous pouvez utiliser cet objet comme une fonction afin d'obtenir des prédictions (n'oubliez pas de passer les entrées sous forme de tenseurs du type approprié) :

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(tf.constant(X_new, dtype=tf.float32))
```

Vous pouvez également charger directement ce SavedModel en appelant la fonction `keras.models.load_model()` :

```
model = keras.models.load_model(model_path)
y_pred = model.predict(tf.constant(X_new, dtype=tf.float32))
```

TensorFlow fournit un petit outil en ligne de commande appelé `saved_model_cli` pour inspecter les SavedModel :

```
$ export ML_PATH="$HOME/ml"      # Pointe sur ce projet, où qu'il se trouve
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['__saved_model_init_op']:
[...]
```

```

signature_def['serving_default']:
    The given SavedModel SignatureDef contains the following input(s):
        inputs['flatten_input'] tensor_info:
            dtype: DT_FLOAT
            shape: (-1, 28, 28)
            name: serving_default_flatten_input:0
    The given SavedModel SignatureDef contains the following output(s):
        outputs['dense_1'] tensor_info:
            dtype: DT_FLOAT
            shape: (-1, 10)
            name: StatefulPartitionedCall:0
    Method name is: tensorflow/serving/predict

```

Un SavedModel contient un ou plusieurs *métagraphes*. Un métagraphe est un graphe de calcul accompagné de définitions de signatures de fonctions (y compris les noms de leurs entrées et sorties, les types et les formes). Chaque métagraphe est identifié par un ensemble d'étiquettes. Par exemple, vous pourriez souhaiter avoir un métagraphe qui contient l'intégralité du graphe de calcul, avec les opérations d'entraînement (celui-ci pourrait être libellé "train"), et un second contenant un graphe de calcul élagué avec uniquement les opérations de prédiction, y compris certaines opérations propres au GPU (celui-ci pourrait être libellé "serve", "gpu"). Cependant, lorsque vous passez un modèle tf.keras à la fonction `tf.saved_model.save()`, elle enregistre par défaut un SavedModel beaucoup plus simple. Il comprend un seul métagraphe libellé "serve", avec deux définitions de signatures, une fonction d'initialisation (nommée `__saved_model_init_op` et que vous pouvez ignorer) et une fonction de service par défaut (nommée `serving_default`). Lors de l'enregistrement d'un modèle tf.keras, cette fonction de service par défaut correspond à la fonction `call()` du modèle qui, évidemment, effectue des prédictions.

L'outil `saved_model_cli` peut également servir à effectuer des prédictions (essentiellement pour des tests). Supposons que vous disposez d'un tableau NumPy (`X_new`) contenant trois images de chiffres écrits à la main et pour lesquelles vous souhaitez obtenir des prédictions. Vous devez commencer par les exporter au format npy de NumPy :

```
np.save("my_mnist_tests.npy", X_new)
```

Exécutez ensuite la commande `saved_model_cli` de la manière suivante :

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
--signature_def serving_default \
--inputs flatten_input=my_mnist_tests.npy
[...] Result for output key dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

La sortie produite par l'outil contient les dix probabilités de classe de chacune des trois instances. Parfait ! Vous disposez à présent d'un SavedModel opérationnel. L'étape suivante consiste à installer TF Serving.

Installer TensorFlow Serving

Il existe différentes manières d'installer TF Serving: à partir d'une image Docker²⁸⁶, du gestionnaire de packages du système, des fichiers sources, etc. Choisissons l'option Docker, que l'équipe TensorFlow recommande fortement, car l'installation est simple, ne perturbe pas le système et apporte de hautes performances. Vous devez commencer par installer Docker (<https://docker.com/>). Puis, téléchargez l'image Docker officielle de TF Serving:

```
$ docker pull tensorflow/serving
```

Vous pouvez alors créer un conteneur Docker pour exécuter cette image:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
-v '$ML_PATH/my_mnist_model:/models/my_mnist_model' \
-e MODEL_NAME=my_mnist_model \
tensorflow/serving
```

```
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

Voilà tout! TF Serving est en cours d'exécution. Il chargera notre modèle MNIST (version 1) et le proposera au travers de gRPC (sur le port 8500) et de REST (sur le port 8501). Voici l'action des options de ligne de commande:

- **-it**

Rend le conteneur interactif (vous pouvez appuyer sur Ctrl-C pour l'arrêter) et affiche la sortie du serveur.

- **--rm**

Supprime le conteneur lorsque vous l'arrêtez (il est inutile d'encombrer votre machine avec des conteneurs interrompus). Toutefois, l'image n'est pas supprimée.

- **-p 8500:8500**

Fait en sorte que le moteur Docker redirige le port TCP 8500 de l'hôte vers le port TCP 8500 du conteneur. Par défaut, TF Serving utilise ce port pour la prise en charge de l'API gRPC.

- **-p 8501:8501**

Redirige le port TCP 8501 de l'hôte vers le port TCP 8501 du conteneur. Par défaut, TF Serving utilise ce port pour la prise en charge de l'API REST.

- **-v "\$ML_PATH/my_mnist_model:/models/my_mnist_model"**

Rend le répertoire \$ML_PATH/my_mnist_model de l'hôte disponible au conteneur *via* le chemin /models/mnist_model. Sur les systèmes Windows, vous

286. Si vous ne connaissez pas Docker, sachez que cet outil vous permet de télécharger facilement un ensemble d'applications préparées sous forme d'une *image Docker* (avec toutes les dépendances et, en général, dans une configuration par défaut adéquate) et de les exécuter sur votre système à l'aide d'un *moteur Docker*. Lorsque vous exéutez une image, le moteur crée un *conteneur Docker* qui assure une parfaite isolation entre les applications et votre propre système (vous pouvez lui accorder un accès limité si vous le souhaitez). Le conteneur est comparable à une machine virtuelle, mais il est beaucoup plus rapide et plus léger, car il se fonde directement sur le noyau de l'hôte. L'image n'a donc pas besoin d'inclure ni d'exécuter son propre noyau.

devrez remplacer / par \ dans le chemin de l'hôte (mais pas dans le chemin du conteneur).

- `-e MODEL_NAME=my_mnist_model`

Fixe la valeur de la variable d'environnement MODEL_NAME du conteneur afin que TF Serving sache quel modèle servir. Par défaut, il recherchera des modèles dans le répertoire `/models` et servira automatiquement la dernière version trouvée.

- `tensorflow/serving`

Le nom de l'image à exécuter.

Revenons à présent à Python et interrogeons ce serveur, tout d'abord par le biais de l'API REST, puis de l'API gRPC.

Interroger TF Serving avec l'API REST

Commençons par créer la requête. Elle doit contenir le nom de la signature de la fonction à invoquer et, bien entendu, les données d'entrée:

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Notez que le format JSON est exclusivement textuel et donc que le tableau NumPy `X_new` doit être converti en une liste Python et mis au format JSON:

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

Transmettons les données d'entrée à TF Serving en envoyant une requête HTTP POST. La bibliothèque `requests` facilite cette opération (elle ne fait pas partie de la bibliothèque standard de Python, mais nous l'avons installée au chapitre 1, lors de la création de l'environnement `conda tensorflow`):

```
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # Lève une exception en cas d'erreur
response = response.json()
```

La réponse est un dictionnaire qui contient une seule clé "predictions". La valeur correspondante est la liste des prédictions. Il s'agit d'une liste Python, que nous convertissons en tableau NumPy et dont nous arrondissons les valeurs réelles à la deuxième décimale:

```
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Super, nous avons les prédictions ! Le modèle est presque sûr à 100 % que la première image est un 7, sûr à 99 % que la deuxième est un 2 et sûr à 96 % que la troisième est un 1.

L'API REST est belle et simple, et fonctionne parfaitement lorsque les données d'entrée et de sortie ne sont pas trop volumineuses. Par ailleurs, quasiment n'importe quelle application cliente peut effectuer des requêtes REST sans compléments logiciels, alors que la disponibilité d'autres protocoles est moindre. Toutefois, elle se fonde sur JSON, un format textuel plutôt verbeux. Par exemple, nous avons dû convertir le tableau NumPy en une liste Python et chaque nombre à virgule flottante a été représenté sous forme d'une chaîne de caractères. Cette approche est très inefficace, à la fois en temps de sérialisation/désérialisation (pour convertir tous les réels en chaînes de caractères, et inversement) et en occupation mémoire, car de nombreuses valeurs ont été représentées avec plus de quinze caractères, ce qui correspond à plus de 120 bits pour des nombres à virgule flottante sur 32 bits ! Cela conduira à une latence élevée et à une occupation importante de la bande passante pour le transfert de tableaux NumPy de grande taille²⁸⁷. Utilisons à la place gRPC.



Pour le transfert d'une grande quantité de données, il est préférable d'utiliser l'API gRPC (si le client la prend en charge), car elle se fonde sur un format binaire compact et un protocole de communication efficace (à base de trames HTTP/2).

Interroger TF Serving avec l'API gRPC

L'API gRPC attend en entrée un protobuf sérialisé `PredictRequest` et produit en sortie un protobuf sérialisé `PredictResponse`. Ces protobufs font partie de la librairie `tensorflow-serving-api`, que nous avons installée au chapitre 1, lors de la création de l'environnement conda `tf2`. Commençons par créer la requête :

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

Ce code crée un protobuf `PredictRequest` et remplit les champs requis, y compris le nom du modèle (défini précédemment), le nom de la signature de la fonction à appeler et les données d'entrée, sous la forme d'un protobuf `Tensor`. La fonction `tf.make_tensor_proto()` crée ce protobuf `Tensor` à partir du tenseur ou du tableau NumPy donné, dans ce cas `X_new`.

287. Pour être honnête, ce problème peut être réduit en sérialisant les données et en les encodant au format Base64 préalablement à la création de la requête REST. Par ailleurs, les requêtes REST peuvent être compressées avec gzip, ce qui réduit énormément la taille des informations transmises.

Nous transmettons ensuite la requête au serveur et obtenons sa réponse (pour cela, nous utilisons la bibliothèque `grpcio`, installée au chapitre 1, lors de la création de l'environnement `conda tensorflow2`):

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

Ce code n'a rien de compliqué. Après les importations, nous créons un canal de communication gRPC avec `localhost` sur le port TCP 8500, puis un service gRPC sur ce canal. Nous l'utilisons pour envoyer une requête, avec un délai d'expiration de 10 secondes (l'appel étant synchrone, il restera bloqué jusqu'à la réception de la réponse ou l'expiration du délai). Dans cet exemple, le canal n'est pas sécurisé (aucun chiffrement, aucune authentification), mais gRPC et TensorFlow Serving prennent en charge les canaux au-dessus de SSL/TLS.

Convertissons ensuite le protobuf `PredictResponse` en un tenseur:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Si vous exécutez ce code et affichez `y_proba.numpy().round(2)`, vous obtiendrez exactement les mêmes probabilités de classe estimées que précédemment. C'est là tout l'intérêt de la chose. En quelques lignes de code, vous pouvez à présent accéder à distance à votre modèle TensorFlow, en utilisant REST ou gRPC.

Déployer une nouvelle version d'un modèle

Créons à présent une nouvelle version du modèle et exportons comme précédemment un `SavedModel` dans le répertoire `my_mnist_model/0002`:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit(...)

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

À intervalles réguliers (le délai est configurable), TensorFlow Serving vérifie l'existence de nouvelles versions d'un modèle. S'il en trouve une, il prend automatiquement en charge la transition. Par défaut, il répond aux éventuelles requêtes en attente avec la version précédente du modèle, tout en traitant les nouvelles requêtes avec la nouvelle version²⁸⁸. Dès que toutes les requêtes en attente ont été traitées, la

²⁸⁸. Si le `SavedModel` contient des instances d'exemples dans le répertoire `assets/extras`, vous pouvez configurer TF Serving pour qu'il exécute le modèle sur ces instances avant qu'il ne commence à traiter les nouvelles requêtes. Il s'agit de l'*échauffement du modèle*. Cela permet de s'assurer que tout est correctement chargé, en évitant de longs temps de réponse pour les premières requêtes.

version précédente du modèle est déchargée. Vous pouvez constater ces opérations dans les journaux de TensorFlow Serving :

```
[...]
reserved resources to load servable {name: my_mnist_model version: 2}
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

Cette approche permet une transition en douceur, mais elle risque d'exiger une très grande quantité de mémoire (en particulier la RAM du GPU, qui est généralement la plus limitée). Dans ce cas, vous pouvez configurer TF Serving pour qu'il traite les requêtes en attente avec la version précédente du modèle et décharge celle-ci avant de charger et d'utiliser la nouvelle. Cette configuration évitera la présence simultanée des deux versions du modèle en mémoire, mais le service sera indisponible pendant une courte période.

Vous le constatez, TF Serving simplifie énormément le déploiement de nouveaux modèles. De plus, si vous découvrez que la version 2 ne fonctionne pas comme attendu, le retour à la version 1 consiste simplement à supprimer le répertoire *my_mnist_model/0002*.



TF Serving dispose d'une autre fonctionnalité très intéressante : la mise en lots automatique. Pour l'activer, ajoutez l'option `--enable_batching` lors du lancement. Lorsque TF Serving reçoit plusieurs requêtes en une courte période de temps (le délai est configurable), il les regroupe automatiquement avant d'invoquer le modèle. Cela permet d'améliorer énormément les performances en exploitant la puissance du GPU. Après que le modèle a retourné les prédictions, TF Serving renvoie chaque prédition au client approprié. Vous pouvez accepter une latence légèrement plus élevée dans le but d'obtenir un débit supérieur en augmentant le délai de regroupement (voir l'option `--batching_parameters_file`).

Si vous pensez que le nombre de requêtes par seconde sera très élevé, vous pouvez déployer TF Serving sur plusieurs serveurs et répartir les requêtes de façon équilibrée (voir la figure 11.2). Il faudra pour cela déployer et gérer de nombreux nœuds TF Serving sur ces serveurs. Pour vous y aider, tournez-vous vers un outil comme Kubernetes (<https://kubernetes.io/>). Il s'agit d'un système open source qui simplifie l'orchestration de nœuds sur de nombreux serveurs. Si vous ne souhaitez pas acheter, maintenir et mettre à niveau toute l'infrastructure matérielle, vous pouvez opter pour des machines virtuelles sur une plateforme de cloud, comme Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud ou toute autre plateforme en tant que service (PaaS, *Platform-as-a-Service*). La gestion de toutes les machines virtuelles, l'orchestration des nœuds (même avec l'aide de Kubernetes), la configuration, le réglage et la supervision de TF Serving, tout cela peut être un travail à plein temps.

Heureusement, certains fournisseurs de services proposent de s'en occuper à votre place. Dans ce chapitre, nous allons utiliser Google Cloud AI Platform car elle est, aujourd'hui, la seule plateforme à disposer de TPU, elle est compatible avec TensorFlow 2, elle propose une suite de services AI intéressants (par exemple, AutoML, Vision API, Natural Language API), et elle m'est familière. Mais il existe bon nombre d'autres fournisseurs, comme Amazon AWS SageMaker et Microsoft AI Platform, également capables de servir des modèles TensorFlow.

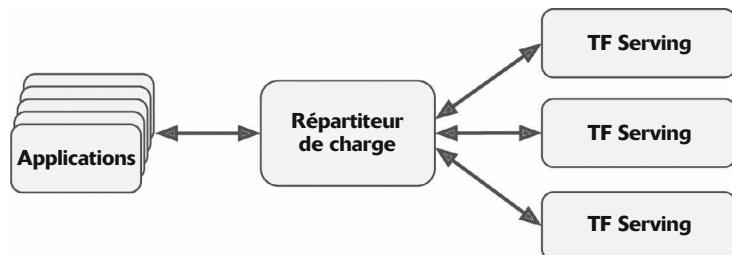


Figure 11.2 – Répartition de charge avec TF Serving

Voyons à présent comment servir notre merveilleux modèle MNIST depuis le cloud !

11.1.2 Créez un service de prédition sur la plateforme GCP AI

Avant de pouvoir déployer un modèle, une phase de préparation est nécessaire :

1. Connectez-vous à votre compte Google, puis rendez-vous sur la console Google Cloud Platform (GCP) à l'adresse <https://console.cloud.google.com/> (voir la figure 11.3). Si vous ne disposez d'aucun compte Google, vous devrez en créer un.

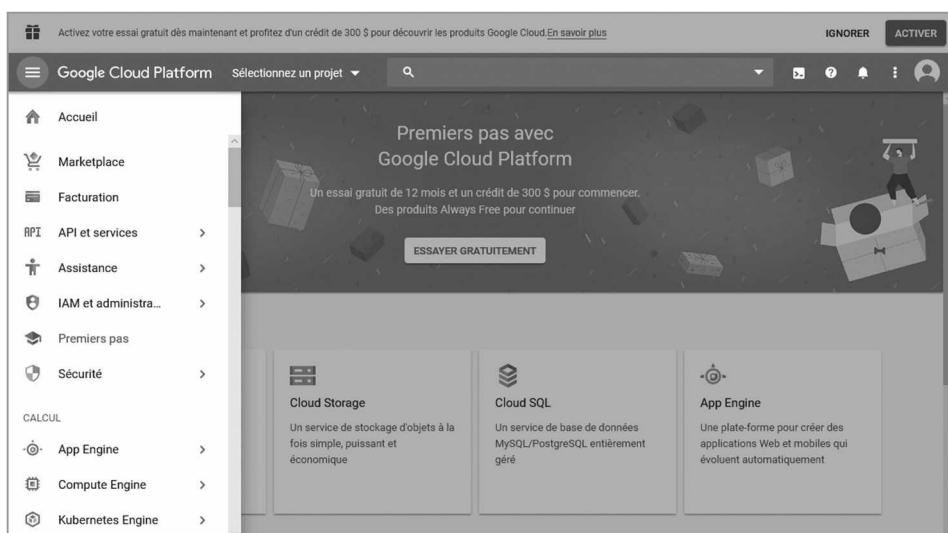


Figure 11.3 – Console de Google Cloud Platform

2. S'il s'agit de votre première utilisation de GCP, vous devrez lire et accepter les conditions d'utilisation. Au moment de l'écriture de ces lignes, il est proposé aux nouveaux utilisateurs un crédit de 300 \$ valable pendant douze mois sur tous les produits GCP. Vous n'aurez besoin que d'une petite partie de cette somme pour payer les services que vous utiliserez dans ce chapitre. Après avoir accepté l'essai gratuit, vous devrez créer un profil de paiement et saisir le numéro de votre carte bancaire. Elle est utilisée pour des vérifications (probablement pour éviter plusieurs inscriptions à l'essai gratuit), mais vous ne serez pas facturé. Activez et mettez à jour votre compte si cela vous est demandé.
3. Si vous avez déjà employé GCP et si votre période d'essai gratuit a expiré, les services utilisés dans ce chapitre vous coûteront un peu d'argent. Le montant sera faible, notamment si vous pensez à désactiver les services lorsque vous n'en avez plus besoin. Avant d'exécuter un service, assurez-vous de bien comprendre les conditions tarifaires, puis acceptez-les. Je décline toute responsabilité si des services finissent par vous coûter plus qu'attendu ! Vérifiez également que votre compte de facturation est actif. Pour cela, ouvrez le menu de navigation et cliquez sur Facturation. Assurez-vous d'avoir configuré une méthode de paiement et que le compte de facturation est actif.
4. Dans GCP, toutes les ressources sont associées à un projet. Cela comprend toutes les machines virtuelles que vous utilisez, les fichiers que vous stockez et les tâches d'entraînement que vous exécutez. Lorsque vous créez un compte, GCP crée automatiquement un projet intitulé « My First Project ». Vous pouvez changer son nom d'affichage en allant dans les paramètres du projet. Pour cela, dans le menu de navigation, choisissez IAM et administration → Paramètres, modifiez le nom du projet et cliquez sur Enregistrer. Notez que le projet possède également un identifiant et un numéro unique. L'identifiant peut être choisi au moment de la création du projet, mais il ne peut plus être changé ensuite. Le numéro du projet est généré automatiquement et ne peut pas être modifié. Pour créer un nouveau projet, cliquez sur Sélectionnez un projet, puis sur Nouveau Projet. Saisissez le nom du projet, modifiez éventuellement son ID, et cliquez sur Créer. Vérifiez que la facturation est active sur ce nouveau projet.



Lorsque vous savez que vous n'aurez besoin de services que pour quelques heures, définissez toujours une alarme pour vous rappeler de les désactiver. Si vous l'oubliez, ils risquent de s'exécuter pendant des jours ou des mois, conduisant à des coûts potentiellement élevés.

5. Puisque vous disposez d'un compte GCP avec une facturation activée, vous pouvez commencer à utiliser les services. Le premier dont vous aurez besoin se nomme Google Cloud Storage (GCS). C'est là que vous placerez les SavedModel, les données d'entraînement et d'autres éléments. Dans le menu de navigation, allez jusqu'à la rubrique Stockage et cliquez sur Stockage → Navigateur. Tous vos fichiers iront dans un ou plusieurs *buckets*. Cliquez sur Créer un bucket et indiquez son nom (vous devrez peut-être activer tout d'abord l'API). Puisque GCS utilise

un seul espace de noms mondial pour les buckets, des noms trop simples, surtout en anglais, risquent d'être indisponibles. Assurez-vous que le nom du bucket est conforme aux conventions de nommage du DNS, car il pourra être utilisé dans des enregistrements DNS. Par ailleurs, les noms des buckets sont publics. Évitez donc d'y inclure des informations privées. Une pratique courante consiste à utiliser votre nom de domaine ou votre nom d'entreprise comme préfixe afin de garantir une certaine unicité, ou d'inclure simplement un nombre aléatoire dans le nom. Choisissez un emplacement pour l'hébergement du bucket. Les valeurs par défaut des autres options doivent convenir. Cliquez sur Créer.

- Transférez le dossier *my_mnist_model* que vous avez créé précédemment (y compris une ou plusieurs versions) dans votre bucket. Pour cela, allez dans le navigateur de GCS, sélectionnez le bucket, puis faites glisser le dossier *my_mnist_model* depuis votre système vers le bucket (voir la figure 11.4). Vous pouvez également cliquer sur Importer des fichiers et sélectionner le dossier *my_mnist_model*. La taille maximale par défaut d'un SavedModel est fixée à 250 Mo, mais il est possible de demander un quota plus élevé.

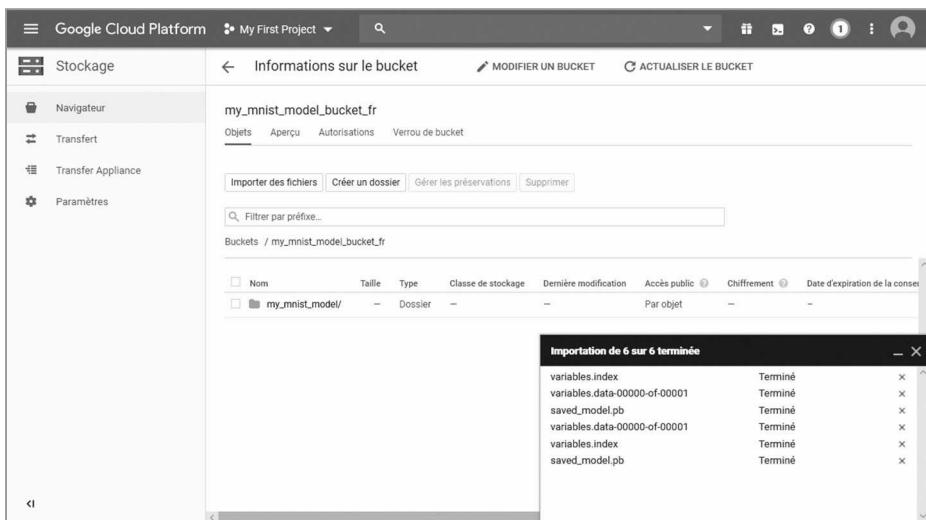


Figure 11.4. – Téléchargement d'un SavedModel vers Google Cloud Storage

- Vous devez à présent configurer AI Platform (précédemment nommé ML Engine) afin qu'il sache quel modèle et quelle version vous souhaitez utiliser. Dans le menu de navigation, aller à la rubrique Intelligence artificielle et cliquez sur AI Platform → Modèles. Cliquez sur Activer une API (l'opération prend quelques minutes), puis cliquez sur Créer un modèle. Indiquez les détails du modèle (voir la figure 19.5) et cliquez sur Créer.

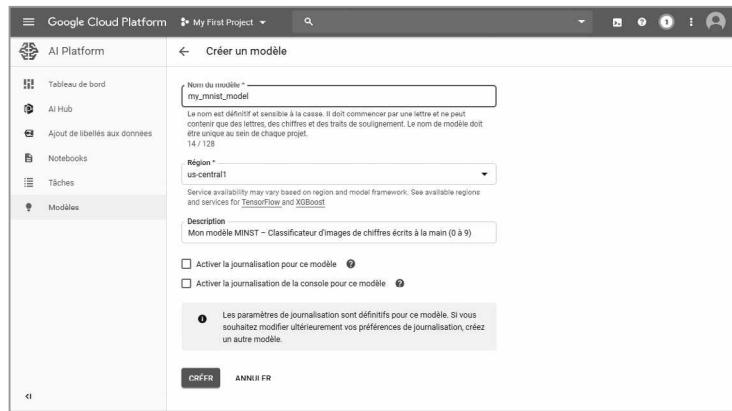


Figure 11.5 – Création d'un nouveau modèle sur Google Cloud AI Platform

8. Puisque vous disposez à présent d'un modèle sur AI Platform, vous devez en créer une version. Dans la liste des modèles, cliquez sur celui que vous venez de créer, puis cliquez sur Créer une version et saisissez les informations demandées (voir la figure 11.6: indiquez le nom, une description, la version de Python (3.5 ou ultérieure), le framework (TensorFlow), la version du framework (2.0 si disponible, ou 1.13)²⁸⁹, la version d'exécution de ML (2.0 si disponible, ou 1.13), le type de machine (pour le moment choisissez Processeur à cœur unique), le chemin du modèle sur GCS (il s'agit du chemin complet vers le dossier de la version courante, par exemple `gs://my-mnist-model-bucket_fn/my_mnist_model/0002/`), le scaling (choisissez automatique), le nombre minimal de noeuds TF Serving qui doivent s'exécuter en continu (laissez ce champ vide). Cliquez sur Enregister.

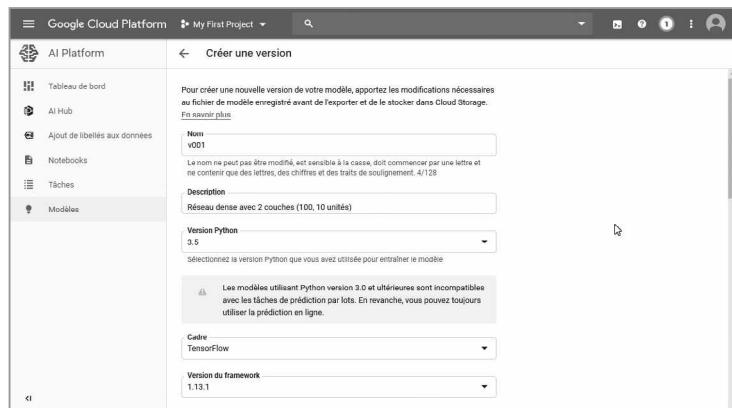


Figure 11.6 – Création d'une nouvelle version de modèle sur Google Cloud AI Platform

289. Au moment de l'écriture de ces lignes, TensorFlow version 2 n'est pas encore disponible sur AI Platform, mais ce n'est pas un problème. Vous pouvez utiliser la version 1.13, qui saura parfaitement exécuter vos SavedModel TF 2.

Félicitations, vous avez déployé votre premier modèle dans le cloud ! Puisque vous avez sélectionné une mise à l'échelle (*scaling*) automatique, AI Platform démarrera d'autres nœuds TF Serving lorsque le nombre de requêtes par seconde augmentera. Il les répartira de façon équilibrée entre tous les nœuds. Si le QPS baisse, des nœuds seront arrêtés automatiquement. Le coût est donc directement lié au QPS (ainsi qu'au type de machine choisi et à la quantité de données stockées sur GCS). Ce modèle de tarification est particulièrement intéressant pour les utilisateurs occasionnels et pour des services qui présentent des pics d'utilisation importants, ainsi que pour les start-up. Il reste faible jusqu'à ce que l'activité de la start-up démarre réellement.



Si vous n'utilisez pas le service de prédiction, AI Platform arrêtera tous les nœuds. Vous ne paieriez donc que pour l'espace de stockage exploité (quelques centimes par giga-octet par mois). Lorsque vous interrogez le service, AI Platform doit démarrer un nœud TF Serving, ce qui prend quelques secondes. Si ce délai est inacceptable, vous devrez fixer le nombre minimum de nœuds TF Serving à 1 au moment de la création de la version du modèle. Bien entendu, cela signifie qu'au moins une machine s'exécutera en permanence et les coûts mensuels seront donc plus élevés.

À présent, interrogeons ce service de prédiction !

11.1.3 Utiliser le service de prédiction

Puisque TF Serving est déjà en exécution sur AI Platform, vous pouvez en principe utiliser le même code que précédemment, pour peu que vous connaissiez l'URL de requête. En revanche, nous avons un problème car GCP prend également en compte le chiffrement et l'authentification. Le chiffrement se fonde sur SSL/TLS et l'authentification utilise des jetons. Un jeton d'authentification secret doit être transmis au serveur à chaque requête. Par conséquent, avant que votre code ne puisse utiliser le service de prédiction (ou tout autre service GCP), il doit obtenir un jeton. Nous verrons comment procéder plus loin, mais vous devez tout d'abord configurer l'authentification et donner à votre application les droits d'accès appropriés à GCP.

Vous avez deux options pour l'authentification :

- Votre application (c'est-à-dire le code client qui interrogera le service de prédiction) peut s'authentifier avec les identifiants d'utilisateur de votre compte Google. De cette façon, l'application dispose des mêmes droits que vous sur GCP, mais c'est certainement plus qu'il ne lui en faut. Par ailleurs, vos identifiants seront indiqués dans l'application et pourront donc être volés par quiconque a accès à l'application. Il aura alors un plein accès à votre compte GCP. Autrement dit, ne choisissez pas cette option ; elle n'est utile que dans de très rares cas (par exemple, lorsque votre application doit accéder au compte GCP de l'utilisateur).
- Le code client peut s'authentifier à l'aide d'un *compte de service*. Il s'agit d'un compte représentant non pas un utilisateur mais une application. Il reçoit en général des droits d'accès très restreints : le strict nécessaire, rien de plus. Il s'agit de l'option recommandée.

Créons donc un compte de service pour votre application. Dans le menu de navigation, cliquez sur IAM et administration → Comptes de service, puis cliquez sur Créer un compte de service. Remplissez le formulaire (nom du compte de service, identifiant, description) et cliquez sur Créer (voir la figure 11.7).

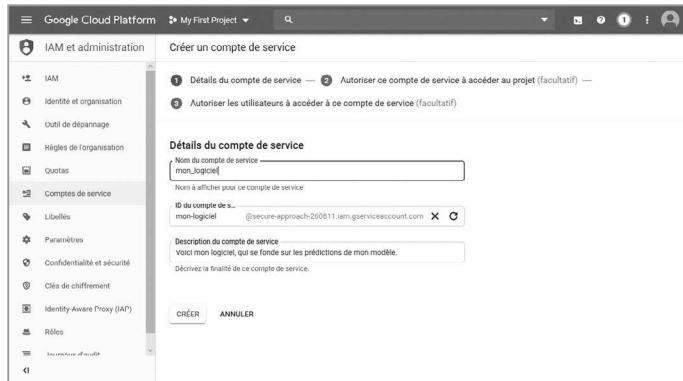


Figure 11.7 – Crédit d'un nouveau compte de service dans Google IAM

Vous devez ensuite affecter des droits d'accès à ce compte. Sélectionnez le rôle Développeur sur ML Engine. Cela permettra au compte de service d'effectuer des prédictions et bien plus encore. Vous pouvez également accorder des accès utilisateurs au compte de service (cela se révélera utile lorsque votre compte d'utilisateur GCP fait partie d'une entreprise et si vous souhaitez que d'autres utilisateurs de l'entreprise soient autorisés à déployer des applications fondées sur ce compte de service ou même à le gérer). Ensuite, cliquez sur Créer une clé pour exporter la clé privée du compte de service, choisissez JSON et cliquez sur Créer. N'oubliez pas de la garder privée !

Parfait ! Écrivons à présent un petit script qui interrogera le service de prédiction. Google fournit plusieurs bibliothèques qui simplifient l'accès à ses services :

- *Google API Client Library*

Cette couche relativement fine se place au-dessus de OAuth 2.0 (<https://oauth.net/>), pour l'authentification, et de REST. Vous pouvez l'utiliser avec tous les services GCP, y compris AI Platform. Cette librairie se nomme `google-api-python-client` (nous l'avons déjà installée au chapitre 1, lors de la création de l'environnement `conda tf2`).

- *Google Cloud Client Libraries*

Ces bibliothèques sont d'un niveau un peu plus élevé. Chacune est dédiée à un service particulier, comme GCS, Google BigQuery, Google Cloud Natural Language et Google Cloud Vision. Elles peuvent toutes être installées avec pip (par exemple, GCS Client Library se nomme `google-cloud-storage`). Lorsqu'il existe une bibliothèque cliente pour un service donné, il est préférable de l'utiliser plutôt que Google API Client Library, car elle met en œuvre toutes

les meilleures pratiques et se sert généralement non pas de REST mais de gRPC, pour de meilleures performances.

Au moment de l'écriture de ces lignes, il n'existe aucune bibliothèque cliente AI Platform. Nous allons donc choisir Google API Client Library. Elle devra utiliser la clé privée du compte de service. Pour la lui indiquer, fixez la valeur de la variable d'environnement `GOOGLE_APPLICATION_CREDENTIALS`, avant le démarrage du script ou à l'intérieur du script :

```
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "ma_cle_de_compte_de_service.json"
```



Si vous déployez votre application dans une machine virtuelle sur Google Cloud Engine (GCE), dans un conteneur en utilisant Google Cloud Kubernetes Engine, en tant qu'application web sur Google Cloud App Engine ou comme un microservice sur Google Cloud Functions, et si la variable d'environnement `GOOGLE_APPLICATION_CREDENTIALS` n'est pas définie, la bibliothèque utilisera le compte de service par défaut du service hébergé (par exemple, le compte de service GCE par défaut si l'application s'exécute sur GCE).

Vous devez ensuite créer un objet de ressources qui enveloppe l'accès aux services de prédiction²⁹⁰:

```
import googleapiclient.discovery

project_id = "secure-approach-260811" # Donnez à cette valeur l'ID
                                         # de votre projet
model_id = "my_mnist_model"
model_path = "projects/{}/models/{}".format(project_id, model_id)
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Vous pouvez ajouter `/versions/0001` (ou tout autre numéro de version) à la variable `model_path` de manière à préciser la version que vous souhaitez interroger. Cela pourra être utile pour les tests A/B ou pour le test d'une nouvelle version sur un petit groupe d'utilisateurs avant de la proposer à tout le monde (selon le principe du *canary release*). Ensuite, écrivons une petite fonction qui utilisera l'objet de ressources pour invoquer le service et obtenir en retour les prédictions :

```
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                      "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name] for pred in response["predictions"]])
```

290. Si vous obtenez une erreur qui indique que le module `google.appengine` est introuvable, précisez `cache_discovery=False` dans l'appel à la méthode `build()` ; voir <https://stackoverflow.com/q/55561354>.

