

Hugues Bersini
Ken Hasselmann

L'intelligence artificielle en pratique avec Python

Recherche, optimisation, apprentissage

2^e édition



● Éditions
EYROLLES

Un livre à la fois théorique et pratique

Cet ouvrage à vocation pédagogique a pour but d'aider les débutants et même les praticiens confirmés de l'intelligence artificielle à mieux faire le tri entre certains mécanismes algorithmiques propres à cette discipline et souvent confondus dont les trois fondamentaux : « la recherche », « l'optimisation » et « l'apprentissage ».

Même si le Web regorge de solutions algorithmiques et de codes clés en main mis à disposition des internautes, ces codes constituent rarement la bonne solution pour faire face à un problème. En effet, il faut souvent prendre du recul, et c'est précisément ce que propose cet ouvrage, pour pouvoir trancher entre les différentes offres algorithmiques (les trois fondamentaux) et choisir celle qui sera la plus appropriée au cas de figure que l'on rencontre.

Dix problèmes très classiques de l'univers algorithmique et de l'IA sont abordés dans la 2^e édition ce livre. Pour chacun, nous allons détailler l'une ou l'autre méthode issue d'un des trois mécanismes fondamentaux (recherche, optimisation ou apprentissage) :

- le jeu du taquin ;
- l'algorithme du plus court chemin (celui qu'on trouve dans les GPS) ;
- le jeu du sudoku ;
- le jeu de Puissance 4 à deux joueurs ;
- le jeu du Tetris ; **Mis à jour**
- le jeu du Snake ;
- la séparation des spams et des non-spams ;
- les règles d'accès au crédit ; **Nouveau**
- les aides au tri de la presse ou des avis de clients ; **Nouveau**
- la reconnaissance sur photo de chiens ou de chats.

À qui s'adresse cet ouvrage ?

- Aux étudiants, en informatique ou pas, qui découvrent l'IA dans leur parcours académique
- Aux informaticiens, même les plus confirmés, qui se sentent de plus en plus déçus devant l'offre pléthorique des recettes d'IA dont ils n'arrivent pas toujours à comprendre « qui fait quoi »

Compléments web

Le code source des exemples du livre en Python est disponible sur le site d'accompagnement www.editions-eyrolles.com/dl/0101094

Au sommaire

Introduction • Jouons au taquin • Découvrir le plus court chemin • Jouons au sudoku • Jouons à Puissance 4 • Jouons au Snake • Jouons à Tetris • Comment reconnaître un spam • Découvrir les règles d'accès au crédit • Aider à trier la presse ou les avis de clients • Comment distinguer un chien d'un chat • Conclusion : les deux IA

Membre de l'Académie royale de Belgique, **Hugues Bersini** enseigne l'informatique et la programmation aux facultés polytechnique et Solvay de l'Université Libre de Bruxelles, dont il dirige le laboratoire d'intelligence artificielle. Il est l'auteur de très nombreuses publications (systèmes complexes, génie logiciel, sciences cognitives et bioinformatique) et de plusieurs ouvrages d'introduction à la programmation, l'intelligence artificielle et les systèmes complexes qui font aujourd'hui autorité dans le monde académique.

Ken Hasselmann est enseignant-chercheur à l'École Centrale des Arts et Métiers (ECAM) à Bruxelles dans le département d'informatique et d'électronique, et doctorant à l'université libre de Bruxelles au sein de l'Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle (IRIDIA). Il y étudie la conception automatique d'algorithmes dans les essaims de robots, et participe notamment au projet européen ERC DEMIURGE dans le cadre de ses recherches.

www.editions-eyrolles.com

IMPRIMÉ EN FRANCE

L'intelligence artificielle en pratique avec Python

CHEZ LE MÊME ÉDITEUR

SUR LE MÊME THÈME

Y. BENZAKI. – **Les data sciences en 100 questions/réponses.**

N°67951, 2020, 126 pages.

M-R. AMINI. ET AL. – **Data science – Cours et exercices.**

N°67410, 2018, 272 pages.

H. WICKHAM, G. GROLEMUND. – **R pour les data sciences.**

N°67571, 2018, 496 pages.

F. PROVOST, T. FAWCETT. – **Data science pour l'entreprise.**

N°67570, 2018, 384 pages.

E. BIERNAT, M. LUTZ. – **Data science : fondamentaux et études de cas.**

N°14243, 2015, 312 pages.

DANS LA MÊME COLLECTION

B. BAILLY. ET AL. – **Les réseaux 5G.**

N°67898, 2020, 580 pages.

K. NOVAK. – **Administration Linux par la pratique – Tome 2.**

N°67949, 2020, 418 pages.

C. DELANNOY. – **Le guide complet du langage C.**

N°67922, 2020, 876 pages.

J. LORIAUX, T. DEFOSSEZ. – **Emailing : développement et intégration.**

N°67849, 2020, 160 pages.

B. BARRÉ. – **Concevez des applications mobiles avec React Native.**

N°67889, 2^e édition, 2019, 224 pages.

S. RINGUEDÉ. – **SAS.**

N°67631, 4^e édition, 2019, 688 pages.

C. BLAESS. – **Développement système sous Linux.**

N°67760, 5^e édition, 2019, 1080 pages.

Retrouvez nos bundles (livres papier + e-book) et livres numériques sur
<http://izibook.eyrolles.com>

Hugues Bersini
Ken Hasselmann

L'intelligence artificielle en pratique avec Python

Recherche, optimisation, apprentissage

2^e édition

● Éditions
EYROLLES

ÉDITIONS EYROLLES

61, bd Saint-Germain

75240 Paris Cedex 05

www.editions-eyrolles.com

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.

© Éditions Eyrolles, 2022, 2023 pour la présente édition, ISBN : 978-2-416-01094-1

Avant-propos

Un peu de cinéma

Dans le troisième opus de la série *Die Hard*, le héros interprété par Bruce Willis doit résoudre un ensemble de problèmes logiques, chaque fois en une minute maximum, sinon un horrible méchant risque de faire exploser New York. L'un de ces problèmes implique deux cruches non graduées, de capacité cinq litres et trois litres, que notre héros peut remplir et vider à une fontaine à proximité. Il lui est demandé de remplir en moins d'une minute la cruche de cinq litres avec exactement quatre litres. C'est un problème idéal pour l'Intelligence Artificielle qui l'aurait, elle, résolu en quelques millisecondes. Bruce Willis anticipe plusieurs actions dans sa tête, même plusieurs séquences d'actions, avant de trouver la bonne solution deux secondes avant l'apocalypse. Il faut remplir la cruche de cinq litres avec celle de trois litres, il reste deux litres vide dans la première. On recommence la même opération avec la cruche de trois litres, il reste cette fois un litre dans la cruche de trois litres. Il suffit alors de vider la cruche de cinq litres, la remplir d'abord avec le reste de la cruche de trois litres, le litre restant, et recommencer l'opération pour obtenir les quatre litres dans la cruche de cinq. Pour l'IA, il suffit de décrire l'état du problème en tant que (x, y) , contenu des deux cruches, et coder chaque action possible sous la forme avant-après, par exemple $(0, 0) \rightarrow (5, 0)$ (remplissage de la première) ou $(x, y) \rightarrow (5, y - (5 - x))$, remplissage de la première avec une partie de la seconde. L'IA peut alors facilement enchaîner les actions et trouver la séquence optimale : $(0, 0)$ puis $(0, 3)$ puis $(3, 0)$ puis $(3, 3)$ puis $(5, 1)$ puis $(0, 1)$ puis $(1, 0)$ puis $(1, 3)$ puis $(4, 0)$. Bruce Willis, doté d'un **QI** bien supérieur à la moyenne, raisonne exactement comme l'IA d'avant, celle des années 1950-1960, celle qui a fait de notre raison la cible à abattre.

Autre film, autre ambiance. Dans ces scènes de cinéma que Steven Spielberg prend tant de plaisir à réaliser, les petits garçons n'ont jamais aucun mal à reconnaître leur maman, que ce soit le héros de *L'empire du soleil*, surtout à la fin du film, ou même le petit robot de *AI* (ça tombe bien)¹. Il leur suffit de percevoir, même de très loin, même dans l'obscurité, même dans une foule, surtout dans une foule, le visage de leur maman, pour que le leur s'illumine. Cette reconnaissance est comme une évidence, un réflexe de survie qui ne nécessite pas le moindre raisonnement, contrairement à celui extrait du film précédent. Il s'agit d'un tout autre type de processus cognitif, bien plus primitif, dont sont dotés les enfants autant que les animaux. Même les singes les plus proches de nous sont totalement inca-

pables de résoudre le problème de *Die Hard*, alors que la reconnaissance d'un visage de primate ou d'humain ne leur demande aucun effort, une dépense minimale de Watts neuronaux. Pour l'IA en revanche, il s'agit là d'un tout autre type de processus cognitifs à reproduire.

Alors que les premiers développements de l'IA cherchaient déjà à reproduire l'ensemble de tous les processus cognitifs humains, ces multiples intelligences, de celles qui nous permettent de raisonner, mais aussi voir, se mouvoir, entendre, parler, écrire, créer... les premières réussites logicielles tinrent surtout à la reproduction de la partie la plus cognitive de nos cerveaux : raisonnement, résolution de problème, maîtrise du langage naturel. On négligea la vision ou la robotique, car supposées beaucoup plus à portée que le jeu d'échec ou le résumé d'un texte (c'est d'une telle simplicité pour nous). Erreur de jugement fatale ! Il fallut attendre de nombreuses années pour que les idées déjà en ébauche aux origines de l'IA (comme les réseaux de neurones) finissent par porter leurs fruits et atteindre les résultats espérés, amenant cette autre tradition de l'IA (ancrée dans nos processus inconscients) à voler la vedette à la version d'origine (ancrée dans nos processus les plus conscients et les plus rationnels). Cette inversion de tendance incite certains chercheurs à aller jusqu'à déclarer que toute la tradition plus cognitive de l'IA, plus logique, plus séquentielle, basée sur la construction d'arbres de raisonnement, et exploitant les connaissances humaines pour simplifier la recherche, l'a conduite dans une impasse².

