Deep Learning avec Keras et TensorFlow

Chez le même éditeur

Big Data et Machine Learning
3e édition
Pirmin Lemberger, Marc Batty, Médéric Morel, Jean-Luc Raffaëlli
272 pages
Dunod, 2019

Introduction au Machine Learning
Chloé-Agathe Azencott
240 pages
Dunod, 2018

Python pour le data scientist
Emmanuel Jakobowicz
304 pages
Dunod, 2018

Le traitement automatique des langues
François-Régis Chaumartin, Pirmin Lemberger
320 pages
Dunod, 2020

O'REILLY®

Deep Learning avec Keras et TensorFlow

Mise en œuvre et cas concrets

2e 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

Conception de la couverture : Karen Montgomery
Illustratrice : Rebecca Demarest

© Dunod, 2017, 2020
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-079066-1

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 sillonneraient 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 Moore[1], 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 1 000 € une carte graphique équipée d'un processeur GPU capable de réaliser des milliers de milliards de calculs par seconde[2]. 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 *connexionnisme*, 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 éparpillé, 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 Python[3]), 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-Learn[4]. 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 (2e 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 renumérotés 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 rencontrerait 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 livre[5] : 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 Simsa, 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://homl.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, 2e 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 suite[6].

Nous commencerons par installer TensorFlow et les autres librairies 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 attaquerons 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* (2e 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* (2e 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.

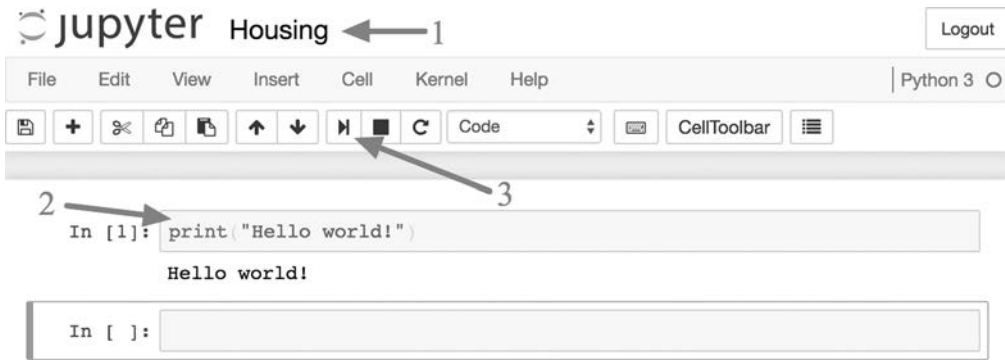

Figure 1.1 – Afficher « Hello world! » dans un notebook Jupyter

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.

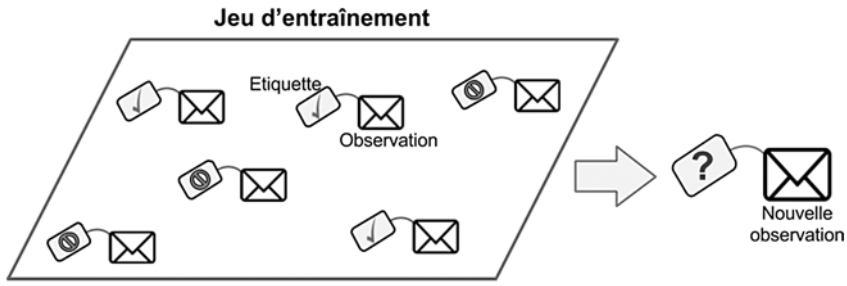

Figure 1.2 – Jeu d'entraînement étiqueté pour une tâche de classification (détection de spam)

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 dimensionalité, dont ceux dédiés à la visualisation des données, sont aussi des exemples d'algorithmes d'apprentissage non supervisé.