Cette fonction prend un tableau NumPy qui contient des images d'entrée. Elle prépare un dictionnaire que la bibliothèque cliente convertira au format JSON (comme nous l'avons fait précédemment). Elle construit ensuite une requête de prédiction et l'exécute. Elle lève une exception si la réponse contient une erreur ou, sinon, extrait les prédictions pour chaque instance et les regroupe dans un tableau NumPy. Voyons si elle fonctionne correctement :

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Oui ! Vous avez à présent un beau service de prédiction qui s'exécute sur le cloud. Il peut grandir automatiquement en fonction du nombre de requêtes par seconde. Vous pouvez l'interroger de n'importe où, de manière sécurisée. De plus, il ne vous coûte pratiquement rien si vous ne l'utilisez pas. Vous ne paierez que quelques centimes par mois par giga-octet d'espace occupé sur GCS. Vous pouvez également obtenir des informations et des indicateurs détaillés en utilisant Google Stackdriver (<https://cloud.google.com/stackdriver/>).

Voyons à présent comment déployer votre modèle sur une application mobile ou un système embarqué.

11.2 DÉPLOYER UN MODÈLE SUR UN PÉRIPHÉRIQUE MOBILE OU EMBARQUÉ

Si vous devez déployer votre modèle sur un périphérique mobile ou embarqué, tout modèle volumineux prendra trop de temps à télécharger et nécessitera une quantité de RAM et une puissance de calcul trop importantes. L'application aura des temps de réponse très longs, l'appareil chauffera et sa batterie se videra rapidement. Pour éviter tout cela, vous devez créer un modèle léger, efficace et « mobile-friendly », mais sans trop sacrifier sa précision.

La bibliothèque TFLite (<https://tensorflow.org/lite>) fournit plusieurs outils²⁹¹ qui vous aideront à déployer vos modèles sur des systèmes mobiles ou embarqués, avec trois objectifs principaux :

- Réduire la taille du modèle, afin de raccourcir les temps de téléchargement et diminuer l'empreinte mémoire.
- Réduire la quantité de calculs nécessaire à chaque prédiction, afin de diminuer la latence, l'utilisation de la batterie et la surchauffe.
- Adapter le modèle aux contraintes du périphérique.

Pour réduire la taille du modèle, le convertisseur de TFLite peut prendre un SavedModel et le compresser dans un format beaucoup plus léger fondé sur FlatBuffers

²⁹¹. Jetez également un œil à Graph Transform Tool (<https://homl.info/tfgtt>) de TensorFlow pour modifier et optimiser des graphes de calcul.

(<https://google.github.io/flatbuffers/>). Cette bibliothèque de sérialisation efficace interplateforme (un peu comme Protocol Buffers) a été créée initialement par Google pour les jeux. Sa conception permet de charger des FlatBuffers directement en mémoire sans aucun prétraitement. Cela permet de réduire le temps de chargement et l'empreinte mémoire. Après que le modèle a été chargé sur le périphérique mobile ou embarqué, l'interpréteur de TFLite l'exécute et le modèle réalise ses prédictions. Voici comment convertir un SavedModel au format FlatBuffers et l'enregistrer dans un fichier `.tflite`:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```



Vous pouvez également enregistrer un modèle `tf.keras` directement dans un fichier FlatBuffers en utilisant `from_keras_model()`.

Le convertisseur optimise également le modèle, à la fois pour le rétrécir et pour diminuer sa latence. Il coupe toutes les opérations inutiles aux prédictions (comme les opérations d'entraînement) et optimise les calculs lorsque c'est possible. Par exemple, $3 \times a + 4 \times a + 5 \times a$ est converti en $(3 + 4 + 5) \times a$. Il tente également de fusionner des opérations. Par exemple, si c'est possible, les couches de normalisation par lots sont réunies dans les opérations d'addition et de multiplication de la couche précédente.

Afin d'avoir une idée de l'optimisation effectuée par TFLite sur un modèle, téléchargez l'un des modèles TFLite préentraînés (<https://hml.info/itemodels>), extrayez le contenu de l'archive, puis ouvrez Netron, un excellent outil de visualisation de graphes (<https://lutzroeder.github.io/netron/>), et téléchargez le fichier `.pb` pour visualiser le modèle d'origine. Il s'agit d'un graphe plutôt volumineux et complexe. Ouvrez ensuite le modèle `.tflite` optimisé et admirez sa beauté !

Une autre manière de réduire la taille de modèle (hormis une simple utilisation d'architectures de réseaux de neurones plus petites) consiste à utiliser des largeurs de bits inférieures. Par exemple, si vous utilisez des demi-nombres à virgule flottante (sur 16 bits) à la place des réels normaux (sur 32 bits), la taille du modèle sera réduite d'un facteur 2, au prix d'une baisse de la précision (généralement faible). De plus, l'entraînement sera plus rapide et la quantité de RAM occupée sur le GPU sera divisée environ par deux.

Le convertisseur de TFLite peut aller encore plus loin, en quantifiant les poids du modèle en entiers sur 8 bits ! Cela permet une réduction d'un facteur 4 par rapport à l'utilisation des nombres à virgule flottante sur 32 bits. L'approche la plus simple se nomme *quantification post-entraînement*. Il s'agit de quantifier les poids après l'entraînement, en utilisant une technique de quantification symétrique relativement basique mais efficace. Elle recherche la valeur absolue maximale des poids, m , puis met en correspondance la plage des nombres à virgule flottante $-m$ à $+m$ et celle des

entiers de -127 à +127. Par exemple (voir la figure 11.8), si les poids vont de -1,5 à +0,8, alors les octets -127, 0 et +127 correspondront, respectivement, aux valeurs à virgule flottante -1,5, 0,0 et +1,5. Notez que, dans une quantification symétrique, 0,0 est toujours associé à 0 (notez également que les valeurs +68 à +127 ne seront pas utilisées, car elles correspondent à des réels de valeur supérieure à +0,8).

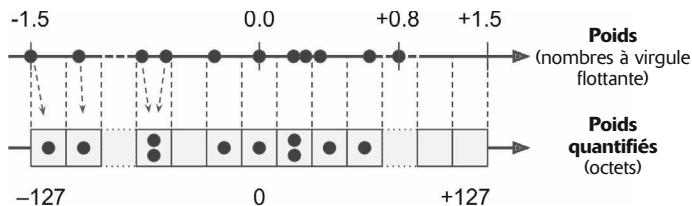


Figure 11.8 – Quantification symétrique de nombres à virgule flottante sur 32 bits en entiers sur 8 bits

Pour réaliser cette quantification post-entraînement, ajoutez simplement `OPTIMIZE_FOR_SIZE` à la liste des optimisations du convertisseur avant d'invoquer la méthode `convert()` :

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

Cette technique réduit énormément la taille du modèle. Il est donc plus rapide à télécharger et à stocker. Toutefois, au moment de l'exécution, les poids quantifiés sont reconvertis en nombres à virgule flottante avant d'être utilisés (les valeurs retrouvées ne sont pas parfaitement identiques aux valeurs d'origine, mais sans être trop éloignées, et la perte de précision est généralement acceptable). Pour éviter qu'elles soient recalculées à chaque fois, les valeurs flottantes récupérées sont placées dans un cache. L'empreinte mémoire n'est donc pas diminuée. De plus, cela ne donne aucune réduction des besoins en calcul.

La manière la plus efficace de réduire la latence et les besoins en puissance consiste à quantifier également les activations. Les calculs se feront alors entièrement sur des entiers, sans recourir à des opérations en virgule flottante. Même lorsque la largeur de bit reste identique (par exemple, des entiers sur 32 bits à la place de nombres à virgule flottante sur 32 bits), le gain est sensible car les calculs sur des entiers demandent moins de cycles processeurs, consomment moins d'énergie et produisent moins de chaleur. Si vous réduisez également la largeur de bit (par exemple, en prenant des entiers sur 8 bits), vous obtenez des accélérations importantes. Par ailleurs, certains dispositifs accélérateurs de réseaux de neurones (comme Edge TPU) ne peuvent traiter que des entiers. Une quantification intégrale des poids et des activations est donc impérative. Cela peut être réalisé après l'entraînement. Puisqu'il faudra une étape de calibration pour trouver la valeur absolue maximale des activations, vous devrez fournir un exemple représentatif de données d'entraînement à TFLite (il n'a pas besoin d'être volumineux), qui les soumettra au modèle et mesurera les statistiques d'activation requises pour la quantification (cette étape est généralement rapide).

Le principal problème de la quantification réside dans la légère perte de précision. Cela équivaut à l'ajout d'un bruit aux poids et aux activations. Si la baisse de précision est trop importante, vous devrez utiliser un *entraînement conscient de la quantification*. Cela signifie ajouter des opérations de quantification factices au modèle afin qu'il apprenne à ignorer le bruit de la quantification pendant l'entraînement. Les poids finaux seront moins sensibles à la quantification. Par ailleurs, l'étape de calibration peut être menée de façon automatique pendant l'entraînement, ce qui simplifie le processus global.

J'ai expliqué les principes de base de TFLite, mais il faudrait un livre complet pour décrire en détail le codage d'une application mobile ou d'un programme embarqué. Heureusement, cet ouvrage existe. Si vous souhaitez en savoir plus sur le développement d'applications TensorFlow pour les périphériques mobiles et embarqués, consultez l'ouvrage *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers* (<https://homl.info/tinyml>) publié par Pete Warden (qui dirige l'équipe TFLite) et Daniel Situnayake aux éditions O'Reilly.

TensorFlow dans le navigateur

Il pourrait être intéressant d'utiliser votre modèle sur un site web, directement dans le navigateur de l'utilisateur, notamment dans les scénarios suivants :

- Lorsque l'application web est souvent employée alors que la connexion de l'utilisateur est intermittente ou lente (par exemple, un site web pour randonneurs). L'exécution du modèle directement sur le client est alors la seule manière d'obtenir un site web fiable.
- Lorsque les réponses du modèle doivent arriver aussi rapidement que possible (par exemple, pour un jeu en ligne). Ne plus avoir à interroger le serveur pour obtenir des prédictions réduit la latence et rend le site web beaucoup plus réactif.
- Lorsque votre service web effectue des prédictions fondées sur des données privées de l'utilisateur et que leur confidentialité doit être assurée. En effectuant les prédictions du côté client, les données privées ne quittent jamais la machine de l'utilisateur²⁹².

Dans toutes ces situations, vous pouvez exporter votre modèle dans un format spécial qui pourra être chargé par la bibliothèque JavaScript TensorFlow.js (<https://tensorflow.org/js>). Elle peut ensuite utiliser votre modèle pour effectuer des prédictions directement dans le navigateur de l'utilisateur. Le projet TensorFlow.js fournit l'outil `tensorflowjs_converter` capable de convertir un SavedModel TensorFlow ou un fichier de modèle Keras dans le format *TensorFlow.js Layers*. Il s'agit d'un répertoire qui contient un ensemble de fichiers de poids éclatés au format binaire et un fichier `model.json` qui décrit l'architecture du modèle et établit les liens entre les fichiers de poids. Ce format est optimisé en vue d'un téléchargement efficace sur le web. Les utilisateurs peuvent ensuite télécharger le modèle et exécuter des prédictions dans le navigateur en utilisant la bibliothèque TensorFlow.js. Voici un extrait de code pour vous faire une idée de cette API JavaScript :

292. Si ce sujet vous intéresse, jetez un œil à l'apprentissage fédéré (<https://tensorflow.org/federated>).

```

import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel(
  'https://example.com/tfjs/model.json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
  
```

De nouveau, pour rendre justice à ce sujet, il faudrait un livre complet. Si vous souhaitez en savoir plus sur TensorFlow.js, consultez l'ouvrage *Practical Deep Learning for Cloud, Mobile, and Edge* (<https://hml.info/tfjsbook>) publié par Anirudh Koul, Siddha Ganju et Meher Kasam aux éditions O'Reilly.

Nous allons voir à présent comment utiliser des GPU pour accélérer les calculs.

11.3 UTILISER DES GPU POUR ACCÉLÉRER LES CALCULS

Au chapitre 3, nous avons décrit plusieurs techniques qui permettent d'accélérer considérablement l'entraînement: meilleure initialisation des poids, normalisation par lots, optimiseurs sophistiqués, etc. Cependant, malgré toutes ces techniques, l'entraînement d'un vaste réseau de neurones sur une seule machine équipée d'un seul processeur peut prendre des jours, voire des semaines.

Dans cette section, nous expliquerons comment accélérer des modèles en utilisant des GPU. Nous verrons également comment distribuer les calculs sur plusieurs processeurs, y compris le CPU et plusieurs GPU (voir la figure 11.9). Pour le moment, l'exécution se fera sur une seule machine, mais, plus loin dans ce chapitre, nous verrons comment distribuer les calculs sur plusieurs serveurs.

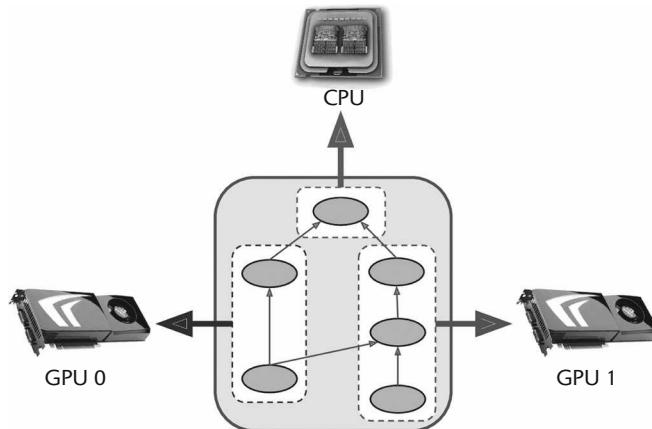


Figure 11.9 – Exécuter un graphe TensorFlow en parallèle sur plusieurs processeurs

Grâce aux GPU, vous n'aurez plus à attendre des jours ou des semaines pour qu'un algorithme d'entraînement se termine, seulement quelques minutes ou heures. Non

seulement l'on gagne ainsi un temps considérable, mais l'on peut également expérimenter plus facilement différents modèles et réentraîner fréquemment les modèles sur des données actualisées.



Pour améliorer les performances de manière significative, il est possible d'ajouter simplement des cartes graphiques à une machine. Cela suffit dans de nombreux cas et il est alors inutile d'utiliser plusieurs machines. Par exemple, l'entraînement d'un réseau de neurones peut en général être aussi rapide avec une seule machine équipée de quatre GPU qu'avec huit GPU sur plusieurs machines, en raison des délais supplémentaires liés aux communications réseau entre les machines. De manière comparable, utiliser un seul GPU puissant est souvent préférable à utiliser plusieurs GPU plus lents.

La première étape consiste à mettre la main sur un GPU. Vous avez deux options : soit acheter votre propre GPU (ou plusieurs), soit utiliser des machines virtuelles équipées d'un GPU dans le cloud. Commençons par la première solution.

11.3.1 Obtenir votre propre GPU

Si vous décidez d'acheter une carte graphique, prenez le temps afin d'effectuer le bon choix. Tim Dettmers a rédigé un billet très intéressant qui vous aidera à choisir une carte (<https://homl.info/66>) ; il l'actualise assez régulièrement. N'hésitez pas à le lire attentivement. Au moment de l'écriture de ces lignes, TensorFlow prend uniquement en charge les cartes Nvidia dotées de la technologie CUDA Compute Capability, en version 3.5 ou ultérieure (<https://homl.info/cudagpus>), mais sa compatibilité avec d'autres fabricants pourrait venir ; bien évidemment, les TPU de Google sont également reconnus. Par ailleurs, bien que les TPU ne soient actuellement disponibles que sur GCP, il est très probable que des cartes de type TPU voient bientôt le jour ; TensorFlow pourrait les prendre en charge. En attendant, vérifiez la compatibilité de vos périphériques dans la documentation de TensorFlow (<https://tensorflow.org/install>).

Si vous optez pour une carte graphique Nvidia, vous devrez installer les pilotes Nvidia adéquats, comme nous l'avons vu au chapitre 1. En installant TensorFlow à l'aide de conda, d'autres bibliothèques sont également automatiquement installées, en particulier :

- la bibliothèque CUDA (*Compute Unified Device Architecture*), qui permet aux développeurs d'exploiter les GPU compatibles CUDA dans toutes sortes de calculs et pas uniquement pour l'accélération graphique ;
- la bibliothèque cuDNN (*CUDA Deep Neural Network*), qui comprend des primitives accélérées par le GPU pour les RNP. Elle fournit des implémentations optimisées des opérations RNP communes, comme les couches d'activation, la normalisation, les convolutions en avant et en arrière, et le pooling (voir le chapitre 6).

TensorFlow utilise CUDA et cuDNN pour contrôler les cartes graphiques et accélérer les calculs (voir la figure 11.10).

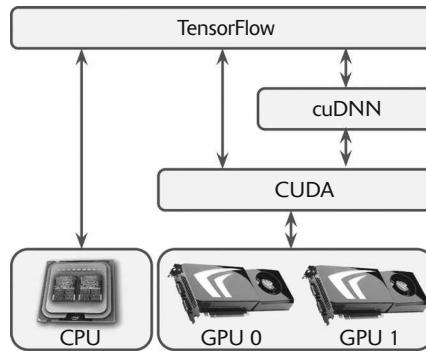


Figure 11.10 – TensorFlow exploite CUDA et cuDNN pour contrôler les GPU et accélérer les RNP

Si vous devez un jour installer TensorFlow avec pip plutôt que conda (par exemple pour installer la toute dernière version de TensorFlow avant qu'elle ne soit disponible *via* conda), il vous faudra alors installer CUDA et cuDNN vous-même. Pour cela, vous devrez vous rendre sur <https://nvidia.com>, créer un compte développeur Nvidia, télécharger et installer les libraries. Assurez-vous de télécharger les versions exactes requises par la version de TensorFlow que vous souhaitez installer. Comme vous pouvez le constater, il est beaucoup plus simple d'utiliser Anaconda.

Après avoir installé la ou les cartes graphiques, ainsi que les pilotes et les bibliothèques nécessaires, vous pouvez utiliser la commande `nvidia-smi` pour vérifier la bonne installation de CUDA. Elle affiche la liste des cartes graphiques disponibles et les processus qui s'exécutent sur chacune d'elles :

```

$ nvidia-smi
Sun Jun  2 10:05:22 2019
+-----+
| NVIDIA-SMI 418.67      Driver Version: 410.79      CUDA Version: 10.0 |
+-----+
| GPU  Name     Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====
|   0  Tesla T4          Off  | 00000000:00:04.0 Off |                  0 |
| N/A   61C     P8    17W /  70W |      0MiB / 15079MiB |      0%  Default |
+-----+
+-----+
| Processes:
| GPU     PID  Type  Process name          GPU Memory |
| =====+=====+=====+=====
| No running processes found
+-----+
  
```

L'équipe de TensorFlow fournit également une image Docker comprenant tout ce dont vous avez besoin pour utiliser TensorFlow avec un GPU. Cependant, pour que le conteneur Docker puisse accéder au GPU, vous devrez installer les pilotes Nvidia

sur la machine hôte, et également installer le Nvidia Container Toolkit (voir <https://github.com/NVIDIA/nvidia-docker>).

Pour vérifier que TensorFlow voit bien les GPU, procédez aux tests suivants :

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

La fonction `is_gpu_available()` vérifie si au moins un GPU est disponible. La fonction `gpu_device_name()` renvoie le nom du premier GPU ; les opérations s'exécuteront par défaut sur celui-ci. La fonction `list_physical_devices()` retourne la liste de tous les GPU disponibles (un seul dans notre exemple)²⁹³.

Et si vous ne souhaitez pas investir du temps et de l'argent dans votre propre carte graphique ? Dans ce cas, vous pouvez simplement utiliser une machine virtuelle avec GPU dans le cloud !

11.3.2 Utiliser une machine virtuelle équipée d'un GPU

Toutes les principales plateformes de cloud proposent aujourd'hui des machines virtuelles avec GPU, certaines déjà configurées avec les pilotes et les bibliothèques dont vous avez besoin (y compris TensorFlow). Google Cloud Platform applique différents quotas de GPU, que ce soit au niveau mondial ou régional : vous ne pouvez pas simplement créer des milliers de machines virtuelles avec GPU sans obtenir au préalable l'autorisation de Google²⁹⁴.

Par défaut, le quota de GPU au niveau mondial étant égal à zéro, vous ne pouvez utiliser aucune machine virtuelle avec GPU. La toute première chose à faire est donc de demander un quota mondial plus élevé. Dans la console GCP, ouvrez le menu de navigation et allez dans IAM et administration → Quotas. Cliquez sur Métrique, cliquez sur Aucune pour décocher tous les lieux, puis recherchez « GPU » et sélectionnez « GPUs (all regions) » pour obtenir les quotas correspondants. Si la valeur de ce quota est égale à zéro (ou insuffisant pour vos besoins), cochez la case qui se trouve à côté (il doit être le seul sélectionné) et cliquez sur Modifier les quotas. Remplissez les informations demandées, puis cliquez sur Envoyer la requête. Il faudra peut-être quelques heures, voire quelques jours, pour que votre demande de quota soit traitée et, généralement, acceptée.

293. Dans ce chapitre, de nombreux exemples de code utilisent des API expérimentales. Il est fort probable qu'elles rejoindront l'API de base dans de futures versions. Si une fonction expérimentale échoue, essayez simplement de retirer le mot `experimental`, et croisez les doigts pour que cela fonctionne. Dans le cas contraire, il est possible que l'API ait légèrement changé. Consultez le notebook Jupyter (voir « `19_training_and_deployment_at_scale.ipynb` » sur <https://github.com/ageron/handson-ml2>), car je m'assurerai qu'il contienne le code correct.

294. On peut supposer que ces quotas sont un moyen d'éviter que des personnes mal intentionnées utilisent GCP avec des cartes bancaires volées pour exploiter des cryptomonnaies.

Par défaut, il existe également un quota sur un GPU par région et par type de GPU. Vous pouvez demander à augmenter ces quotas: cliquez sur Métrique, cliquez sur Aucune pour décocher toutes les métriques, recherchez « GPU », et sélectionnez le type de GPU souhaité (par exemple, NVIDIA P4 GPUs). Ouvrez ensuite la liste déroulante Zone, cliquez sur Aucune pour décocher toutes les métriques, puis choisissez l'emplacement; cochez les cases qui se trouvent à côté des quotas que vous souhaitez modifier, puis cliquez sur Modifier les quotas pour remplir une demande.

Après que vos demandes de quotas GPU ont été approuvées, vous pouvez, en un rien de temps, créer une machine virtuelle équipée d'un ou de plusieurs GPU en utilisant des *images de machines virtuelles pour le Deep Learning* proposées par Google Cloud AI Platform: allez sur <https://console.cloud.google.com/compute/instances/create?step=1®ion=us-central1>, cliquez sur Accéder à la console, puis cliquez sur Lancer sur Compute Engine, et remplissez le formulaire de configuration de la machine virtuelle. Notez que tous les types de GPU ne sont pas disponibles dans toutes les zones, et que certaines ne disposent d'aucun GPU (changez la zone pour voir tous les types de GPU disponibles). N'oubliez pas de sélectionner le Framework TensorFlow 2.0 et de cocher « Install NVIDIA GPU driver automatically on first startup ». Il est également préférable de cocher « Enable access to JupyterLab via URL instead of SSH », car il sera alors très facile de lancer un notebook Jupyter s'exécutant sur cette machine virtuelle et fonctionnant avec JupyterLab (une autre interface web pour exécuter des notebooks Jupyter).

Après que la machine virtuelle a été créée, allez dans la rubrique Intelligence artificielle du menu de navigation, puis cliquez sur AI Platform → Notebooks. Lorsque l'instance de Notebook apparaît dans la liste (puisque cela peut prendre quelques minutes, cliquez de temps en temps sur Actualiser), cliquez sur son lien Ouvrir JupyterLab. Cela déclenchera l'exécution de JupyterLab sur la machine virtuelle et ouvrira l'interface dans le navigateur. Vous pouvez créer des notebooks et exécuter tout code dans cette machine virtuelle, en profitant de ses GPU !

Si votre objectif n'est que d'effectuer quelques tests rapides ou de partager facilement des notebooks avec vos collègues, Colaboratory peut être une bonne solution.

11.3.3 Colaboratory

La manière la plus simple et la moins chère d'accéder à une machine virtuelle avec GPU consiste à utiliser Colaboratory (ou Colab). C'est gratuit ! Allez sur <https://colab.research.google.com/> et créez un nouveau notebook Python 3. L'opération va créer un notebook Jupyter stocké sur votre Google Drive (vous pouvez également ouvrir n'importe quel notebook de GitHub ou Google Drive, ou même envoyer vos propres notebooks). L'interface utilisateur de Colab est comparable à celle de Jupyter, excepté le fait que vous pouvez partager et utiliser les notebooks comme des documents Google Docs normaux, sans oublier quelques autres différences mineures (comme la possibilité de créer des widgets pratiques en plaçant des commentaires spéciaux dans le code).

Lorsque vous ouvrez un notebook Colab, il s'exécute sur une machine virtuelle Google gratuite réservée à ce notebook, appelée *Colab Runtime* (voir la figure 11.11). Par défaut, le Runtime est uniquement équipé d'un CPU, mais vous pouvez changer

cela en allant dans Exécution → Modifier le type d'exécution, en sélectionnant GPU dans la liste déroulante Accélérateur matériel, puis en cliquant sur Enregistrer. Vous avez même la possibilité de choisir TPU ! (Oui, vous pouvez effectivement utiliser gratuitement un TPU ; nous reviendrons sur les TPU plus loin dans ce chapitre, alors, pour le moment, choisissez simplement GPU.)

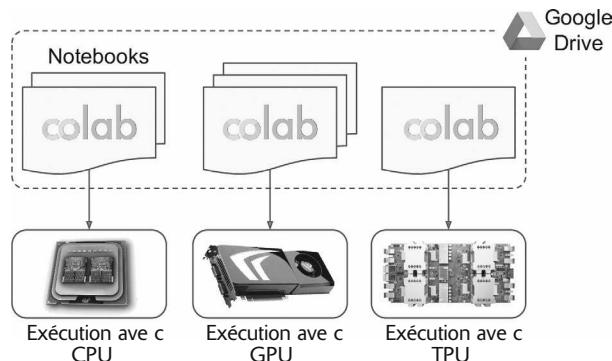


Figure 11.11 – Colab Runtime et notebooks

Colab présente quelques restrictions. Tout d'abord, le nombre de notebooks Colab exécutables simultanément est limité (à cinq par type de Runtime, actuellement). Ensuite, comme le stipule la FAQ, « Les meilleures ressources sont allouées en priorité aux utilisateurs qui interagissent avec Colaboratory plutôt qu'à ceux qui exécutent des calculs longs. Les types de ressources matérielles disponibles et/ou leur durée d'utilisation peuvent être temporairement limités pour les utilisateurs qui exécutent de tels calculs. Veuillez noter que l'utilisation de Colaboratory pour le minage de cryptomonnaie est interdite et peut vous valoir d'être exclu du programme ». De plus, l'interface web se déconnectera automatiquement du Colab Runtime si vous la laissez inactive pendant un certain temps (≈ 30 minutes). Lorsque vous vous reconnectez au Colab Runtime, il a pu être réinitialisé. Prévoyez donc de toujours exporter les données importantes (par exemple, en les téléchargeant ou en les enregistrant sur Google Drive). Même si vous ne vous déconnectez jamais, le Colab Runtime s'arrêtera automatiquement après douze heures, car il n'est pas fait pour les calculs longs.

Malgré ces limitations, il s'agit d'un outil fantastique pour effectuer facilement des tests, obtenir des résultats rapidement et collaborer avec vos collègues.

11.3.4 Gérer la RAM du GPU

Par défaut, TensorFlow s'octroie automatiquement l'intégralité de la RAM disponible sur toutes les cartes graphiques la première fois qu'on démarre une session ; l'objectif est de limiter la fragmentation de la mémoire du GPU. Cela signifie que si vous tentez de démarrer un second programme TensorFlow (ou tout programme qui a besoin du GPU), il sera rapidement en manque de mémoire. Toutefois, cette situation est plus

rare que vous pourriez le penser car, en général, un seul programme TensorFlow s'exécutera sur une machine. Le plus souvent, il s'agira d'un script d'entraînement, d'un noeud TF Serving ou d'un notebook Jupyter. Si, pour une raison ou pour une autre, vous devez exécuter plusieurs programmes (par exemple, pour entraîner en parallèle deux modèles différents sur la même machine), vous devrez répartir la mémoire du GPU entre ces processus de façon plus équilibrée.

Si votre machine est équipée de plusieurs cartes graphiques, une solution simple consiste à affecter chacune d'elle à un seul processus. Pour cela, il suffit de fixer la variable d'environnement `CUDA_VISIBLE_DEVICES` de sorte que chaque processus voie uniquement les cartes graphiques appropriées. Fixez également la variable d'environnement `CUDA_DEVICE_ORDER` à `PCI_BUS_ID` pour être certain que chaque identifiant fait toujours référence à la même carte graphique. Par exemple, si vous avez quatre cartes graphiques, vous pouvez lancer deux programmes, en affectant deux GPU à chacun d'eux. Il suffit d'exécuter des commandes semblables aux suivantes dans deux fenêtres de terminal distinctes :

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python program_1.py
# Et dans un autre terminal :
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python program_2.py
```

Le programme 1 voit uniquement les cartes graphiques 0 et 1, nommées respectivement `/gpu:0` et `/gpu:1`, tandis que le programme 2 voit uniquement les cartes 2 et 3 (nommées respectivement `/gpu:1` et `/gpu:0` (remarquez l'ordre). Tout fonctionne parfaitement (voir la figure 11.12). Bien entendu, vous pouvez également définir ces variables d'environnement dans Python en fixant les valeurs de `os.environ["CUDA_DEVICE_ORDER"]` et de `os.environ["CUDA_VISIBLE_DEVICES"]`, tant que vous le faites avant d'utiliser TensorFlow.

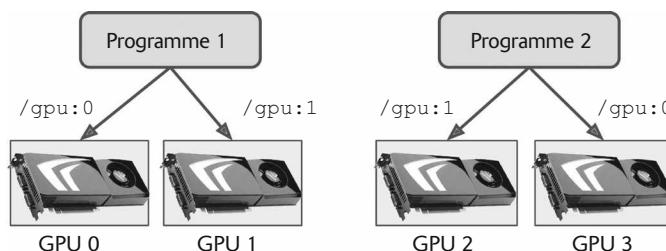


Figure 11.12 – Chaque programme reçoit deux GPU

Une autre option consiste à indiquer à TensorFlow de ne réserver qu'une partie de la mémoire. Il faut le faire immédiatement après avoir importé TensorFlow. Par exemple, pour que TensorFlow s'octroie uniquement 2 Gio de la mémoire de chaque carte graphique, il faut créer un *périphérique GPU virtuel* (ou *périphérique GPU logique*) pour chaque processeur graphique physique et limiter sa mémoire à 2 Gio (c'est-à-dire 2048 Mio) :

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
```

```
gpu,
[tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

En supposant que vous disposez de quatre GPU, chacun équipé d'au moins 4 Gio de mémoire, deux programmes comme celui-ci peuvent à présent s'exécuter en parallèle, chacun utilisant les quatre mêmes cartes graphiques (voir la figure 11.13).

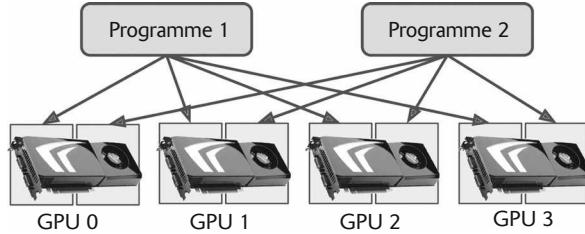


Figure 11.13 – Chaque programme peut utiliser les quatre GPU, mais uniquement 2 Gio de leur RAM

Si vous exécutez la commande `nvidia-smi` pendant que les deux programmes s'exécutent, vous devriez constater que chacun possède 2 Gio de RAM sur chaque carte :

```
$ nvidia-smi
[...]
+-----+
| Processes:
| GPU     PID   Type  Process name          GPU Memory |
| GPU     PID   Type  Process name          Usage      |
+=====+
|    0     2373   C   /usr/bin/python        2241MiB  |
|    0     2533   C   /usr/bin/python        2241MiB  |
|    1     2373   C   /usr/bin/python        2241MiB  |
|    1     2533   C   /usr/bin/python        2241MiB  |
[...]
```

Une autre solution consiste à indiquer à TensorFlow de ne prendre de la mémoire que lorsqu'il en a besoin (cela doit également être effectué immédiatement après avoir importé TensorFlow) :

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

Vous pouvez également fixer la variable d'environnement `TF_FORCE_GPU_ALLOW_GROWTH` à `true`. Dans ce cas, TensorFlow ne libérera jamais la mémoire qu'il a reçue (de nouveau, afin d'éviter sa fragmentation), excepté lorsque le programme se termine. Avec cette méthode, il sera probablement plus difficile de garantir un comportement déterministe (par exemple, un programme peut s'arrêter parce qu'un autre programme a utilisé plus de mémoire qu'autorisé). Par conséquent, en production, il est préférable d'opter pour l'une des options précédentes. Cependant, il existe des cas où cette approche se révèle très utile, par exemple lorsque vous utilisez une machine pour exécuter plusieurs notebooks Jupyter, plusieurs d'entre eux utilisant TensorFlow. C'est la raison pour laquelle la variable d'environnement `TF_FORCE_GPU_ALLOW_GROWTH` est fixée à `true` dans les Colab Runtime.

Enfin, dans certains cas, vous voudrez diviser un GPU en deux *GPU virtuels*, ou plus, par exemple pour tester un algorithme de distribution (une façon pratique d'essayer les exemples de code donnés dans la suite de ce chapitre, même si vous disposez d'un seul GPU, comme dans un Colab Runtime). Le code suivant divise le premier GPU en deux périphériques virtuels, chacun disposant de 2 Gio de RAM (de nouveau, cela doit être effectué juste après avoir importé TensorFlow) :

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Ces deux périphériques virtuels seront nommés `/gpu:0` et `/gpu:1`, et vous pouvez placer des opérations et des variables sur chacun d'eux comme s'il s'agissait de deux GPU indépendants. Voyons à présent comment TensorFlow choisit les processeurs sur lesquels il va placer des variables et exécuter des opérations.

11.3.5 Placer des opérations et des variables sur des processeurs

Le livre blanc²⁹⁵ sur TensorFlow décrit un algorithme de *placement dynamique* convivial qui distribue « automagiquement » des opérations sur tous les processeurs disponibles, en prenant en compte des facteurs comme le temps de calcul mesuré lors d'exécutions précédentes du graphe, des estimations de la taille des tenseurs d'entrée et de sortie pour chaque opération, la quantité de RAM disponible sur chaque périphérique, les délais de communication lors de transfert des données vers et depuis les processeurs, et des recommandations et des contraintes fournies par l'utilisateur. En pratique, cet algorithme s'est révélé moins efficace qu'un petit jeu de règles de placement définies par l'utilisateur. L'équipe TensorFlow a donc finalement retiré le placement dynamique.

Cela dit, `tf.keras` et `tf.data` font généralement un bon travail de placement des opérations et des variables (par exemple, les calculs lourds sur le GPU et le traitement des données sur le CPU). Mais, pour un plus grand contrôle, vous pouvez également placer des opérations et des variables manuellement sur chaque processeur :

- Comme nous venons de l'indiquer, les opérations de prétraitement des données seront généralement placées sur le CPU, tandis que les opérations du réseau de neurones iront sur les GPU.
- Les GPU disposent généralement d'une bande passante de communication relativement limitée. Il est donc important d'éviter les transferts de données inutiles vers et depuis les GPU.
- Puisque l'ajout de mémoire destinée au processeur d'une machine est simple et peu onéreux, elle en est généralement bien pourvue. En revanche, la mémoire du GPU est intégrée à la carte graphique, ce qui en fait une ressource limitée et onéreuse. Par conséquent, si une variable n'est pas utile lors des quelques étapes d'entraînement suivantes, elle doit probablement être placée sur le CPU (par exemple, les datasets sont généralement confiés au CPU).

295. Martín Abadi *et al.*, « TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems », Google Research whitepaper (2015) : <https://homl.info/67>.

Par défaut, toutes les variables et toutes les opérations seront placées sur le premier GPU (nommé `/gpu:0`), à l'exception de celles qui n'ont pas de noyau GPU²⁹⁶, qui iront sur le CPU (nommé `/cpu:0`). L'attribut `device` d'un tenseur ou d'une variable indique le processeur sur lequel l'élément a été placé²⁹⁷:

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

Vous pouvez, pour le moment, ignorer le préfixe `/job:localhost/replica:0/task:0` (il permet de placer des opérations sur d'autres machines dans le cadre d'une partition TensorFlow ; nous présenterons les jobs, les répliques et les tâches plus loin dans ce chapitre). Vous le constatez, la première variable a été placée sur le GPU 0, qui correspond au processeur par défaut. En revanche, la seconde variable a été placée sur le CPU, car il n'existe aucun noyau GPU pour les variables entières (ou pour les opérations qui impliquent des tenseurs entiers). TensorFlow s'est donc replié sur le CPU.

Pour placer une opération sur un processeur différent de celui par défaut, utilisez un contexte `tf.device()`:

```
>>> with tf.device('/cpu:0'):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```



Le CPU est toujours considéré comme un seul processeur (`/cpu:0`), même si votre machine est équipée d'un processeur à plusieurs coeurs. Une opération confiée au CPU peut s'exécuter en parallèle sur plusieurs coeurs si elle dispose d'un noyau multithread.