Dans ce livre, nous nous inscrivons en faux et, parmi nos missions premières, nous souhaitons réhabiliter ces premiers algorithmes de l'IA, de ceux que l'on retrouve abondamment dans les entreprises aujourd'hui, comme lorsque vous faites une recherche sur Google, jouez aux échecs, utilisez Waze ou Uber. Il est vrai que trois facteurs conjoncturels importants sont à l'origine du succès de ces nouveaux algorithmes d'IA bien davantage axés sur l'apprentissage : d'abord quelques innovations algorithmiques (de celles que l'on trouve par exemple dans les architectures plus récentes des réseaux de neurones), mais surtout une quantité extraordinaire de données, comme autant de témoignages du génie humain dont les algorithmes d'apprentissage peuvent s'inspirer même sans les comprendre, et finalement des processeurs informatiques qui se sont extraordinairement « musclés » en puissance de calcul, recourant notamment à de plus en plus de parallélisme (ainsi l'ascendance des GPU sur les CPU pour les réseaux de neurones profonds). Nous en parlerons également.

-
1. Alors qu'il cogitait à la réalisation de son film *Artificial Intelligence*, inspiré de la nouvelle *Les supertoys durent tout l'été* de Brian Aldiss, un des problèmes taraudant considérablement Stanley Kubrick, connu pour son perfectionnisme maladif, était l'apparence du petit garçon robot, David, le protagoniste du film. Lorsque Spielberg hérita de ce projet à la mort de Kubrick, il ne s'encombra pas de difficultés techniques outre mesure et choisit Haley Joel Osment, qui s'efforça, par la mécanique de sa gestuelle et son statisme, de jouer au robot. Dans le film, une famille en mal d'enfants se voit confier David, représentant d'une nouvelle génération robotique frôlant la perfection et qui, alors qu'il est programmé pour être capable d'un véritable amour filial pour ses parents, surtout sa mère, souffre de l'absence de réciprocité de cette dernière, qui ne parvient pas à s'affranchir de la nature siliconée de son petit. On retrouve le problème relationnel parents/enfants si cher à Spielberg, rendu plus aigu encore par la difficulté dans laquelle se trouve la mère. Elle peine à voir dans David autre chose qu'un toasteur, capable de quelques expressions touchantes mais sans plus, et cela malgré les effets appuyés de séduction déployés à tout venant par le petit robot.
 2. Afin de prendre position dans ce débat, nous recommandons la lecture de *Rebooting Artificial Intelligence we can trust* de Gary Marcus et Ernest Davis ou *Architects of intelligence, the truth about AI from the people building it* de Martin Ford.

Contenu et lectorat du livre

Plusieurs algorithmes d'IA, parmi les plus fameux et les plus usités, vont vous être illustrés, décorés et présentés dans ce livre. Il s'agit d'algorithmes de recherche (A^*), d'optimisation (algorithmes génétiques) et d'apprentissage (renforcement et classification, à partir de réseaux de neurones ou pas). Ils le seront à travers des applications aussi simples que populaires et très certainement connues par la plupart d'entre vous. Nous décrirons d'abord l'algorithme de résolution de manière conceptuelle, puis aborderons plus en détail certaines parties du code permettant à cet algorithme de s'exécuter pour les applications en question. Les codes seront accessibles dans leur intégralité sur un site web dédié et donc récupérables et exécutables en l'état. Nous nous intéresserons également à certaines techniques propres au traitement du langage naturel (comme l'algorithme LDA) et au traitement d'image (comme les filtres convolutifs), deux domaines applicatifs qui ont vécu des améliorations assez conséquentes ces vingt dernières années.

Nous avons fait le choix du langage de programmation Python pour l'écriture et la présentation des codes. Ce choix pourrait sembler paradoxal quand vous lirez souvent dans l'ouvrage qu'un des critères les plus importants dans le choix d'un algorithme s'avère le temps de calcul que son exécution réclame. En effet, Python étant un langage interprété, ayant privilégié par ailleurs la simplicité d'écriture sur la vitesse d'exécution, il n'est nullement recommandé pour celle-ci. C'est bien évidemment cette simplicité d'usage qui en fait depuis quelques années le langage de prédilection pour l'enseignement de l'informatique et de la programmation ; donc, assez logiquement, pour l'enseignement des algorithmes d'IA. Nous ne cherchons donc pas à déroger à cette tradition pédagogique, dont une des conséquences bénéfiques premières est l'extraordinaire offre logicielle sur le Web, notamment pour résoudre les mêmes applications que nous présenterons dans la suite de l'ouvrage. Lorsque nous comparons plusieurs algorithmes quant à leur vitesse d'exécution, nous passons évidemment totalement sous silence le langage de programmation utilisé. Libre à chacun d'entre vous, une fois comprise et assimilée la logique qui se cache derrière l'algorithme, de le transposer dans votre langage de programmation favori, afin de récupérer tout ce temps d'exécution perdu dont Python pourrait plaider coupable. Nous souhaitons aussi le plus souvent révéler la logique algorithmique (codée en Python) qui se dissimule derrière les performances des logiciels, afin de décourager le lecteur de se ruer sans discernement sur les bibliothèques qui implémentent l'algorithme et facilitent son usage, mais en en dissimulant ce cœur algorithmique. La compréhension profonde de ce dernier nous intéresse bien davantage que ses performances. C'est la pédagogie de l'IA qui nous importe, plus que son exploitation empressée.

Ce livre s'adresse à tous les étudiants, en informatique ou pas, qui découvrent l'IA dans leur parcours académique, mais aussi à tous les informaticiens, même les plus confirmés, qui se sentent de plus en plus décontenancés devant l'offre pléthorique des recettes d'IA dont ils n'arrivent pas toujours à comprendre « qui fait quoi ». Il est la conséquence de trente années d'enseignement de l'IA et des milliers d'étudiants que nous avons vu effectuer très souvent le mauvais choix algorithmique pour le problème qu'ils étaient censés résoudre. Parmi les raisons de ces errements : un manque de recul et de questionnement sur le problème, l'absence d'une connaissance plus en profondeur des recettes de l'IA (y compris les plus anciennes), le panurgisme et la facilité d'accès aux recettes les plus récentes et les plus à la mode, tel l'apprentissage profond. Les apprenants pourront, après

avoir assimilé la logique algorithmique, facilement télécharger et exécuter les codes Python, quitte à les remanier, les améliorer et les transposer pour un problème du même acabit.

Remerciements

Hugues Bersini souhaite vivement remercier son co-auteur, Ken Hasselmann, dont il a eu la grande chance de bénéficier de l'accompagnement toutes ces dernières années dans son enseignement de l'IA. Il souhaite aussi remercier tous les membres historiques et ceux toujours présents du laboratoire IRIDIA, qu'il dirige avec son ami et formidable chercheur Marco Dorigo. Il s'agit de centaines et de centaines de chercheurs qui ont fait la réputation d'un laboratoire devenu un incontournable de l'IA dans le monde entier. Il souhaite remercier les milliers d'étudiants qu'il espère avoir sensibilisés, même plus, passionnés à l'IA, surtout ces deux dernières années, que la crise de Covid a rendu nettement plus malaisées, tellement moins agréables pour eux. Finalement, c'est à toute l'équipe d'Eyrolles, avec laquelle il contribue depuis tant d'années et qui continue à le soutenir et l'accompagner avec grand professionnalisme, qu'il souhaite adresser sa plus sincère reconnaissance. Rien ne fait mieux ressortir des lignes de code que la page blanche sur laquelle elles s'écrivent. Et combien de fois lui a-t-il été donné de vivre un eureka salvateur, en feuilletant les pages d'un livre d'informatique ou de programmation, dans un métro, un train, à l'arrière d'une voiture ou faiblement éclairé par une lampe de chevet la tête posée sur l'oreiller. Le livre survivra, même dans sa version papier, surtout dans sa version papier, grâce aux éditeurs qui le défendent avec tant de passion.

Ken Hasselmann tient aussi à remercier Hugues Bersini, pour sa confiance et pour lui avoir offert l'opportunité de participer à l'élaboration de cet ouvrage, et toute l'équipe d'Eyrolles pour l'avoir rendu possible. Il tient à vivement remercier toute l'équipe d'IRIDIA pour l'ambiance chaleureuse et familiale qui règne au sein du laboratoire ! Pour finir, il souhaite remercier ses amis et sa famille pour leur soutien indéfectible et leur amour inconditionnel.