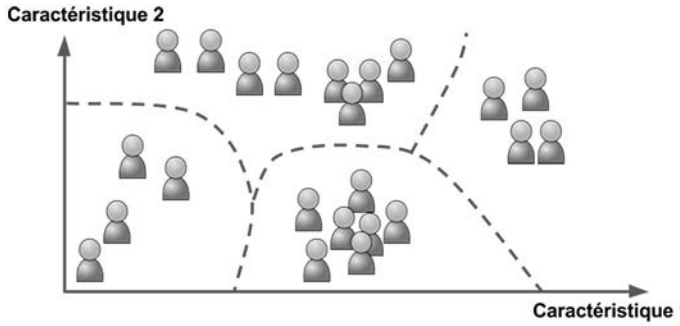

Figure 1.3 – Le partitionnement, un exemple d'apprentissage non supervisé

Résumons : on distingue les tâches d'apprentissage supervisé (classification, régression…), et les tâches d'apprentissage non supervisé (partitionnement, détection d'anomalie, réduction de dimensionalité...). 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 prédite 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

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

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

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

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

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

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

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

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 $\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n$, et que l'on notera dans ce livre $\boldsymbol{\theta}^T \mathbf{x}$.[8]
- $h_{\boldsymbol{\theta}}$ 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 prédiction 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 $i^{\text{ième}}$ élément correspond à la prédiction du modèle pour la $i^{\text{ième}}$ 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)

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

- 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 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 $\boldsymbol{\theta}$ 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

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

Dans cette équation :

- $\hat{\boldsymbol{\theta}}$ est la valeur de $\boldsymbol{\theta}$ 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)
```

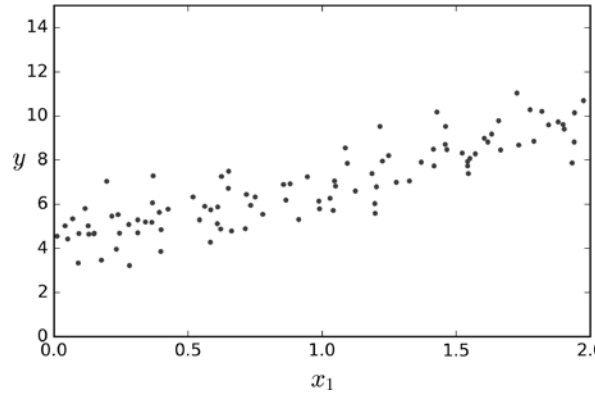

Figure 1.4 – Jeu de données généré aléatoirement

Calculons maintenant $\hat{\boldsymbol{\theta}}$ 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 matriciels[9] :

```
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 $y = 4 + 3x_1 +$ 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 $\hat{\boldsymbol{\theta}}$:

```
>>> 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()
```

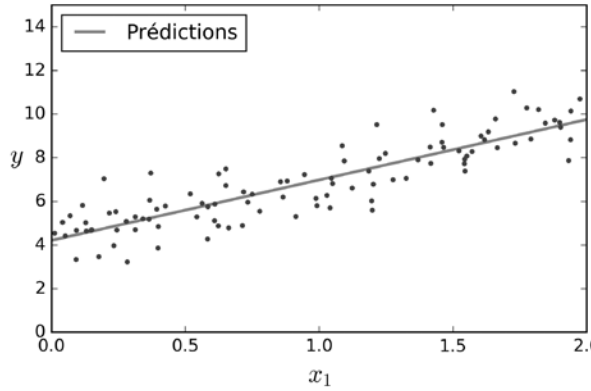

Figure 1.5 – Prédictions du modèle de régression linéaire

Effectuer une régression linéaire avec Scikit-Learn est très simple[10] :

```
>>> 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 $\hat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, où $\mathbf{X}^{+}$ est le pseudo-inverse de $\mathbf{X}$ (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 $\mathbf{X}$ en produit de trois matrices $\mathbf{U}$, $\boldsymbol{\Sigma}$ et $\mathbf{V}^{T}$ (voir `numpy.linalg.svd()`). Le pseudo-inverse se calcule ensuite ainsi : $\mathbf{X}^{+}=\mathbf{V}\boldsymbol{\Sigma}^{+}\mathbf{U}^{T}$. Pour calculer la matrice $\boldsymbol{\Sigma}^{+}$, l'algorithme prend $\boldsymbol{\Sigma}$ 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 $\mathbf{X}^{T}\mathbf{X}$ 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 $\mathbf{X}^{T}\mathbf{X}$, 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 grosso modo 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 $\boldsymbol{\theta}$, puis progresse en direction du gradient descendant. Lorsque le gradient est nul, vous avez atteint un minimum !

En pratique, vous commencez par remplir $\boldsymbol{\theta}$ 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).