Si vous tentez de placer explicitement une opération ou une variable sur un processeur qui n'existe pas ou pour lequel il n'existe aucun noyau, vous recevrez une exception. Cependant, dans certains cas, vous pourriez préférer un repli sur le CPU. Par exemple, si votre programme peut s'exécuter à la fois sur des machines équipées uniquement d'un CPU et sur des machines dotées de GPU, vous pourriez souhaiter que TensorFlow ignore la directive `tf.device("/gpu:*)` sur les machines à CPU uniquement. Vous pouvez appeler `tf.config.set_soft_device_placement(True)` pour cela, juste après l'importation de TensorFlow. Si une demande de placement échoue, TensorFlow se replie alors sur les règles de placement

296. Nous l'avons vu au chapitre 4, un noyau est l'implémentation d'une variable ou d'une opération pour un type de données et un type de processeur spécifiques. Par exemple, il existe un noyau GPU pour l'opération `float32 tf.matmul()`, mais pas pour `int32 tf.matmul()` (uniquement un noyau CPU).

297. Vous pouvez également appeler `tf.debugging.set_log_device_placement(True)` pour consigner tous les placements sur les processeurs.

par défaut (autrement dit, le GPU 0 par défaut si présent et s'il existe un noyau GPU, sinon le CPU 0).

Voyons à présent comment TensorFlow exécute toutes ces opérations sur plusieurs processeurs.

11.3.6 Exécution en parallèle sur plusieurs processeurs

Nous l'avons vu au chapitre 4, l'un des avantages des fonctions TF réside dans leur possibilité de parallélisme. Examinons cela d'un peu plus près. Lorsque TensorFlow exécute une fonction TF, il commence par analyser son graphe afin de trouver la liste des opérations à évaluer et compte le nombre de dépendances de chacune. Il ajoute ensuite chaque opération n'ayant aucune dépendance (c'est-à-dire chaque opération source) dans la file d'évaluation du processeur de cette opération (voir la figure 11.14). Après qu'une opération a été évaluée, le compteur de dépendances de chaque opération qui en dépend est décrémenté. Lorsque ce compteur atteint zéro, l'opération correspondante est placée dans la file d'évaluation de son processeur. Une fois que tous les noeuds dont a besoin TensorFlow ont été évalués, leurs sorties sont retournées.

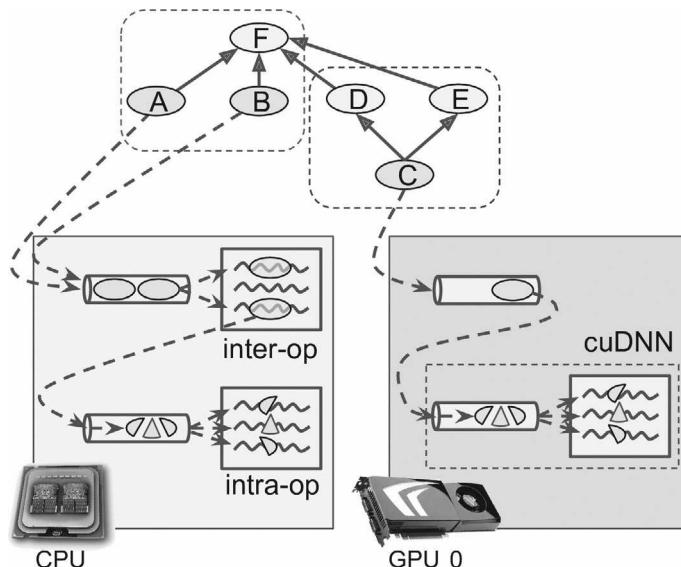


Figure 11.14 – Exécution en parallèle d'un graphe TensorFlow

Les opérations qui se trouvent dans la file d'évaluation du CPU sont distribuées à un pool de threads appelé *pool de threads inter-op*. Si le CPU dispose de plusieurs coeurs, ces opérations sont effectivement évaluées en parallèle. Certaines opérations disposent de noyaux CPU multithreads. Dans ce cas, ces noyaux divisent leurs tâches en plusieurs sous-opérations, qui sont placées dans une autre file d'évaluation et distribuées à un second pool de threads appelé *pools de threads intra-op* (il est partagé par tous les noyaux de CPU multithreads). En résumé, de multiples opérations et sous-opérations peuvent être évaluées en parallèle sur différents coeurs de CPU.

Pour le GPU, les choses sont un peu plus simples. Les opérations dans la file d'évaluation d'un GPU sont traitées séquentiellement. Toutefois, la plupart disposent de noyaux GPU multithreads, souvent implémentées par des bibliothèques dont dépend TensorFlow, comme CUDA et cuDNN. Ces implémentations disposent de leurs propres pools de threads et exploitent en général autant de threads de GPU qu'elles le peuvent (c'est la raison pour laquelle un pool de threads inter-op est inutile dans les GPU ; chaque opération occupe déjà la plupart des threads du GPU).

Par exemple, sur la figure 11.14, les opérations A, B et C sont des opérations source et peuvent donc être évaluées immédiatement. Les opérations A et B sont placées sur le CPU et sont donc envoyées à la file d'évaluation du CPU, puis transmises au pool de threads inter-op et immédiatement évaluées en parallèle. Puisque l'opération A dispose d'un noyau multithread, ses calculs sont divisés en trois parties, exécutées en parallèle par le pool de threads intra-op. L'opération C va dans la file d'évaluation du GPU 0 et, dans cet exemple, son noyau GPU utilise cuDNN, qui gère son propre pool de threads intra-op et exécute les opérations sur plusieurs threads de GPU en parallèle. Supposons que C se termine en premier. Les compteurs de dépendances de D et de E sont décrémentés et arrivent à zéro. Ces deux opérations sont envoyées à la file d'évaluation du GPU 0 et sont exécutées séquentiellement. Notez que C n'est évaluée qu'une seule fois, même si D et E en dépendent. Supposons que B finisse ensuite. Le compteur de dépendances de F passe alors de 4 à 3 et, puisqu'il n'est pas égal à 0, F n'est pas encore exécutée. Dès que A, D et E sont terminées, le compteur de dépendances de F atteint 0 et cette opération est placée dans la file d'évaluation du CPU et est évaluée. Pour finir, TensorFlow retourne les sorties demandées.

TensorFlow ajoute également un petit traitement magique lorsque la fonction TF modifie une ressource avec état, comme une variable. Il s'assure que l'ordre d'exécution correspond à l'ordre dans le code, même s'il n'existe aucune dépendance explicite entre les instructions. Par exemple, si la fonction TF contient `v.assign_add(1)` suivie de `v.assign(v * 2)`, TensorFlow fait en sorte que ces opérations soient exécutées dans cet ordre.



Vous pouvez contrôler le nombre de threads dans les pools inter-op en appelant `tf.config.threading.set_inter_op_parallelism_threads()`. Pour fixer le nombre de threads intra-op, utilisez `tf.config.threading.set_intra_op_parallelism_threads()`. Ces ajustements seront utiles si vous ne souhaitez pas que TensorFlow utilise tous les coeurs du CPU ou si vous souhaitez qu'il reste monothread²⁹⁸.

Vous disposez à présent de tout le nécessaire pour exécuter n'importe quelle opération sur n'importe quel processeur et exploiter la puissance de vos GPU ! Voici quelques propositions :

- Vous pouvez entraîner plusieurs modèles en parallèle, chacun sur son propre GPU. Écrivez simplement un script d'entraînement pour chaque modèle et exécutez-les en parallèle, en fixant `CUDA_DEVICE_ORDER` et `CUDA_`

298. Cette option peut se révéler utile pour garantir une parfaite reproductibilité, comme il est expliqué dans la vidéo disponible à l'adresse <https://homl.info/repro>; elle est basée sur TF 1.

`VISIBLE_DEVICES` de sorte que chaque site ne voie qu'un seul GPU. Cette approche convient parfaitement à l'ajustement d'un hyperparamètre, car vous pouvez entraîner en parallèle plusieurs modèles avec des hyperparamètres différents. Si vous disposez d'une seule machine avec deux GPU et s'il faut une heure pour entraîner un modèle sur un GPU, l'entraînement de deux modèles en parallèle, chacun sur son GPU dédié, ne prendra qu'une heure.

- Vous pouvez entraîner un modèle sur un seul GPU et effectuer tous les prétraitements en parallèle sur le CPU, en utilisant la méthode `prefetch()` du dataset²⁹⁹ pour préparer à l'avance les quelques lots suivants. Ils seront alors prêts au moment où le GPU en aura besoin (voir le chapitre 5).
- Si votre modèle prend deux images en entrée et les traite à l'aide de deux CNN, avant de réunir leur sortie, il s'exécutera probablement plus rapidement si vous placez chaque CNN sur un GPU différent.
- Vous pouvez créer un ensemble efficace : placez simplement un modèle entraîné différent sur chaque GPU. Vous obtiendrez toutes les prédictions plus rapidement, pour ensuite générer la prédition finale de l'ensemble.

Voyons à présent comment *entraîner* un seul modèle sur plusieurs GPU.

11.4 ENTRAÎNER DES MODÈLES SUR PLUSIEURS PROCESSEURS

Il existe deux approches principales à l'entraînement d'un seul modèle sur plusieurs processeurs : le *parallélisme du modèle*, dans lequel le modèle est réparti sur les processeurs, et le *parallélisme des données*, dans lequel le modèle est répliqué sur chaque processeur et chaque réplica est entraîné sur un sous-ensemble des données. Examinons de plus près ces deux options, avant d'entraîner un modèle sur plusieurs GPU.

11.4.1 Parallélisme du modèle

Jusque-là, nous avons exécuté chaque réseau de neurones sur un seul processeur. Comment pouvons-nous exécuter un seul réseau de neurones sur plusieurs processeurs ? Pour cela, nous devons découper notre modèle en morceaux séparés et exécuter chacun d'eux sur un processeur différent. Malheureusement, cette méthode est relativement complexe et dépend totalement de l'architecture du réseau de neurones. Lorsqu'un réseau est intégralement connecté, le parallélisme du modèle apporte généralement peu de gain (voir la figure 11.15).

299. Au moment de l'écriture de ces lignes, la prélecture des données se fait uniquement vers la mémoire du CPU, mais vous pouvez utiliser `tf.data.experimental.prefetch_to_device()` pour que la prélecture des données soit dirigée vers le processeur de votre choix. Le GPU ne perdra alors pas de temps à attendre le transfert des données.

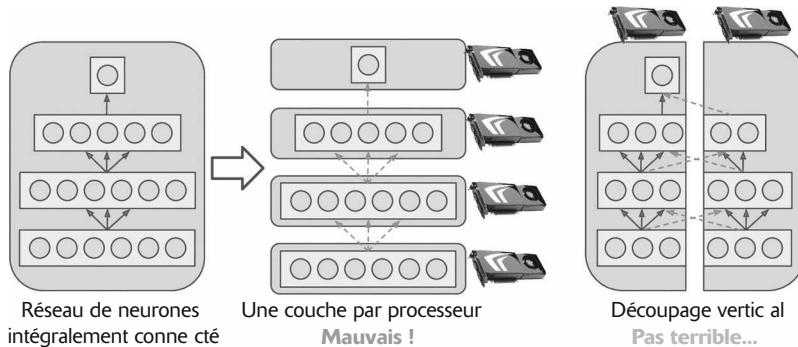


Figure 11.15 – Découpage d'un réseau de neurones intégralement connecté

Intuitivement, on pourrait penser qu'un découpage facile du modèle consiste à placer chaque couche du réseau sur un processeur différent, mais cette solution fonctionne très mal car chaque couche doit attendre la sortie de la couche précédente pour opérer. Dans ce cas, pourquoi ne pas découper le réseau verticalement, par exemple en plaçant la moitié gauche de chaque couche sur un processeur, et la moitié droite sur un autre ? Les résultats sont un peu meilleurs, car les deux moitiés de chaque couche peuvent effectivement travailler en parallèle, mais le problème est que chaque moitié de la couche suivante a besoin de la sortie des deux moitiés précédentes. Cela conduit donc à un grand nombre de communications entre les processeurs (représentées par les flèches en pointillé), qui risquent d'annuler totalement le bénéfice des calculs parallèles. En effet, les communications de ce type sont lentes, notamment lorsqu'elles se font entre des machines séparées.

Certaines architectures de réseaux de neurones, comme les réseaux de neurones convolutifs (voir le chapitre 6), comprennent des couches qui ne sont que partiellement connectées aux couches inférieures. Dans ce cas, il est beaucoup plus facile d'en répartir efficacement des portions entre les processeurs (voir la figure 11.16).

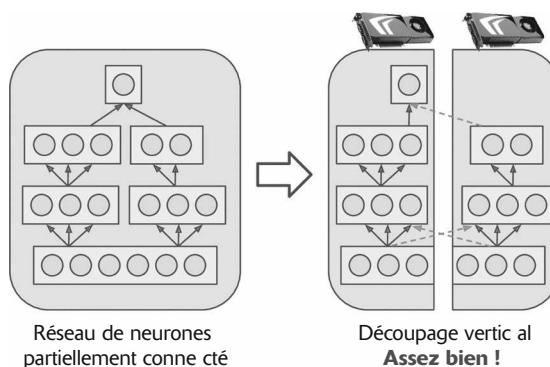


Figure 11.16 – Découpage d'un réseau de neurones partiellement connecté

Les réseaux de neurones récurrents profonds (voir le chapitre 7) peuvent être divisés de façon plus efficace sur plusieurs GPU. Si vous découpez horizontalement un tel réseau et que vous placez chaque couche sur un processeur différent, et si vous alimentez le réseau avec une série d'entrée à traiter, alors, pendant la première étape, un seul processeur est actif (travaillant sur la première valeur de la série), pendant la deuxième, deux le sont (la deuxième couche traite la sortie produite par la première couche pour la première valeur, tandis que la première couche traite la deuxième valeur), et lorsque le signal atteint la couche de sortie, tous les processeurs sont actifs simultanément (voir la figure 11.17). Cette approche implique un grand nombre de communications entre les processeurs, mais, puisque chaque cellule peut être relativement complexe, l'intérêt d'en exécuter plusieurs en parallèle peut, en théorie, contrebalancer le coût des communications. En pratique, une pile normale de couches LSTM s'exécutant sur un seul GPU donne de meilleures performances.

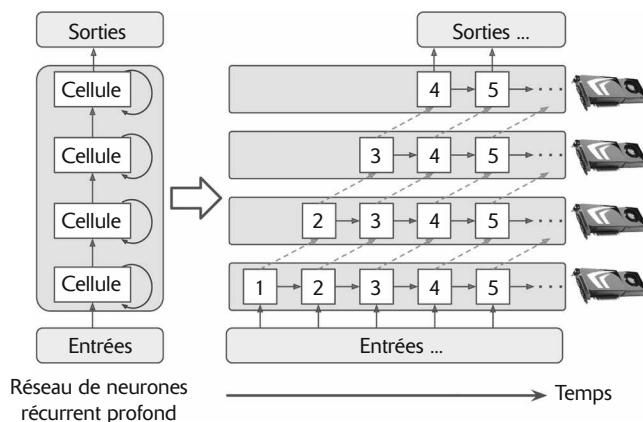


Figure 11.17 – Découpage d'un réseau de neurones récurrent profond

En résumé, le parallélisme du modèle peut accélérer l'exécution ou l'entraînement de certains types de réseaux de neurones, mais pas tous. Il exige également une attention et un ajustement particuliers afin que les processeurs qui communiquent le plus soient placés sur la même machine³⁰⁰. Voyons à présent une option plus simple et généralement plus efficace, le parallélisme des données.

11.4.2 Parallélisme des données

Une autre façon de paralléliser l'entraînement d'un réseau de neurones consiste à le répliquer sur chaque processeur et à exécuter simultanément une étape d'entraînement sur tous les réplicas, chacun travaillant sur un mini-lot différent. La moyenne des gradients déterminés par chaque réplica est ensuite calculée et permet d'actualiser

300. Si vous souhaitez aller plus loin avec le parallélisme du modèle, intéressez-vous à Mesh TensorFlow (<https://github.com/tensorflow/mesh>).

les paramètres du modèle. Cette approche se nomme *parallélisme des données*, dont il existe de nombreuses variantes. Décrivons les plus importantes.

Parallélisme des données avec mise en miroir

L'approche certainement la plus simple consiste à refléter tous les paramètres du modèle sur tous les GPU et de procéder à la même mise à jour des paramètres sur chaque GPU. De cette manière, les répliques restent toujours parfaitement identiques. Cette stratégie de *mise en miroir* se révèle plutôt efficace, en particulier lorsqu'on utilise une seule machine (voir la figure 11.18).

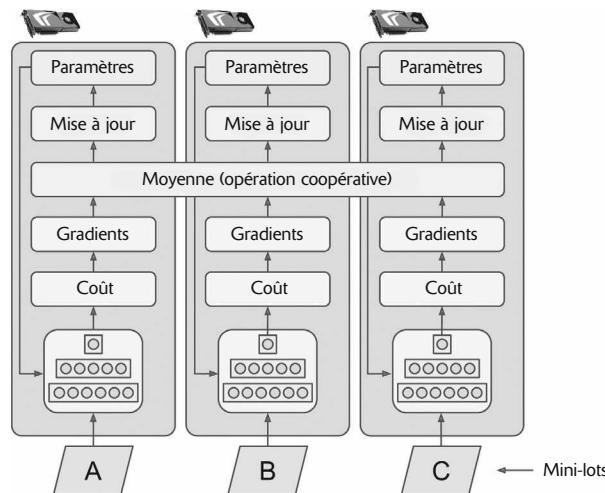


Figure 11.18 – Parallélisme des données selon la stratégie de miroir

La partie complexe de cette approche réside dans le calcul efficace de la moyenne de tous les gradients provenant de tous les GPU et de la distribution du résultat sur tous les GPU. La solution réside dans un algorithme *AllReduce*. Dans cette classe d'algorithmes, plusieurs nœuds collaborent pour réaliser efficacement une opération de réduction (comme calculer la moyenne, la somme et le maximum), tout en garantissant que tous les nœuds obtiennent le même résultat final. Heureusement, il existe des implémentations prêtes à l'emploi pour ces algorithmes.

Parallélisme des données avec centralisation des paramètres

Une autre approche consiste à stocker les paramètres du modèle en dehors des GPU qui s'occupent des calculs (appelés *ouvriers*), par exemple sur le CPU (voir la figure 11.19). Dans une configuration distribuée, tous les paramètres peuvent être placés sur un ou plusieurs serveurs à CPU uniquement (appelés *serveurs de paramètres*), leur seul rôle étant d'héberger et de mettre à jour ces paramètres.

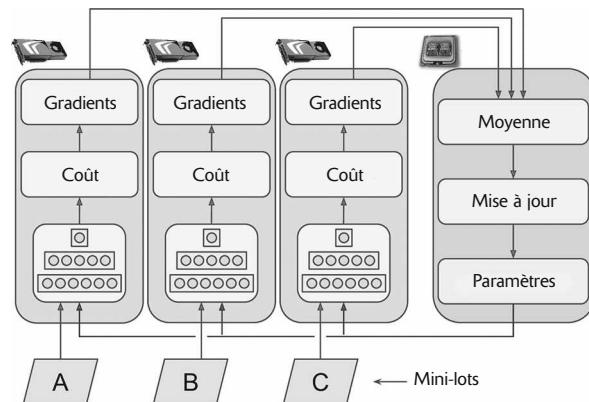


Figure 11.19 – Parallélisme des données avec paramètres centralisés

Alors que la stratégie avec mise en miroir impose une mise à jour synchronisée des poids sur tous les GPU, cette approche centralisée autorise des mises à jour synchrones ou asynchrones. Voyons les avantages et les inconvénients de ces deux options.

Mises à jour synchrones

Dans le cas des mises à jour synchrones, l'agrégateur attend que tous les gradients soient disponibles avant d'en calculer la moyenne et de les passer à l'optimiseur, qui met à jour les paramètres du modèle. Lorsqu'un réplica a terminé le calcul de ses gradients, il doit attendre que les paramètres soient mis à jour avant de pouvoir traiter le mini-lot suivant. L'inconvénient est que certains processeurs peuvent être plus lents que d'autres et que, à chaque étape, les plus rapides doivent tous attendre les plus lents. Par ailleurs, les paramètres seront copiés sur chaque processeur presque au même moment (immédiatement après l'application des gradients), ce qui risque de saturer la bande passante des serveurs de paramètres.



Afin de diminuer le temps d'attente à chaque étape, il est possible d'ignorer les gradients des réplicas les plus lents (habituellement les 10 % les plus lents). Par exemple, on peut exécuter 20 réplicas, mais n'agrégier à chaque étape que les gradients des 18 réplicas les plus rapides et ignorer ceux des 2 retardataires. Dès que les paramètres sont actualisés, les 18 premiers réplicas peuvent commencer à travailler immédiatement, sans avoir à attendre les 2 réplicas les plus lents. Cette configuration est généralement présentée comme ayant 18 réplicas plus 2 réplicas de recharge³⁰¹.

301. Le terme peut introduire une légère confusion, car il semble indiquer que certains réplicas sont particuliers, ne faisant rien. En réalité, ils sont tous équivalents et travaillent dur pour faire partie des plus rapides à chaque étape d'entraînement. Les perdants ne sont pas nécessairement les mêmes à chaque étape (sauf si certains processeurs sont réellement plus lents que d'autres). Toutefois, cela signifie qu'en cas de dysfonctionnement d'un serveur, l'entraînement se poursuivra sans problème.

Mises à jour asynchrones

Avec les mises à jour asynchrones, dès qu'un réplica a terminé le calcul des gradients, il les utilise immédiatement pour actualiser les paramètres du modèle. L'agrégation est absente (l'étape « moyenne » sur la figure 11.19 disparaît), tout comme la synchronisation. Les réplicas travaillent indépendamment les uns des autres. Puisque aucun réplica n'attend les autres, cette approche permet de réaliser un plus grand nombre d'entraînements par minute. Par ailleurs, même si les paramètres doivent toujours être copiés sur chaque processeur à chaque étape, cette opération est réalisée à des moments différents pour chaque réplica, ce qui réduit les risques de saturation de la bande passante.

Le parallélisme des données avec mises à jour asynchrones est très attrayant, en raison de sa simplicité, de l'absence de délai de synchronisation et d'une meilleure utilisation de la bande passante. Toutefois, malgré son comportement convenable en pratique, il est assez surprenant qu'il fonctionne ! Au moment où un réplica a terminé le calcul des gradients à partir de certaines valeurs des paramètres, ceux-ci auront été actualisés à plusieurs reprises par d'autres réplicas (en moyenne $N - 1$ fois s'il y a N réplicas) et rien ne garantit que les gradients calculés pointeront toujours dans la bonne direction (voir la figure 11.20). Lorsque les gradients sont fortement obsolètes, ils sont appelés *gradients périmés (stale gradient)*. Ils peuvent ralentir la convergence, en introduisant du bruit et des effets de secousse (la courbe d'apprentissage peut contenir des oscillations temporaires), et peuvent même faire diverger l'algorithme d'entraînement.

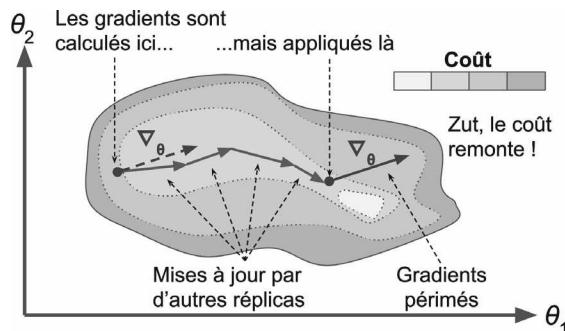


Figure 11.20 – Gradients périmés lors des mises à jour asynchrones

Il existe quelques solutions pour diminuer les effets des gradients périmés :

- abaisser le taux d'apprentissage ;
- ignorer les gradients périmés ou les réduire ;
- ajuster la taille du mini-lot ;
- démarrer les quelques premières époques en utilisant un seul réplica (*phase d'échauffement*). Les gradients périmés ont tendance à avoir un effet plus important au début de l'entraînement, lorsqu'ils sont grands et que les

paramètres ne sont pas encore entrés dans une vallée de la fonction de coût, ce qui peut conduire différents réplicas à pousser les paramètres dans des directions assez différentes.

Un article³⁰² publié par l'équipe Google Brain en avril 2016 évalue différentes approches. Il conclut que le parallélisme des données avec mises à jour synchrones et quelques réplicas de rechange était le plus efficace, en raison d'une convergence plus rapide et de la production d'un meilleur modèle. Cependant, cela reste un sujet de recherche très actif et il n'est pas encore possible d'écartier les mises à jour asynchrones.

Saturation de la bande passante

Que les mises à jour se fassent de façon synchrone ou asynchrone, le parallélisme des données avec centralisation des paramètres implique la transmission des paramètres du modèle depuis les serveurs de paramètres vers tous les réplicas au début de chaque étape d'entraînement, et celle des gradients dans le sens inverse à la fin de chaque étape. De façon comparable, dans la stratégie avec mise en miroir, les gradients produits par chaque GPU doivent être partagés avec chaque autre GPU. Malheureusement, cela signifie qu'à un certain stade l'ajout d'un GPU supplémentaire n'améliorera pas les performances, car le temps passé à échanger les données avec la RAM du GPU (potentiellement au travers du réseau) annulera l'accélération obtenue par la répartition de la charge de calcul. À partir de là, l'ajout d'autres GPU ne fera qu'augmenter la saturation et ralentir l'entraînement.



Avec certains modèles, notamment ceux relativement petits et entraînés sur un jeu de données très grand, il est souvent préférable d'effectuer l'entraînement sur une seule machine avec un seul GPU avec une bande passante mémoire élevée.

La saturation est encore plus importante avec les grands modèles denses, car ils impliquent la transmission d'un grand nombre de paramètres et de gradients. Elle est moindre avec les petits modèles (mais le gain obtenu grâce à la parallélisation est faible) et avec les grands modèles creux, car la plupart des gradients sont généralement à zéro et peuvent être transmis de façon efficace. Jeff Dean, initiateur et responsable du projet Google Brain, a rapporté (<https://homl.info/69>) des accélérations d'un facteur 25 à 40 lors de la distribution des calculs sur 50 GPU pour des modèles denses, et d'un facteur 300 pour des modèles plus creux entraînés sur 500 GPU. Cela montre bien que les modèles creux sont mieux adaptés à un parallélisme plus étendu. Voici quelques exemples concrets :

- Neural Machine Translation : accélération d'un facteur 6 sur 8 GPU;
- Inception/ImageNet : accélération d'un facteur 32 sur 50 GPU;
- RankBrain : accélération d'un facteur 300 sur 500 GPU.

302. Jianmin Chen *et al.*, « Revisiting Distributed Synchronous SGD » (2016) : <https://homl.info/68>.

Au-delà de quelques dizaines de GPU pour un modèle dense et de quelques centaines pour un modèle creux, la saturation entre en scène et les performances se dégradent. De nombreuses recherches étant menées pour tenter de résoudre ce problème (choix d'architectures P2P à la place de serveurs de paramètres centralisés, compression du modèle avec perte, optimisation du moment de la transmission et de son contenu, etc.), il est fort probable que la parallélisation des réseaux de neurones fasse de grands progrès dans les prochaines années.

En attendant, pour réduire le problème de saturation, il vaut mieux regrouper les GPU sur quelques serveurs parfaitement interconnectés. Vous pouvez également essayer d'abaisser la précision des paramètres à virgule flottante du modèle de 32 bits (`tf.float32`) à 16 bits (`tf.bfloat16`). Cela permet de diviser par deux la quantité de données à transférer, avec peu d'impact sur le taux de convergence ou sur les performances du modèle. Enfin, si vous utilisez des paramètres centralisés, vous pouvez les répartir sur plusieurs serveurs de paramètres. En ajoutant des serveurs de paramètres, vous réduisez la charge du réseau sur chacun et limitez le risque de saturation de la bande passante.

Entraînons à présent un modèle sur plusieurs GPU !

11.4.3 Entrainer à grande échelle en utilisant l'API *Distribution Strategies*

L'entraînement de nombreux modèles peut très bien se faire sur un seul GPU, voire sur un CPU. Mais s'il prend trop de temps, vous pouvez essayer de le distribuer sur plusieurs GPU sur la même machine. S'il reste encore trop long, vous pouvez opter pour des GPU plus puissants ou ajouter d'autres GPU à la machine. Si votre modèle a besoin de calculs intensifs (comme des multiplications de grandes matrices), il s'exécutera plus rapidement sur des GPU puissants et vous pouvez même essayer d'utiliser des TPU sur Google Cloud AI Platform, qui seront encore plus efficaces sur de tels modèles.

Mais si vous n'avez pas la possibilité d'équiper votre machine d'autres GPU et si les TPU ne sont pas pour vous (par exemple, il est possible que votre modèle n'en profite pas tant que ça ou que vous préfériez utiliser votre propre infrastructure matérielle), vous pouvez envisager l'entraînement sur plusieurs serveurs, chacun avec de multiples GPU (si cela n'est toujours pas suffisant, vous pouvez, en dernier recours, essayer d'ajouter le parallélisme du modèle, mais cela demande beaucoup d'efforts). Dans cette section, nous verrons comment entraîner des modèles à grande échelle, en commençant par plusieurs GPU sur la même machine (ou des TPU), puis nous passerons à plusieurs GPU sur plusieurs machines.

Heureusement, TensorFlow dispose d'une API très simple qui prend en charge toute la complexité de la méthode : l'API *Distribution Strategies*. Pour entraîner un modèle Keras sur tous les GPU disponibles (sur une seule machine, pour le moment) en utilisant le parallélisme des données avec la stratégie de mise en miroir, créez un objet `MirroredStrategy`, invoquez sa méthode `scope()` de façon à obtenir un contexte de distribution et enveloppez la création et la compilation

du modèle dans ce contexte. Ensuite, appelez la méthode `fit()` du modèle de manière classique :

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # Doit être divisible par le nombre de répliques
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

`tf.keras` est conscient de la distribution. Par conséquent, dans ce contexte `MirroredStrategy`, il sait qu'il doit répliquer toutes les variables et toutes les opérations sur tous les GPU disponibles. Notez que la taille du lot d'entraînement doit être divisible par le nombre de répliques, car la méthode `fit()` le scindera automatiquement sur tous les répliques. Voilà tout ! L'entraînement sera généralement beaucoup plus rapide qu'en utilisant un seul processeur et la modification du code est vraiment minimale.

Dès que l'entraînement du modèle est terminé, vous pouvez l'utiliser pour réaliser efficacement des prédictions. Appelez la méthode `predict()` pour qu'elle répartisse automatiquement le lot sur tous les répliques, effectuant les prédictions en parallèle (de nouveau, la taille du lot doit être divisible par le nombre de répliques). Si vousappelez la méthode `save()` du modèle, il sera enregistré non pas comme un modèle en miroir avec plusieurs répliques, mais comme un modèle normal. Par conséquent, lorsque vous le chargez, il s'exécutera en tant que modèle normal, sur un seul processeur (par défaut le GPU 0, ou, en l'absence de GPU, sur le CPU). Si vous souhaitez le charger et l'exécuter sur tous les processeurs disponibles, vous devez appeler `keras.models.load_model()` depuis un contexte de distribution :

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

Pour n'utiliser qu'un sous-ensemble de tous les GPU disponibles, passez-en la liste au constructeur de `MirroredStrategy` :

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

Par défaut, la classe `MirroredStrategy` exploite la bibliothèque NCCL de Nvidia (*NVIDIA Collective Communications Library*) pour le calcul de la moyenne avec `AllReduce`, mais vous pouvez modifier ce fonctionnement en fixant l'argument `cross_device_ops` à une instance de la classe `tf.distribute.HierarchicalCopyAllReduce` ou bien de la classe `tf.distribute.ReductionToOneDevice`. NCCL s'oriente par défaut vers la classe `tf.distribute.NcclAllReduce`, qui se révèle généralement plus rapide, mais cela dépend du nombre et du type de GPU. Il vous faudra peut-être essayer les autres options³⁰³.

303. Pour de plus amples informations sur les algorithmes `AllReduce`, consultez le billet rédigé par Yuichiro Ueno (<https://homl.info/uenopost>) et la page sur la mise à l'échelle avec NCCL (<https://homl.info/ncclalgo>).

Si vous voulez essayer le parallélisme des données avec centralisation des paramètres, remplacez `MirroredStrategy` par `CentralStorageStrategy`:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

Dans l'argument `compute_devices`, vous pouvez également préciser la liste des processeurs qui serviront d'ouvriers (par défaut, tous les GPU disponibles seront utilisés) et, dans l'argument `parameter_device`, le processeur qui devra stocker les paramètres (par défaut, il s'agira du CPU, ou du GPU s'il n'y en a qu'un).

Voyons à présent comment entraîner un modèle sur un groupe de serveurs TensorFlow !

11.4.4 Entraîner un modèle sur une partition TensorFlow

Une *partition TensorFlow* est un groupe de processus TensorFlow qui s'exécutent en parallèle, en général sur différentes machines, et qui communiquent les uns avec les autres pour réaliser une tâche, comme entraîner ou exécuter un réseau de neurones. Chaque processus TF de la partition est appelé *tâche*, ou *serveur TF*. Il possède une adresse IP, un port et un type (également appelé son *rôle* ou *job*). Le type peut être "worker", "chief", "ps" (pour serveur de paramètres) ou "evaluator":

- Chaque *ouvrier* (*worker*) effectue des calculs, en général sur une machine dotée d'un ou plusieurs GPU.
- Le *maître* (*chief*) effectue lui aussi des calculs (il s'agit d'un ouvrier) mais réalise des opérations supplémentaires, comme remplir des journaux TensorBoard ou effectuer des points de sauvegarde. Il n'y a qu'un seul maître dans une partition. Si aucun maître n'est désigné, le premier ouvrier le devient.
- Un *serveur de paramètres* (*ps*) conserve uniquement des valeurs de variables et se trouve généralement sur une machine équipée seulement d'un CPU. Ce type de tâches n'est utilisé qu'avec `ParameterServerStrategy`.
- Un *évaluateur* (*evaluator*) prend évidemment en charge l'évaluation.