Table des matières

Introduction	1
Installation de Python	5
Installation de poetry	6
Sous Windows	6
Sous Linux	7
Fichiers des différents projets	7
Installer et utiliser les projets	8
 CHAPITRE 1	
Jouons au taquin	9
Principe – Choix algorithmique	9
Recherche : A*	11
Le logiciel Python du taquin (8Puzzle) en pratique	14
Recherche A*	15
Apprentissage par renforcement : le Q-learning	16
Application au taquin	19
L'algorithme d'optimisation génétique	22
 CHAPITRE 2	
Découvrir le plus court chemin	27
Principe – Choix algorithmique	28
Recherche du plus court chemin	28
Problème du voyageur de commerce	31
Le logiciel Python de la recherche du plus court chemin en pratique	33
 CHAPITRE 3	
Jouons au sudoku	37
Principe – Choix algorithmique	38
Le logiciel Python du sudoku en pratique	42
Recherche best-first	42
Algorithme génétique	43

CHAPITRE 4

Jouons à Puissance 4 47

Principe – Choix algorithmique	47
Le logiciel Python de Puissance 4 en pratique	53
Min-Max	53
MCTS	56

CHAPITRE 5

Jouons au Snake 61

Principe – Choix algorithmique	61
Réseaux de neurones artificiels	64
Retour au Snake	68
Le logiciel Python du Snake en pratique	70
Recherche A*	70
Réseau de neurones et algorithme génétique	73

CHAPITRE 6

Jouons à Tetris 83

Principe – Choix algorithmique	84
L'apprentissage par renforcement pour le Tetris	85
Le logiciel Python du Tetris en pratique	89
Apprentissage par renforcement neuronal	89
Algorithme génétique	94

CHAPITRE 7

Comment reconnaître un spam..... 101

Principe – Choix algorithmique	102
Le logiciel Python du détecteur de spams en pratique	105
Classificateur naïf de Bayes	106

CHAPITRE 8

Découvrir les règles d'accès au crédit..... 109

Principe – Choix algorithmique	109
Démarrons par un petit récit personnel	109
Et l'apprentissage vient à la rescousse	111
Comment fonctionnent les arbres de décision ?	112
De l'arbre à la forêt	117
De l'éthique de l'apprentissage machine	117
L'arbre de décision en pratique	119
Construire un arbre de décision en utilisant sklearn	124

CHAPITRE 9

Aider à trier la presse ou les avis de clients 127

Principe – Choix algorithmique	127
Supervisé et non supervisé	127
L'algorithme LDA (Latent Dirichlet Allocation)	129
Word2Vec et Doc2Vec – Catégoriser le sens profond et non plus les mots	131
Le logiciel Python de la catégorisation de documents en pratique	132
LDA	132
Représenter les documents par un vecteur de « sens »	136

CHAPITRE 10

Comment distinguer un chien d'un chat 139

Principe – Choix algorithmique	140
Le logiciel Python de la reconnaissance chien/chat en pratique	147
Apprentissage profond	147

Conclusion : les deux IA 151**Index 159**

Introduction

Cet écrit à vocation essentiellement pédagogique a pour but d'aider les débutants et même les praticiens confirmés de l'Intelligence Artificielle à mieux faire le tri entre certains mécanismes algorithmiques propres à cette discipline et souvent confondus, dont les trois fondamentaux que sont « la recherche », « l'optimisation » et « l'apprentissage ». Le besoin d'un tel écrit s'est fait sentir lorsque l'un des auteurs, qui enseigne cette discipline informatique depuis plus de trente ans, a constaté de manière accrue la grande confusion régnant dans l'esprit de ses étudiants lorsque ces derniers choisissent de programmer ce qui leur semble l'algorithme le plus désigné et prometteur pour résoudre certains problèmes :

- réussir le jeu du taquin ou des problèmes de sudoku ;
- affronter des joueurs humains au Puissance 4, aux dames ou aux échecs ;
- jouer aux vétérans Tetris ou Snake, ou pour contrôler Super Mario dans le jeu vidéo du même nom ;
- trouver le chemin le plus court dans un graphe ou le parcours le plus rapide traversant toutes les villes pour un voyageur de commerce ;
- séparer les spams des non-spams et, au-delà, organiser un ensemble de textes selon les thématiques qu'ils couvrent ;
- découvrir les règles qui distinguent un débiteur fiable de son contraire et reconnaître soit un chien soit un chat sur une photo qu'on lui présente.

De fait, les trois mécanismes les plus souvent confondus ou hybridés sans grand discernement afin de s'attaquer à ces problèmes sont effectivement « la recherche », « l'optimisation » et « l'apprentissage ». C'est aussi un fait certain que la sémantique n'aide pas, surtout lorsqu'on apprend à chercher à optimiser, que l'on optimise son apprentissage ou que l'on cherche à apprendre...

De la même manière, le Web regorge de solutions algorithmiques et de codes mis à disposition pour toujours ces mêmes types de problèmes (beaucoup d'enseignants en Intelligence Artificielle y ont recours pour leurs vertus pédagogiques). Une manière d'aborder et de résoudre le problème est exposée, le code est téléchargeable, de nombreuses bibliothèques disponibles sont prêtes à l'emploi, mais ce mode de résolution est rarement justifié et, surtout, peu comparé avec d'autres possibles. Et en effet, au risque de nous répéter, pour chaque problème, il existe de multiples manières algorithmiques

miques de s'y attaquer. Ce qui manque donc, et c'est la raison de cet ouvrage, c'est le recul, la prise de distance nécessaire pour, entre ces différentes offres algorithmiques, parvenir à trancher. Les sciences cognitives nous enseignent, que, avant de bien résoudre un problème, il faut d'abord le comprendre au mieux, c'est-à-dire concevoir le modèle mental adéquat d'où découlera naturellement la voie algorithmique la plus prometteuse. En effet, plusieurs raisons expliquent la confusion.

- 1 Il peut sembler au premier abord possible de programmer un algorithme dérivé indifféremment de l'un de ces trois mécanismes fondamentaux et qui, de fait, réussit à résoudre le problème après un temps d'exécution donné. Prenons, par exemple, le problème du taquin, le premier cas d'école que nous approfondirons par la suite. Composé d'un certain nombre de carreaux, il consiste pour le joueur à déplacer un certain nombre de fois un carreau vide dans les quatre directions cardinales possibles afin que tous les carreaux, au départ dans une configuration quelconque, se retrouvent placés de façon ordonnée. Cette séquence optimale de coups, idéalement la plus courte, peut être découverte à la fois par l'algorithme de recherche A^* (le vétéran de tous les algorithmes d'IA, mais sans une ride), par une méthode d'optimisation appelée algorithme génétique ou par un apprentissage par renforcement de type Q-learning (très à la mode aujourd'hui et plus encore dans sa version « Deep »). Dès lors, lequel choisir ?

Nous en discuterons dans le prochain chapitre et nous verrons qu'il est indispensable pour un praticien de l'IA de réussir à différencier ces algorithmes concurrents selon divers critères :

- certains propres à l'ordinateur censé les exécuter : temps de calcul, capacité d'occupation de la mémoire vive ;
- d'autres propres à la tâche du programmeur : complexité de l'algorithme et de sa programmation, sa décomposition facile, connaissance a priori de certaines voies résolutive déjà bien connues par les experts humains et possibilité de traduire cette expertise sous une forme algorithmique directement exploitable. Pourquoi ne pas faciliter la vie de l'algorithme (et surtout économiser ses ressources énergétiques) en partant de toute cette compétence que l'homme a acquise pour ce problème ? Pourquoi l'algorithme doit-il réinventer la roue ? Le cerveau humain doit-il jeter toutes ses armes au pied de la CPU machine ?

Comme toute situation de décision multicritère, le choix de l'algorithme adéquat pour le problème en question peut dériver de la priorité donnée à un critère plutôt qu'un autre ou d'un compromis judicieux entre ces différents critères.

- 2 Une autre raison importante est l'existence de multiples possibilités de mélanges entre ces différents mécanismes, transformant l'IA en une espèce de boîte à outils ou de couteau suisse, desquels l'ingénieur sort l'un ou l'autre de ces outils et bricole, en les hybridant plus ou moins maladroitement (les assemblages sont multiples). Citons quelques exemples, toujours pour le taquin. L'algorithme A^* requiert pour fonctionner une heuristique permettant d'évaluer et privilégier, parmi plusieurs configurations intermédiaires du jeu, laquelle est la plus prometteuse sur le chemin de la solution (i.e. à partir de laquelle on disposera les carreaux dans le bon ordre en le moins de coups possible). Or, il n'est pas toujours aisé de connaître une telle heuristique. Plusieurs voies peuvent être possibles, les

meilleures permettant d'atteindre la solution en minimisant le temps de calcul pour un processeur qui ne se perd plus dans des chemins de traverse. Elles peuvent dès lors être apprises en les tentant et en les testant et en espérant que, une fois la meilleure mise à jour, elle se généralisera à toutes les situations de taquin, au-delà de celles qui ont justement servi à la découvrir. Ce qu'un apprentissage doit toujours réussir (on le comprendra mieux plus tard), c'est généraliser ce que l'on a trouvé à partir de quelques situations types à toutes les nouvelles situations qui n'ont pas servi pour le trouver mais qui lui ressemblent.

Nous voyons bien là un mélange entre un algorithme de recherche et un autre d'apprentissage, permettant au premier de fonctionner plus efficacement ou plus rapidement encore. Un autre exemple de mélange bien connu est le réseau de neurones : un algorithme d'IA vieux de plus de soixante ans, mais qui a regagné en jeunesse, vigueur et popularité dans sa version *relookée* dite de l'apprentissage profond. De tels réseaux ont besoin pour fonctionner d'une phase d'apprentissage basée sur un algorithme d'optimisation censé découvrir, par une « sorte » d'essai/erreur guidé, les meilleurs paramètres possibles reliant les neurones entre eux et assurant le bon fonctionnement de l'ensemble (nous détaillerons cela par la suite). Comme nous le verrons, apprentissage et optimisation ont généralement partie liée. C'est aussi un algorithme d'apprentissage qui peut tenter de découvrir les meilleurs paramètres d'un algorithme d'optimisation type génétique (également rencontré par la suite), cherchant à découvrir, en évaluant le moins possible, le meilleur itinéraire d'un voyageur de commerce devant traverser une succession de villes pour revenir à son point de départ. Comme on l'aura compris, ce cocktail algorithmique, cette possibilité de mélange à l'infini, ne contribue pas à faciliter le tri pour identifier correctement quel candidat algorithmique s'applique à quel type de problème (dès lors que plusieurs peuvent y contribuer).