Pour démarrer une partition TensorFlow, vous devez commencer par la définir, c'est-à-dire préciser l'adresse IP, le port TCP et le type de chaque tâche. Par exemple, la *spécification de partition* suivante définit une partition possédant trois tâches (deux ouvriers et un serveur de paramètres; voir la figure 11.21). La spécification est un dictionnaire avec une clé par job, et les valeurs sont des listes d'adresses de tâches (*IP:port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",    # /job:worker/task:0
        "machine-b.example.com:2222"      # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"]   # /job:ps/task:0
}
```

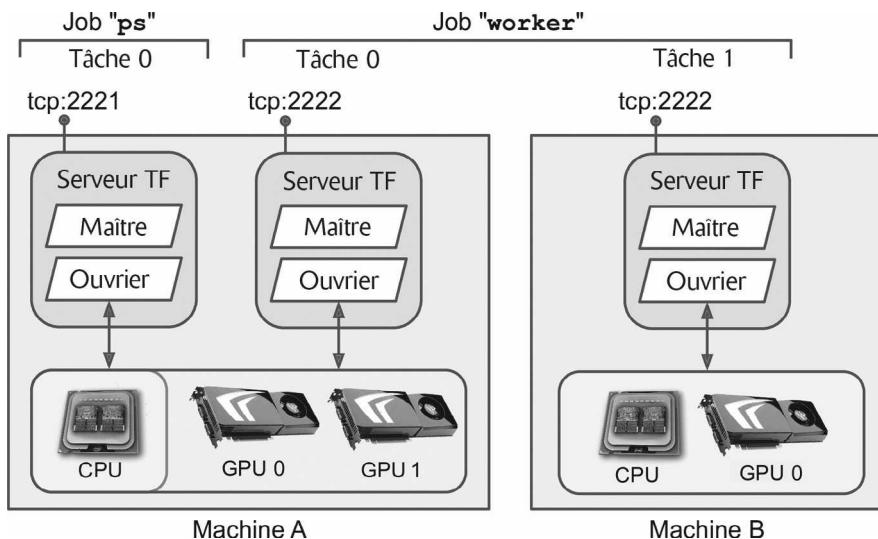


Figure 11.21 – Partition TensorFlow

De façon générale, il y a une tâche par machine, mais, comme le montre cet exemple, vous pouvez configurer plusieurs tâches sur la même machine (si elles partagent les mêmes GPU, assurez-vous que la mémoire est correctement répartie).



Par défaut, chaque tâche de la partition peut communiquer avec toutes les autres tâches. Assurez-vous que votre pare-feu est configuré de façon à autoriser les communications entre ces machines sur les ports indiqués (il est généralement plus simple d'utiliser le même port sur toutes les machines).

Lorsque vous démarrez une tâche, vous devez lui fournir la spécification de partition et lui indiquer son type et son indice (par exemple, ouvrier 0). Pour tout préciser en même temps (spécification de partition et type et indice de la tâche courante), l'approche la plus simple consiste à définir la variable d'environnement `TF_CONFIG` avant de démarrer TensorFlow. Sa valeur doit être un dictionnaire JSON contenant une spécification de partition (sous la clé "cluster") et le type et l'indice de la tâche courante (sous la clé "task"). Par exemple, la variable d'environnement `TF_CONFIG` suivante utilise la partition que nous venons de définir et indique que la tâche à démarrer est le premier ouvrier :

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```



En règle générale, la variable `TF_CONFIG` sera définie en dehors de Python de façon à ne pas fixer le type et l'indice de la tâche courante dans le code (il pourra ainsi être utilisé sur tous les ouvriers).

Entraînons à présent un modèle sur une partition. Nous commencerons avec la stratégie de mise en miroir car elle est étonnamment simple. Nous devons tout d'abord donner la valeur appropriée à la variable `TF_CONFIG` pour chaque tâche. Il ne doit y avoir aucun serveur de paramètres (la clé "ps" est retirée de la spécification de partition) et, en général, vous opterez pour un seul ouvrier par machine. Vérifiez bien qu'un indice différent est attribué à chaque tâche. Terminez en exécutant le code d'entraînement suivant sur chaque ouvrier :

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile(...)

batch_size = 100    # Doit être divisible par le nombre de répliques
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Vous avez bien vu, le code n'a pas changé, à l'exception de l'utilisation de `MultiWorkerMirroredStrategy` (dans les prochaines versions, il est probable que `MirroredStrategy` prenne en charge à la fois le mode mono- et le mode multimachines). Au lancement de ce script sur les premiers ouvriers, ils restent bloqués à l'étape `AllReduce`. Mais, dès que le dernier ouvrier démarre, l'entraînement débute et vous les voyez tous progresser au même rythme (puisque'ils se synchronisent à chaque étape).

Pour cette stratégie de distribution, vous avez le choix entre deux implémentations de `AllReduce`: un algorithme `AllReduce` en anneau fondé sur des communications réseau gRPC, et une implémentation de NCCL. Le meilleur algorithme dépendra du nombre d'ouvriers, du nombre et du type de GPU, et du réseau. Par défaut, TensorFlow sélectionnera l'algorithme approprié à partir de certaines heuristiques, mais vous pouvez le choisir en passant `CollectiveCommunication.RING` ou `CollectiveCommunication.NCCL` (du package `tf.distribute.experimental`) au constructeur de la stratégie.

Si vous préférez mettre en place un parallélisme des données asynchrone avec des serveurs de paramètres, changez la stratégie en `ParameterServerStrategy`, ajoutez un ou plusieurs serveurs de paramètres et configurez `TF_CONFIG` de façon appropriée pour chaque tâche. Même si les ouvriers vont travailler de façon asynchrone, les répliques sur chaque ouvrier opéreront de façon synchrone.

Enfin, si vous avez accès à des TPU sur Google Cloud (<https://cloud.google.com/tpu/>), vous pouvez créer un `TPUStrategy` (puis l'utiliser comme les autres stratégies) :

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```



Si vous êtes un chercheur, vous pourriez être éligible à une utilisation gratuite des TPU. Pour de plus amples informations, rendez-vous sur <https://tensorflow.org/tfrc>.

Vous pouvez à présent entraîner des modèles sur plusieurs GPU et plusieurs serveurs. Vous pouvez vous féliciter ! Si vous souhaitez entraîner un modèle vaste, vous aurez besoin de nombreux GPU, sur de nombreux serveurs. Il vous faudra alors acheter un grand nombre de matériels ou gérer un grand nombre de machines virtuelles dans le cloud. Le plus souvent, il sera moins pénible et moins onéreux d'utiliser un service de cloud que de provisionner et de gérer toute cette infrastructure vous-même, uniquement lorsque vous en avez besoin. Voyons comment procéder sur GCP.

11.4.5 Exécuter des tâches d'entraînement volumineuses sur Google Cloud AI Platform

Si vous décidez d'utiliser Google AI Platform, vous pouvez déployer une tâche d'entraînement en utilisant le même code d'entraînement que celui exécuté sur votre propre partition TF. La plateforme se chargera de provisionner et de configurer autant de machines virtuelles avec GPU que vous le souhaitez (sous la contrainte de vos quotas).

Pour démarrer la tâche, vous aurez besoin de l'outil en ligne de commande `gcloud` fourni avec le SDK Google Cloud (<https://cloud.google.com/sdk/>). Vous pouvez soit installer le SDK sur votre machine, soit utiliser Google Cloud Shell sur GCP. Il s'agit d'un terminal disponible directement dans votre navigateur web. Il s'exécute sur une machine virtuelle Linux gratuite (Debian), sur laquelle le SDK est déjà installé et configuré. Cloud Shell est disponible partout dans GCP : cliquez simplement sur l'icône Activer Cloud Shell en partie supérieure droite de la page (voir la figure 11.22).



Figure 11.22 – Activer Google Cloud Shell

Si vous préférez installer le SDK sur votre machine, vous devrez, après son installation, l'initialiser en exécutant `gcloud init`. Vous devrez vous connecter à GCP et accorder l'accès à vos ressources GCP, puis sélectionner le projet GCP concerné (si vous avez plusieurs projets), ainsi que la région dans laquelle doit s'exécuter la tâche. La commande `gcloud` permet d'accéder à toutes les fonctionnalités de GCP, y compris celles employées précédemment. Il est inutile de passer à chaque fois par l'interface web ; vous pouvez écrire des scripts qui démarrent ou arrêtent les machines virtuelles, déplient des modèles ou effectuent toute autre opération GCP.

Avant de pouvoir exécuter le job d'entraînement, vous devez écrire le code d'entraînement, exactement comme vous l'avez fait précédemment pour une configuration distribuée (par exemple, en utilisant `ParameterServerStrategy`).

AI Platform s'occupera de fixer la valeur de `TF_CONFIG` sur chaque machine virtuelle. Ensuite, vous pouvez le déployer et l'exécuter sur une partition TF à l'aide d'une commande semblable à la suivante :

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
    --region asia-southeast1 \
    --scale-tier PREMIUM_1 \
    --runtime-version 2.0 \
    --python-version 3.5 \
    --package-path /my_project/src/trainer \
    --module-name trainer.task \
    --staging-bucket gs://my-staging-bucket \
    --job-dir gs://my-mnist-model-bucket/trained_model \
    --
    --my-extra-argument1 foo --my-extra-argument2 bar
```

Examînons ces options. La commande démarre une tâche d'entraînement nommée `my_job_20190531_164700`, dans la région `asia-southeast1`, en utilisant un *niveau d'évolutivité* `PREMIUM_1`: cela correspond à 20 ouvriers (y compris le maître) et 11 serveurs de paramètres (la liste des niveaux d'évolutivité est disponible à l'adresse <https://homl.info/scaletiers>). Toutes ces machines virtuelles seront basées sur la version d'exécution 2.0 d'AI Platform (une configuration de machine virtuelle qui comprend TensorFlow 2.0 et de nombreux autres packages)³⁰⁴ et sur Python 3.5. Le code d'entraînement se trouve dans le répertoire `/my_project/src/trainer` et la commande `gcloud` le placera automatiquement dans un package pip qui sera envoyé à GCS sur `gs://my-staging-bucket`. Ensuite, AI Platform va démarrer plusieurs machines virtuelles, y déployer ce package et exécuter le module `trainer.task`. Enfin, l'argument `--job-dir` et les arguments supplémentaires (c'est-à-dire tous ceux qui viennent après le séparateur `--`) seront passés au programme d'entraînement. La tâche maîtresse se servira généralement de `--job-dir` pour savoir où enregistrer le modèle final sur GCS, dans ce cas `gs://my-mnist-model-bucket/trained_model`.

Voilà tout ! Dans la console GCP, vous pouvez ouvrir le menu de navigation, aller à la rubrique Intelligence Artificielle et ouvrir AI Platform → Tâches. Vous devriez voir votre tâche en cours d'exécution et, si vous cliquez dessus, vous obtiendrez des graphiques montrant l'utilisation du CPU, du GPU et de la mémoire pour chaque tâche. Cliquez sur Afficher les journaux pour accéder aux journaux détaillés en utilisant Stackdriver.



Si vous placez les données d'entraînement sur GCS, vous pouvez y accéder en créant un `tf.data.TextLineDataset` ou un `tf.data.TFRecordDataset`. Il suffit d'utiliser des chemins GCS comme noms de fichiers (par exemple, `gs://my-data-bucket/my_data_001.csv`). Ces jeux de données se fondent sur le package `tf.io.gfile` pour l'accès aux fichiers, qu'ils soient locaux ou sur GCS (vérifiez que le compte de service utilisé ait accès à GCS).

³⁰⁴. Au moment de l'écriture de ces lignes, la version d'exécution 2.0 n'est pas encore disponible, mais elle devrait l'être lorsque vous les lirez. La liste des versions d'exécution disponibles se trouve à l'adresse <https://homl.info/runtimes>.

Si vous souhaitez expérimenter quelques valeurs d'hyperparamètres, vous pouvez simplement exécuter plusieurs tâches et indiquer les valeurs des hyperparamètres dans les arguments supplémentaires transmis aux tâches. Toutefois, si vous souhaitez tester de nombreux hyperparamètres de façon efficace, il est préférable d'utiliser le service de réglage d'hyperparamètres proposé par AI Platform.

11.4.6 Réglages d'hyperparamètres en boîte noire sur AI Platform

AI Platform fournit un puissant service de réglage d'hyperparamètres avec optimisation bayésienne appelé Google Vizier³⁰⁵ (<https://homl.info/vizier>). Pour en profiter, vous devez passer un fichier de configuration YAML au moment de la création de la tâche (--config tuning.yaml). Par exemple :

```
trainingInput:
  hyperparameters:
    goal: MAXIMIZE
    hyperparameterMetricTag: accuracy
    maxTrials: 10
    maxParallelTrials: 2
    params:
      - parameterName: n_layers
        type: INTEGER
        minValue: 10
        maxValue: 100
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: momentum
        type: DOUBLE
        minValue: 0.1
        maxValue: 1.0
        scaleType: UNIT_LOG_SCALE
```

Ce code indique à AI Platform que nous souhaitons maximiser l'indicateur nommé "accuracy" et que la tâche effectuera au maximum 10 essais (chaque essai exécutera le code d'entraînement afin d'entraîner le modèle à partir de zéro), avec un maximum de 2 essais en parallèle. Nous souhaitons qu'il ajuste deux hyperparamètres : `n_layers` (un entier entre 10 et 100) et `momentum` (un réel entre 0,1 et 1,0). L'argument `scaleType` stipule la distribution de probabilité *a priori* pour la valeur de l'hyperparamètre. `UNIT_LINEAR_SCALE` représente une distribution uniforme (c'est-à-dire aucune préférence *a priori*), tandis que `UNIT_LOG_SCALE` indique que notre croyance *a priori* est que la valeur optimale se trouvera près de la valeur maximale (l'autre distribution possible est `UNIT_REVERSE_LOG_SCALE`, à utiliser si l'on pense *a priori* que la valeur optimale sera proche de la valeur minimale).

Les arguments `n_layers` et `momentum` sont passés au code d'entraînement en tant qu'arguments de la ligne de commande. Ce code est, bien entendu, supposé les utiliser. Comment le code d'entraînement retourne-t-il l'indicateur à AI Platform afin que celui-ci puisse choisir les valeurs d'hyperparamètres qui seront

³⁰⁵. Daniel Golovin *et al.*, «Google Vizier: A Service for Black-Box Optimization», *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), 1487-1495: <https://homl.info/vizier>.

utilisées pour l'essai suivant ? AI Platform surveille le répertoire de sortie (indiqué par `--job-dir`) et attend l'arrivée d'un fichier d'événements (voir le chapitre 2) contenant les résumés d'un indicateur nommé "accuracy" (ou tout autre nom d'indicateur donné par `hyperparameterMetricTag`), puis lit ces valeurs. Votre code d'entraînement doit donc simplement utiliser le rappel `TensorBoard()` (que vous utiliserez de toute manière pour la supervision) !

Lorsque la tâche est terminée, toutes les valeurs des hyperparamètres utilisées dans chaque essai et la précision résultante sont disponibles dans la sortie de la tâche (accessible sur la page AI Platform → Tâches).



Les tâches AI Platform peuvent également être utilisées pour exécuter efficacement votre modèle sur de grandes quantités de données. Chaque ouvrier peut lire une partie des données depuis GCS, effectuer des prédictions et les enregistrer sur GCS.

Vous disposez à présent de tous les outils et de toutes les connaissances nécessaires à la création d'architectures de réseaux de neurones performantes et à leur entraînement à grande échelle en utilisant diverses stratégies de distribution, sur votre propre infrastructure ou dans le cloud. Vous pouvez même effectuer une optimisation bayésienne puissante pour affiner les hyperparamètres !

11.5 EXERCICES

1. Que contient un `SavedModel`? Comment pouvez-vous inspecter son contenu ?
2. Quand devez-vous utiliser TF Serving ? Quelles sont ses principales caractéristiques ? Donnez quelques outils que vous pouvez utiliser pour le mettre en service.
3. Comment déployez-vous un modèle sur plusieurs instances de TF Serving ?
4. Quand devez-vous utiliser l'API gRPC à la place de l'API REST pour interroger un modèle servi par TF Serving ?
5. De quelles manières TFLite réduit-il la taille d'un modèle afin qu'il puisse s'exécuter sur un périphérique mobile ou embarqué ?
6. Qu'est-ce qu'un entraînement conscient de la quantification et pourquoi en auriez-vous besoin ?
7. Expliquez ce que sont le parallélisme du modèle et le parallélisme des données. Pourquoi conseille-t-on généralement ce dernier ?
8. Lors de l'entraînement d'un modèle sur plusieurs serveurs, quelles stratégies de distribution pouvez-vous appliquer ? Comment choisissez-vous celle qui convient ?
9. Entraînez un modèle (celui que vous voulez) et déployez-le sur TF Serving ou Google Cloud AI Platform. Écrivez le code client qui l'interrogera au travers de l'API REST ou de l'API gRPC. Actualisez

le modèle et déployez la nouvelle version. Votre code client interrogera alors cette nouvelle version. Revenez à la première.

10. Entraînez n'importe quel modèle sur plusieurs GPU sur la même machine en utilisant `MirroredStrategy` (si vous n'avez pas accès à des GPU, vous pouvez employer `Colaboratory` dans le mode d'exécution avec GPU et créer deux GPU virtuels). Entraînez de nouveau le modèle en utilisant `CentralStorageStrategy` et comparez les temps d'entraînement.
11. Entraînez un petit modèle sur Google Cloud AI Platform, en utilisant le réglage d'hyperparamètres en boîte noire.

Les solutions de ces exercices sont données à l'annexe A.

Le mot de la fin

Avant de clore le dernier chapitre de cet ouvrage, je voudrais vous remercier de l'avoir lu jusqu'au bout. J'espère sincèrement que sa lecture vous a procuré autant de plaisir que moi à l'écrire, et qu'il sera utile à vos projets, petits ou grands.

Si vous découvrez des erreurs, avertissez-moi. Plus généralement, j'aimerais connaître votre avis, alors n'hésitez pas à me contacter par l'intermédiaire des éditions Dunod ou du projet GitHub [ageron/handson-ml2](#) ou @aureliengeron sur Twitter.

Le meilleur conseil que je puisse vous donner est de pratiquer et de pratiquer encore. Essayez de réaliser les exercices proposés, de jouer avec les notebooks Jupyter, de rejoindre Kaggle.com ou d'autres communautés ML, de suivre des cours sur l'apprentissage automatique, de lire des articles, de participer à des conférences et de rencontrer des experts. Il est également plus facile de progresser si vous avez un projet concret sur lequel travailler, que ce soit dans un cadre professionnel ou pour le plaisir (idéalement pour les deux). Par conséquent, si vous avez toujours rêvé de construire quelque chose, allez-y ! Avancez progressivement. Ne visez pas la Lune tout de suite, mais restez focalisé sur votre projet et construisez-le morceau par morceau. Vous devrez faire preuve de patience et de persévérance, mais, dès que votre robot marchera, que votre chatbot sera opérationnel ou que vous aurez réussi à construire ce que vous souhaitez, cela sera extrêmement gratifiant.

Mon plus grand souhait est que cet ouvrage vous incite à créer une application ML exceptionnelle qui bénéficiera à tout le monde. Quelle sera-t-elle ?

Annexe A

Solutions des exercices



Les solutions des exercices de programmation figurent dans les notebooks Jupyter disponibles en ligne sur <https://github.com/ageron/handson-ml2>.³⁰⁶

CHAPITRE 1 : LES FONDAMENTAUX DU MACHINE LEARNING

1. Si vous avez un jeu d’entraînement comportant des millions de variables, vous pouvez utiliser une descente de gradient stochastique ou une descente de gradient par mini-lots, ou encore une descente de gradient ordinaire si le jeu d’entraînement tient en mémoire. Mais vous ne pouvez pas utiliser l’équation normale, car le temps de calcul augmente très rapidement avec le nombre de variables (plus que quadratiquement).
2. Si les caractéristiques composant votre jeu d’entraînement ont des échelles très différentes, la fonction de coût aura la forme d’un bol allongé, et les algorithmes de descente de gradient mettront plus longtemps à converger. Pour résoudre ce problème, vous devez normaliser toutes les variables (c’est-à-dire les ramener à la même échelle) avant d’entraîner le modèle. Par contre, l’équation normale ne nécessite pas de changement d’échelle. Par ailleurs, si vous utilisez un modèle régularisé (ridge, lasso, elastic net...), il est d’autant plus important de normaliser les variables, car sans cela vous pourriez aboutir à une solution bien moins performante : dans un tel modèle, les poids des variables sont

^{306.} Les corrigés se trouvent en fin des notebooks des différents chapitres, sachant que le chapitre 1 du présent livre correspond au chapitre 4 sur Github, puis le chapitre 2 correspond au chapitre 10 sur Github, le chapitre 3 au 11, etc. jusqu’au 11, correspondant au 19.

contraints de rester petits, donc les variables dont les valeurs sont plus petites que les autres auront tendance à être ignorées.

3. Une descente de gradient ne peut pas rester bloquée sur un minimum local lors de l'entraînement d'un modèle de régression logistique, car la fonction de coût est convexe³⁰⁷.
4. Si la fonction à optimiser est convexe (comme par exemple pour une régression linéaire ou une régression logistique) et si le taux d'apprentissage n'est pas trop élevé, alors tous les algorithmes de descente de gradient s'approcheront du minimum global et produiront au final des modèles assez similaires. Cependant, si vous ne réduisez pas graduellement le taux d'apprentissage, les descentes de gradient stochastique et par mini-lots ne convergeront jamais véritablement : au lieu de cela, elles continueront à errer autour du minimum global. Cela signifie que, même si vous les laissez s'exécuter pendant très longtemps, ces algorithmes de descente de gradient produiront des modèles légèrement différents.
5. Si l'erreur de validation augmente régulièrement après chaque époque, alors il se peut que le taux d'apprentissage soit trop élevé et que l'algorithme diverge. Si l'erreur d'entraînement augmente également, alors c'est clairement là qu'est le problème et vous devez réduire le taux d'apprentissage. Par contre, si l'erreur d'entraînement n'augmente pas, votre modèle surajuste le jeu d'entraînement et vous devez arrêter l'entraînement.
6. Du fait de leur nature aléatoire, rien ne garantit qu'une descente de gradient stochastique ou qu'une descente de gradient par mini-lots fera des progrès à chaque itération de l'entraînement. Par conséquent, si vous arrêtez immédiatement l'entraînement dès que l'erreur de validation augmente, vous risquez de vous arrêter bien trop tôt, alors que le minimum n'est pas atteint. Une meilleure solution consiste à sauvegarder le modèle à intervalles réguliers, puis, lorsqu'il ne s'est plus amélioré pendant assez longtemps (ce qui signifie probablement qu'il ne fera jamais mieux), vous pouvez revenir au meilleur modèle sauvegardé.
7. La descente de gradient stochastique est celle dont l'itération d'entraînement est la plus rapide étant donné qu'elle ne prend en compte qu'une seule observation d'entraînement à la fois, et par conséquent c'est celle qui arrive le plus rapidement à proximité du minimum global (ou la descente de gradient par mini-lots, lorsque la taille du lot est très petite). Cependant, seule la descente de gradient ordinaire convergera effectivement, si le temps d'entraînement est suffisant. Comme expliqué ci-dessus, les descentes de gradient stochastique et par mini-lots continueront à errer autour du minimum, à moins de réduire graduellement le taux d'apprentissage.

307. Le segment de droite reliant deux points quelconques de la courbe ne traverse jamais la courbe.

8. Si l'erreur de validation est beaucoup plus élevée que l'erreur d'entraînement, c'est vraisemblablement parce que votre modèle surajuste le jeu d'entraînement. Un des remèdes consiste à réduire le degré polynomial : un modèle ayant moins de degrés de liberté risque moins de surajuster. Vous pouvez également essayer de régulariser le modèle en ajoutant par exemple une pénalité ℓ_2 (régression de crête) ou ℓ_1 (lasso) à la fonction de coût : ceci réduira aussi les degrés de liberté du modèle. Enfin vous pouvez essayer d'accroître la taille du jeu d'entraînement.
9. Si l'erreur d'entraînement et l'erreur de validation sont à peu près égales et assez élevées, le modèle sous-ajuste vraisemblablement le jeu d'entraînement, ce qui signifie que le biais est important. Vous devez essayer de réduire l'hyperparamètre de régularisation α .
10. Voyons voir :
 - Un modèle avec un peu de régularisation donne en général de meilleurs résultats qu'un modèle sans aucune régularisation, c'est pourquoi vous préférerez en général la régression ridge à la régression linéaire simple³⁰⁸.
 - La régression lasso utilise une pénalité ℓ_1 qui tend à ramener les coefficients de pondération à zéro exactement. Ceci conduit à des modèles creux, où tous les poids sont nuls sauf les plus importants. C'est une façon de sélectionner automatiquement des variables, ce qui est judicieux si vous soupçonnez que seules certaines d'entre elles ont de l'importance. Si vous n'en êtes pas sûr, préférez la régression ridge.
 - Elastic net est préférée en général à lasso, car cette dernière méthode peut se comporter de manière désordonnée dans certains cas (lorsque plusieurs variables sont fortement corrélées ou lorsqu'il y a plus de variables que d'observations d'entraînement). Cependant, cela fait un hyperparamètre supplémentaire à ajuster. Si vous souhaitez une régression lasso mais sans comportement désordonné, vous pouvez utiliser elastic net avec un `l1_ratio` proche de 1.
11. Si vous voulez classer des photos en extérieur/intérieur et jour/nuit, sachant qu'il ne s'agit pas de classes s'excluant mutuellement (les 4 combinaisons sont possibles), vous devez entraîner deux classificateurs de régression logistique.

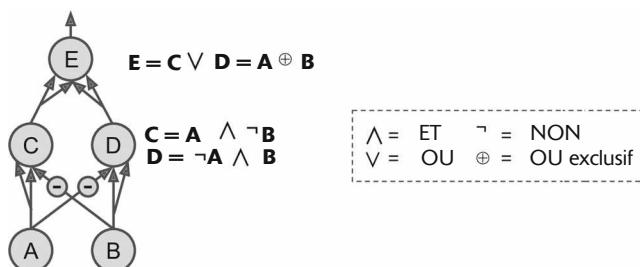
Pour la solution de l'exercice 12, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³⁰⁹

³⁰⁸. De plus, la résolution de l'équation normale nécessite l'inversion d'une matrice, mais cette matrice n'est pas toujours inversible. Par contraste, la matrice de la régression ridge est toujours inversible.

³⁰⁹. Voir « `04_training_linear_models.ipynb` ».

CHAPITRE 2 : INTRODUCTION AUX RÉSEAUX DE NEURONES ARTIFICIELS AVEC KERAS

- Visitez TensorFlow Playground (<https://playground.tensorflow.org/>) et jouez avec, comme décrit dans l'énoncé de cet exercice.
- Le réseau de neurones suivant, fondé sur les neurones artificiels de base, calcule $A \oplus B$ (où \oplus représente le OU exclusif), en exploitant le fait que $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. Il existe d'autres solutions, par exemple en prenant $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$, ou encore $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$, etc.



- Un perceptron classique convergera uniquement si le jeu de données peut être séparé de façon linéaire et ne sera pas capable d'estimer des probabilités de classes. À l'inverse, un classificateur à régression logistique convergera vers une bonne solution même si le jeu de données n'est pas séparable linéairement et produira des probabilités de classes. Si vous remplacez la fonction d'activation du perceptron par la fonction d'activation logistique (ou la fonction d'activation softmax en cas de neurones multiples) et si vous l'entraînez à l'aide de la descente de gradient (ou tout autre algorithme d'optimisation qui minimise la fonction de coût, comme l'entropie croisée), il devient équivalent à un classificateur à régression logistique.
- La fonction d'activation logistique a été un élément-clé de l'entraînement des premiers PMC, car sa dérivée est toujours différente de zéro, et la descente de gradient peut donc toujours aller vers le bas de la pente. Lorsque la fonction d'activation est une fonction échelon, la pente est nulle et la descente de gradient ne peut donc pas se déplacer.
- La fonction échelon, la fonction logistique (sigmoïde), la tangente hyperbolique (tanh) et la fonction ReLU (Rectified Linear Unit) (voir la figure 2.8) sont des fonctions d'activation répandues. D'autres exemples sont donnés au chapitre 3, comme ELU et des variantes de ReLU.
- Le PMC décrit dans la question est constitué d'une couche d'entrée avec 10 neurones intermédiaires, suivie d'une couche cachée de 50 neurones artificiels, et d'une couche de sortie avec 3 neurones

- artificiels. Tous les neurones artificiels utilisent la fonction d'activation ReLU.
- a. La forme de la matrice d'entrée \mathbf{X} est $m \times 10$, où m représente la taille du lot d'entraînement.
 - b. La forme du vecteur des poids \mathbf{W}_h de la couche cachée est 10×50 , et la longueur de son vecteur de termes constants \mathbf{b}_h est 50.
 - c. La forme du vecteur des poids \mathbf{W}_o de la couche de sortie est 50×3 , et la longueur de son vecteur de termes constants \mathbf{b}_o est 3.
 - d. La forme de la matrice de sortie du réseau \mathbf{Y} est $m \times 3$.
 - e. $\mathbf{Y} = \text{ReLU}(\text{ReLU}(\mathbf{X}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_o + \mathbf{b}_o)$. Rappelons que la fonction ReLU se contente de remplacer par des zéros les termes négatifs de la matrice. Par ailleurs, notez que lorsque vous additionnez un vecteur de termes constants et une matrice, le vecteur est ajouté à chacune des lignes de la matrice. Cette transformation est appelée *broadcasting*.
7. Pour classer des messages électroniques dans les catégories *spam* et *ham*, la couche de sortie d'un réseau de neurones a besoin d'un seul neurone, indiquant, par exemple, la probabilité que le message soit non sollicité. Pour estimer une probabilité, vous pouvez généralement utiliser la fonction d'activation logistique dans la couche de sortie. Si, à la place, vous voulez traiter le jeu MNIST, vous aurez besoin de dix neurones dans la couche de sortie et la fonction logistique doit être remplacée par la fonction d'activation softmax, car celle-ci est capable de prendre en charge des classes multiples en produisant une probabilité par classe. Si le réseau de neurones doit prédire les prix des maisons, vous avez alors besoin d'un neurone de sortie, sans aucune fonction d'activation dans la couche de sortie³¹⁰.
8. La rétropropagation est une technique que l'on utilise pour entraîner les réseaux de neurones artificiels. Elle commence par calculer les gradients de la fonction de coût par rapport à chaque paramètre du modèle (tous les poids et les termes constants), puis elle réalise une étape de descente de gradient en utilisant les gradients calculés. Cette étape de rétropropagation est généralement effectuée des milliers ou des millions de fois, en utilisant de nombreux lots d'entraînement, jusqu'à ce que les paramètres du modèle convergent vers des valeurs qui (avec un peu de chance) minimisent la fonction de coût. Pour calculer les gradients, la rétropropagation utilise une différentiation automatique en mode inverse (elle n'était pas appelée ainsi lorsque la rétropropagation a été imaginée, et elle a été réinventée à plusieurs reprises).

³¹⁰. Lorsque les valeurs à prédire ont des échelles très variables, il est possible de prédire le logarithme de la valeur cible plutôt que directement celle-ci. Il suffit de calculer l'exponentielle de la sortie du réseau de neurones pour obtenir la valeur estimée, car $\exp(\log(v)) = v$.

La différentiation automatique en mode inverse effectue une passe en avant sur le graphe de calcul, déterminant la valeur de chaque nœud pour le lot d’entraînement courant, puis réalise une passe en arrière, calculant tous les gradients en une seule fois (voir l’annexe B). Quelle est donc la différence ? La rétropropagation fait référence à l’intégralité du processus d’entraînement d’un réseau de neurones artificiels, qui implique plusieurs étapes de rétropropagation, chacune calculant des gradients et utilisant ceux-ci pour effectuer une étape de descente de gradient. En comparaison, la différentiation automatique en mode inverse n’est qu’une technique de calcul efficace des gradients, employée par la rétropropagation.

9. Voici une liste de tous les hyperparamètres que vous pouvez ajuster dans un PMC de base : le nombre de couches cachées, le nombre de neurones dans chaque couche cachée et la fonction d’activation utilisée dans chaque couche cachée et dans la couche de sortie³¹¹. En général, la fonction d’activation ReLU (ou l’une de ses variantes ; voir le chapitre 3) se révèle un bon choix par défaut pour les couches cachées. Pour la couche de sortie, la fonction d’activation logistique convient aux classifications binaires, la fonction d’activation softmax est adaptée aux classifications à classes multiples, et aucune fonction d’activation n’est requise pour une régression.
Si le PMC surajuste les données d’entraînement, vous pouvez essayer de réduire le nombre de couches cachées, ainsi que leur nombre de neurones.
10. Consultez le notebook Jupyter disponible à l’adresse <https://github.com/ageron/handson-ml2>.³¹²

CHAPITRE 3 : ENTRAÎNEMENT DE RÉSEAUX DE NEURONES PROFONDS

1. Non, tous les poids doivent être échantillonnés indépendamment et ils ne doivent pas avoir la même valeur initiale. L’échantillonnage aléatoire des poids a pour objectif important de casser les symétries : si tous les poids possèdent la même valeur initiale, même différente de zéro, la symétrie n’est pas rompue (tous les neurones d’une couche donnée sont équivalents) et la rétropropagation ne sera pas en mesure d’y remédier. Concrètement, cela signifie que tous les neurones d’une couche donnée auront toujours le même poids, ce

311. Le chapitre 3 présente de nombreuses techniques qui ajoutent des hyperparamètres supplémentaires : type d’initialisation des poids, hyperparamètres de la fonction d’activation (par exemple, niveau de fuite dans Leaky ReLU), seuil d’écritage des gradients, type d’optimiseur et ses hyperparamètres (par exemple, l’inertie avec un `MomentumOptimizer`), type de régularisation pour chaque couche et les hyperparamètres de la régularisation (par exemple, taux de dropout), etc.

312. Voir « `10_neural_nets_with_keras.ipynb` ».

qui revient à n'avoir qu'un seul neurone par couche, en beaucoup plus lent. Il est quasi impossible qu'une telle configuration converge vers une bonne solution.

2. Il est tout à fait approprié d'initialiser les termes constants à zéro. Certaines personnes préfèrent les initialiser comme les poids, et cela convient également ; la différence n'est pas vraiment importante.
3. Voici quelques avantages de la fonction SELU par rapport à la fonction ReLU :
 - Elle accepte des valeurs négatives et la sortie moyenne des neurones de n'importe quelle couche est donc généralement plus proche de zéro qu'avec la fonction ReLU (qui ne génère jamais de valeurs de sortie négatives). Cela aide à réduire les effets du problème de disparition des gradients.
 - Elle possède toujours une dérivée différente de zéro, ce qui évite le problème de mort des unités que l'on peut rencontrer avec la fonction ReLU.
 - Si les conditions sont bonnes (c'est-à-dire lorsque le modèle est séquentiel, lorsque les poids sont initialisés à l'aide de l'initialisation de LeCun, lorsque les entrées sont normalisées et lorsqu'il n'y a pas de couches ni de régularisation incompatible, comme le dropout et la régularisation ℓ_1), alors la fonction d'activation SELU garantit l'autonormalisation du modèle, ce qui résout le problème d'explosion/disparition des gradients.
4. La fonction d'activation SELU constitue un bon choix par défaut. Si le réseau de neurones doit être le plus rapide possible, vous pouvez choisir à la place l'une des variantes de Leaky ReLU (par exemple, une simple Leaky ReLU avec la valeur d'hyperparamètre par défaut). En raison de sa simplicité, la fonction d'activation ReLU a souvent la préférence de nombreuses personnes, malgré les meilleures performances générales des fonctions SELU et Leaky ReLU. Sa capacité à produire en sortie exactement zéro peut se révéler utile dans certains cas (voir le chapitre 9). De plus, elle peut parfois profiter des implémentations optimisées ainsi que des accélérations matérielles. La tangente hyperbolique (tanh) sera intéressante dans la couche de sortie si vous devez générer un nombre entre -1 et 1, mais elle n'est plus beaucoup utilisée aujourd'hui dans les couches cachées (excepté dans les réseaux récurrents). La fonction d'activation logistique convient également dans la couche de sortie lorsque vous devez estimer une probabilité (par exemple, pour une classification binaire), mais on la retrouve rarement dans les couches cachées (il existe des exceptions, par exemple pour la couche de codage des autoencodeurs variationnels ; voir le chapitre 9). Enfin, la fonction d'activation softmax est utile dans la couche de sortie afin de produire des probabilités pour les classes mutuellement exclusives, mais elle est rarement, voire jamais, employée dans les couches cachées.

5. Si la valeur de l'hyperparamètre `momentum` est trop proche de 1 (par exemple, 0,99999) lorsqu'un optimiseur SGD est utilisé, l'algorithme va prendre beaucoup de vitesse et, avec un peu de chance, aller en direction du minimum global, mais il va dépasser celui-ci en raison de son inertie. Il va ensuite ralentir et revenir, réaccélérer, aller de nouveau trop loin, etc. Il peut osciller ainsi à de nombreuses reprises avant de converger. Globalement, le temps de convergence sera beaucoup plus long qu'avec une valeur de `momentum` plus faible.
6. Pour produire un modèle creux (c'est-à-dire avec la plupart des poids égaux à zéro), une solution consiste à entraîner le modèle normalement, puis à mettre à zéro les poids très faibles. Pour une dispersion encore plus importante, vous pouvez appliquer une régularisation ℓ_1 pendant l'entraînement afin de pousser l'optimiseur vers une dispersion. Une troisième option consiste à utiliser TF-MOT (*TensorFlow Model Optimization Toolkit*).
7. Oui, le dropout ralentit l'entraînement, en général d'un facteur 2 environ. Toutefois, il n'a aucun impact sur l'inférence, car il n'est actif que pendant l'entraînement. MC Dropout est identique à dropout pendant l'entraînement, mais il reste actif pendant l'inférence. Chaque inférence est donc légèrement ralenti. Toutefois, avec MC Dropout, l'inférence est généralement exécutée au moins dix fois pour obtenir de meilleures prédictions. Autrement dit, la réalisation des prédictions est ralenti d'un facteur 10 ou plus.

Pour la solution de l'exercice 8, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹³

CHAPITRE 4 : MODÈLES PERSONNALISÉS ET ENTRAÎNEMENT AVEC TENSORFLOW

1. TensorFlow est une bibliothèque open source pour les calculs numériques, particulièrement bien adaptée et ajustée au Machine Learning à grande échelle. Sa base est comparable à NumPy, mais elle prend également en charge les GPU, les calculs distribués, l'analyse d'un graphe de calcul et son optimisation (avec un format de graphe portable qui vous permet d'entraîner un modèle TensorFlow dans un environnement et de l'exécuter dans un autre), une API d'optimisation basée sur la différentiation automatique en mode inverse et plusieurs autres API, comme `tf.keras`, `tf.data`, `tf.image`, `tf.signal`, etc. PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2 et Chainer sont d'autres bibliothèques populaires pour le Deep Learning.

313. Voir « `11_training_deep_neural_networks.ipynb` ».

2. Bien que la bibliothèque TensorFlow offre la majorité des fonctionnalités que l'on trouve dans NumPy, elle n'en est pas un remplaçant direct, pour plusieurs raisons. Tout d'abord, les noms des fonctions ne sont pas toujours identiques (par exemple `tf.reduce_sum()` à la place de `np.sum()`). Ensuite, certaines fonctions n'affichent pas exactement le même comportement (par exemple, `tf.transpose()` crée une copie transposée d'un tenseur, tandis que l'attribut `T` de NumPy crée une vue transposée, sans réelle copie des données). Enfin, les tableaux NumPy sont modifiables, contrairement aux tenseurs TensorFlow (mais vous pouvez employer un `tf.Variable` si vous avez besoin d'un objet modifiable).
3. Les appels `tf.range(10)` et `tf.constant(np.arange(10))` retournent tous deux un tenseur à une dimension contenant les entiers 0 à 9. Toutefois, le premier utilise des entiers sur 32 bits tandis que le second utilise des entiers sur 64 bits. Par défaut, TensorFlow est en mode 32 bits, tandis que NumPy est en mode 64 bits.
4. Outre les tenseurs normaux, TensorFlow propose plusieurs autres structures de données, comme les tenseurs creux, les tableaux de tenseurs, les tenseurs irréguliers, les files d'attente, les tenseurs chaînes de caractères et les ensembles. Ces deux derniers sont en réalité représentés sous forme de tenseurs normaux, mais TensorFlow fournit des fonctions spéciales pour les manipuler (dans `tf.strings` et `tf.sets`).
5. En général, vous pouvez simplement implémenter une fonction de perte personnalisée sous forme de fonction Python normale. Cependant, si elle doit accepter des hyperparamètres (ou tout autre état), elle doit être une sous-classe de `keras.losses.Loss` et implémenter les méthodes `__init__()` et `call()`. Pour que les hyperparamètres de la fonction de perte soient enregistrés avec le modèle, vous devez également implémenter la méthode `get_config()`.
6. À l'instar des fonctions de perte personnalisées, la plupart des indicateurs peuvent être définis sous forme de fonctions Python normales. Mais, si votre indicateur personnalisé doit prendre en charge des hyperparamètres (ou tout autre état), il doit dériver de la classe `keras.metrics.Metric`. Par ailleurs, si le calcul de l'indicateur sur l'intégralité d'une époque diffère du calcul de l'indicateur moyen sur tous les lots de cette époque (par exemple, pour les indicateurs de précision et de rappel), vous devez dériver de la classe `keras.metrics.Metric` et implémenter les méthodes `__init__()`, `update_state()` et `result()` afin de gérer en continu l'indicateur tout au long de chaque époque. Vous devez également implémenter la méthode `reset_states()`, sauf s'il suffit de réinitialiser toutes les variables à 0,0. Pour que l'état soit

- enregistré avec le modèle, la méthode `get_config()` doit être implémentée.
7. Vous devez distinguer les composants internes du modèle (c'est-à-dire les couches ou les blocs de couches réutilisables) et le modèle lui-même (c'est-à-dire l'objet qui sera entraîné). Les premiers doivent dériver de la classe `keras.layers.Layer`, tandis que le dernier doit être une sous-classe de `keras.models.Model`.
 8. L'écriture d'une boucle d'entraînement personnalisée est une opération assez complexe et ne doit être envisagée qu'en cas de besoin avéré. Keras fournit plusieurs outils pour personnaliser l'entraînement sans avoir à écrire une boucle personnalisée : les rappels, les régulariseurs personnalisés, les contraintes personnalisées, les pertes personnalisées, etc. Dans la mesure du possible, vous devez les employer au lieu d'écrire une boucle d'entraînement personnalisée, car un tel développement est sujet aux erreurs et le code produit sera difficile à réutiliser. Toutefois, dans certains cas, vous n'aurez pas le choix, par exemple si vous souhaitez utiliser différents optimiseurs pour différentes parties du réseau de neurones, comme dans l'article Wide & Deep (<https://homl.info/widedeep>). Une boucle d'entraînement personnalisée pourra également être utile lors du débogage ou pour comprendre précisément comment fonctionne l'entraînement.
 9. Les composants Keras personnalisés doivent être convertibles en fonctions TF. Autrement dit, vous devez vous limiter autant que possible aux opérations TF et respecter toutes les règles données au paragraphe 4.4.2 « Règles d'une fonction TF ». Si vous devez absolument inclure du code Python quelconque dans un composant personnalisé, vous pouvez soit le placer dans une opération `tf.py_function()` (mais cela réduit les performances et limite la portabilité du modèle), soit indiquer `dynamic=True` lors de la création de la couche personnalisée ou du modèle (ou préciser `run_eagerly=True` lors de l'appel à la méthode `compile()` du modèle).
 10. La liste des règles à respecter lors de la création d'une fonction TF est donnée au paragraphe 4.4.2 « Règles d'une fonction TF ».
 11. La création d'un modèle Keras dynamique peut être utile pour le débogage. En effet, les composants personnalisés ne seront pas compilés en fonctions TF et vous pourrez utiliser n'importe quel débogueur Python pour analyser le code. Elle pourra également se révéler utile si vous souhaitez inclure du code Python quelconque dans votre modèle (ou dans le code d'entraînement), y compris des appels aux bibliothèques externes. Pour qu'un modèle soit dynamique, vous devez préciser `dynamic=True` au moment de sa création. Vous pouvez également fixer `run_eagerly` à `True` lors de l'invocation de la méthode `compile()` du modèle. Lorsqu'un modèle est

dynamique, Keras n'est pas en mesure d'utiliser les fonctionnalités de graphe de TensorFlow. L'entraînement et les inférences seront donc ralentis et vous n'aurez pas la possibilité d'exporter le graphe de calcul. La portabilité du modèle sera donc limitée.

Pour les solutions des exercices 12 et 13, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹⁴

CHAPITRE 5 : CHARGEMENT ET PRÉTRAITEMENT DE DONNÉES AVEC TENSORFLOW

1. Le chargement et le prétraitement efficaces d'un jeu de données volumineux peuvent représenter un véritable défi d'ingénierie. L'API Data simplifie ces opérations. Elle fournit de nombreuses fonctionnalités, y compris le chargement de données à partir de différentes sources (comme des fichiers textuels ou binaires), la lecture de données en parallèle à partir de plusieurs sources, leur transformation, l'entrelacement des enregistrements, le mélange des données, leur mise en lots et leur prélecture.
2. Le découpage d'un jeu de données volumineux en plusieurs fichiers permet son mélange à un niveau grossier, avant de le mélanger à un niveau plus fin en utilisant un tampon de mélange. Cela permet également de prendre en charge des jeux de données trop gros pour tenir sur une seule machine. Il est également plus facile de manipuler des milliers de petits fichiers qu'un énorme fichier, comme décomposer les données en plusieurs sous-ensembles. Enfin, si les données sont réparties dans plusieurs fichiers distribués sur plusieurs serveurs, il est possible de télécharger simultanément plusieurs fichiers à partir de différents serveurs, améliorant ainsi l'utilisation de la bande passante.
3. Vous pouvez employer TensorBoard pour visualiser des données de profilage : si le GPU n'est pas pleinement utilisé, alors le pipeline d'entrée constitue probablement le goulot d'étranglement. Pour résoudre ce problème, vous pouvez faire en sorte que les données soient lues et prétraitées dans plusieurs threads en parallèle, et effectuer une prélecture de quelques lots. Si cela ne suffit pas à l'exploitation totale du GPU pendant l'entraînement, assurez-vous que le code de prétraitement est optimisé. Vous pouvez également essayer d'enregistrer le jeu de données dans plusieurs fichiers TFRecord et, si nécessaire, d'effectuer une partie du prétraitement à l'avance afin qu'elle ne se fasse pas à la volée pendant l'entraînement (TF Transform peut vous aider sur ce point). Si nécessaire, prenez une machine avec une plus grande capacité de calcul et une plus

314. Voir « `12_custom_models_and_training_with_tensorflow.ipynb` ».

grande quantité de mémoire. Assurez-vous que la bande passante du GPU est suffisamment importante.

4. Un fichier TFRecord est constitué d'une suite d'enregistrements binaires quelconques. Vous pouvez stocker absolument n'importe quelles données binaires dans chaque enregistrement. Cependant, en pratique, la plupart des fichiers TFRecord contiennent des suites de protobufs sérialisés. Il est ainsi possible de bénéficier des avantages des protobufs, comme le fait qu'ils puissent être lus facilement sur diverses plateformes et langages, et que leur définition puisse être modifiée ultérieurement en conservant la rétrocompatibilité.
5. L'avantage du format du protobuf `Example` tient dans les opérations TensorFlow qui permettent de l'analyser sans que vous ayez à définir votre propre format (les fonctions `tf.io.parse*example()`). Il est suffisamment souple pour pouvoir représenter des instances dans la plupart des jeux de données. Toutefois, s'il ne répond pas à vos besoins, vous pouvez définir votre propre protobuf, le compiler avec `protoc` (en précisant les arguments `--descriptor_set_out` et `--include_imports` de façon à exporter le descripteur de protobuf) et utiliser la fonction `tf.io.decode_proto()` pour analyser les protobufs sérialisés (un exemple est donné dans la section « Custom protobuf » du notebook³¹⁵). La procédure est plus complexe et impose le déploiement du descripteur avec le modèle, mais elle est réalisable.
6. Lorsque des fichiers TFRecord sont utilisés, leur compression sera généralement activée s'ils doivent être téléchargés par le script d'entraînement. En effet, elle permet de réduire leur taille et donc le temps de téléchargement. En revanche, si les fichiers se trouvent sur la même machine que le script d'entraînement, il est préférable de la désactiver afin de ne pas gaspiller du temps processeur dans la décompression.
7. Voici les avantages et les inconvénients de chaque option de prétraitement:
 - Si les données sont prétraitées lors de la création des fichiers de données, le script d'entraînement s'exécutera plus rapidement, car il n'aura pas à effectuer ces étapes à la volée. Dans certains cas, la taille des données prétraitées sera également plus petite que celle des données d'origine. Vous économiserez alors de la place et accélérerez les téléchargements. Il peut être également utile de matérialiser les données prétraitées, par exemple pour les inspecter ou les archiver, mais cette approche présente quelques inconvénients. Tout d'abord, il peut être difficile de mener des expériences fondées sur différentes logiques de prétraitement si un jeu de données prétraitées doit être généré pour chaque

315. Voir « 13_loading_and_preprocessing_data.ipynb » sur <https://github.com/ageron/handson-ml2>.

cas. Ensuite, si vous souhaitez effectuer une augmentation des données, vous devrez matérialiser de nombreuses variantes du jeu de données, ce qui occupera un espace important sur le disque et prendra beaucoup de temps. Enfin, le modèle entraîné attendra des données prétraitées et vous devrez donc ajouter le code de prétraitement dans l'application avant qu'elle n'appelle le modèle.

- Si les données sont prétraitées dans un pipeline tf.data, il est beaucoup plus facile d'ajuster la logique de prétraitement et d'appliquer une augmentation des données. Par ailleurs, tf.data simplifie la construction de pipelines de prétraitement extrêmement efficaces (par exemple, avec le multithread et la prélecture). Toutefois, un tel prétraitement des données ralentira l'entraînement. De plus, chaque instance d'entraînement sera prétraitée une fois par époque et non pas une seule fois lorsque le prétraitement se fait au moment de la création des fichiers de données. Enfin, le modèle entraîné attendra toujours des données prétraitées.
 - Si vous ajoutez des couches de prétraitement à votre modèle, vous n'aurez à écrire le code correspondant qu'une seule fois pour l'entraînement et l'inférence. Si votre modèle doit être déployé sur de nombreuses plateformes différentes, vous n'aurez pas à écrire le code de prétraitement à plusieurs reprises. De plus, vous n'encourez pas le risque d'utiliser la mauvaise logique de prétraitement pour votre modèle, puisqu'elle fera partie du modèle. En revanche, le prétraitement des données ralentira l'entraînement et chaque instance d'entraînement sera prétraitée une fois par époque. Par ailleurs, les opérations de prétraitement du lot courant s'exécuteront par défaut sur le GPU (vous ne bénéficiez pas du prétraitement parallèle sur le CPU ni de la prélecture). Heureusement, les couches de prétraitement Keras prévues pour bientôt devraient être en mesure d'extraire les opérations de prétraitement des couches de prétraitement et de les exécuter au sein du pipeline tf.data. Vous bénéficiez alors de l'exécution multithread sur le CPU et de la prélecture.
 - En utilisant TF Transform pour le prétraitement, vous bénéficiez des avantages des options précédentes. Les données prétraitées sont matérialisées, chaque instance est prétraitée une seule fois (l'entraînement est accéléré) et les couches de prétraitement sont générées automatiquement (le code de prétraitement est écrit une seule fois). Le principal inconvénient est que vous devez apprendre à utiliser cet outil.
8. Voyons comment encoder des caractéristiques de catégorie et du texte :
- Pour encoder une caractéristique de catégorie qui présente un ordre naturel, comme les critiques cinématographiques (par exemple,

« mauvais », « moyen », « bon »), la solution la plus simple consiste à utiliser un encodage ordinal. Les catégories sont triées selon leur ordre naturel et chacune est associée à son rang (par exemple, « mauvais » correspond à 0, « moyen » à 1, et « bon » à 2). Cependant, la plupart des caractéristiques de catégorie n'ont pas un tel ordre naturel. Par exemple, il n'en existe pas pour les professions ni pour les pays. Dans ce cas, vous pouvez employer un encodage *one-hot* ou, s'il existe de nombreuses catégories, des plongements.

- Pour le texte, une solution se fonde sur la représentation par sacs de mots. Une phrase est représentée par un vecteur dénombrant la quantité de chaque mot possible. Puisque les mots communs sont généralement peu importants, vous utiliserez TF-IDF pour diminuer leur poids. Au lieu de compter des mots, il est également fréquent de compter les *n*-grammes, qui sont des suites de *n* mots consécutifs – beau et simple. Une autre solution consiste à encoder chaque mot en utilisant des plongements de mots, éventuellement préentraînés. Au lieu d'encoder des mots, il est également possible d'encoder chaque lettre ou des jetons de sous-mots (par exemple, décomposer « smartest » en « smart » et « est »). Ces deux dernières approches sont décrites dans le chapitre 8.

Pour les solutions des exercices 9 et 10, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹⁶

CHAPITRE 6: VISION PAR ORDINATEUR ET RÉSEAUX DE NEURONES CONVOLUTIFS

1. Dans le contexte de la classification des images, voici les principaux avantages d'un CNN par rapport à un RNP intégralement connecté:
 - Puisque les couches consécutives ne sont que partiellement connectées et puisqu'il réutilise massivement ses poids, un CNN possède un nombre de paramètres bien inférieur à un RNP intégralement connecté, ce qui rend son entraînement plus rapide, réduit les risques de surajustement et nécessite une quantité de données d'entraînement moindre.
 - Lorsqu'un CNN a appris un noyau capable de détecter une caractéristique particulière, il peut la détecter n'importe où dans l'image. À l'inverse, lorsqu'un RNP apprend une caractéristique en un endroit donné, il ne peut la détecter qu'en cet endroit précis. Puisque les images possèdent généralement des caractéristiques très répétitives, la généralisation des CNN est bien meilleure que celle des RNP dans les tâches de traitement d'images, comme la classification, en utilisant des exemples d'entraînement moins nombreux.

316. Voir « 13_loading_and_preprocessing_data.ipynb ».

- Enfin, un RNP n'a aucune connaissance préalable de l'organisation des pixels; il ne sait pas que les pixels voisins sont proches. L'architecture d'un CNN intègre cette connaissance. Les couches inférieures identifient généralement des caractéristiques dans de petites zones des images, tandis que les couches supérieures combinent les caractéristiques de plus bas niveau pour identifier des caractéristiques plus larges. Cela fonctionne bien avec la plupart des images naturelles, donnant aux CNN un avantage décisif par rapport au RNP.
2. Calculons le nombre de paramètres d'un CNN. Puisque sa première couche de convolution comprend 3×3 noyaux et puisque l'entrée est constituée de trois canaux (rouge, vert et bleu), chaque carte de caractéristiques possède $3 \times 3 \times 3$ poids, plus un terme constant. Cela représente 28 paramètres par carte de caractéristiques. Puisque cette première couche de convolution a 100 cartes de caractéristiques, nous arrivons à un total de 2 800 paramètres. La deuxième couche de convolution comprend 3×3 noyaux et son entrée est l'ensemble des 100 cartes de caractéristiques de la couche précédente. Chaque carte de caractéristiques a donc $3 \times 3 \times 100 = 900$ poids, plus un terme constant. Cette couche contient 200 cartes de caractéristiques, donc elle possède $900 \times 200 = 180\,200$ paramètres. Enfin, la troisième et dernière couche de convolution comprend également 3×3 noyaux et son entrée est constituée des 200 cartes de caractéristiques de la couche précédente. Par conséquent, chaque carte de caractéristiques implique $3 \times 3 \times 200 = 1800$ poids, plus un terme constant. Avec ses 400 cartes de caractéristiques, cette couche a donc $1800 \times 400 = 720\,400$ paramètres. Au total, le CNN possède donc $2\,800 + 180\,200 + 720\,400 = 903\,400$ paramètres.

Calculons à présent la quantité minimale de RAM nécessaire à ce réseau de neurones lorsqu'il effectue une prédiction pour une seule instance. Commençons par déterminer la taille d'une carte de caractéristiques pour chaque couche. Puisque nous utilisons un pas de 2 et le remplissage "same", les dimensions horizontale et verticale des cartes de caractéristiques sont divisées par deux à chaque couche (et arrondies si nécessaire). Par conséquent, puisque les canaux d'entrée font 200×300 pixels, les cartes de caractéristiques de la première couche ont une taille de 100×150 , celles de la deuxième, une taille de 50×75 , et celles de la troisième, une taille de 25×38 . Puisque 32 bits font 4 octets et que la première couche de convolution comprend 100 cartes de caractéristiques, elle a besoin de $4 \times 100 \times 150 \times 100 = 6$ millions d'octets (6 Mo). La deuxième couche occupe $4 \times 50 \times 75 \times 200 = 3$ millions d'octets (3 Mo). Enfin, la troisième couche demande $4 \times 25 \times 38 \times 400 = 1\,520\,000$ octets (environ 1,5 Mo). Cependant, lorsqu'une couche a été calculée, la mémoire occupée par la couche précédente peut

être libérée et il faudra donc seulement $6 + 3 = 9$ millions d'octets (9 Mo) de RAM (lorsque la deuxième couche vient d'être calculée, mais que la mémoire occupée par la première couche n'a pas encore été libérée). Nous devons également ajouter la mémoire occupée par les paramètres du CNN. Nous sommes arrivés à 903 400 paramètres, chacun occupant 4 octets, ce qui fait 3 613 600 octets (environ 3,6 Mo) supplémentaires. La quantité de RAM totale nécessaire est (au moins) de 12 613 600 octets (environ 12,6 Mo).

Terminons en calculant la quantité de RAM minimale nécessaire à l'entraînement du CNN sur un mini-lot de 50 images. Pendant l'entraînement, TensorFlow met en œuvre une rétropropagation qui a besoin de toutes les valeurs calculées pendant la passe en avant, jusqu'à ce que la passe en arrière débute. Nous devons donc déterminer la RAM totale requise par toutes les couches pour une seule instance et multiplier ce résultat par 50. À ce stade, nous pouvons compter en mégaoctets plutôt qu'en octets. Nous avons déterminé précédemment que les trois couches ont respectivement besoin de 6 Mo, 3 Mo et 1,5 Mo pour chaque instance. Cela représente un total de 10,5 Mo par instance. Pour 50 instances, la quantité de RAM totale est donc de 525 Mo. Ajoutons cela à la mémoire nécessaire aux images d'entrée, c'est-à-dire $50 \times 4 \times 200 \times 300 \times 3 = 36$ millions d'octets (36 Mo), plus la mémoire occupée par les paramètres du modèle, environ 3,6 Mo (calculée précédemment), plus un peu de mémoire pour les gradients (nous pouvons l'oublier car elle sera libérée au fur et à mesure de la progression de la rétropropagation vers les couches inférieures). Nous arrivons à un total approximatif de $525 + 36 + 3,6 = 564,6$ Mo. Et ce n'est réellement qu'un strict minimum optimiste.

3. Si votre GPU vient à manquer de mémoire pendant l'entraînement d'un CNN, voici cinq actions que vous pouvez effectuer pour tenter de résoudre le problème (autres qu'acheter une carte graphique avec une plus grande quantité de RAM):
 - Réduire la taille du mini-lot.
 - Diminuer la dimensionnalité en utilisant un pas plus grand dans une ou plusieurs couches.
 - Supprimer une ou plusieurs couches.
 - Utiliser des nombres à virgule flottante sur 16 bits à la place de 32 bits.
 - Distribuer le CNN sur plusieurs processeurs.
4. Une couche de pooling maximum ne possède aucun paramètre, tandis qu'une couche de convolution en utilise beaucoup (voir les questions précédentes).
5. Une couche de normalisation de réponse locale permet aux neurones les plus actifs d'inhiber ceux situés aux mêmes emplacements mais

dans des cartes de caractéristiques voisines. De cette manière, les cartes de caractéristiques différentes ont tendance à se spécialiser et à se distinguer, et se voient obligées d'explorer une plage de caractéristiques plus importante. Cette normalisation se rencontre habituellement dans les couches inférieures de façon à obtenir des caractéristiques de bas niveau plus nombreuses, que les couches supérieures pourront exploiter.

6. Les principales innovations de l'architecture AlexNet par rapport à LeNet-5 sont une largeur et une profondeur plus importantes, et un empilement des couches de convolution directement l'une au-dessus de l'autre à la place d'un empilement d'une couche de pooling au-dessus d'une couche de convolution. La principale innovation de GoogLeNet vient des *modules Inception*, grâce auxquels le réseau peut être beaucoup plus profond qu'avec les architectures de CNN précédentes, pour un nombre de paramètres moindre. La principale innovation de ResNet réside dans l'introduction des connexions de saut, qui permettent d'aller bien au-delà des 100 couches. Sa simplicité et son uniformité sont également des éléments innovants. La principale innovation de SENet vient de l'utilisation d'un bloc SE (un réseau dense de deux couches) après chaque module Inception dans un réseau Inception ou après chaque unité résiduelle dans un ResNet, de façon à recalibrer l'importance relative des cartes de caractéristiques. Enfin, la principale innovation de Xception réside dans l'utilisation de couches de convolution séparables en profondeur, qui examinent séparément les motifs spatiaux et les motifs en profondeur.
7. Les réseaux entièrement convolutifs sont des réseaux constitués exclusivement de couches de convolution et de couches de pooling. Les FCN sont en mesure de traiter efficacement des images quelles que soient leur hauteur et leur largeur (tout au moins au-dessus de la taille minimale). Ils sont les plus utiles pour la détection d'objets et la segmentation sémantique, car ils n'ont besoin d'examiner l'image qu'une seule fois (au lieu d'exécuter à plusieurs reprises un CNN sur différentes parties de l'image). Si vous disposez d'un CNN ayant des couches denses au sommet, vous pouvez convertir celles-ci en couches de convolution afin d'obtenir un FCN. Il suffit de remplacer la couche dense la plus basse par une couche de convolution dont la taille du noyau est égale à la taille de la sortie de la couche, avec un filtre par neurone dans la couche dense, et avec le remplissage "valid". En général, le pas doit être égal à 1, mais, si vous le souhaitez, vous pouvez lui donner une valeur plus élevée. La fonction d'activation doit être identique à celle de la couche dense. Les autres couches denses peuvent être converties de la même manière, mais en utilisant des filtres 1×1 . En réalité, il est possible de convertir un

CNN entraîné de cette manière, en modifiant de façon appropriée la forme des matrices de poids des couches denses.

8. La principale difficulté technique de la segmentation sémantique vient du fait qu'un grand nombre d'informations spatiales sont perdues dans un CNN alors que le signal traverse chaque couche, en particulier dans les couches de pooling et celles dont le pas est supérieur à 1. Ces informations spatiales doivent être retrouvées d'une manière ou d'une autre pour obtenir une bonne précision de prédiction de la classe de chaque pixel.

Pour les solutions des exercices 9 à 11, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹⁷

CHAPITRE 7 : TRAITEMENT DES SÉRIES AVEC DES RNR ET DES CNN

1. Voici quelques applications des RNR :
 - Pour un RNR série-vers-série : prédire la météo (ou n'importe quelle autre série chronologique), traduction automatique (en utilisant une architecture encodeur-décodeur), sous-titrage vidéo, reconnaissance vocale, génération musicale (ou toute autre génération de séries), identification des accords dans une chanson).
 - Pour un RNR série-vers-vecteur : classification d'échantillons musicaux par genre musical, analyse de sentiment pour une critique littéraire, prédire le mot auquel un patient aphasiqe pense en fonction des lectures effectuées grâce à des implants cérébraux, prédire la probabilité qu'une personne souhaite regarder un film en fonction de son historique de visionnage (l'une des nombreuses implémentations possibles du *filtrage collaboratif* pour un système de recommandation).
 - Pour un RNR vecteur-vers-série : génération de légendes pour des images, création d'une liste de lecture musicale en fonction d'un plongement de l'artiste en cours d'écoute, génération d'une mélodie fondée sur un ensemble de paramètres, localisation des piétons dans une image (par exemple, une vidéo provenant de la caméra d'une voiture autonome).
2. Une couche de RNR doit avoir des entrées à trois dimensions : la première dimension correspond à la dimension de lot (sa taille est celle du lot), la deuxième représente le temps (sa taille est le nombre d'étapes temporelles), et la troisième contient les entrées de chaque étape temporelle (sa taille est le nombre de caractéristiques d'entrées par étape temporelle). Par exemple, si vous souhaitez traiter un lot contenant 5 séries chronologiques de 10 étapes temporelles chacune,

317. Voir « 14_deep_computer_vision_with_cnns.ipynb ».

avec 2 valeurs par étape temporelle (par exemple, la température et la vitesse du vent), la forme sera [5, 10, 2]. Les sorties ont également trois dimensions, les deux premières étant identiques à celles des entrées, la dernière étant le nombre de neurones. Par exemple, si une couche de RNR comprenant 32 neurones traite le lot que nous venons de mentionner, la sortie aura la forme [5, 10, 32].

3. Pour construire un RNR série-vers-série profond avec Keras, vous devez préciser `return_sequences=True` pour toutes les couches de RNR. Pour construire un RNR série-vers-vecteur, vous devez indiquer `return_sequences=True` pour toutes les couches de RNR, à l'exception de la couche supérieure, pour laquelle `return_sequences` doit avoir la valeur `False` (vous pouvez ne pas fixer cet argument, car sa valeur par défaut est `False`).
4. Si vous disposez de séries chronologiques univariées quotidiennes et si vous souhaitez prévoir les sept jours suivants, l'architecture de RNR la plus simple est une pile de couches de RNR (toutes avec `return_sequences` égal à `True`, à l'exception de la couche de RNR supérieure), en utilisant sept neurones dans la couche de sortie. Vous pouvez ensuite entraîner ce modèle en utilisant des fenêtres aléatoires sur les séries chronologiques (par exemple, des séquences de trente jours consécutifs en entrée, et un vecteur contenant les valeurs des sept jours suivants comme cible). Il s'agit d'un RNR série-vers-vecteur. En indiquant `return_sequences=True` pour toutes les couches de RNR, vous créez un RNR série-vers-série, que vous pouvez entraîner en utilisant des fenêtres aléatoires sur les séries chronologiques, avec des séquences de la même longueur que les entrées et les cibles. Chaque série cible doit avoir sept valeurs par étape temporelle (par exemple, pour l'étape temporelle t , la cible doit être un vecteur contenant les valeurs aux étapes temporelles $t + 1$ à $t + 7$).
5. Les deux principales difficultés de l'entraînement des RNR sont l'instabilité des gradients (explosion ou disparition) et une mémoire à court terme très limitée. Ces deux problèmes empêtent lorsque les séries sont longues. Pour alléger le problème d'instabilité des gradients, vous pouvez utiliser un taux d'apprentissage plus faible, une fonction d'activation saturante, comme la tangente hyperbolique (le choix par défaut), et, éventuellement, l'écrêtage de gradients, la normalisation par couche ou un dropout à chaque étape temporelle. Pour traiter le problème de mémoire à court terme limitée, vous pouvez employer des couches LSTM ou GRU (cela aide aussi à réduire le problème d'instabilité des gradients).
6. L'architecture d'une cellule LSTM semble complexe, mais elle n'est pas si compliquée lorsque l'on comprend la logique sous-jacente. La cellule possède un vecteur d'état à court terme et un vecteur d'état à long terme. À chaque étape temporelle, les entrées et l'état à court

terme précédent sont passés à une cellule de RNR et trois portes : la porte d'oubli choisit ce qui doit être retiré de l'état à long terme, la porte d'entrée choisit quelle partie de la sortie de la cellule de RNR simple doit être ajoutée à l'état à long terme, et la porte de sortie décide de la partie de l'état à long terme qui doit être sortie lors de cette étape temporelle (après être passée par la fonction d'activation tanh). Le nouvel état à court terme est égal à la sortie de la cellule. Voir la figure 7.9.

7. Une couche de RNR est fondamentalement séquentielle. Pour calculer des sorties à l'étape temporelle t , elle doit commencer par calculer les sorties de toutes les étapes temporelles précédentes. La parallélisation est donc impossible. D'un autre côté, une couche de convolution à une dimension est bien adaptée à la parallélisation, car elle ne maintient pas d'état entre les étapes temporelles. Autrement dit, elle n'a aucune mémoire : la sortie de n'importe quelle étape temporelle peut être calculée à partir d'une petite fenêtre de valeurs sur les entrées sans avoir à connaître toutes les valeurs passées. De plus, puisqu'une telle couche de convolution n'est pas récurrente, elle souffre moins de l'instabilité des gradients. Dans un RNR, une ou plusieurs couches de convolution à une dimension pourront permettre un prétraitement efficace des entrées, par exemple pour réduire leur résolution temporelle (sous-échantillonnage) et ainsi aider les couches de RNR à détecter des motifs à long terme. En réalité, il est possible de n'utiliser que des couches de convolution, par exemple en construisant une architecture WaveNet.
8. Pour classer des vidéos en fonction du contenu visuel, une architecture possible consisterait à prendre, par exemple, une trame par seconde, à passer chaque trame à un même réseau de neurones convolutif (par exemple, un modèle Xception préentraîné, éventuellement figé si le jeu de données n'est pas volumineux), à transmettre la sortie de ce CNN à un RNR série-vers-vecteur, et, enfin, à passer sa sortie au travers d'une couche softmax qui donnerait toutes les probabilités de classes. Pour l'entraînement, l'entropie croisée peut être choisie comme fonction de coût. Si vous souhaitez utiliser également le son pour la classification, vous pouvez utiliser une pile de couches de convolution à pas à une dimension de manière à réduire la résolution temporelle, en passant de milliers de trames audio par seconde à juste une trame par seconde (pour correspondre au nombre d'images par seconde), et concaténer la série de sortie aux entrées du RNR série-vers-vecteur (le long de la dernière dimension).