- 3** Une dernière raison, un peu moins fondamentale et plus conjoncturelle, tient aux phénomènes de mode qui, comme partout ailleurs, ont un impact très important sur le choix et la popularité de ces algorithmes. Il n'aura échappé à aucun adepte de l'IA combien l'apprentissage machine et son représentant actuel le plus *sexy*, l'apprentissage profond, ont le vent en poupe aujourd'hui et se retrouvent projetés sur le devant de toutes les scènes logicielles. Pour de nombreux chercheurs en IA, quelque problème qui soit, quelque processus cognitif qui soit devraient pouvoir se traiter et se résoudre définitivement par l'apprentissage profond. Tout ce que l'IA a produit jusque-là ne conduirait qu'à un désolant cul-de-sac, soixante années de recherche et développement à jeter par les fenêtres ou reléguer au musée. Plusieurs de ces stars aujourd'hui vont même jusqu'à affirmer, à tort, que c'en est fini de la programmation, l'ordinateur pouvant tout résoudre par simple apprentissage, y compris s'autoprogrammer. Il suffirait de lui montrer quelques exemples et il retrouverait tout par lui-même. C'est bien évidemment ce qui amène beaucoup de nos étudiants à vouloir attaquer, leur enthousiasme le disputant à leur naïveté, le jeu du Puissance 4 ou du Snake avec, par exemple, un Deep-Q-learning (mélange de réseau de neurones profond et de l'apprentissage par renforcement, à découvrir par la suite), quand bien même cela revient à se munir d'un bazooka pour se débarrasser d'une mouche. C'est rendu d'autant plus possible et accessible au vu de la puissance de calcul actuelle de nos

ordinateurs. Les pionniers de l'IA étaient, à raison, beaucoup plus attentifs aux limites imposées par la vitesse des processeurs et la capacité de mémoire vive. Et encore, les quelques degrés additionnels du réchauffement climatique ne leur pendaient pas au nez.

Les premières publications sur les réseaux de neurones datent des années quarante et, lors de la fameuse conférence de Dartmouth donnant naissance à ce nouveau champ disciplinaire qu'était l'IA, on parla déjà abondamment d'apprentissage machine et de réseaux de neurones. Lors de cette même conférence, Arthur Samuel proposa une méthodologie d'apprentissage dite par renforcement qui permettait à un logiciel d'apprendre tout seul à jouer aux dames de façon optimale. AlphaZero (le meilleur logiciel actuel pour le jeu de go), une autre découverte soi-disant révolutionnaire, reprend pour l'essentiel ces mêmes idées soixante ans plus tard. Toutefois, c'est la voie de la programmation et de la résolution de problème qui prirent à l'époque le dessus sur ces réseaux de neurones. Pour ces mêmes pionniers de l'IA, la bonne idée était de mettre de l'humain dans les machines, que ces dernières s'inspirent de nos processus cognitifs. Avaient-ils fondamentalement tort, comme leurs héritiers le prétendent aujourd'hui ?

Le vénérable chercheur qu'est un des auteurs a vécu de nombreux prétendus hivers et étés de cette discipline (au prix d'une certaine lassitude) et déjà, à l'orée des années 1980, on parlait de la revanche de ces grisonnants réseaux de neurones sur l'autre versant de l'IA dit symbolique, plus ancré dans le raisonnement explicite et, justement, basé sur les algorithmes de recherche. Ces modes, donc, se succèdent et se ressemblent ; elles s'en vont et s'en viennent, en fonction des succès pratiques accomplis par cette discipline. Face à de tels engouements passagers, il est judicieux de prendre quelque recul, de raison garder, de résister aux effets de mode et de ne comparer le comportement et les performances de ces algorithmes qu'à partir de critères suffisamment objectivables : complexité algorithmique, temps de calcul, occupation de la mémoire, compréhensibilité de l'algorithme. Nous ne pensons modestement pas avoir vécu une succession d'hivers et d'étés, pas plus qu'une progression de cette discipline ponctuée de révolutions comme on le lit souvent, mais un parcours bien plus linéaire, en progrès constant et modifiant la cible des faisceaux des projecteurs en fonction des résultats très pratiques obtenus par l'une ou l'autre approche. Rien de plus.

Dans les chapitres qui suivent, nous allons attaquer dix problèmes très classiques de l'univers algorithmique et de l'IA. Pour chacun, nous allons proposer l'une ou l'autre méthode issue d'un des trois mécanismes fondamentaux (recherche, optimisation ou apprentissage). Cela permettra aisément au lecteur de bien faire la différence entre les trois et de saisir lequel (si aucun mélange n'est possible) est le mieux adapté au problème donné. Pour chacun, nous proposerons un ou plusieurs code(s) Python disponibles dans un dépôt GitHub public et accessibles également via un site web¹, ou sur la page du livre sur le site d'Eyrolles². Ces dix problèmes seront dans l'ordre :

-
1. Pour chaque projet, vous trouverez sur le site quelques informations complémentaires pour y accéder, le télécharger et l'exécuter. <https://iridia-ulb.github.io/AI-book/>
 2. <https://www.editions-cyrolles.com/dl/0101094>

- le jeu du taquin ;
- l'algorithme du plus court chemin (celui que l'on trouve dans les GPS) ;
- le jeu du sudoku ;
- le jeu Puissance 4 à deux joueurs ;
- le jeu du Tetris ;
- le jeu du Snake ;
- la séparation des spams et des non-spams ;
- la découverte des règles d'attribution d'un crédit bancaire ;
- l'organisation thématique de documents textuels non étiquetés ;
- la reconnaissance sur photo de chiens ou de chats.

Nous débuterons chaque chapitre par une capture d'écran de l'exécution du logiciel Python, suivie d'un descriptif plus conceptuel des méthodes et mécanismes à l'œuvre dans la réalisation du logiciel. Finalement, nous présenterons quelques parties du code réalisant très concrètement ces mécanismes décrits précédemment.

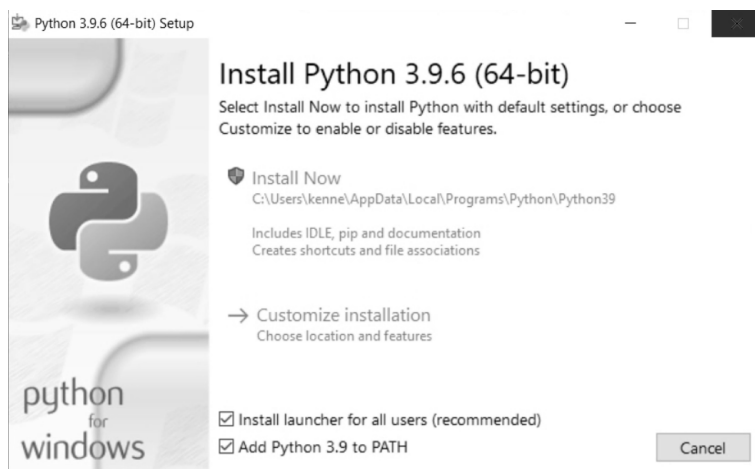
Installation de Python

Nous faisons l'hypothèse que vous utilisez Windows 64 bits en version 7 et plus, ou une distribution récente et à jour basée sur GNU/Linux telle que Ubuntu 20.

Sous GNU/Linux, Python est préinstallé dans la plupart des distributions modernes, nous partons donc du principe que Python 3 (>3.7) est installé sur votre machine.

Sous Windows, rendez-vous tout d'abord sur le site web de Python et téléchargez l'exécutable de la dernière version (3.9.6 à l'heure d'écriture de ces lignes) : <https://www.python.org/downloads/>.

Figure I-1
Installation de Python.



Lancez l'installation classique en cochant bien la case *Add Python 3.9 to PATH*.

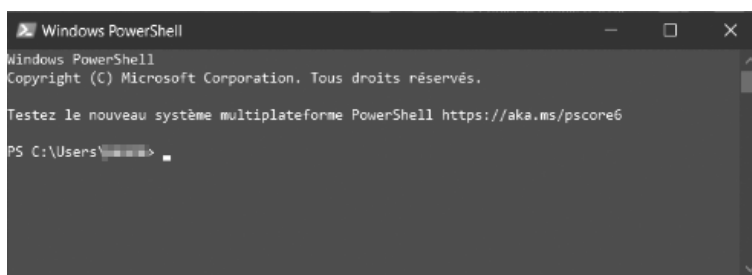
Votre installation du langage sera alors disponible dans l'invite de commande de Windows ou dans Powershell. Nous vous conseillons l'utilisation de ce dernier. En programmation, il est de bonne pratique de savoir se servir un minimum de l'invite de commande sous Windows et sous Linux, sans nécessairement avoir d'interface graphique, et ainsi de pouvoir réaliser de simples programmes et les exécuter au plus proche de la machine, tout en comprenant vraiment ce que vous faites.