Pour les solutions des exercices 9 et 10, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹⁸

318. Voir « 15_processing_sequences_using_rnns_and_cnns.ipynb ».

CHAPITRE 8 : TRAITEMENT AUTOMATIQUE DU LANGAGE NATUREL AVEC LES RNR ET LES ATTENTIONS

1. Les RNR sans état ne peuvent capturer que des motifs dont la longueur est inférieure ou égale à la taille des fenêtres sur lesquelles le RNR est entraîné. À l'inverse, les RNR avec état peuvent capturer des motifs à plus long terme. Toutefois, l'implémentation d'un RNR avec état est beaucoup plus complexe, notamment pour la préparation correcte du jeu de données. De plus, ces RNR n'affichent pas toujours de meilleures performances, en partie parce que les lots consécutifs ne sont pas indépendants et distribués de façon identique. La descente de gradients apprécie peu les jeux de données de ce genre.
2. En général, la traduction d'une phrase mot à mot donne un très mauvais résultat. Par exemple, la phrase anglaise « It is up to you » signifie « C'est à toi de décider », mais sa traduction un mot à la fois pourrait donner « Il est haut à toi ». Pardon ? Il est nettement préférable de lire tout d'abord la phrase dans son intégralité, puis de la traduire. Un RNR série-vers-série de base commencerait à traduire une phrase immédiatement après la lecture du premier mot, tandis qu'un RNR encodeur-décodeur lira l'intégralité de la phrase, puis la traduira. Cela dit, nous pouvons imaginer un RNR série-vers-série de base qui produirait un silence dès qu'il ne sait pas ce qu'il doit dire ensuite (à l'instar des interprètes lors des traductions en direct).
3. Pour prendre en charge les séries d'entrée de longueur variable, il est possible de remplir les séries les plus courtes de sorte que toutes les séquences d'un lot aient la même longueur et d'utiliser les masques pour s'assurer que le RNR ignore le jeton de remplissage. Pour de meilleures performances, vous pouvez également créer des lots contenant des séries de taille similaire. Les tenseurs irréguliers peuvent contenir des séries de longueur variable et tf.keras finira par les prendre en charge, ce qui simplifiera énormément le traitement des séries d'entrée de longueur variable (ce n'est pas le cas au moment de l'écriture de ces lignes). En ce qui concerne les séries de sortie de longueur variable, si leur taille est connue à l'avance (par exemple, si vous savez qu'elle est identique à celle de la série d'entrée), vous devez simplement configurer la fonction de perte pour qu'elle ignore les jetons venant après la fin de la série. De façon comparable, le code qui utilisera le modèle doit ignorer les jetons qui se trouvent après la fin de la série. Mais, en général, la longueur de la série de sortie n'est pas connue à l'avance et la solution consiste à entraîner le modèle afin qu'il termine chaque série par un jeton de fin.
4. La recherche en faisceaux est une solution d'amélioration des performances d'un modèle encodeur-décodeur entraîné, par exemple dans un système de traduction automatique neuronale. L'algorithme conserve une liste restreinte des k séries de sortie les plus

prometteuses (par exemple, les trois premières) et, à chaque étape, tente de leur ajouter un mot ; il garde ensuite uniquement les k séries les plus probables. Le paramètre k est la *largeur du faisceau*. Plus le faisceau est large, plus la puissance de calcul et la quantité mémoire requises sont importantes, mais plus le système est précis. Au lieu de choisir immédiatement à chaque étape le prochain mot le plus probable de façon à étendre une seule série, cette technique permet au système d'explorer plusieurs phrases prometteuses simultanément. C'est une technique bien adaptée à la parallélisation. Vous pouvez implémenter la recherche en faisceaux relativement facilement grâce à TensorFlow Addons.

5. Le mécanisme d'attention est une technique employée initialement dans les modèles encodeur-décodeur pour donner au décodeur un accès direct à la série d'entrée, lui permettant de prendre charge des séries d'entrée plus longues. À chaque étape temporelle du décodeur, l'état courant du décodeur et sa sortie complète sont traités par un modèle d'alignement qui produit un score d'alignement pour chaque étape temporelle d'entrée. Ce score indique la partie de l'entrée la plus pertinente pour l'étape temporelle courante du décodeur. La somme pondérée de la sortie de l'encodeur (pondérée par le score d'alignement) est transmise au décodeur, qui produit l'état suivant du décodeur et la sortie pour cette étape temporelle. Le mécanisme d'attention a pour principal avantage de permettre au modèle encodeur-décodeur de traiter des séries d'entrée plus longues. Par ailleurs, grâce au score d'alignement, le modèle est plus facile à déboguer et à interpréter. Par exemple, s'il fait une erreur, vous pouvez examiner la partie de l'entrée à laquelle il prêtait attention à ce moment-là, ce qui pourra vous aider à diagnostiquer le problème. Le mécanisme d'attention est également au cœur de l'architecture Transformer, dans les couches Multi-Head Attention. Voir la prochaine réponse.
6. Dans l'architecture Transformer, la couche Multi-Head Attention est la plus importante (l'architecture Transformer d'origine en contient 18, y compris 6 couches Masked Multi-Head Attention). Elle est au centre de modèles de langage, comme BERT et GPT-2. Son rôle est de permettre au modèle d'identifier les mots présentant le meilleur alignement avec chaque autre mot, et d'exploiter ensuite ces indices contextuels pour améliorer la représentation de chaque mot.
7. La technique softmax échantillonnée est utilisée pour l'entraînement d'un modèle de classification lorsque les classes sont nombreuses (des milliers). Elle calcule une approximation de la perte d'entropie croisée à partir du logit prédit par le modèle pour la classe appropriée et les logits prédits pour un échantillon de mots incorrects. L'entraînement s'en trouve considérablement amélioré par rapport à un calcul softmax sur tous les logits suivi d'une estimation de la

perte d'entropie croisée. Après l'entraînement, le modèle peut être employé de façon habituelle, en utilisant la fonction softmax normale pour calculer toutes les probabilités de classes en fonction de tous les logits.

Pour les solutions des exercices 8 à 11, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³¹⁹

CHAPITRE 9 : APPRENTISSAGE DE PRÉSENTATIONS ET APPRENTISSAGE GÉNÉRATIF AVEC DES AUTOENCODEURS ET DES GAN

1. Voici quelques-uns des principaux usages des autoencodeurs :
 - extraction de caractéristiques ;
 - préentraînement non supervisé ;
 - réduction de la dimensionnalité ;
 - modèles génératifs ;
 - détection d'anomalies (un autoencodeur a généralement du mal à reconstruire des aberrations).
2. Si vous voulez entraîner un classificateur en ayant une grande quantité de données d'entraînement non étiquetées, mais seulement quelques milliers d'instances étiquetées, vous pouvez commencer par entraîner un autoencodeur profond sur le jeu de données complet (étiqueté + non étiqueté), puis réutiliser sa moitié inférieure pour le classificateur (c'est-à-dire réutiliser les couches allant jusqu'à la couche de codage incluse), en entraînant celui-ci avec les données étiquetées. Si vous avez peu de données étiquetées, vous voudrez probablement figer les couches réutilisées pendant l'entraînement du classificateur.
3. Le fait qu'un autoencodeur reconstruise parfaitement ses entrées ne signifie pas nécessairement qu'il est un bon autoencodeur ; peut-être s'agit-il simplement d'un autoencodeur sur-complet qui a appris à copier ses entrées vers la couche de codage, puis vers les sorties. En réalité, même si la couche de codage ne contient qu'un seul neurone, un autoencodeur très profond a la possibilité d'apprendre à faire correspondre chaque instance d'entraînement à un codage différent (par exemple, la première instance peut être associée à 0,001, la deuxième à 0,002, la troisième à 0,003, etc.) et il peut apprendre « par cœur » à reconstruire la bonne instance d'entraînement pour chaque codage. Il reconstruirait parfaitement ses entrées sans réellement apprendre de motifs utiles dans les données. Dans la pratique, il est peu probable qu'une telle correspondance se produise,

³¹⁹ Voir « 16_nlp_with_rnns_and_attention.ipynb ».

mais elle illustre le fait qu'une reconstruction parfaite ne garantit pas que l'autoencodeur a appris quelque chose d'intéressant. En revanche, s'il produit de très mauvaises reconstructions, il s'agit alors presque à coup sûr d'un mauvais autoencodeur. Pour évaluer la performance d'un autoencodeur, il est possible de mesurer la perte de reconstruction (par exemple, calculer la MSE, la moyenne quadratique des sorties moins les entrées). De nouveau, une perte de reconstruction élevée est un bon signe que l'autoencodeur est mauvais, mais une perte de reconstruction faible ne garantit pas qu'il est bon. Vous devez également évaluer l'autoencodeur en fonction de son utilisation. Par exemple, s'il sert au préentraînement non supervisé d'un classificateur, vous devez aussi évaluer les performances de celui-ci.

4. Un autoencodeur sous-complet possède une couche de codage plus petite que les couches d'entrée et de sortie. Si elle est plus grande, alors il s'agit d'un autoencodeur sur-complet. Un autoencodeur fortement sous-complet court le risque d'échouer dans la reconstruction des entrées. Un autoencodeur sur-complet risque de se contenter de recopier les entrées sur les sorties, sans rien apprendre d'intéressant.
5. Pour lier les poids d'une couche d'encodage à la couche de décodage correspondante, vous pouvez simplement rendre les poids du décodeur égaux à la transposée des poids du décodeur. Le nombre de paramètres du modèle est ainsi divisé par deux, avec pour conséquence un entraînement qui converge souvent plus rapidement pour une quantité de données d'entraînement moindre, et une diminution du risque de surajustement du jeu d'entraînement.
6. Un modèle génératif est un modèle capable de produire aléatoirement des sorties qui ressemblent aux instances d'entraînement. Par exemple, après un entraînement réussi sur le jeu de données MNIST, un modèle génératif peut servir à générer de façon aléatoire des images de chiffres réalistes. La distribution de la sortie est en général comparable à celle des données d'entraînement. Par exemple, puisque MNIST comprend de nombreuses images de chaque chiffre, le modèle génératif va sortir approximativement le même nombre d'images de chacun d'eux. Certains modèles génératifs possèdent des paramètres, par exemple pour générer uniquement certains types de sorties. L'autoencodeur variationnel est un exemple d'autencodeur génératif.
7. Un réseau génératif antagoniste est une architecture de réseau de neurones constituée de deux parties, le générateur et le discriminateur, dont les objectifs sont opposés. Le but du générateur est de produire des instances semblables à celles présentes dans le jeu d'entraînement afin de tromper le discriminateur. Celui-ci doit différencier les instances réelles et les instances générées. À chaque itération d'entraînement, le discriminateur est entraîné comme un

classificateur binaire normal, puis le générateur est entraîné de façon à maximiser l'erreur du discriminateur. Des GAN sont employés dans les traitements d'images complexes, comme la super-résolution, la colorisation, l'édition élaborée d'images (remplacer des objets par des arrière-plans réalistes), convertir une simple esquisse en une image photoréaliste ou prédire les trames suivantes dans une vidéo. Ils servent également à augmenter un jeu de données (pour entraîner d'autres modèles), à générer d'autres types de données (par exemple du texte, de l'audio ou des séries chronologiques) et à identifier les faiblesses dans d'autres modèles et les corriger.

8. L'entraînement des GAN est réputé difficile, en raison de la dynamique complexe entre le générateur et le discriminateur. La plus grande difficulté se nomme mode collapse. Elle correspond au moment où le générateur produit des sorties présentant très peu de diversité. Par ailleurs, l'entraînement peut être terriblement instable. Il peut commencer parfaitement, pour soudainement se mettre à osciller ou à diverger, sans raison apparente. Les GAN sont également très sensibles au choix des hyperparamètres.

Pour les solutions des exercices 9, 10 et 11, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³²⁰

CHAPITRE 10 : APPRENTISSAGE PAR RENFORCEMENT

1. L'apprentissage par renforcement est un domaine de l'apprentissage automatique dont l'objectif est de créer des agents capables d'effectuer des actions dans un environnement en faisant en sorte qu'elles maximisent les récompenses au fil du temps. Il existe de nombreuses différences entre l'apprentissage par renforcement et l'apprentissage automatique « normal », supervisé ou non. En voici quelques-unes :
 - Dans l'apprentissage supervisé ou non supervisé, l'objectif est généralement de découvrir des motifs dans les données afin de pouvoir faire des prédictions. Dans l'apprentissage par renforcement, l'objectif est de trouver une bonne politique.
 - Contrairement à l'apprentissage supervisé, la « bonne » réponse n'est pas explicitement donnée à l'agent. Il doit apprendre par tâtonnement.
 - Contrairement à l'apprentissage non supervisé, il existe une forme de supervision, au travers des récompenses. On ne précise pas à l'agent comment effectuer une tâche, mais on lui indique s'il fait des progrès ou s'il échoue.
 - Dans l'apprentissage par renforcement, l'agent doit trouver le bon équilibre entre explorer l'environnement, rechercher de

³²⁰. Voir « 17_autoencoders_and_gans.ipynb ».

nouvelles manières de recevoir des récompenses et exploiter les sources de récompenses qu'il connaît déjà. À l'inverse, les systèmes d'apprentissage supervisé ou non supervisé n'ont pas à s'occuper de l'exploration et exploitent simplement les données d'entraînement qui leur sont fournies.

- Dans l'apprentissage supervisé ou non supervisé, les instances d'entraînement sont généralement indépendantes (en fait, elles sont souvent mélangées). Dans l'apprentissage par renforcement, des observations consécutives ne sont généralement *pas* indépendantes. Puisqu'un agent peut rester un certain temps dans la même région de l'environnement avant de se déplacer, des observations consécutives seront fortement corrélées. Dans certains cas, une mémoire (tampon) de rejet est mise en place afin que l'algorithme d'entraînement obtienne des observations assez indépendantes.
2. Voici quelques applications possibles de l'apprentissage par renforcement, autres que celles mentionnées au chapitre 10 :

Personnalisation musicale

L'environnement correspond à une radio web personnalisée pour l'utilisateur. L'agent est le logiciel qui décide de la chanson que l'utilisateur va écouter ensuite. Les actions possibles sont soit de jouer n'importe quelle chanson du catalogue (l'agent doit essayer de choisir une chanson que l'utilisateur appréciera), soit de passer une publicité (il doit essayer de choisir une publicité qui intéressera l'utilisateur). L'agent obtient une petite récompense chaque fois que l'utilisateur écoute une chanson, une récompense plus importante chaque fois qu'il écoute une publicité, une récompense négative lorsqu'il saute une chanson ou une publicité, et une récompense très négative lorsqu'il quitte l'application.

Marketing

L'environnement correspond au service marketing de votre entreprise. L'agent est le logiciel qui détermine les utilisateurs ciblés par une campagne de mailing, à partir de leur profil et de leur historique d'achat (pour chaque client, il existe deux actions possibles: envoyer et ne pas envoyer). Il reçoit une récompense négative en fonction du coût de la campagne de mailing et une récompense positive en fonction des revenus estimés générés par cette campagne.

Livraison de produits

L'agent contrôle une flotte de camions de livraison, en décidant des produits qu'ils doivent prendre dans les dépôts, où ils doivent se rendre, ce qu'ils doivent livrer, etc. Il reçoit des récompenses positives pour les produits livrés à temps et des récompenses négatives pour les livraisons retardées.

3. Lors de l'estimation de la valeur d'une action, les algorithmes d'apprentissage par renforcement additionnent généralement toutes les récompenses auxquelles cette action conduit, en donnant plus de poids aux récompenses immédiates et moins de poids aux récompenses ultérieures (en considérant qu'une action a plus d'influence sur le futur proche que sur le futur éloigné). Pour modéliser ce fonctionnement, un taux de rabais est généralement appliqué à chaque étape temporelle. Par exemple, avec un taux de rabais égal à 0,9, une récompense de 100 qui sera reçue dans deux étapes temporelles futures compte uniquement pour $0,9^2 \times 100 = 81$ lors de l'estimation de la valeur de l'action. Le taux de rabais peut être vu comme une mesure de la valeur du futur relativement au présent. S'il est très proche de 1, alors le futur compte presque autant que le présent. S'il est très proche de 0, alors seules les récompenses immédiates importent. Bien entendu, cela impacte énormément la politique optimale. Si le futur est valorisé, vous êtes prêt à souffrir immédiatement dans l'objectif d'éventuelles récompenses ultérieures. En revanche, si le futur n'est pas valorisé, vous prenez simplement toute récompense immédiate trouvée, sans miser sur l'avenir.
4. Pour mesurer les performances d'un agent d'apprentissage par renforcement, vous pouvez simplement additionner les récompenses obtenues. Dans un environnement simulé, vous pouvez exécuter de nombreux épisodes et examiner le total des récompenses qu'il reçoit en moyenne (et éventuellement tenir compte du minimum, du maximum, de l'écart-type, etc.).
5. Le problème d'affectation du crédit est lié au fait qu'un agent d'apprentissage par renforcement qui reçoit une récompense ne peut pas connaître directement celles de ses actions passées qui ont contribué à cette récompense. Ce problème survient souvent lorsque le délai entre une action et les récompenses résultantes est important (par exemple, dans le jeu Pong d'Atari, il peut y avoir plusieurs dizaines d'étapes temporelles entre le moment où l'agent frappe la balle et le moment où il gagne le point). Pour le minimiser, une solution consiste à donner à l'agent des récompenses à plus court terme, lorsque c'est possible. En général, cela nécessite des connaissances préalables sur la tâche. Par exemple, dans le cas d'un agent qui apprend à jouer aux échecs, on peut le récompenser non pas uniquement lorsqu'il gagne la partie, mais chaque fois qu'il capture une pièce de l'adversaire.
6. Il est fréquent qu'un agent reste un certain temps dans la même région de son environnement. Pendant cette période, toutes ses expériences seront très similaires et cela peut introduire une forme de biais dans l'algorithme d'apprentissage. L'agent peut peaufiner sa politique pour cette région de l'environnement, mais ses performances ne seront pas aussi bonnes lorsqu'il en sortira. Pour résoudre ce problème, on peut

employer une mémoire de rejet. Au lieu de fonder son apprentissage uniquement sur les expériences immédiates, l'agent apprendra à partir d'une mémoire de ses expériences passées, récentes et moins récentes (voilà peut-être pourquoi nous rêvons la nuit : pour rejouer nos expériences du jour et mieux apprendre d'elles).

7. Un algorithme RL de politique hors ligne (*off-policy*) apprend la valeur de la politique optimale (c'est-à-dire la somme des récompenses avec rabais qu'il peut attendre pour chaque état si l'agent agit de façon optimale), alors que l'agent suit une politique différente. L'apprentissage Q est un bon exemple de ce type d'algorithme. À l'inverse, un algorithme de politique en ligne (*on-policy*) apprend la valeur de la politique que l'agent exécute réellement (ce qui inclut à la fois l'exploration et l'exploitation).

Pour les solutions des exercices 8, 9 et 10, consultez le notebook Jupyter disponible à l'adresse <https://github.com/ageron/handson-ml2>.³²¹

CHAPITRE 11 : ENTRAÎNEMENT ET DÉPLOIEMENT À GRANDE ÉCHELLE DE MODÈLES TENSORFLOW

1. Un SavedModel contient un modèle TensorFlow, y compris son architecture (un graphe de calcul) et ses poids. Il est stocké sous forme d'un répertoire contenant un fichier `saved_model.pb`, qui définit le graphe de calcul (représenté sous forme de protobuf sérialisé), et d'un sous-répertoire `variables` contenant des valeurs de variables. Pour les modèles qui comprennent un grand nombre de poids, ces valeurs de variables peuvent être réparties dans plusieurs fichiers. Un SavedModel inclut également un sous-répertoire `assets` qui peut contenir des données supplémentaires, comme des fichiers de vocabulaire, des noms de classes ou des instances d'exemples pour le modèle. De façon plus précise, un SavedModel peut contenir un ou plusieurs métagraphes. Un métagraph est un graphe de calcul accompagné de définitions de signatures de fonctions (y compris les noms de leurs entrées et sorties, les types et les formes). Chaque métagraph est identifié par un ensemble d'étiquettes. Pour inspecter un SavedModel, vous pouvez utiliser l'outil en ligne de commande `saved_model_cli` ou l'examiner dans Python après l'avoir chargé avec `tf.saved_model.load()`.
2. TF Serving permet de déployer de multiples modèles TensorFlow (ou plusieurs versions du même modèle) et de les rendre facilement accessibles à toutes vos applications au travers d'une API REST ou gRPC. En utilisant les modèles directement dans vos applications, le déploiement d'une nouvelle version d'un modèle sur toutes les

321. Voir « `18_reinforcement_learning.ipynb` ».

applications est plus complexe. Le développement d'un microservice qui enveloppe un modèle TF demande du travail supplémentaire et il sera difficile d'obtenir les nombreuses fonctionnalités de TF Serving. TF Serving peut surveiller un répertoire et déployer automatiquement les modèles qui y sont placés, sans avoir à modifier ni même redémarrer les applications pour qu'elles bénéficient de nouvelles versions des modèles. Il est rapide, testé sous toutes les coutures et s'adapte parfaitement aux besoins. Il prend en charge les tests A/B de modèles expérimentaux et le déploiement de nouvelles versions d'un modèle à un groupe d'utilisateurs (dans ce cas, le modèle est appelé un *canari*). Il est également capable de regrouper des requêtes individuelles en lots pour les exécuter conjointement sur le GPU. Pour déployer TF Serving, vous pouvez partir des fichiers sources, mais il est plus simple d'employer une image Docker. Pour déployer un groupe d'images Docker de TF Serving, vous pouvez utiliser un outil d'orchestration, comme Kubernetes, ou vous tourner vers une solution hébergée, comme Google Cloud AI Platform.

3. Pour déployer un modèle sur plusieurs instances de TF Serving, vous devez simplement configurer ces instances pour qu'elles surveillent le même répertoire de modèles, puis exporter votre nouveau modèle au format SavedModel dans un sous-répertoire.
4. L'API gRPC est plus efficace que l'API REST. Toutefois, ses bibliothèques clientes sont moins répandues et, en activant la compression dans l'API REST, vous pouvez arriver à des performances comparables. Par conséquent, l'API gRPC est la plus utile lorsque vous avez besoin des plus hautes performances possible et lorsque les clients ne sont pas limités à l'API REST.
5. Pour réduire la taille d'un modèle de sorte qu'il puisse s'exécuter sur un périphérique mobile ou embarqué, TFLite met en œuvre plusieurs techniques :
 - Son convertisseur est capable d'optimiser un SavedModel : il rétrécit le modèle et réduit sa latence. Pour cela, il élague toutes les opérations non indispensables aux prédictions (comme les opérations d'entraînement) et optimise et fusionne des opérations lorsque c'est possible.
 - Le convertisseur peut également effectuer une quantification post-entraînement. Cette technique réduit énormément la taille du modèle, qui devient plus rapide à télécharger et à stocker.
 - Le modèle optimisé est enregistré au format FlatBuffer, qui peut être chargé directement en RAM sans analyse préalable. Le temps de chargement et l'empreinte mémoire s'en trouvent réduits.
6. Dans un entraînement conscient de la quantification, on ajoute au modèle des opérations de quantification factices pendant l'entraînement. Cela permet au modèle d'apprendre à ignorer le

bruit lié à la quantification ; les poids finaux seront moins sensibles à la quantification.

7. Dans le parallélisme du modèle, le modèle est découpé en plusieurs parties, qui sont exécutées en parallèle sur plusieurs processeurs, dans le but d'accélérer l'entraînement ou l'inférence. Dans le parallélisme des données, plusieurs répliques identiques du modèle sont créées et déployés sur plusieurs processeurs. À chaque itération d'entraînement, chaque réplica reçoit un lot de données différent et calcule les gradients de la perte conformément aux paramètres du modèle. Dans le parallélisme des données en mode synchrone, les gradients de tous les répliques sont ensuite agrégés et l'optimiseur effectue une étape de descente de gradient. Les paramètres peuvent être centralisés (par exemple, sur des serveurs de paramètres) ou copiés sur tous les répliques avec une synchronisation assurée par un algorithme AllReduce. Dans le parallélisme des données en mode asynchrone, les paramètres sont centralisés et les répliques s'exécutent indépendamment les uns des autres, chacun mettant directement à jour les paramètres centraux à la fin de chaque itération d'entraînement, sans attendre les autres répliques. De façon générale, le parallélisme des données permet une plus grande accélération de l'entraînement que le parallélisme du modèle. Cela vient essentiellement du fait qu'il demande moins de communications entre les processeurs. Par ailleurs, son implémentation est plus facile et il travaille de la même manière quel que soit le modèle. En revanche, le parallélisme du modèle impose une analyse du modèle afin de déterminer la meilleure façon de le découper en morceaux.
8. Lors de l'entraînement d'un modèle sur plusieurs serveurs, vous pouvez utiliser les stratégies de distribution suivantes:
 - `MultiWorkerMirroredStrategy` effectue un parallélisme des données avec mise en miroir. Le modèle est répliqué sur tous les serveurs et processeurs disponibles, et chaque réplica reçoit un lot de données différent à chaque itération d'entraînement et calcule ses propres gradients. La moyenne des gradients est calculée et transmise à tous les répliques à l'aide d'une implémentation distribuée d'AllReduce (par défaut NCCL), et tous les répliques effectuent la même étape de descente de gradient. Cette stratégie est la plus simple car tous les serveurs et processeurs sont traités de la même manière. De plus, ses performances sont plutôt bonnes. De façon générale, vous devriez employer cette stratégie. Sa principale limite est que le modèle doit tenir dans la mémoire de chaque réplica.
 - `ParameterServerStrategy` réalise un parallélisme des données en mode asynchrone. Le modèle est répliqué sur tous les processeurs sur tous les ouvriers, et les paramètres sont éclatés sur tous les serveurs de paramètres. Chaque ouvrier dispose de sa

propre boucle d’entraînement, qui s’exécute de façon asynchrone par rapport aux autres ouvriers. Lors de chaque itération d’entraînement, un ouvrier reçoit son propre lot de données et récupère la dernière version des paramètres du modèle à partir des serveurs de paramètres. Il calcule ensuite les gradients de la perte en fonction de ces paramètres et les envoie aux serveurs de paramètres. Ceux-ci réalisent une étape de descente de gradient à partir des gradients reçus. Cette stratégie est généralement plus lente que la précédente et un peu plus difficile à déployer, car elle exige une gestion des serveurs de paramètres. Toutefois, elle se révélera utile pour l’entraînement de modèles extrêmement volumineux qui ne tiennent pas dans la mémoire du GPU.

Pour les solutions des exercices 9, 10 et 11, consultez le notebook Jupyter disponible à l’adresse <https://github.com/ageron/handson-ml2>.³²²

^{322.} Voir « 19_training_and_deploying_at_scale.ipynb ».

Annexe B

Différentiation automatique

Cette annexe explique comment fonctionne la différentiation automatique dans TensorFlow et la compare à d'autres solutions.

Supposons que nous définissons la fonction $f(x, y) = x^2y + y + 2$ et que nous ayons besoin de ses dérivées partielles $\partial f / \partial x$ et $\partial f / \partial y$, par exemple pour effectuer une descente de gradient (ou tout autre algorithme d'optimisation). Nous avons le choix entre une différentiation manuelle, une approximation par différences finies, une différentiation automatique en mode direct et une différentiation automatique en mode inverse. TensorFlow met en œuvre cette dernière solution, mais pour bien la comprendre, examinons d'abord les autres.

DIFFÉRENTIATION MANUELLE

La première option consiste à prendre un crayon et une feuille de papier, et à exploiter nos connaissances en algèbre pour dériver l'équation appropriée. Pour la fonction $f(x, y)$ définie précédemment, cela n'a rien de très complexe. Nous utilisons simplement cinq règles :

- La dérivée d'une constante est 0.
- La dérivée de λx est λ (où λ est une constante).
- La dérivée de x^λ est $\lambda x^{\lambda-1}$; la dérivée de x^2 est donc $2x$.
- La dérivée d'une somme de fonctions est la somme des dérivées de ces fonctions.
- La dérivée de λ fois une fonction est λ fois la dérivée de la fonction.

En appliquant ces règles, nous obtenons les dérivées suivantes.

Équation B.1 – Dérivées partielles de $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Avec des fonctions plus complexes, cette méthode peut devenir très fastidieuse et le risque d'erreurs n'est pas négligeable. Heureusement, il existe d'autres approches, notamment l'approximation par différences finies.

APPROXIMATION PAR DIFFÉRENCES FINIES

Rappelons que la dérivée $h'(x_0)$ d'une fonction $h(x)$ au point x_0 correspond à la pente (coefficients directeur) de la fonction en ce point. Plus précisément, la dérivée est la limite de la pente d'une ligne droite passant par ce point x_0 et un autre point x de la fonction, lorsque x s'approche indéfiniment de x_0 (voir l'équation B.2).

Équation B.2 – Dérivée d'une fonction $h(x)$ au point x_0

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} \end{aligned}$$

Par conséquent, si nous voulons calculer la dérivée partielle de $f(x, y)$ par rapport à x , pour $x = 3$ et $y = 4$, nous pouvons simplement calculer $f(3 + \varepsilon, 4) - f(3, 4)$ et diviser le résultat par ε , en prenant ε très petit. Ce type d'approximation numérique de la dérivée est appelé *approximation par différences finies* et cette équation spécifique est le *taux d'accroissement*. C'est exactement ce que fait le code suivant :

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Malheureusement, le résultat n'est pas précis, et c'est encore pire pour les fonctions plus complexes. Les valeurs exactes sont, respectivement, 24 et 10, alors que nous obtenons à la place :

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Pour calculer les deux dérivées partielles, nous appelons $f()$ au moins trois fois (elle a été appelée à quatre reprises dans le code précédent, mais il peut être optimisé). Si nous avions 1 000 paramètres, nous devrions appeler $f()$ au moins 1 001 fois. Dans le cas des grands réseaux de neurones, l'approximation par différences finies manque donc totalement d'efficacité.

Toutefois, cette différentiation est tellement simple à mettre en œuvre qu'elle constitue un bon outil de vérification de l'implémentation des autres méthodes. Par exemple, si elle contredit la dérivée manuelle d'une fonction, il est probable que cette dernière comporte une erreur.

Pour le moment, nous avons vu deux façons de calculer des gradients: la différentiation manuelle et l'approximation par différences finies. Malheureusement, aucune des deux ne convient à l'entraînement d'un réseau de neurones à grande échelle. Tournons-nous vers la différentiation automatique, en commençant par le mode direct.

DIFFÉRENTIATION AUTOMATIQUE EN MODE DIRECT

La figure B.1 illustre le fonctionnement de la différentiation automatique en mode direct sur une fonction encore plus simple, $g(x, y) = 5 + xy$. Le graphe de cette fonction est représenté en partie gauche. La différentiation automatique en mode direct produit le graphe donné en partie droite. Il représente la dérivée partielle $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (nous pouvons obtenir de façon similaire la dérivée partielle par rapport à y).

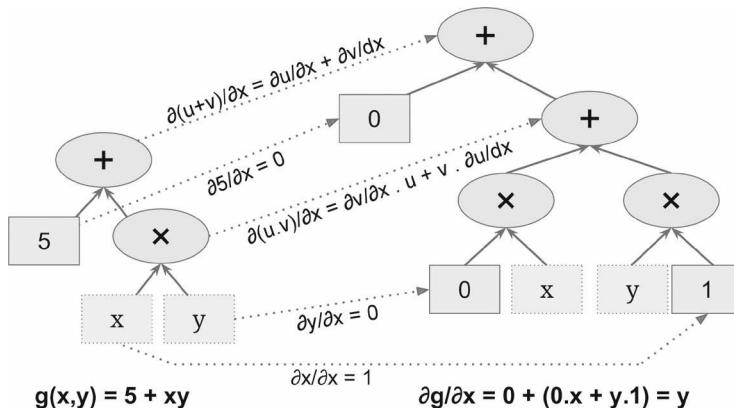


Figure B.1 – Différentiation automatique en mode direct

L'algorithme parcourt le graphe de calcul, depuis les entrées vers les sorties. Il commence par déterminer la dérivée partielle des nœuds feuilles. Le nœud de constante (5) retourne la constante 0, car la dérivée d'une constante est toujours 0. Le nœud de variable x retourne la constante 1 car $\frac{\partial x}{\partial x} = 1$, et celui de y retourne la

constante 0 car $\partial y / \partial x = 0$ (si nous recherchions la dérivée partielle par rapport à y , ce serait l'inverse).

Nous pouvons à présent remonter le graphe jusqu'au nœud de multiplication dans la fonction g . Les mathématiques nous indiquent que la dérivée d'un produit de deux fonctions u et v est $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + v \times \frac{\partial u}{\partial x}$. Nous pouvons donc construire une grande partie du graphe de droite, qui représente $0 \times x + y \times 1$.

Enfin, nous pouvons aller jusqu'au noeud d'addition dans la fonction g . Nous l'avons mentionné précédemment, la dérivée d'une somme de fonctions est la somme des dérivées de ces fonctions. Nous devons donc simplement créer un noeud d'addition et le relier aux parties du graphe que nous avons déjà déterminées. Nous obtenons la dérivée partielle correcte : $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

Il est possible de simplifier cette équation (énormément). Il suffit d'appliquer quelques étapes d'élagage à ce graphe de calcul pour nous débarrasser de toutes les opérations inutiles. Nous obtenons alors un graphe beaucoup plus petit constitué d'un seul nœud : $\partial g / \partial x = y$. Dans ce cas, la simplification est relativement aisée, mais la différentiation automatique en mode direct d'une fonction plus complexe peut produire un graphe volumineux difficile à simplifier et conduisant à des performances sous-optimales.

Notez que nous avons commencé avec un graphe de calcul et que la différentiation automatique en mode direct a produit un autre graphe de calcul. Il s'agit d'une *définition symbolique*, qui présente deux caractéristiques intéressantes. Premièrement, après que le graphe de calcul de la dérivée a été généré, nous pouvons l'employer à de nombreuses reprises pour calculer les dérivées de la fonction donnée pour n'importe quelle valeur de x et de y . Deuxièmement, nous pouvons effectuer de nouveau une différentiation automatique en mode direct sur le graphe résultant pour obtenir les dérivées secondes (autrement dit, les dérivées des dérivées), si jamais nous en avions besoin. Nous pouvons même calculer les dérivées de troisième ordre, etc.

Mais il est également possible d'effectuer une différentiation automatique en mode direct sans construire un graphe (c'est-à-dire non pas de façon symbolique mais numérique), simplement en calculant des résultats intermédiaires à la volée. Une façon de procéder se fonde sur les *nombres duaux*, qui sont des nombres (bizarres mais fascinants) de la forme $a + b\epsilon$, où a et b sont des réels, et ϵ un nombre infinitésimal de sorte que $\epsilon^2 = 0$ (mais $\epsilon \neq 0$). Vous pouvez voir le nombre dual $42 + 24\epsilon$ comme $42,0000\dots000024$, avec une infinité de 0 (il s'agit évidemment d'une simplification, juste pour donner une idée de ce que sont les nombres duaux). Un nombre dual est représenté en mémoire sous forme d'un couple de nombres à virgule flottante. Par exemple, $42 + 24\epsilon$ est représenté par le couple $(42,0; 24,0)$.

Les nombres duaux peuvent être additionnés, multipliés, etc., comme le montre l'équation B.3.

Équation B.3 – Quelques opérations sur les nombres duaux

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Plus important, on peut montrer que $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$ et donc que le calcul de $h(a + \epsilon)$ nous donne en une seule opération à la fois $h(a)$ et la dérivée $h'(a)$. La figure B.2 illustre le calcul de la dérivée partielle de $f(x, y)$ par rapport à x en $x = 3$ et $y = 4$ à l'aide des nombres duaux. Il suffit de calculer $f(3 + \epsilon, 4)$ pour obtenir un nombre dual dont la première composante est égale à $f(3, 4)$, et la seconde à $\frac{\partial f}{\partial x}(3, 4)$.

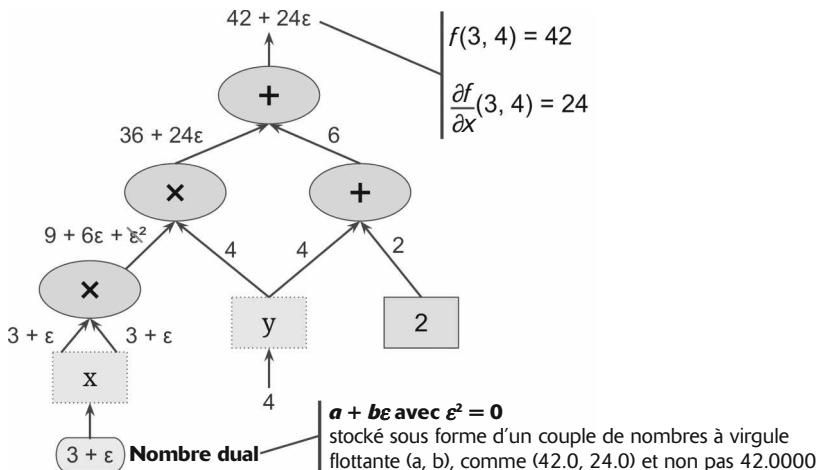


Figure B.2 – Différentiation automatique en mode direct avec des nombres duaux

Pour calculer $\frac{\partial f}{\partial x}(3, 4)$, nous devons à nouveau parcourir le graphe, mais cette fois-ci avec $x = 3$ et $y = 4 + \epsilon$.