Nous vous invitons donc, si ce n'est déjà fait, à vous renseigner sur les commandes usuelles à utiliser lors d'une session dans l'invite de commande (par exemple : `cd`, `ls`, `mv`, `cp`, `rm`...), qui vous seront très utiles !

Powershell est facilement accessible depuis Windows. Lancez une recherche dans les programmes (au clavier : touche *Windows*, puis taper « powershell »).

Figure I-2

Ligne de commande
Powershell.



Installation de poetry

Poetry est le gestionnaire de dépendances que nous avons choisi d'utiliser pour les différents exemples de ce livre. Il permet d'exécuter des programmes Python dans un environnement virtuel et gère automatiquement l'installation, la gestion et l'utilisation de ces environnements virtuels. Cela vous permettra de bien isoler les différents projets entre eux (ainsi que leurs dépendances), c'est pratique également si certains projets dépendent de bibliothèques dont les versions diffèrent d'un projet à l'autre.

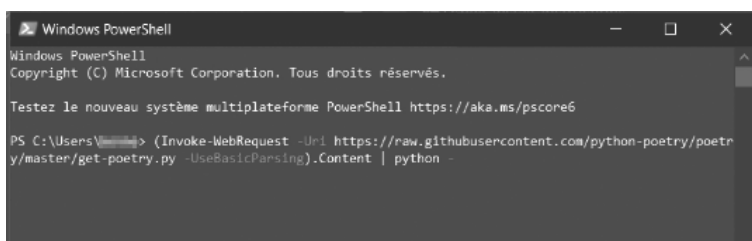
Sous Windows

Vous retrouverez les instructions d'installation de poetry sur le site web : <https://python-poetry.org/docs/#windows-powershell-install-instructions>.

Il vous suffit pour cela de démarrer Powershell, d'y copier la ligne suivante et d'appuyer sur *Entrée*.

```
(Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py -UseBasicParsing).Content | python -
```

Figure I-3
Installation de poetry
avec Powershell.



Une fois ceci exécuté, il n'y aura plus besoin de le refaire ; poetry aura automatiquement été installé et aura détecté votre installation de Python. Redémarrez ensuite Powershell.

Si cette commande ne fonctionne pas, référez-vous à la documentation de poetry dans la section *Alternative installation methods* : <https://python-poetry.org/docs/#alternative-installation-methods-not-recommended>.

Sous Linux

Vous pouvez utiliser les dépôts de votre distribution, par exemple apt sous Ubuntu/Debian :

```
sudo apt install poetry
```

Si poetry n'est pas présent dans les dépôts de votre distribution, installez-le manuellement en copiant la ligne suivante dans un terminal de commande (<https://python-poetry.org/docs/#osx--linux--bashonwindows-install-instructions>) :

```
curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py |  
python3 -
```

Puis redémarrez votre terminal.

Fichiers des différents projets

Rendez-vous sur la page associée au livre sur le site d'Eyrolles ou directement sur GitHub pour récupérer le dépôt contenant les différents projets.

Téléchargez l'archive zip (ou utilisez le gestionnaire de versions git) pour récupérer l'ensemble des codes et le placer dans votre dossier principal (par exemple : `C:\Users\votrenom` ou `C:\Users\votrenom\Documents`) et enfin désarchivez-le (*Extraire tout* dans le menu contextuel).

Ensuite, ouvrez Powershell et naviguez jusqu'à ce dossier. Pour cela, utilisez la commande `cd` suivie du nom du dossier (notez que la touche *Tab* du clavier offre une autocomplétion des commandes de la plus grande utilité).

Installer et utiliser les projets

Une fois dans le dossier principal, les différents projets s'installent de façon analogue. Rendez-vous dans le dossier du projet (par exemple, `cd 8Puzzle` depuis le dossier principal).

Puis entrez la commande suivante (une seule fois pour chaque projet) :

```
poetry install
```

Ou souvent dans le Powershell de Windows :

```
python -m poetry install
```

Une fois les dépendances installées et l'environnement créé, vous n'avez plus qu'à exécuter le code de chaque projet à l'aide de la commande `run` de poetry ; par exemple, pour le taquin (`8Puzzle`), cela donne la commande suivante :

```
poetry run python3 main.py
```

Ou souvent dans le Powershell de Windows :

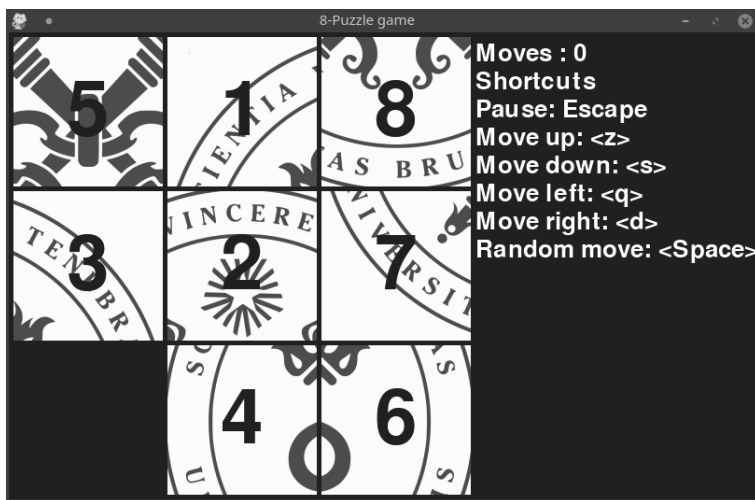
```
python -m poetry run python main.py
```

1

Jouons au taquin

Figure 1–1

Capture d'écran du logiciel Python mis à votre disposition pour le taquin.



Principe – Choix algorithmique

Rappelons que ce jeu consiste pour le joueur à déplacer un carreau vide dans l'une des quatre directions cardinales (s'il est près du bord, certains mouvements deviennent impossibles) afin de placer tous les carreaux de manière uniquement ordonnée (figure 1–1).

Figure 1-2

Un jeu possible du taquin :
à gauche dans sa configuration initiale
et à droite dans sa configuration finale,
celle désirée.

Configuration initiale			Configuration finale		
1	2	3	1	2	3
5	6		4	5	6
7	8	4	7	8	

Ce jeu est simple et tous les concepts fondamentaux de l'IA ainsi que les bases de toutes les solutions algorithmiques peuvent y être parfaitement illustrés.

Commençons par le concept d'**état**, fondamental en IA, qui décrit la configuration du jeu à un moment donné ; ici, le positionnement de tous les carreaux, y compris le carreau vide. Un problème en IA démarre en général par une configuration initiale de cet état, une manière formelle de le représenter, et se définit par la configuration finale désirée dans ce même formalisme. On conçoit le caractère universel d'une telle définition, ce qui laissa penser aux précurseurs de l'IA que les algorithmes de résolution qu'ils mettraient au point conserveraient ce même caractère d'universalité. Sans surprise d'ailleurs, un des algorithmes les plus célèbres de l'IA des origines s'appelait modestement le GPS (*General Problem Solver*¹).

Puis vient le concept d'**action** ; dans ce cas-ci, le déplacement du carreau vide dans les positions successives possibles qui permet de faire évoluer l'état comme seul résultat de l'action. Là aussi, toute la dimension cognitive de l'IA se trouve illustrée par ce concept. L'humain est capable de prédire l'effet de ses actions sur le monde et, de ce fait, d'échafauder des plans d'action en enchaînant les prédictions et en s'autorisant des retours en arrière si ces inférences ne mènent nulle part, tout comme les algorithmes d'IA (on parle de *backtracking*). Ces algorithmes de recherche furent dès les années 1950, à la naissance de l'IA, une parfaite transposition informatique d'une démarche cognitive conçue comme universelle et vue comme la quintessence de l'intelligence humaine : résoudre un problème en enchaînant les actions/prédictions jusqu'à découvrir le meilleur enchaînement possible, le bon plan.

On voit alors se dessiner un arbre dont les nœuds sont les états et les branches les actions qui permettent de descendre de nœud à nœud, jusqu'à aboutir au nœud cible. Enfin vient le concept de **solution** du problème, ici la succession d'actions (ou de coups dans le cas d'un jeu), idéalement la plus courte, qui, à partir de n'importe quelle configuration initiale, conduira à la configuration ordonnée. La solution pour le taquin est donc une succession de longueur minimale de coups gagnants.

1. Comme le dit Wikipédia : « Le General Problem Solver (GPS) est un programme informatique créé en 1959 par Herbert Simon, Cliff Shaw et Allen Newell dans le but de construire un solveur de problèmes universel. N'importe quel problème formalisé peut en principe être résolu par GPS, par exemple des preuves de théorèmes, des problèmes géométriques et des parties d'échecs. GPS était le premier programme à séparer sa base de données (tables) de sa stratégie de résolution de problèmes. »

Attaquons ce même problème à l'aide de deux des trois mécanismes fondamentaux de l'IA : la recherche et l'apprentissage. L'optimisation serait aussi envisageable et nous esquisserons juste sans approfondir l'algorithme génétique en question, que nous expliquerons plus en détail pour le jeu du sudoku. En effet, ces trois mécanismes sont applicables ici et tous les trois peuvent mener à la solution, même si, dans ce cas précis, nous allons chercher à vous convaincre que le premier, le plus ancien par ailleurs, s'impose très naturellement.

Recherche : A*

Cette approche, historiquement emblématique des premiers succès très pratiques de l'IA, consiste, en partant de la configuration initiale, à explorer tous les coups possibles afin de construire un arbre se ramifiant vers le bas : de coup en coup. À chaque niveau, on obtient autant de nouvelles configurations qu'il y a de coups possibles, jusqu'à obtenir la solution au dernier niveau. La recherche en largeur d'abord, illustrée à la figure 1–3, revient en effet à tenter à chaque niveau tous les coups possibles, en repérant les configurations communes entre plusieurs branches et en interdisant les cycles (c'est-à-dire en évitant de revenir sur une configuration déjà rencontrée précédemment). Il est clair que cette recherche finira par atteindre la configuration solution en le moins de coups possible, si une séquence qui y conduit existe en effet (le jeu pourrait être insoluble). Dès que la configuration désirée est trouvée, on conçoit aisément qu'elle aura été atteinte par une séquence minimale de coups.

Une alternative est la recherche en profondeur d'abord (figure 1–4), qui consiste à s'enfoncer de plus en plus dans l'arbre, en sélectionnant à chaque niveau un seul coup possible (fixant un ordre de priorité entre les coups), mais capable de rebrousser chemin et d'emprunter une voie alternative si cette succession de choix de coups mène à un cul-de-sac (plus aucun coup possible). Cette option, sans avoir besoin de vous le démontrer, est susceptible de trouver une séquence de coups gagnants, mais n'est plus assurée de vous révéler la séquence optimale. Elle s'arrêtera à la première trouvée.

Figure 1–3

Résolution du jeu du taquin
par une recherche en largeur
d'abord.

<https://courses.cs.washington.edu/courses/cse473/12au/slides/lect3.pdf>

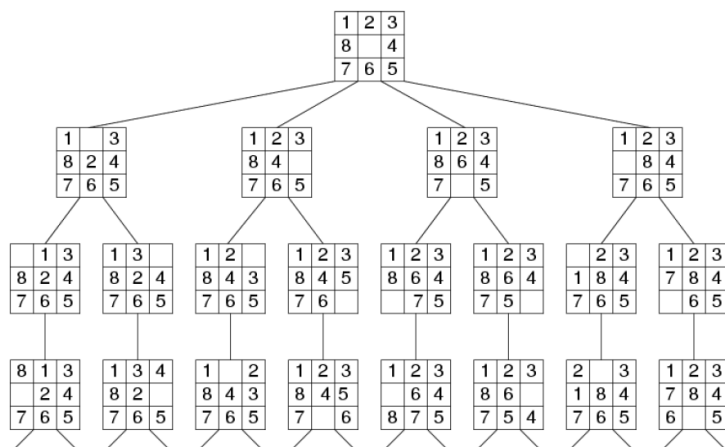
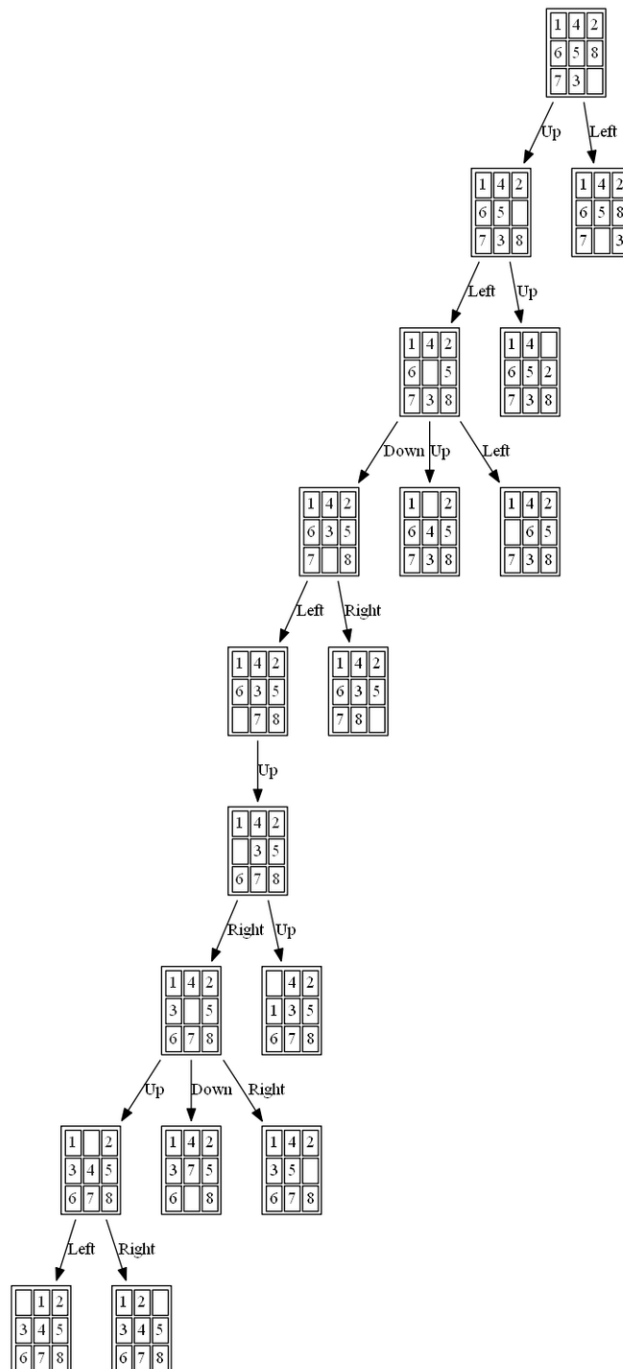


Figure 1-4

Résolution du jeu du taquin
par une recherche
en profondeur d'abord.

<https://sandipanweb.wordpress.com/2017/03/16/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/>

using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/

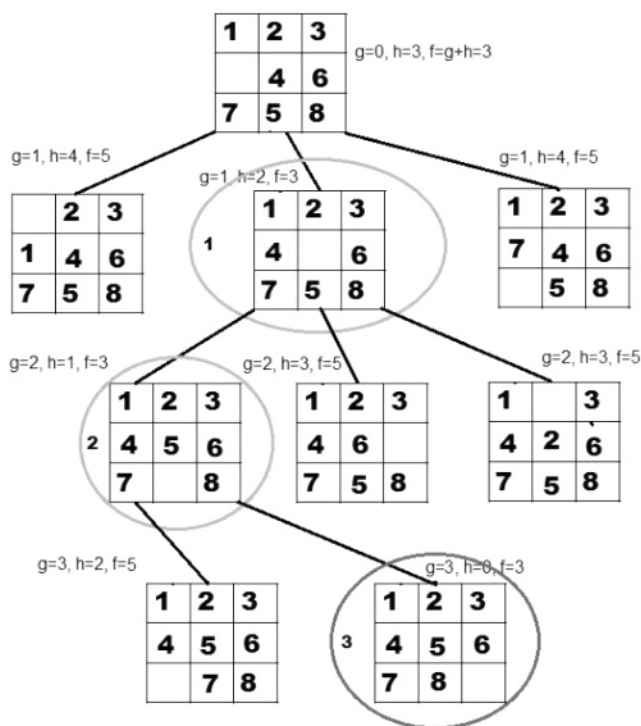


Un des algorithmes les plus fameux de l'IA, nommé A^* , le meilleur algorithme conçu pour résoudre le taquin et que nous comprendrons mieux encore à l'aide du code Python qui accompagne ce chapitre (quelque peu schématisé à la figure 1-5), se situe entre les deux précédents et privilégie les coups à jouer selon une évaluation des configurations intermédiaires obtenues, sommant à la fois le nombre de coups effectués jusqu'à présent (souvent désigné par g et appelé coût de la configuration) et la qualité de la configuration obtenue jusqu'ici sur le chemin de la solution (par exemple, le nombre de carreaux mal positionnés par rapport à la solution). Cette deuxième partie de l'évaluation, souvent désignée par h pour « heuristique », fut la très belle innovation de l'algorithme A^* par rapport à ses prédécesseurs. Nous y reviendrons par la suite, quand il sera question du problème du plus court chemin. La valeur g témoigne de la qualité de la solution obtenue jusqu'ici, alors que la valeur h , autrement plus subtile, relève d'une prédiction sur la solution future, une estimation. Comme toute heuristique et estimation future, elle possède aussi en son sein de quoi nous fourvoyer, nous emmener dans un cul-de-sac ou une solution de moindre qualité. g dit toujours la vérité, rien que la vérité, alors que h nous enjoint de prendre des risques et s'avère susceptible de nous perdre. Pourquoi un tel h alors, quand g seul pourrait parfaitement faire l'affaire ? Pour une unique raison : gagner du temps, nous éviter d'évaluer des solutions qui finiront par s'avérer de bien moindre qualité.

Figure 1-5

Illustration de l'algorithme A^*
pour le jeu du taquin.

<https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288?gi=fa9b0fd6a73d>



Cette heuristique est une autre preuve de l'extraordinaire source d'inspiration pour l'IA que constitue la cognition humaine. En effet, nous aussi cherchons bien souvent à économiser du temps ou d'autres ressources précieuses, en estimant les chances de succès de certaines de nos décisions avant de les prendre. Nous empruntons cette petite rue pour chercher à nous garer dans un centre-ville saturé de voitures, car nous prédisons qu'il y a plus de chances de y trouver une place de stationnement, au risque de refaire le tour du centre-ville si notre prédiction s'avère trompeuse. Le choix de la prochaine configuration sur laquelle A* se concentre sera donc celle qui minimise $f=g+h$. Si certaines conditions de l'heuristique sont remplies, comme surtout sa non-surestimation de l'évaluation finale de la solution (par exemple, le nombre de coups qu'il a fallu effectuer pour arriver à la solution), on a la garantie que la solution finalement obtenue sera bien la meilleure (la plus rapide) pour arriver à la configuration désirée. Une heuristique très souvent utilisée dans le cas du taquin est la distance de Manhattan, qui s'obtient en calculant et sommant le nombre de déplacements (horizontaux et verticaux) qui conduirait à la configuration idéale. Toutefois, on pourrait parfaitement en imaginer d'autres, comme simplement le nombre de cases mal placées.