La différentiation automatique en mode direct est donc bien plus précise que l'approximation par différences finies, mais elle souffre du même inconvénient majeur, tout au moins en cas de nombreuses entrées et de peu de sorties (ce que l'on trouve avec les réseaux de neurones) : elle demande 1 000 passes dans le graphe pour calculer les dérivées partielles sur 1 000 paramètres. C'est dans cette situation que la différentiation automatique en mode inverse brille : elle est capable de les calculer toutes en seulement deux passes dans le graphe.

DIFFÉRENTIATION AUTOMATIQUE EN MODE INVERSE

La différentiation automatique en mode inverse est la méthode implementée par TensorFlow. Elle commence par un parcours du graphe dans le sens avant (des entrées

vers la sortie) pour calculer la valeur de chaque noeud. Elle procède ensuite à une seconde passe, cette fois-ci en sens inverse (de la sortie vers les entrées) pour calculer toutes les dérivées partielles. Le « mode inverse » du nom de cette différentiation automatique vient de cette seconde passe sur le graphe, dans laquelle les gradients vont dans le sens inverse. La figure B.3 illustre cette seconde phase. Au cours de la première, toutes les valeurs des noeuds ont été calculées, en partant de $x = 3$ et $y = 4$. Ces valeurs sont données en partie inférieure droite de chaque noeud (par exemple, $x \times x = 9$). Pour faciliter la lecture, les noeuds sont libellés n_1 à n_7 . Le noeud de sortie est n_7 : $f(3, 4) = n_7 = 42$.

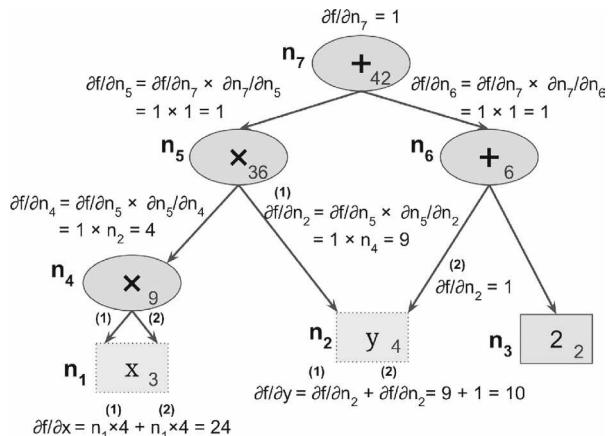


Figure B.3 – Différentiation automatique en mode inverse

L'idée est de descendre progressivement dans le graphe, en calculant la dérivée partielle de $f(x, y)$ par rapport à chaque noeud consécutif, jusqu'à atteindre les noeuds de variables. Pour cela, la différentiation automatique en mode inverse s'appuie énormément sur la *règle de la chaîne* (ou théorème de dérivation des fonctions composées), donnée par l'équation B.4.

Équation B.4 – Règle de la chaîne

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Puisque n_7 est le noeud de sortie, $f = n_7$ et $\frac{\partial f}{\partial n_7} = 1$.

Continuons à descendre dans le graphe jusqu'à n_5 : quelle est la variation de f lorsque n_5 varie? La réponse est $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. Puisque nous savons déjà que

$\frac{\partial f}{\partial n_7} = 1$, nous avons uniquement besoin de $\frac{\partial n_7}{\partial n_5}$. Puisque n_7 effectue simplement la somme $n_5 + n_6$, nous trouvons que $\frac{\partial n_7}{\partial n_5} = 1$, et donc $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Nous pouvons à présent poursuivre vers le nœud n_4 : quelle est la variation de f lorsque n_4 varie? La réponse est $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Puisque $n_5 = n_4 \times n_2$, nous déterminons que $\frac{\partial n_5}{\partial n_4} = n_2$, et donc $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

Le processus continue jusqu'à atteindre la base du graphe. À ce stade, nous aurons calculé toutes les dérivées partielles de $f(x,y)$ au point $x = 3$ et $y = 4$. Dans cet exemple,

nous trouvons $\frac{\partial f}{\partial x} = 24$ et $\frac{\partial f}{\partial y} = 10$. Il semble que nous ayons bon !

La différentiation automatique en mode inverse est une technique très puissante et précise, en particulier lorsque nous avons de nombreuses entrées et peu de sorties. En effet, elle ne demande qu'une seule passe en avant et une passe en arrière par sortie pour calculer toutes les dérivées partielles de toutes les sorties par rapport à toutes les entrées. Lors de l'entraînement des réseaux de neurones, nous souhaitons généralement minimiser la perte. Nous avons par conséquent une seule sortie (la perte), et deux passes sur le graphe suffisent pour calculer les gradients. La différentiation automatique accepte également les fonctions qui ne sont pas différentiables en tout point, à condition de lui demander de calculer les dérivées partielles en des points où elles sont bien différentiables.

À la figure B.3, les résultats numériques sont calculés à la volée, à chaque noeud. TensorFlow procède différemment, en créant à la place un nouveau graphe de calcul. Autrement dit, il met en œuvre une différentiation automatique en mode inverse *symbolique*. De cette manière, le graphe qui sert au calcul des gradients de la perte par rapport à tous les paramètres du réseau de neurones n'est généré qu'une seule fois et peut être exécuté ensuite à de nombreuses reprises, dès que l'optimiseur a besoin de calculer les gradients. De plus, cela permet de calculer des dérivés d'ordre supérieur, si nécessaire.



Si vous voulez implémenter une nouvelle opération TensorFlow de bas niveau en C++ et si vous souhaitez la rendre compatible avec la différentiation automatique, vous devrez fournir une fonction qui retourne les dérivées partielles des sorties de la fonction par rapport à ses entrées. Par exemple, supposons que vous implémentiez une fonction qui calcule le carré de son entrée, $f(x) = x^2$. Dans ce cas, vous devez fournir la fonction dérivée correspondante $f'(x) = 2x$.

Annexe C

Autres architectures de RNA répandues

Dans cette annexe, nous passons rapidement en revue quelques architectures de réseaux de neurones historiquement importantes, mais qui sont moins employées aujourd’hui que les perceptrons multicouches profonds (chapitre 2), les réseaux de neurones convolutifs (chapitre 6), les réseaux de neurones récurrents (chapitre 7) ou les autoencodeurs (chapitre 9). Elles sont souvent citées dans la littérature et certaines sont encore utilisées dans de nombreuses applications. C'est pourquoi il est intéressant de les connaître. De plus, nous présentons les *machines de Boltzmann profondes* (DBN, *deep belief network*), qui représentaient l'état de l'art du Deep Learning jusqu'au début des années 2010. Elles font encore l'objet d'une recherche très active et elles n'ont peut-être pas encore dit leur dernier mot.

RÉSEAUX DE HOPFIELD

Les réseaux de Hopfield ont été présentés pour la première fois par W. A. Little en 1974, puis rendus populaires par J. Hopfield en 1982. Il s'agit de réseaux à *mémoire associative*: on commence par leur apprendre certains motifs, puis, lorsqu'ils rencontrent un nouveau motif, ils produisent (avec un peu de chance) le motif appris le plus proche. Ils se sont révélés utiles notamment dans la reconnaissance des caractères, avant d'être dépassés par d'autres approches. On entraîne tout d'abord le réseau en lui fournissant des exemples d'images de caractères (chaque pixel binaire correspond à un neurone) et, lorsqu'on lui montre une nouvelle image de caractère, il produit en sortie, après quelques itérations, le caractère appris le plus proche.

Ce sont des graphes intégralement connectés (voir la figure C.1), car chaque neurone est connecté à chaque autre neurone. Puisque les images de la figure font 6×6 pixels, le réseau de neurones représenté à gauche devrait contenir 36 neurones (et 630 connexions), mais, pour une meilleure lisibilité, nous avons dessiné un réseau bien plus petit.

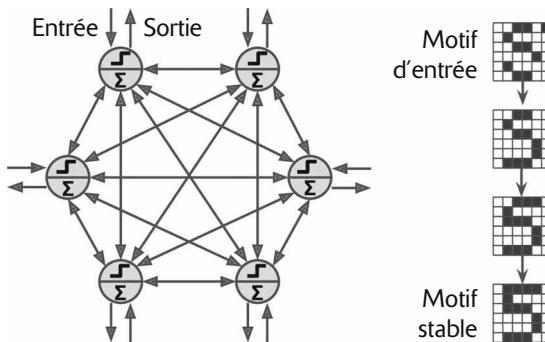


Figure C.1 – Réseau de Hopfield

L'algorithme d'entraînement se fonde sur la loi de Hebb. Pour chaque image d'entraînement, le poids entre deux neurones est augmenté si les pixels correspondants sont tous deux allumés ou éteints, mais diminué si l'un des pixels est allumé et l'autre, éteint.

Pour montrer une nouvelle image au réseau, vous activez simplement les neurones qui correspondent aux pixels allumés. Le réseau calcule ensuite la sortie de chaque neurone, et l'on obtient ainsi une nouvelle image. Vous pouvez prendre cette nouvelle image et répéter l'intégralité du processus. Au bout d'un certain temps, le réseau parvient à un état stable. En général, cela correspond à l'image d'entraînement qui ressemble le plus à l'image d'entrée.

Une fonction d'énergie est associée à un réseau de Hopfield. À chaque itération, l'énergie diminue, ce qui garantit à terme une stabilisation du réseau dans un état de faible énergie. L'algorithme d'entraînement ajuste les poids de sorte que le niveau d'énergie des motifs d'entraînement diminue. De cette manière, le réseau finira normalement par se stabiliser dans l'une de ces configurations de faible énergie. Malheureusement, il est possible que des motifs qui ne faisaient pas partie du jeu d'entraînement finissent également par avoir une énergie faible. Le réseau se stabilise donc parfois dans une configuration qui n'a pas été apprise. Il s'agit alors de *faux motifs*.

Les réseaux de Hopfield ont également pour autre inconvénient majeur une difficulté à grandir. Leur capacité de stockage est approximativement égale à 14 % du nombre de neurones. Par exemple, pour classer des images de 28×28 pixels, il faut un réseau de Hopfield avec 784 neurones intégralement connectés, et donc 306 936 poids. Un tel réseau ne pourrait apprendre qu'environ 110 caractères différents (14 % de 784). Cela fait beaucoup de paramètres pour une si petite capacité de stockage.

MACHINES DE BOLTZMANN

Les *machines de Boltzmann* ont été inventées en 1985 par Geoffrey Hinton et Terrence Sejnowski. À l'instar des réseaux de Hopfield, ce sont des RNA intégralement connectés, mais elles se fondent sur des *neurones stochastiques*: au lieu que la valeur de sortie soit décidée par une fonction échelon déterministe, ces neurones produisent un 1

avec une certaine probabilité, sinon un 0. La fonction de probabilité employée par ces RNA repose sur la distribution de Boltzmann (utilisée en mécanique statistique), d'où leur nom. L'équation C.1 donne la probabilité qu'un neurone particulier produise un 1.

Équation C.1 – Probabilité que le $i^{\text{ème}}$ neurone produise un 1

$$p(s_i^{(\text{étape suivante})} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{i,j}s_j + b_i}{T}\right)$$

- s_j est l'état du $j^{\text{ème}}$ neurone (0 ou 1).
- $w_{i,j}$ est le poids de la connexion entre les $i^{\text{ème}}$ et $j^{\text{ème}}$ neurones. Notez que $w_{i,i}$ est fixé à 0.
- b_i est le terme constant du $i^{\text{ème}}$ neurone. On peut implémenter ce terme en ajoutant un neurone de terme constant au réseau.
- N est le nombre de neurones dans le réseau.
- T est un nombre qui donne la *température* du réseau; plus la température est élevée, plus la sortie est aléatoire (plus la probabilité approche de 50 %).
- σ est la fonction logistique.

Dans les machines de Boltzmann, les neurones sont séparés en deux groupes: les *unités visibles* et les *unités cachées* (voir la figure C.2). Tous les neurones opèrent de la même manière stochastique, mais ce sont les unités visibles qui reçoivent les entrées et qui fournissent les sorties.

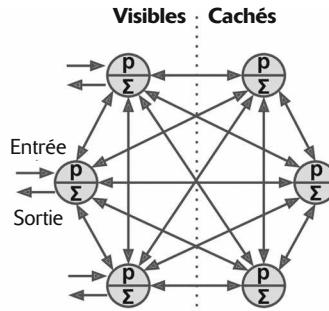


Figure C.2 – Machine de Boltzmann

En raison de sa nature stochastique, une machine de Boltzmann ne parviendra jamais à se stabiliser dans une configuration figée, mais basculera sans cesse entre plusieurs configurations. Si elle s'exécute pendant une durée suffisamment longue, la probabilité d'observer une configuration particulière sera fonction non pas de la configuration d'origine, mais des poids des connexions et des termes constants (de la même manière, si l'on mélange un jeu de cartes suffisamment longtemps, la configuration du jeu ne dépend plus de l'état initial). Lorsque le réseau atteint cet état dans lequel la configuration d'origine est «oubliée», il est dit en *équilibre thermique* (même si sa configuration

change en permanence). En fixant correctement les paramètres du réseau, en le laissant atteindre un équilibre thermique et en observant son état, il est possible de simuler une grande diversité de lois de probabilité. Il s'agit d'un *modèle génératif*.

Entraîner une machine de Boltzmann signifie trouver les paramètres qui permettront au réseau d'approcher la loi de probabilité du jeu d'entraînement. Par exemple, s'il y a trois neurones visibles et si le jeu d'entraînement comprend 75 % de triplets (0, 1, 1), 10 % de triplets (0, 0, 1) et 15 % de triplets (1, 1, 1), alors, au terme de l'entraînement d'une machine de Boltzmann, vous pouvez l'utiliser pour générer des triplets binaires aléatoires avec la même loi de probabilité. Par exemple, environ 75 % des triplets générés par cette machine de Boltzmann seront (0, 1, 1).

Un tel modèle génératif peut être exploité de différentes manières. Par exemple, s'il est entraîné sur des images et si l'on fournit au réseau une image partielle ou brouillée, il la « réparera » automatiquement de manière raisonnable. Vous pouvez également utiliser un modèle génératif pour la classification : il suffit pour cela d'ajouter quelques neurones visibles pour encoder la classe de l'image d'entraînement (par exemple, ajouter dix neurones visibles et pendant l'entraînement activer uniquement le cinquième lorsque l'image d'entraînement représente un 5). Ensuite, lorsqu'une nouvelle image sera fournie, le réseau activera automatiquement les neurones visibles appropriés, indiquant ainsi la classe de l'image (par exemple, il allumera le cinquième neurone visible si l'image représente un 5).

Malheureusement, il n'existe aucune technique vraiment efficace pour entraîner des machines de Boltzmann. En revanche, des algorithmes assez puissants ont été développés pour entraîner des *machines de Boltzmann restreintes*.

MACHINES DE BOLTZMANN RESTREINTES

Une machine de Boltzmann restreinte (RBM, *Restricted Boltzmann Machines*) n'est rien d'autre qu'une machine de Boltzmann dépourvue de connexions entre les unités visibles d'une part, et entre les unités cachées d'autre part ; les connexions existent uniquement entre les unités visibles et les unités cachées. Par exemple, la figure C.3 représente une RBM avec trois unités visibles et quatre unités cachées.

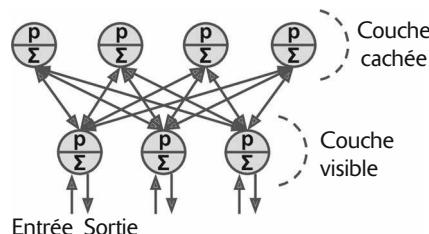


Figure C.3 – Machine de Boltzmann restreinte

Un algorithme d'entraînement très efficace, dit de *divergence contrastive*, a été proposé³²³ en 2005 par Miguel Á. Carreira-Perpiñán et Geoffrey Hinton. Voici son fonctionnement. Pour chaque instance d'entraînement \mathbf{x} , l'algorithme commence par la transmettre au réseau en fixant l'état des unités visibles à x_1, x_2, \dots, x_n . Ensuite, l'état des unités cachées est calculé en appliquant l'équation stochastique C.1 donnée précédemment. Vous obtenez un vecteur caché \mathbf{h} (où h_i est égal à l'état de la $i^{\text{ème}}$ unité). L'état des unités visibles est ensuite calculé, en appliquant la même équation stochastique. Cela donne un vecteur \mathbf{x}' . Puis, l'état des unités cachées est de nouveau calculé, produisant un vecteur \mathbf{h}' . Le poids de chaque connexion peut alors être actualisé en appliquant la règle de l'équation C.2, où η est le taux d'apprentissage.

Équation C.2 – Mise à jour d'un poids par divergence contrastive

$$w_{i,j} \leftarrow w_{i,j} + \eta (\mathbf{x}\mathbf{h}^T - \mathbf{x}'\mathbf{h}'^T)$$

L'avantage de cet algorithme est qu'il est inutile d'attendre que le réseau atteigne un équilibre thermique. Il fait un aller-retour-aller, et c'est tout. Il est incomparablement plus efficace que les algorithmes précédents et cette efficacité a été essentielle aux premiers succès du Deep Learning fondé sur l'empilement de multiples RBM.

MACHINES DE BOLTZMANN PROFONDES

Il est possible d'empiler plusieurs couches de RBM. Les unités cachées du RBM de premier niveau servent d'unités visibles au RBM de deuxième niveau, et ainsi de suite. Un tel empilement de RBM est appelé *machine de Boltzmann profonde* (DBN, Deep Belief Net).

Yee-Whye Teh, l'un des étudiants de Geoffrey Hinton, a observé qu'il était possible d'entraîner des DBN une couche à la fois à l'aide de la divergence contrastive, en commençant par les couches inférieures, puis en remontant progressivement vers les couches supérieures. Cette observation a été à l'origine d'un article³²⁴ révolutionnaire qui a déclenché le tsunami du Deep Learning en 2006.

À l'instar des RBM, les DBN apprennent à reproduire la loi de probabilité de leurs entrées, sans aucune supervision. Ils sont toutefois bien meilleurs, pour la même raison que les réseaux de neurones profonds sont plus performants que les réseaux peu profonds : les données du monde réel sont souvent organisées en motifs hiérarchiques et les DBN exploitent cette particularité. Leurs couches inférieures découvrent des caractéristiques de bas niveau dans les données d'entrée, tandis que les couches supérieures apprennent des caractéristiques de haut niveau.

Comme les RBM, les DBN opèrent essentiellement sans supervision, mais vous pouvez également les entraîner de manière supervisée en ajoutant des unités visibles représentant des étiquettes. Par ailleurs, les DBN présentent l'intérêt de pouvoir être

323. Miguel Á. Carreira-Perpiñán et Geoffrey E. Hinton, «On Contrastive Divergence Learning», *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics* (2005), 59-66 : <https://hml.info/135>.

324. Geoffrey E. Hinton et al., « A Fast Learning Algorithm for Deep Belief Nets », *Neural Computation*, 18 (2006), 1527-1554 : <https://hml.info/136>.

entraînés de façon semi-supervisée. La figure C.4 illustre un DBN configuré pour cette forme d'apprentissage.

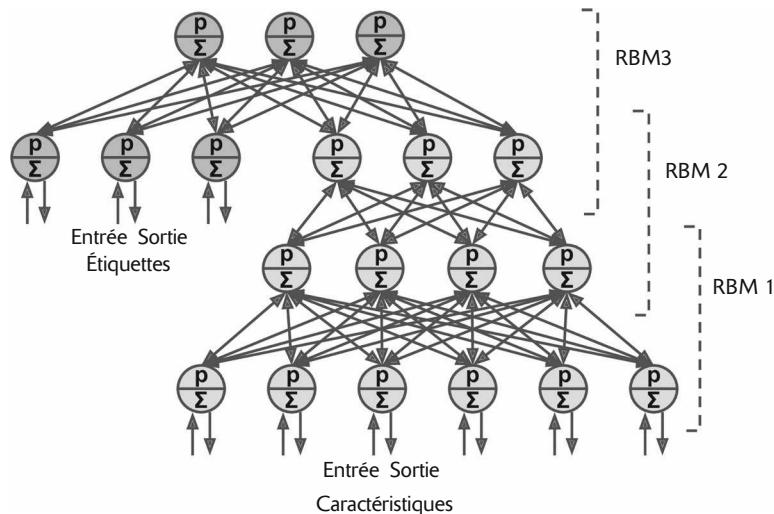


Figure C.4 – Machine de Boltzmann profonde configurée pour un apprentissage semi-supervisé

Tout d'abord, RBM 1 est entraîné sans supervision et apprend les caractéristiques de bas niveau présentes dans les données d'entraînement. Ensuite, RBM 2 est entraîné à partir des unités cachées de RBM 1, qui lui servent d'entrées, de nouveau sans supervision. Il apprend des caractéristiques de plus haut niveau (notons que les unités cachées de RBM 2 comprennent uniquement les trois unités de droite, sans les unités d'étiquettes). Voilà l'idée générale ; plusieurs autres RBM peuvent être empilés ainsi. Jusque-là, l'entraînement a été intégralement non supervisé. Enfin, RBM 3 est entraîné en utilisant en entrée les unités cachées de RBM 2 et des unités visibles supplémentaires pour représenter les étiquettes cibles (par exemple, un vecteur *one-hot* donnant la classe de l'instance). Il apprend à associer des caractéristiques de haut niveau à des étiquettes d'entraînement. Cette étape est supervisée.

Au terme de l'entraînement, si vous fournissez une nouvelle instance à RBM 1, le signal se propage jusqu'à RBM 2, puis jusqu'au sommet de RBM 3, pour revenir ensuite vers les unités d'étiquettes. Si le réseau a bien appris, alors l'étiquette appropriée sera révélée. Voilà comment un DBN peut servir à la classification.

Cette approche semi-supervisée présente un grand avantage : il n'est pas nécessaire de disposer d'une grande quantité de données d'entraînement étiquetées. Si les RBM non supervisés réalisent un bon travail, une petite quantité d'instances d'entraînement étiquetées par classe suffira. C'est comme un jeune enfant qui apprend à reconnaître des objets sans supervision. Lorsque vous lui montrez une chaise et dites « chaise », il peut associer le mot « chaise » à la classe des objets qu'il a déjà appris à reconnaître par lui-même. Vous n'avez pas besoin de désigner chaque chaise individuelle en annonçant « chaise » ; seuls quelques exemples suffiront (ils doivent tout

de même être suffisamment nombreux pour que l'enfant sache que vous faites référence à la chaise et non au chat assis dessus, ou à la couleur de la chaise ou encore à son dossier ou ses pieds).

Étonnamment, les DBN peuvent également travailler en sens inverse. Si vous activez l'une des unités d'étiquettes, le signal se propage jusqu'aux unités cachées de RBM 3, puis descend vers RBM 2 et RBM 1. Une nouvelle instance est alors générée par les unités visibles de RBM 1. Elle ressemble en général à une instance normale de la classe dont vous avez activé l'unité d'étiquette. Cette capacité de génération des DBN est très puissante. Elle a été employée, par exemple, pour générer automatiquement des légendes d'images, et inversement. Tout d'abord, un DBN est entraîné (sans supervision) pour apprendre des caractéristiques dans des images, et un autre DBN est entraîné (de nouveau sans supervision) pour apprendre des caractéristiques dans des jeux de légendes (par exemple, « voiture » vient souvent avec « automobile »). Ensuite, un RBM est empilé au-dessus des deux DBN et entraîné avec un jeu d'images et leur légende. Il apprend à relier des caractéristiques de haut niveau dans des images avec des caractéristiques de haut niveau dans des légendes. Si vous fournissez ensuite la photo d'une voiture au DBN d'images, le signal se propage dans le réseau jusqu'au RBM supérieur, et redescend vers le DBN de légendes, générant une légende. En raison de la nature stochastique des RBM et des DBN, la légende change de façon aléatoire, mais elle correspond généralement bien à l'image. En générant quelques centaines de légendes, il est fort probable que celles qui reviennent le plus souvent constituent une bonne description de l'image³²⁵.

CARTES AUTOADAPTATIVES

Les *cartes autoadaptatives* (SOM, *Self-Organizing Map*) sont assez différentes de tous les autres types de réseaux de neurones dont nous avons parlé jusqu'à présent. Elles sont utilisées pour générer une représentation en petites dimensions d'un jeu de données en grandes dimensions, le plus souvent pour la visualisation, le partitionnement ou la classification. Les neurones sont répartis sur une carte (généralement en 2D pour la visualisation, mais le nombre de dimensions peut être quelconque), comme l'illustre la figure C.5, et chaque neurone possède une connexion pondérée avec chaque entrée (la figure montre uniquement deux entrées, mais elles sont généralement beaucoup plus nombreuses, puisque l'objectif des SOM est de réduire la dimensionnalité).

325. Pour plus de détails et une démonstration, voir la vidéo publiée par Geoffrey Hinton à l'adresse <https://homl.info/137>

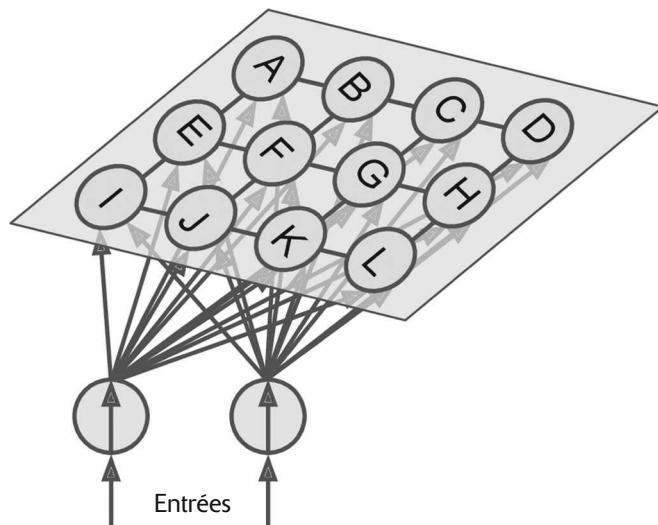


Figure C.5 – Carte autoadaptative

Après l’entraînement du réseau, vous pouvez lui fournir une nouvelle instance, qui n’activera qu’un seul neurone (donc un point sur la carte) : celui dont le vecteur de poids est le plus proche du vecteur d’entrée. En général, les instances qui sont proches dans l’espace d’entrée d’origine vont activer des neurones qui sont proches sur la carte. C’est pourquoi les SOM conviennent bien à la visualisation (vous pouvez notamment identifier facilement des groupes d’instances similaires sur la carte), mais également à des applications comme la reconnaissance vocale. Par exemple, si chaque instance représente l’enregistrement audio d’une personne qui prononce une voyelle, alors différentes prononciations de la voyelle « a » activeront des neurones dans la même zone de la carte, tandis que les instances de la voyelle « e » activeront des neurones dans une autre zone. Les sons intermédiaires activeront généralement des neurones intermédiaires sur la carte.



Il existe une différence importante entre cette technique de réduction de la dimensionnalité et d’autres techniques plus classiques telles que l’analyse en composantes principales³²⁶. Avec les cartes autoadaptatives, toutes les instances sont en effet associées à un nombre discret de points dans l’espace de faible dimension (un point par neurone). Lorsque les neurones sont très peu nombreux, cette technique tient plus du partitionnement que de la réduction de dimensionnalité.

L’algorithme d’entraînement n’est pas supervisé. Tous les neurones sont en concurrence avec les autres. Les poids sont tout d’abord initialisés de façon aléatoire. Une instance d’entraînement est ensuite choisie aléatoirement et transmise au réseau. Tous les neurones calculent la distance entre leur vecteur de poids et le vecteur d’entrée (un fonctionnement très différent des neurones artificiels vus jusqu’à présent).

326. Voir le chapitre 8 de l’ouvrage *Machine Learning avec Scikit-Learn*, A. Géron, Dunod (2^e édition, 2019).

Le neurone qui mesure la plus petite distance l'emporte et ajuste son vecteur de poids pour se rapprocher plus encore du vecteur d'entrée. Il devient alors le favori lors des compétitions suivantes sur d'autres entrées similaires à celle-ci. Il implique également les neurones voisins, qui actualisent leur vecteur de poids pour le rapprocher légèrement du vecteur d'entrée (mais pas autant que le neurone victorieux). L'algorithme choisit ensuite une autre instance d'entraînement et répète le processus, encore et encore. Les neurones voisins ont ainsi tendance à se spécialiser progressivement dans des entrées comparables³²⁷.

327. Imaginons une classe de jeunes enfants ayant des compétences à peu près équivalentes. L'un d'eux se révèle légèrement meilleur que les autres au basket. Cela l'encourage à pratiquer encore plus, notamment avec ses amis. Après un certain temps, ce groupe de copains devient si bon au basket que les autres enfants ne peuvent pas rivaliser. Mais ce n'est pas un problème, car ces autres jeunes se seront spécialisés dans d'autres domaines. À terme, la classe est constituée de petits groupes spécialisés.

Annexe D

Structures de données spéciales

Dans cette annexe, nous présentons très rapidement les structures de données reconnues par TensorFlow, autres que les tenseurs à virgule flottante ou entiers classiques. Cela comprend les chaînes de caractères, les tenseurs irréguliers, les tenseurs creux, les tableaux de tenseurs, les ensembles et les files d'attente.

CHAÎNES DE CARACTÈRES

Ces tenseurs contiennent des chaînes de caractères d'octets et se révéleront particulièrement utiles pour le traitement automatique du langage naturel (voir le chapitre 8) :

```
>>> tf.constant(b"hello world")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'hello world'>
```

Si vous construisez un tenseur avec une chaîne de caractères Unicode, TensorFlow choisit automatiquement le codage UTF-8 :

```
>>> tf.constant("café")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

Il est également possible de créer des tenseurs qui représentent des chaînes de caractères Unicode, simplement en créant un tableau d'entiers 32 bits représentant chacun un seul point de code Unicode³²⁸ :

```
>>> tf.constant([ord(c) for c in "café"])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

328. Si les points de code Unicode ne vous sont pas familiers, consultez la page <https://homl.info/unicode>.



Dans les tenseurs de type `tf.string`, la longueur de la chaîne n'est pas indiquée dans la forme du tenseur. Autrement dit, les chaînes de caractères sont considérées comme des valeurs atomiques. En revanche, dans un tenseur de chaînes de caractères Unicode (c'est-à-dire un tenseur `int32`), la longueur de la chaîne fait partie de la forme du tenseur.

Le package `tf.strings` offre plusieurs fonctions de manipulation des tenseurs chaînes de caractères, comme `length()` pour dénombrer les octets dans une chaîne d'octets (ou dénombrer les points de code si vous indiquez `unit="UTF8_CHAR"`), `unicode_encode()` pour convertir un tenseur chaîne de caractères Unicode (tenseur `int32`) en un tenseur chaîne de caractères d'octets, et `unicode_decode()` pour la conversion inverse :

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
    numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

Vous pouvez également manipuler des tenseurs qui contiennent plusieurs chaînes de caractères :

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
    numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[ 67  97 102 233  67 111 102 102 101 101  99  97
 102 102 232 21654 21857], shape=(17,), dtype=int32),
row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
[99, 97, 102, 102, 232], [21654, 21857]]>
```

Notez que les chaînes de caractères décodées sont enregistrées dans un `RaggedTensor`. Qu'est-ce donc ?

TENSEURS IRRÉGULIERS

Un tenseur irrégulier (*ragged tensor*) est un tenseur de type particulier qui représente une liste de tableaux de tailles différentes. Plus généralement, il s'agit d'un tenseur avec une ou plusieurs *dimensions irrégulières*, autrement dit des dimensions dont les parties peuvent avoir différentes longueurs. Dans le tenseur irrégulier `r`, la deuxième dimension est une dimension irrégulière. Dans tous les tenseurs irréguliers, la première dimension est toujours une dimension régulière (également appelée *dimension uniforme*).

Tous les éléments du tenseur irrégulier `r` sont des tenseurs normaux. Examinons par exemple le deuxième élément :

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

Le package `tf.ragged` fournit plusieurs fonctions pour créer et manipuler de tels tenseurs. Créons un second tenseur irrégulier avec `tf.ragged.constant()` et concaténons-le au premier, le long de l'axe 0 :

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

Le résultat n'est pas trop surprenant. Les tenseurs de `r2` ont été ajoutés après les tenseurs de `r` sur l'axe 0. Que se passe-t-il si nous concaténons `r` et un autre tenseur irrégulier le long de l'axe 1 ?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

Cette fois-ci, vous remarquerez que le $i^{\text{ème}}$ tenseur de `r` et le $i^{\text{ème}}$ tenseur de `r3` ont été concaténés. Le résultat est plus inhabituel, car tous ces tenseurs peuvent avoir des longueurs différentes.

Si vous invoquez la méthode `to_tensor()`, le tenseur est converti en un tenseur normal par remplissage des tenseurs plus courts avec des zéros afin d'obtenir des longueurs identiques (la valeur par défaut est modifiable via l'argument `default_value`) :

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[ 67,    97,   102,   233,      0,      0],
       [ 67,   111,   102,   102,   101,   101],
       [ 99,    97,   102,   102,   232,      0],
       [21654, 21857,      0,      0,      0,      0]], dtype=int32)>
```

De nombreuses opérations TF acceptent des tenseurs irréguliers. La liste complète se trouve dans la documentation de la classe `tf.RaggedTensor`.