Ce célèbre algorithme de l'IA qui, malgré son grand âge, n'a pas pris une seule ride est clairement le plus rapide en temps de calcul, car il est impossible de trouver la séquence de coups optimale en un temps de calcul et un nombre inférieur d'exécution d'instructions. La qualité de cet algorithme dépend bien évidemment de celle de son heuristique et de l'assurance de la condition (la non-surestimation) qui, si elle n'est pas vérifiée, peut ralentir la recherche (sachant qu'il est toujours possible de rebrousser chemin et repartir d'une nouvelle configuration). En l'absence de cette heuristique, le « filet de sûreté » reste la recherche par largeur d'abord qui, au prix d'une mémorisation de toutes les configurations obtenues sur les différentes branches de l'arbre, est également assurée de trouver la solution optimale, mais en un temps de calcul cette fois nettement plus long. En effet, dans le cas du taquin, le g vaut 1 à chaque coup (nous verrons que cette valeur est plus « informative » dans le cas du plus court chemin) et y recourir ne nous apporte rien de plus qu'une simple stratégie en largeur d'abord.

Le logiciel Python du taquin (8Puzzle) en pratique

Le projet du taquin se présente sous la forme suivante :

- un fichier `Game_UI.py` qui contient le code en charge de l'interface graphique du jeu ;
- le programme principal dans `main.py` ;
- deux autres fichiers `EightPuzzle_RL.py` et `EightPuzzle_astar.py`, qui contiennent respectivement les algorithmes de l'apprentissage par renforcement et de la recherche A*.

Recherche A*

Attardons-nous sur quelques extraits intéressants du code pour la recherche A*. Commençons par la partie centrale de l'algorithme :

```
def solveAI(puzzle):
    """
    Implementation of the A* algorithm to solve the 8-puzzle game.

    :param puzzle: The puzzle instance
    :return: The sequence of positions of the blank tile in order
             to solve the puzzle.
             This corresponds to the path to go from the initial
             to the winning configuration.
    """
    start = puzzle.tiles
    q = [(0, start, [start[-1]])]
    # We transform q into a priority queue (heapq)
    heapq.heapify(q)
    g_scores = {str(start): 0}

    while len(q) != 0:
        current = heapq.heappop(q)
        puzzle.tiles = current[1]

        if puzzle.iswin():
            print("Found solution:", current[2])
            return current[2]

        for m in moves(puzzle):
            # for all moves, g is the current cost
            g = g_scores[str(current[1])] + 1
            # f is the sum of the current cost + heuristic
            f = g + heuristic(puzzle, m)
            if str(m) not in g_scores or g < g_scores[str(m)]:
                heapq.heappush(q, (f, m, current[2] + [m[-1]]))
                g_scores[str(m)] = g
```

On retrouve ici la structure classique de l'algorithme A*. On démarre avec la liste des carreaux (donc l'état initial du jeu) dans `start`, puis on l'insère dans la queue. Ici, la queue est une `heapq`. En Python, une `heapq` est une liste de priorités qui trie automatiquement ses éléments de manière à ce que le premier de la liste soit toujours celui avec la plus grande valeur et le dernier avec la plus petite. Lorsqu'on stocke un *tuple*, comme c'est le cas ici, la valeur prise en compte est la première du tuple. On voit ici que l'élément initial, la racine de l'arbre de recherche, a une valeur de 0. On garde aussi en mémoire les scores des nœuds qui sont visités dans le dictionnaire `g_scores`.

Ensuite, on récupère la dernière valeur de la queue grâce à `heappop`, on vérifie s'il s'agit du nœud terminal, auquel cas on stoppe l'algorithme (on a alors trouvé la solution), sinon on parcourt les nœuds fils du nœud courant, grâce à la fonction `moves`, qui génère les candidats pour les ajouter à la

queue. Pour chaque nœud fils, on calcule son score g , qui correspond au coût de la solution, et on calcule l'heuristique h , grâce à la fonction `heuristic`. Finalement, on ajoute les deux pour trouver f , la valeur de priorité du nœud. On ajoute ensuite le nœud à la queue dans le cas où il n'a jamais été visité ou s'il a été visité mais avec un coût (score g) supérieur.

La fonction `heuristic`, ci-après, reprend le calcul de la distance de Manhattan entre un nœud et la solution attendue à la fin du jeu. Pour chaque case, on regarde la valeur absolue de la distance sur les axes verticaux et horizontaux de la case par rapport à la position qu'elle devrait occuper.

```
def heuristic(puzzle, n):
    """
    Computes the Manhattan distance of all tiles corresponding
    to the heuristic used in the A* algorithm.

    :param puzzle: The puzzle instance
    :param n: Configuration for which we want to compute
               the total Manhattan distance.
    :return: Total Manhattan distances for all tiles
             in configuration.
    """
    dist = 0
    for i in range(9):
        # Sum of Manhattan distances for all tiles
        dist += abs(n[i][0] - puzzle.winCdt[i][0]) + abs(
            n[i][1] - puzzle.winCdt[i][1])
    return dist
```

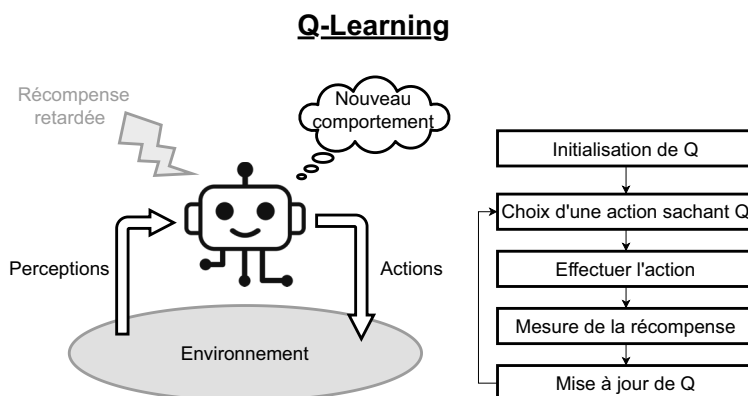
Apprentissage par renforcement : le Q-learning

Envisageons une autre possibilité pour résoudre le taquin : l'apprentissage par renforcement appelé Q-Learning. Nous allons de fait pouvoir démarrer une première étude comparative entre les méthodologies de recherche et d'apprentissage et nous rendre compte qu'une telle analyse n'est pas des plus évidentes. À l'origine, les algorithmes d'apprentissage par renforcement partent d'une idée toute simple (figure 1–6) : un agent autonome cherche à optimiser une séquence d'actions dans son environnement, en les exécutant au fur et à mesure et en recevant, à l'issue de chacune de ces séquences, un « renforcement », c'est-à-dire une évaluation de cette séquence par rapport à l'objectif désiré. C'est comme participer au jeu « du chaud et du froid », lorsque l'on vous guide par des « plus chaud » et des « encore plus chaud ». Ce renforcement est souvent retardé, c'est-à-dire qu'il ne survient pas à la suite de chaque action, mais à l'issue d'une séquence de ces actions.

Petit à petit, en se basant sur ces retours d'expérience, ces carottes et ces bâtons, l'agent cherche à découvrir la meilleure séquence d'actions possibles pour le mener à bien. Les bases du Q-learning, une de ces méthodes d'apprentissage par renforcement des plus connues, requièrent, là aussi, les éléments de départ et de conceptualisation suivants : état, actions et but poursuivi, d'où en partie la possible confusion avec les algorithmes de recherche, car les conditions initiales de la méthode semblent en effet exactement les mêmes.

Figure 1-6

L'apprentissage par renforcement en général et son représentant le plus illustre : le Q-learning.

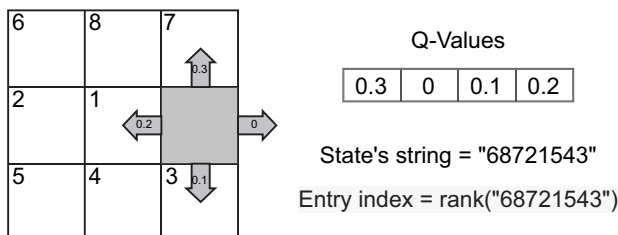


En fait, l'algorithme ici va essayer d'associer à chaque état rencontré (chaque configuration du taquin) la meilleure action possible, afin de parvenir au but poursuivi de façon optimale. On parle d'apprentissage, car ces meilleures actions possibles (ici les meilleurs coups) vont être découvertes par essais/erreurs, en s'appuyant sur les renforcements perçus et accumulés au cours du temps. À la différence du A* et de la recherche en général, qui résout le problème en un coup, le logiciel apprend ici tout en jouant de mieux en mieux. Une fois l'apprentissage terminé, on a la solution optimale, idéalement la même que celle découverte par A*.

Comment procède l'algorithme du Q-learning dans le cas du taquin ? On définit toutes les configurations possibles (tous les états possibles). Chaque configuration est représentée par une chaîne de neuf chiffres, chacun de ces chiffres prenant une valeur entre 1 et 9 (figure 1-7). Il y a donc neuf configurations possibles. Pour chacune d'entre elles, on définit les quatre déplacements possibles auxquels on associe une valeur appelée la Q-valeur, initialisée aléatoirement au début et qui évalue la pertinence de ce coup dans cet état donné (plus la Q-valeur est élevée, plus ce coup semble être le meilleur à jouer). Dans la figure qui suit, on découvre les quatre Q-valeurs pour les quatre coups possibles à jouer dans la configuration 687219543.