TENSEURS CREUX

TensorFlow sait aussi représenter efficacement les *tenseurs creux* (*sparse tensors*), c'est-à-dire les tenseurs contenant principalement des zéros. Créez simplement un `tf.SparseTensor`, en précisant les indices et les valeurs des éléments différents de zéro, ainsi que la forme du tenseur. Les indices doivent être fournis conformément à l'ordre normal de lecture (de gauche à droite, et de haut en bas). Si vous avez un doute, utilisez `tf.sparse.reorder()`. Vous pouvez convertir un tenseur creux en un tenseur dense (normal) avec `tf.sparse.to_dense()` :

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                        values=[1., 2., 3.],
                        dense_shape=[3, 4])
```

```
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Notez que les possibilités de manipulation des tenseurs creux sont inférieures à celles des tenseurs denses. Par exemple, si vous pouvez multiplier un tenseur creux par une valeur scalaire, produisant un nouveau tenseur creux, il est impossible d'ajouter une valeur scalaire à un tenseur creux, car cela ne donnerait pas un tenseur creux :

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

TABLEAUX DE TENSEURS

Un `tf.TensorArray` représente une liste de tenseurs. Cette structure de données se révélera pratique dans les modèles dynamiques qui incluent des boucles, car elle permettra d'accumuler des résultats et d'effectuer ultérieurement des statistiques. Vous pouvez lire ou écrire des tenseurs dans n'importe quel emplacement du tableau :

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => retourne (et extrait !) tf.constant([3., 10.])
```

Notez que la lecture d'un élément l'extrait du tableau, en le remplaçant par un tenseur de même forme rempli de zéros.



Lorsque vous écrivez dans le tableau, vous devez réaffecter la sortie au tableau, comme nous l'avons fait dans l'exemple de code. Dans le cas contraire, votre code fonctionnera parfaitement en mode pressé (*eager mode*) mais échouera en mode graphe (ces modes ont été décrits au chapitre 4).

Lors de la création d'un `TensorArray`, vous devez indiquer sa taille (`size`), excepté en mode graphe. Vous pouvez également ne pas la préciser et, à la place, indiquer `dynamic_size=True`, mais les performances en pâtiroent. Par conséquent, si vous connaissez la taille à l'avance, vous devez la fixer. Vous devez également préciser `dtype`, et tous les éléments doivent avoir la même forme que le premier ajouté au tableau.

Tous les éléments peuvent être empilés dans un tenseur normal en invoquant la méthode `stack()` :

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

ENSEMBLES

TensorFlow prend en charge les ensembles d'entiers ou de chaînes de caractères (mais pas les ensembles de réels). Il les représente à l'aide de tenseurs normaux. Par exemple, l'ensemble {1, 5, 9} est représenté par le tenseur [[1, 5, 9]]. Notez que celui-ci doit avoir au moins deux dimensions et que l'ensemble doit se trouver dans la dernière. Par exemple, [[1, 5, 9], [2, 5, 11]] est un tenseur qui contient deux ensembles indépendants, {1, 5, 9} et {2, 5, 11}. Si certains ensembles sont plus courts que d'autres, vous devez les remplir à l'aide d'une valeur de remplissage (par défaut 0, mais vous pouvez la modifier).

Le package `tf.sets` comprend plusieurs fonctions de manipulation des ensembles. Par exemple, créons deux ensembles et déterminons leur union (le résultat étant un tenseur creux, nous appelons `to_dense()` pour l'afficher) :

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

Vous pouvez également calculer l'union de plusieurs couples d'ensembles simultanément :

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
   [ 0, 10, 13,  0,  0]], dtype=int32)>
```

Pour changer la valeur de remplissage, indiquez-la dans `default_value` lors de l'appel à `to_dense()` :

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
   [ 0, 10, 13, -1, -1]], dtype=int32)>
```



La valeur par défaut de `default_value` est 0. Par conséquent, lorsque vous manipulez des ensembles de chaînes de caractères, vous devez nécessairement changer `default_value` (par exemple, en la fixant à la chaîne vide).

Parmi les autres fonctions disponibles dans `tf.sets`, nous trouvons par exemple `difference()`, `intersection()` et `size()`, dont le rôle est évident. Pour vérifier si un ensemble contient ou non certaines valeurs, vous pouvez calculer l'intersection de cet ensemble et de ces valeurs. Pour ajouter des valeurs à un ensemble, calculez l'union de l'ensemble et des valeurs.

FILES D'ATTENTE

Une *file d'attente* est une structure de données dans laquelle vous pouvez ajouter des enregistrements de données, pour les en extraire plus tard. TensorFlow implémente plusieurs types de files d'attente dans le package `tf.queue`. Elles étaient très importantes pour la mise en œuvre efficace du chargement des données et des pipelines de prétraitement, mais l'API `tf.data` les a pratiquement rendues obsolètes (excepté peut-être dans quelques rares cas) car elle est beaucoup plus simple d'emploi et fournit tous les outils nécessaires à la construction de pipelines efficaces. Toutefois, par souci d'exhaustivité, examinons-les rapidement.

La file d'attente la plus simple est de type premier entré/premier sorti (FIFO, *first-in, first-out*). Pour la construire, vous devez préciser le nombre d'enregistrements qu'elle contiendra. Par ailleurs, chaque enregistrement étant un tuple de tenseurs, vous devez préciser le type de chaque tenseur et, éventuellement, leur forme. Par exemple, le code suivant crée une file d'attente FIFO avec un maximum de trois enregistrements, chacun contenant un tuple constitué d'un entier sur 32 bits et d'une chaîne de caractères. Il y ajoute ensuite deux enregistrements, affiche sa taille (2 à ce stade) et extrait un enregistrement :

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

Il est également possible d'ajouter et d'extraire plusieurs enregistrements à la fois (pour cette dernière opération, vous devez préciser les formes au moment de la création de la file d'attente) :

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Voici les autres types de files d'attente proposés :

- `PaddingFIFOQueue`

Identique à `FIFOQueue`, mais sa méthode `dequeue_many()` prend en charge l'extraction de plusieurs enregistrements de formes différentes. Elle remplit automatiquement les enregistrements les plus courts afin de garantir que tous les enregistrements du lot ont la même forme.

- `PriorityQueue`

Cette file d'attente extrait les enregistrements selon la priorité qui leur a été donnée. Cette priorité est un entier sur 64 bits défini par le premier élément de chaque enregistrement. Les enregistrements qui ont la priorité la plus faible seront extraits en premier. Les enregistrements de même priorité seront extraits selon l'ordre FIFO.

- `RandomShuffleQueue`

Les enregistrements de cette file d'attente sont extraits dans un ordre aléatoire. Elle a été utile pour la mise en œuvre d'un tampon de mélange avant l'arrivée de `tf.data`.

Lorsqu'une file d'attente est déjà pleine, l'appel à la méthode `enqueue*` () pour ajouter un nouvel enregistrement reste bloqué jusqu'à ce qu'un enregistrement soit retiré par un autre thread. De la même manière, si une file d'attente est vide et si vous essayez d'extraire un enregistrement, la méthode `dequeue*` () reste bloquée jusqu'à ce qu'un enregistrement soit ajouté à la file d'attente par un autre thread.

Annexe E

Graphes TensorFlow

Dans cette annexe, nous étudions les graphes générés par les fonctions TF (voir le chapitre 4).

FONCTIONS TF ET FONCTIONS CONCRÈTES

Les fonctions TF sont polymorphiques. Autrement dit, elles acceptent des entrées de types (et de formes) différents. Prenons, par exemple, la fonction `tf_cube()` suivante :

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Chaque fois que vous appelez une fonction TF avec une nouvelle combinaison de types ou de formes d'entrées, elle génère une nouvelle *fonction concrète* ayant son propre graphe adapté à cette combinaison spécifique. Une telle combinaison de types et de formes d'arguments est appelée *signature d'entrée*. Lorsque vousappelez la fonction TF avec une signature d'entrée qu'elle a déjà rencontrée, elle réutilise la fonction concrète générée précédemment.

Par exemple, pour l'appel `tf_cube(tf.constant(3.0))`, la fonction TF réutilise la fonction concrète déjà employée pour `tf_cube(tf.constant(2.0))` (pour des tenseurs scalaires de type float32). En revanche, elle générera une nouvelle fonction concrète pour les appels `tf_cube(tf.constant([2.0]))` ou `tf_cube(tf.constant([3.0]))` (pour des tenseurs de type float32 et de forme [1]), et encore une autre pour l'appel `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (pour des tenseurs de type float32 et de forme [2, 2]).

Vous pouvez obtenir la fonction concrète qui correspond à une combinaison particulière d'entrées en invoquant la méthode `get_concrete_function()` de la fonction TF. Elle s'utilise ensuite comme une fonction normale, mais ne prenant en

charge qu'une seule signature d'entrée (dans l'exemple suivant, des tenseurs scalaires de type float32) :

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

La figure E.1 montre la fonction TF `tf_cube()` après que nous avons exécuté `tf_cube(2)` et `tf_cube(tf.constant(2.0))`. Deux fonctions concrètes ont été générées, une pour chaque signature, chacune avec son propre *graphe de fonction* (`FuncGraph`) optimisé et sa propre *définition de fonction* (`FunctionDef`). Une définition de fonction pointe sur les parties du graphe qui correspondent aux entrées et aux sorties de fonction. Dans chaque `FuncGraph`, les nœuds (les ovales) représentent des opérations (par exemple, puissance, constante ou emplacements pour des arguments comme `x`), tandis que les arêtes (les flèches pleines entre les opérations) représentent les tenseurs qui traverseront le graphe. La fonction concrète illustrée sur la gauche est spécialisée dans le traitement de `x = 2`. TensorFlow a réussi à la simplifier en la laissant produire uniquement la valeur 8 à chaque fois (notez que la définition de fonction ne comprend aucune entrée). La fonction concrète illustrée sur la droite est spécialisée dans le traitement des tenseurs scalaires de type float32 et n'a pas pu être simplifiée. Si nous appelons `tf_cube(tf.constant(5.0))`, la deuxième fonction concrète est utilisée, l'opération de paramètre pour `x` produira 5,0, puis l'opération de puissance calculera $5.0^{**} 3$, et la sortie sera 125,0.

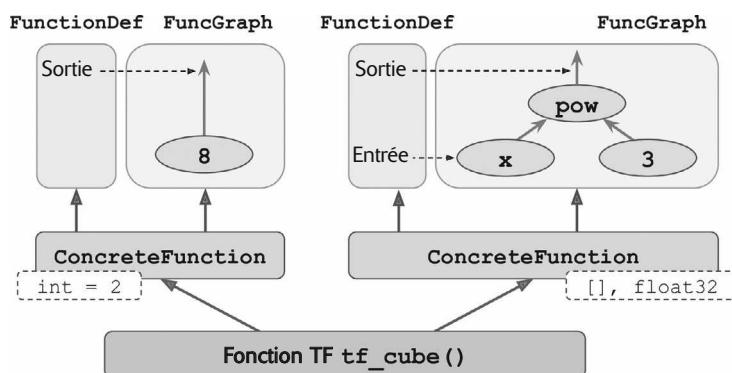


Figure E.1 – La fonction TF `tf_cube()`, avec ses fonctions concrètes et ses graphes de fonctions

Dans ces graphes, les tenseurs sont des *tenseurs symboliques*. Autrement dit, ils n'ont pas de valeur réelle, juste un type de données, une forme et un nom. Ils représentent les futurs tenseurs qui traverseront le graphe dès qu'une valeur réelle sera donnée au paramètre `x` et que le graphe sera exécuté. Grâce aux tenseurs symboliques, il est possible de préciser à l'avance la connexion des opérations et TensorFlow

peut inférer récursivement les types de données et les formes de tous les tenseurs à partir des types de données et des formes de leurs entrées.

À présent, entrons un peu plus dans les détails et voyons comment accéder aux définitions de fonctions et aux graphes de fonctions, et comment explorer les opérations et les tenseurs d'un graphe.

EXPLORER LES DÉFINITIONS ET LES GRAPHES DE FONCTIONS

Pour accéder au graphe de calcul d'une fonction concrète, vous pouvez utiliser l'attribut `graph`. La liste de ses opérations est donnée par la méthode `get_operations()` du graphe :

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

Dans cet exemple, la première opération représente l'argument d'entrée `x` (il s'agit d'un *paramètre* ou `Placeholder`). La deuxième « opération », représente la constante 3. La troisième opération correspond à l'élévation à la puissance (`**`). La dernière représente la sortie de cette fonction (il s'agit d'une opération identité qui se contente de copier la sortie de l'opération d'addition³²⁹).

Chaque opération possède une liste de tenseurs d'entrée et de sortie auxquels vous pouvez facilement accéder *via* les attributs `inputs` et `outputs`. Par exemple, obtenons la liste des entrées et les sorties de l'opération d'élévation à la puissance :

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

329. Vous pouvez l'ignorer car elle n'existe que pour des raisons techniques. Elle évite que les fonctions TF ne divulguent des structures internes.

Ce graphe de calcul est représenté à la figure E.2.

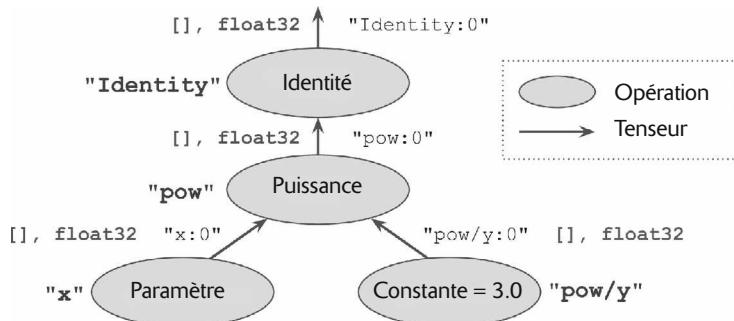


Figure E.2 – Exemple de graphe de calcul

Notez que chaque opération est nommée. Par défaut, il s'agit du nom de l'opération (par exemple, "pow"), mais vous pouvez le définir manuellement lors de l'appel à l'opération (par exemple, `tf.pow(x, 3, name="autre_nom")`). Si le nom existe déjà, TensorFlow ajoute automatiquement un suffixe unique (par exemple, "pow_1", "pow_2", etc.). Chaque tenseur a également un nom unique, toujours celui de l'opération qui produit ce tenseur plus :0 s'il s'agit de la première sortie de l'opération, :1 s'il s'agit de la deuxième sortie, et ainsi de suite. Vous pouvez obtenir une opération ou un tenseur à partir de leur nom grâce aux deux méthodes `get_operation_by_name()` et `get_tensor_by_name()` du graphe :

```

>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>

```

La fonction concrète contient également la définition de fonction (représenté par un protobuf³³⁰), qui comprend la signature de la fonction. Cette signature permet à la fonction concrète de savoir dans quels paramètres placer les valeurs d'entrée et quels tenseurs retourner :

```

>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
    name: "x"
    type: DT_FLOAT
}
output_arg {
    name: "identity"
    type: DT_FLOAT
}

```

Attardons-nous à présent sur le traçage.

330. Les protobufs sont des objets de format Protocol Buffers, un format binaire répandu décrit au chapitre 5.

AU PLUS PRÈS DU TRAÇAGE

Modifions la fonction `tf_cube()` pour qu'elle affiche son entrée :

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Puis, appelons-la :

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

Le résultat semble correct, mais examinez ce qui a été affiché : `x` est un tenseur symbolique ! Il possède une forme et un type de données, mais aucune valeur. Il a également un nom ("`x:0`"). Cela vient du fait que la fonction `print()` n'étant pas une opération TensorFlow, elle s'exécute uniquement lors du traçage de la fonction Python, ce qui se produit en mode graphe, avec des arguments remplacés par des tenseurs symboliques (de même type et forme, mais sans valeur). Puisque la fonction `print()` n'a pas été capturée dans le graphe, les appels suivants à `tf_cube()` avec des tenseurs scalaires de type `float32` n'affichent rien :

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

En revanche, si nous appelons `tf_cube()` avec un tenseur de forme ou de type différent, ou avec une nouvelle valeur Python, la fonction est de nouveau tracée et la fonction `print()` est appelée :

```
>>> result = tf_cube(2) # Nouvelle valeur Python : traçage !
x = 2
>>> result = tf_cube(3) # Nouvelle valeur Python : traçage !
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # Nouvelle forme : traçage !
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # Nouvelle forme :
# traçage !
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Même forme :
# aucun traçage
```



Si votre fonction inclut des traitements secondaires en Python (si, par exemple, elle enregistre des journaux sur le disque), sachez que le code correspondant ne s'exécutera qu'au moment du traçage de la fonction (c'est-à-dire chaque fois que la fonction TF est appelée avec une nouvelle signature d'entrée). Il est préférable de supposer que la fonction peut être tracée (ou non) lorsque la fonction TF est appelée.

Dans certains cas, vous pourriez souhaiter limiter une fonction TF à une signature d'entrée particulière. Par exemple, supposons que vous sachiez que vous appellerez toujours une fonction TF avec des lots d'images de 28×28 pixels, mais que les lots aient des tailles très différentes. Vous ne voulez pas que TensorFlow génère une

fonction concrète différente pour chaque taille de lot, ni compter sur lui pour déterminer quand utiliser None. Dans ce cas, vous pouvez préciser la signature d'entrée de la manière suivante :

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # Enlever la moitié des lignes et colonnes
```

Cette fonction TF acceptera tout tenseur de type float32 et de forme $[*, 28, 28]$, et réutilisera à chaque fois la même fonction concrète :

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # OK. Traçage de la fonction
preprocessed_images = shrink(img_batch_2) # OK. Même fonction concrète
```

En revanche, si vous tentez d'appeler cette fonction TF avec une valeur Python ou un tenseur ayant un type de donnée ou une forme non pris en charge, une exception est levée :

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3) # ValueError! Signature inattendue
```

CAPTURER LE FLUX DE CONTRÔLE AVEC AUTOGRAPH

Si votre fonction contient une simple boucle `for`, quel comportement attendez-vous ? Prenons, par exemple, une fonction qui ajoute 10 à son entrée en additionnant dix fois la valeur 1 :

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

Elle fonctionne parfaitement mais, lorsque nous examinons son graphe, nous constatons l'absence totale de boucle. Elle contient uniquement dix opérations d'addition !

```
>>> add_10(tf.constant(0))
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...]
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

En réalité, ce résultat fait sens. Lorsque la fonction a été tracée, la boucle s'est exécutée à dix reprises et l'opération `x += 1` a été effectuée dix fois. Puisque que la fonction était en mode graphe, elle a enregistré cette opération dix fois dans le graphe. Vous pouvez considérer cette boucle `for` comme une boucle « statique » déroulée lorsque le graphe est créé.

Si vous préférez que le graphe contienne une boucle « dynamique » (autrement dit, une boucle qui s'exécute lorsque le graphe est exécuté), vous pouvez la créer manuellement à l'aide de l'opération `tf.while_loop()`, mais cette solution est peu intuitive (voir l'exemple donné dans la section « Using AutoGraph to Capture Control Flow » du notebook³³¹ du chapitre 4). À la place, il est plus simple d'utiliser la fonctionnalité *AutoGraph* de TensorFlow décrite au chapitre 4. En réalité, *AutoGraph* est activé par défaut (si jamais vous deviez le désactiver, passez `autograph=False` à `tf.function()`). Mais alors, s'il est actif, pourquoi n'a-t-il pas capturé la boucle `for` dans la fonction `add_10()`? Pour la simple raison qu'il capture uniquement les boucles `for` qui itèrent non pas sur `range()` mais sur `tf.range()`. Vous avez ainsi deux possibilités :

- Si vous utilisez `range()`, la boucle `for` sera statique et ne sera exécutée qu'au moment du traçage de la fonction. La boucle sera « déroulée » en un ensemble d'opérations pour chaque itération, comme nous l'avons vu dans l'exemple.
- Si vous utilisez `tf.range()`, la boucle sera dynamique et sera incluse dans le graphe lui-même (mais ne s'exécutera pas au cours du traçage).

Examinons le graphe produit en remplaçant `range()` par `tf.range()` dans la fonction `add_10()` :

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

Vous le constatez, le graphe contient à présent une opération de boucle `While`, comme si vous aviez appelé la fonction `tf.while_loop()`.

GÉRER DES VARIABLES ET D'AUTRES RESSOURCES DANS DES FONCTIONS TF

Dans TensorFlow, les variables et les autres objets avec état, comme les files d'attente ou les datasets, sont des *ressources*. Les fonctions TF leur prêtent une attention particulière. Toute opération qui met à jour une ressource est considérée comme ayant un état, et les fonctions TF s'assurent que les opérations avec état sont exécutées dans leur ordre d'apparition (*a contrario* des opérations sans état, qui peuvent être exécutées en parallèle et dont l'ordre d'exécution n'est pas garanti). De plus, toute ressource passée en argument d'une fonction TF est passée par référence : la fonction peut donc la modifier. Par exemple :

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)
```

³³¹ Voir « 12_custom_models_and_training_with_tensorflow.ipynb » sur <https://github.com/ageron/handson-ml2>.

```
increment(counter) # counter vaut à présent 1
increment(counter) # counter vaut à présent 2
```

Si vous examinez la définition de fonction, le premier argument est marqué comme étant une ressource :

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

Il est également possible d'utiliser un `tf.Variable` défini en dehors de la fonction, sans le passer explicitement comme un argument :

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

La fonction TF le traitera comme un premier argument implicite et aura la même signature (à l'exception du nom de l'argument). Cependant, l'utilisation de variables globales pouvant rapidement devenir compliquée, il est préférable d'envelopper les variables (et les autres ressources) dans des classes. Bonne nouvelle, `@tf.function` fonctionne parfaitement avec les deux méthodes :

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)
```



Avec les variables TF, n'utilisez pas `=`, `+=`, `-=`, ni tout autre opérateur d'affectation Python. À la place, vous devez invoquer les méthodes `assign()`, `assign_add()` et `assign_sub()`. Toute tentative d'utilisation d'un opérateur d'affectation Python conduit à la levée d'une exception au moment de l'appel de la méthode.

Un bon exemple de cette approche orientée objet est, évidemment, `tf.keras`. Voyons comment utiliser des fonctions TF avec `tf.keras`.

UTILISER DES FONCTIONS TF AVEC TF.KERAS (OU NON)

Par défaut, chaque fonction, couche ou modèle personnalisé que vous utilisez avec `tf.keras` sera automatiquement converti en fonction TF. Vous n'avez rien à faire ! Cependant, dans certains cas, vous préférerez désactiver cette conversion automatique, par exemple, si votre code personnalisé ne peut pas être converti en fonction TF ou si vous souhaitez simplement le déboguer, ce qui est beaucoup plus facile en mode pressé. Pour cela, il suffit de préciser `dynamic=True` lors de la création du modèle ou de l'une de ses couches :

```
model = MyModel(dynamic=True)
```

Si votre couche ou modèle personnalisé doit toujours être dynamique, vous pouvez à la place invoquer le constructeur de la classe de base avec `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Vous pouvez également indiquer `run_eagerly=True` lors de l'appel à la méthode `compile()`:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
               run_eagerly=True)
```

Vous savez à présent comment les fonctions TF prennent en charge le polymorphisme (avec plusieurs fonctions concrètes), comment les graphes sont générés automatiquement avec AutoGraph et le traçage, ce à quoi ils ressemblent, comment explorer leurs opérations et tenseurs symboliques, comment prendre en charge les variables et les ressources, et comment utiliser des fonctions TF avec tf.keras.

Index

A

A2C (Advantage Actor-Critic) 422
A3C (Asynchronous Advantage Actor-Critic) 422
accuracy 6
acteur-critique 422
acteur-critique à avantages 422
acteur-critique asynchrone à avantages 422
acteur-critique soft 422
AdaGrad 124
Adam 126
AdaMax 127
ajout d'une marge de zéros 216
AlexNet 231
algorithmes de gradients de politique 382
algorithmes génétiques 375
algorithmes REINFORCE 382
Anaconda 3
analyse de sentiments 301
analyse d'opinions 301
apprentissage antagoniste 262
apprentissage hebbien 56
apprentissage non supervisé 7
apprentissage par différence temporelle 390
apprentissage par renforcement 371

apprentissage Q 391
apprentissage Q par approximation 394
apprentissage Q profond 394
apprentissage résiduel 238
apprentissage supervisé 6
architecture Transformer 319
arrêt précoce 37
article BERT 328
article ELMo 327
article GPT 327
article GPT-2 328
article ULMFiT 327
attention à têtes multiples 323
attention de Bahdanau 316
attention de Luong 316
attention multiplicative 316
attention par concaténation 316
attention visuelle 318
augmentation des données 233
auto-attention 321
autodiff 59
autoencodeurs 119, 331, 332
autoencodeurs convolutifs 343
autoencodeurs débruiteurs 345
autoencodeurs empilés 336
autoencodeurs épars 346

autoencodeurs génératifs 350
 autoencodeurs probabilistes 350
 autoencodeurs profonds 336
 autoencodeurs récurrents 344
 autoencodeurs sous-complets 334
 autoencodeurs variationnels 349
 AutoGraph 176
 average pooling 226

B

bag of words 206
 biais 29
 bias 8
 BN (Batch Normalization) 109
 boucles d'entraînement personnalisées 172
 BPTT (backpropagation through time) 270

C

calcul automatique des gradients 168
 calculs logiques 53
 California Housing 67
 carte autoadaptative 525
 carte de caractéristiques 218
 cellule de longue mémoire à court terme 282
 cellule de mémoire 268
 cellule d'unité récurrente à porte 286
 cellule GRU 286
 cellule LSTM 282
 CGAN (Conditional GAN) 363
 chaînes de caractères 529
 chaînes de Markov 386
 Char-RNN 292
 cible 6
 classificateur d'images 66, 247
 classification 6, 250
 CNN (Convolutional Neural Network) 213, 228
 codages 331
 coefficients de pondération 8
 Colab 452
 Colaboratory 452
 complexité algorithmique 13
 composants personnalisés 155
 conda 3
 connexions de raccourci 238
 connexions de saut 238
 connexions judas 285
 contraintes personnalisées 157
 convolution 215

couche BN 109
 couche d'attention 316
 couche d'écart-type par mini-lot 365
 couche de concaténation en profondeur 235
 couche de convolution 215
 couche de convolution séparable 241
 couche de normalisation par pixel 366
 couche dense 55
 couche de pooling 224
 couche de pooling maximum 224
 couche de pooling moyen 226
 couche intégralement connectée 55
 couche Multi-Head Attention 323, 326
 couche récurrente bidirectionnelle 312
 couche Scaled Dot-Product Attention 323
 couches de prétraitement 205
 couches personnalisées 161
 courbes d'apprentissage 27
 CPU 147
 credit assignment problem 380
 CUDA 449
 cuDNN 449
 Curiosity-Based Exploration 423
 1cycle 131

D

dataset imbriqué 295
 dataset plat 295
 DBN (Deep Belief Net) 523
 DCGAN (Deep Convolutional GAN) 360
 DDQN (Dueling DQN) 401
 décodeur 269, 308, 333
 décomposition en valeurs singulières 13
 Deep Q-Learning 394
 déplier le réseau dans le temps 266
 depthwise separable convolution layer 241
 deque 395
 descente de gradient 14
 descente de gradient ordinaire 17
 descente de gradient par mini-lots 23
 descente de gradient stochastique 20
 détection d'anomalies 7
 détection d'objets 252
 différenciation 274
 différenciation automatique 59, 168, 511
 discount rate 380
 discriminateur 355
 discrimination par mini-lots 360
 disparition des gradients 102

divergence de Kullback-Leibler 45
 DNN (Deep Neural Network) 58
 données d'entraînement 6
 DQN (Deep Q-Network) 394
 DQN double 400
 dropout 135
 dropout rate 136
 duel de DQN 401

E

early stopping 37
 échantillon 6
 échantillonnage préférentiel 401
 échéancier d'apprentissage 21, 130
 écrêtage de gradient 115
 elastic net 36
 ELMo (Embeddings from Language Models) 327
 ELU (Exponential Linear Unit) 106
 embedding 201
 enchaîner des transformations 183
 encodage one-hot 199
 encodeur 269, 308, 333
 encodeur-décodeur 269
 ensembles 154, 533
 entraînement autosupervisé 121
 entraînement en parallèle 460
 entraîner un autoencodeur à la fois 342
 entraîner un modèle 10
 entropie croisée 44, 45
 EOS (end-of-sequence) 308
 epoch 21
 époque 21
 équation d'optimalité de Bellman 388
 équation normale 11
 équilibre de Nash 359
 erreur irréductible 30
 espace de paramètres 17
 étiquette 6
 exactitude 6, 30
 exécution en parallèle 458
 experience replay 360
 explicabilité 318
 exploration fondée sur la curiosité 423
 explosion des gradients 102

F

Fashion MNIST 67
 Faster R-CNN 262

FCN (réseaux entièrement convolutifs) 253
 feature map 218
 files d'attente 154, 534
 filtres 217
 fonction d'activation 61
 fonction d'activation personnalisée 157
 fonction de coût 10, 39
 fonction de Heaviside 54
 fonction de perte personnalisée 154
 fonction échelon 54
 fonction logistique 39
 fonction logit 39
 fonction TensorFlow 175
 forget gate 283
 format TFRecord 192
 frontières de décision 41

G

GAN (Generative Adversarial Network) 119, 331, 332, 355
 GAN conditionnel 363
 GAN convolutif profond 360
 générateur 355
 GoogLeNet 234
 GPU 147, 448
 gradient accéléré de Nesterov 123
 gradient clipping 115
 gradient de politique 375
 graphes TensorFlow 175, 537
 greedy layer-wise pretraining 120
 greedy layer-wise training 342
 GRU (Gated Recurrent Unit) 286

H

HANDSON-ML2 2
 hyperas 91
 hyperband 92
 hyperopt 91
 hyperparameter tuning 14
 hyperparamètre 14

I

ImageNet 111
 IMDb reviews 301
 Inception-v4 240
 indicateur à états 159
 indicateur de référence 272
 indicateur en continu 159

indicateur personnalisé 158
 indice de Jaccard 251
 inertie 122
 initialisation aléatoire 14
 initialisation de Glorot 104
 initialisation de He 104
 initialisation de LeCun 104
 initialiseurs personnalisés 157
 input gate 283
 instabilité des gradients 102, 279
 instance 6
 intercept 8
 interpolation sémantique 354
 IoU (Intersection over Union) 251
 IS (Importance Sampling) 401
 itération sur la valeur 388

J

jeu d'entraînement 6
 jeu de validation 14
 Jupyter 4

K

Keras 65
 Keras Tuner 92
 K-fold cross-validation 89

L

ℓ_1 31, 33
 ℓ_2 31
 label 6
 largeur du faisceau 313
 leaky ReLU 105
 learning schedule 21, 130
 LeNet-5 230
 localisation 250
 logarithme de cote 39
 logit 39
 log loss 40
 log-odds 39
 LRN (Local Response Normalization) 232
 LSTM (Long Short-Term Memory) 282
 LTU (unité linéaire à seuil) 54

M

Machine Learning 6
 machines à vecteurs de support 50

machines de Boltzmann 520
 machines de Boltzmann profondes 523
 machines de Boltzmann restreintes 120, 522
 mAP (mean Average Precision) 257
 Mask R-CNN 262
 masquage 305
 max pooling 224
 MC Dropout 279
 MDP (Markov decision processes) 387
 mécanismes d'attention 314, 317
 mélanger les données 184
 mini-batch discrimination 360
 mini-batch gradient descent 23
 minimum global 16
 minimum local 16
 MLM (Masked Language Model) 328
 MNIST 67
 mode collapse 359
 modèle creux 34, 129
 modèle d'alignement 316
 modèle de régression linéaire 10
 modèles ARIMA (autoregressive integrated moving average) 274
 modèles de moyenne mobile pondérée 274
 modèles personnalisés 164
 modèles préentraînés 246, 247
 momentum 122
 Momentum Optimization 122
 Monte Carlo (MC) Dropout 138
 moyenne de la précision moyenne 257
 MPC de régression 77
 MSE 10

N

Nadam 128
 NAG (Nesterov Accelerated Gradient) 123
 neurone biologique 51
 neurone de biais 55
 neurone d'entrée 55
 neurone de terme constant 55
 NMT (Neural Machine Translation) 308
 nombre de couches cachées 93
 nombre de neurones par couche cachée 94
 normalisation de réponse locale 232
 normalisation par couches 280
 normalisation par lots 109
 norme de Manhattan 31
 normes ℓ_k 31
 notebooks Jupyter 4

noyau de pooling 224
NSP (Next Sentence Prediction) 329

O

observation d'entraînement 6
off-policy 392
on-policy 393
opérations de convolution 260
optimisation avec inertie 122
optimisation avec inertie de Nesterov 123
optimisation de politique proximale 423
optimiseur 95
output gate 284
OvA 45
OvR 45

P

parallelisme des données 462
parallelisme du modèle 460
partitionnement 7
pas 14
peephole connections 285
PER (Prioritized Experience Replay) 401
perceptron 54
perceptron multicouche *Voir* PMC
performance 6
personnalisation 154
perte de reconstruction 334
perte logistique 40
PG (Policy Gradients) 382
planification 1cycle 131
planification à la performance 131
planification du taux d'apprentissage 129
planification par exponentielle 131
planification par puissance 130
plateforme GCP AI 437
plongements 201, 304, 308
plongements de mots 202
plongements positionnels 321
plongements préentraînés 307
PMC (perceptron multicouche) 57-58
PMC de classification 63
PMC de régression 62
poids 8
policy 374
politique 374, 378
politique en ligne 393
politique hors ligne 392
pooling 224

porte d'entrée 283
porte de sortie 284
porte d'oubli 283
positional embedding 321
PPO (Proximal Policy Optimization) 423
précision 30
préentraînement à partir d'une tâche seconde 121
préentraînement glouton par couche 120
préentraînement non supervisé 119, 340
prélecture 189
PReLU (Parametric leaky ReLU) 106
prétraiter les données 187
prévision de plusieurs étapes temporelles à l'avance 275
problème d'affectation de crédit 380
problème de mémoire à court terme 282
processus de décision markoviens 387
protobufs de TensorFlow 194
Protocol Buffers 193
pseudo-inverse de Moore-Penrose 13

Q

Q-Learning 391
quantification post-entraînement 445
Q-Value Iteration 388

R

ragged tensor 530
rappels 30, 84
RBM (Restricted Boltzmann Machines) 120, 522
recall 30
recherche en faisceau 313
récompenses 372
reconstructions 334
rectangle d'encadrement 250, 251
recuit simulé 21
réduction de la dimensionnalité 7, 331
régression 7
régression lasso 33
régression logistique 38
régression logistique multinomiale 43
régression logit 38
régression polynomiale 24
régression ridge 30
régression softmax 43
régularisation 8, 134
régularisation de Tikhonov 30

régularisation ℓ_1 135
 régularisation ℓ_2 135
 régularisation max-norm 141
 régulariseurs personnalisés 157
 rejet d'expériences 360
 rejet d'expériences à priorités 401
 ReLU 61
 remplissage par zéros 216
 rendement 380
 représentations latentes 331
 réseau antagoniste génératif 119
 réseau de correspondance 366
 réseau de génération 333
 réseau de Hopfield 519
 réseau de neurones convolutif 213
 réseau de neurones Wide & Deep 78
 réseau de reconnaissance 333
 réseau de synthèse 367
 réseau Q profond 394
 réseau série-vers-série 269
 réseau série-vers-vecteur 269
 réseau vecteur-vers-série 269
 residual learning 238
 ResNet 237
 ResNet-34 244
 rétropropagation dans le temps 270
 rétropropagation tronquée dans le temps 295
 return 380
 RL (Reinforcement Learning) 371
 RMSE 10
 RMSProp 126
 RNA (réseaux de neurones artificiels) 49
 RNP (réseaux de neurones profonds) 58
 RNR (réseaux de neurones récurrents) 265
 RNR à caractères 292
 RNR avec état 292, 299
 RNR bidirectionnels 312
 RNR sans état 292
 RReLU (Randomized leaky ReLU) 106

S

sac de mots 206
 SAC (Soft Actor-Critic) 422
 saisonnalité 274
 SavedModel 429
 Scikit-Optimize 92
 segmentation sémantique 259
 SELU (Scaled ELU) 107
 SENet 242

séries chronologiques 271
 séries chronologiques multivariées 271
 séries chronologiques univariées 271
 séries longues 279
 simulated annealing 21
 skip connections 238
 Sklearn-Deap 92
 softplus 62
 SOM (Self-Organizing Map) 525
 SOS (start-of-sequence) 308
 sous-ajustement 8
 sparse tensors 531
 Spearmint 92
 stateful metric 159
 step 54
 streaming metric 159
 StyleGAN 366
 super-résolution 261
 surajustement 8, 134
 SVD 13
 SVM 50

T

tableaux de tenseurs 153, 532
 taille des lots 95
 TALN (traitement automatique du langage naturel) 121, 265
 tangente hyperbolique 60
 target 6
 taux d'apprentissage 14, 15, 95
 taux d'apprentissage adaptatif 125
 taux d'apprentissage constant par morceaux 131
 taux d'apprentissage égalisé 365
 taux de dilatation 288
 taux de rabais 380
 taux d'extinction 136
 TD (Temporal Difference Learning) 390
 tendance 274
 tenseurs 149
 tenseurs chaînes de caractères 153
 tenseurs creux 153, 531
 tenseurs de masque 305
 tenseurs irréguliers 153, 530
 TensorBoard 85
 TensorFlow 3, 146
 TensorFlow 2 66
 TensorFlow Addons 311
 TensorFlow dans le navigateur 447

TensorFlow Datasets 208
TensorFlow Serving 429
terme constant 8
test de Turing 291
TF-Agents 402
TF Transform 207
TLU (unité logique à seuil) 54
tolérance 20
TPU 147
traduction automatique neuronale 315
traduction automatique neuronale simple 308
training set 6
transfer learning 116
transfert d'apprentissage 116
truncated backpropagation through time 295

V

valeur d'état optimale 387
valeurs Q 388
validation croisée 27
validation croisée en K passes 89
variance 29

vecteur de pondération 31, 33
vecteur de sous-gradient 36
vecteur des paramètres 9
vecteur des prédictions 10
vecteur des valeurs 9
vecteur des valeurs cibles 11
VGGNet 237

W

WaveNet 288
wrappers d'environnement 406

X

Xception 241

Y

YOLO (You Only Look Once) 255

Z

zero padding 216
ZSL (Zero-Shot Learning) 328