Figure 1-7

Représentation de l'état du taquin avec les quatre Q-valeurs associées aux quatre actions possibles.



L'apprentissage peut alors démarrer en partant d'une configuration initiale. Le choix de l'action dans chaque configuration se fait sur la base de ces Q-valeurs, en favorisant l'exploration (choix au hasard) au début de l'apprentissage et l'exploitation vers la fin (l'action avec la plus grande Q-valeur est privilégiée, car on a appris de plus en plus de choses). Cette stratégie de choix

d'action est un des paramètres de l'algorithme. Une version simple serait de prendre deux fois sur trois un coup au hasard et une fois sur trois le coup avec la meilleure Q-valeur, ou encore de faire un choix probabiliste proportionnel aux Q-valeurs. Après chaque coup dans un état donné, on met à jour la Q-valeur du coup à peine exécuté de la façon suivante :

$$Q\text{-valeur}(action, \text{état}) \leftarrow (1 - \alpha) \times Q\text{-valeur}(action, \text{état}) + \alpha (\text{renforcement} + \gamma \max (Q\text{-valeur}(action, \text{état}_{\text{suivant}})))$$

Cette modification tient compte de la Q-valeur de la meilleure action dans l'état suivant (celui résultant de cette action), atténuée d'un γ inférieur à 1, et du renforcement obtenu comme résultat de cette action. En scrutant l'équation de cette mise à jour, on peut assez facilement déduire que toutes les Q-valeurs tendent vers la somme des renforcements obtenus en choisissant la meilleure action dans chacune des cases, chacun de ces renforcements (r_n) se voyant atténué par un γ inférieur à 1, de manière à pondérer davantage les renforcements les plus récents.

$$Q\text{-valeur} \leftarrow \sum_0^{\infty} r_n \gamma^n$$

Ainsi, dans le cas du taquin, on pourrait imaginer une valeur de ce renforcement qui vaille 1 à chaque coup et, à la fin d'une séquence, dans le cas du dernier coup, qui soit le nombre de carreaux encore mal placés. Avec pareil choix, la meilleure action serait celle qui minimise cette somme de renforcements. On pourrait aussi recevoir comme alternative un renforcement de 0 pour chaque coup, à l'exception de celui qui conduit à la configuration optimale et qui, lui, recevrait la valeur 1. Les Q-valeurs tendraient donc vers la valeur γ^n , avec n le nombre de coups pour atteindre la configuration optimale, que l'on chercherait cette fois à minimiser tout en maximisant les Q-valeurs.

Une condition indispensable au bon fonctionnement de cet algorithme est celle de sa « Markovianité ». Tant le prochain état que le renforcement obtenu en passant d'un état à l'autre ne peuvent dépendre que de l'état présent et non pas de la manière dont on y est parvenu. C'est très clairement le cas avec le taquin, pour lequel nul aléatoire ni indéterminisme ne vient interférer avec le déroulement du jeu. Pour chaque configuration intermédiaire, c'est toujours la même suite de coups optimaux qui conduit à la configuration optimale.

Au fur et à mesure de ces essais/erreurs, on conçoit aisément que, progressivement, et pour chacun des neuf états, les quatre Q-valeurs associées à chaque état tendent vers le nombre minimal des déplacements nécessaires pour aboutir à la configuration optimale. On dit progressivement, car il faut de très nombreuses séquences de jeux pour que les Q-valeurs convergent en effet vers leur valeur optimale. Les bases sont mêmes posées pour une résolution plus large du problème, car débouchant sur la solution optimale quelle que soit la configuration initiale. Il suffit de faire partir l'algorithme de n'importe laquelle de ces configurations possibles. La question essentielle qui se pose plutôt est : pourquoi parle-t-on d'apprentissage à la différence des alternatives, pour lesquelles il est question de recherche et d'optimisation (que nous aborderons dans le prochain chapitre). L'apprentissage, lorsqu'il converge idéalement, nous conduit ici au meilleur coup possible pour chaque configuration du taquin. On a donc bien appris quelque chose. Or, A^* , a priori, nous donne simplement la meilleure séquence pour une configuration initiale donnée. Et là, c'est plutôt

de recherche qu'il s'agit. Mais, il ne vous aura pas échappé que retrouver le résultat du Q-learning n'est en rien difficile pour A^* . Il suffit de le démarrer de plusieurs configurations initiales différentes et de mémoriser pour chacune la séquence des meilleurs coups possibles. A^* retrouverait alors ce que le Q-Learning obtient mais, et on vous le garantit, de manière beaucoup plus rapide et beaucoup plus efficace. Pourquoi dès lors s'obstiner à faire de l'apprentissage par renforcement ?

Les adeptes de ce type d'apprentissage, de plus en plus nombreux (car on trouve cette approche à l'œuvre dans le meilleur joueur logiciel de go, de nombreux jeux vidéo et dans l'apprentissage des robots), voient dans cette approche la possibilité de justement pallier par la force brute de l'ordinateur des angles morts de notre cognition humaine, des aspects du problème que notre entendement d'ingénieur n'a pas réussi à découvrir ou élucider. Dans le cas du taquin, nous possédons une connaissance parfaite du jeu et de l'effet de chacune de nos actions, mais ce n'est pas toujours le cas de tous les problèmes que l'IA se doit d'affronter. Par exemple, lorsqu'un robot explore son environnement, l'effet de ses actions sur l'évolution de son état n'est pas connu ; le robot le découvre au fur et à mesure. Il en va de même pour une voiture autonome qui devrait circuler dans le centre de Paris. Ces agents ne possèdent pas un modèle aussi parfait et déterminé du monde dans lequel ils se déploient. Il ne vous aura pas échappé que le monde est un peu plus complexe et imprédictible que le taquin.

Cela peut être aussi le cas d'un jeu vidéo pour lequel l'état se borne à l'image 2D de l'écran car il est difficile de mettre en mémoire toutes les évolutions de cette même image en réponse à nos actions. Doté d'une connaissance parfaite de l'état du monde et de l'effet des actions sur l'évolution de ces états, il n'est nul besoin de véritable apprentissage ; une recherche de type A^* ou programmation dynamique, consciente de la nature « arborée » de la construction des solutions et de la possible disponibilité d'une évaluation des coups intermédiaires, semble idéalement faire l'affaire, car conduit à la solution pour un coup de calcul minimum. On privilégiera, pour ces mêmes raisons, ce type d'algorithme pour tous les jeux à un joueur, comme le sudoku, le solitaire, le Rubik's Cube ou autres jeux du même acabit.

Le seul espace algorithmique possible qu'une approche de type apprentissage pourrait occuper dans ce cas est la découverte, par essais/erreurs, de la meilleure heuristique possible parmi plusieurs alternatives, en souhaitant (mais c'est souvent le cas pour un même type de jeu) que cette heuristique se généralise pour toutes les instances de quelque taille qui soit de ce même jeu (par exemple un taquin de n'importe quelle dimension). Il s'agirait alors de lancer plusieurs versions de A^* (chacune avec sa propre heuristique) et de sélectionner celle qui obtient la solution en le moins de coups possible. Nous avons vu que, selon l'heuristique choisie et pour autant qu'elle reste admissible, c'est-à-dire sous-estimant le véritable coût de ces choix d'action, la vitesse de convergence de A^* vers la séquence optimale peut beaucoup varier. Cette heuristique restant le seul élément de l'algorithme qui peut prêter à incertitude et jouant pourtant un rôle essentiel dans l'économie de calculs de l'algorithme, pourquoi ne pas l'apprendre par essais/erreurs ?

Application au taquin

Découvrons maintenant plus concrètement la partie du programme qui traite le jeu en adoptant l'apprentissage par renforcement. Elle est séparée logiquement en deux :

- une partie apprentissage, où l'algorithme va jouer des parties pour apprendre et enrichir sa connaissance des états du jeu et les enregistrer dans sa table Q ;
- et une partie jeu, où l'on peut utiliser une table Q que l'on vient d'entraîner et la tester sur une instance du jeu.

La fonction principale qui gère l'entraînement est `trainingAI` (`8Puzzle_RL.py`, ligne 150) ; c'est elle qui va lancer l'entraînement à proprement parler sans sa boucle.

On appelle *policy* de l'algorithme, souvent notée π , la stratégie déterminée par l'algorithme à un moment donné. Le but de l'apprentissage par renforcement est de trouver la stratégie optimale (*optimal policy*), notée π^* .

Afin de sauvegarder l'état du jeu dans la table Q , il est tout d'abord converti dans un format plus commode à stocker dans une table. Les cases sont simplement numérotées de gauche à droite et de haut en bas dans l'ordre dans lequel elles apparaissent dans le jeu. Ainsi, la chaîne de caractères 123456789 représente l'état final du jeu (la case vide n'est pas représentée).

L'apprentissage a lieu ensuite dans la fonction `train` présentée ci-après (`8Puzzle_RL.py`) :

```
def train(self, nbGames):
    """
    Trains the AI by playing a specified number of games.
    The games are randomly generated.

    :param nbGames:    An integer corresponding to
                        the number of games to train the AI on.
    :return:           Integer corresponding
                        to total number of moves.
    """
    nbMoves = 0
    for _ in range(nbGames):
        state = self.generateGame()
        while state != "123456789":
            newAction = self.selectNewAction(state)
            state = self.playRound(state, newAction)
            nbMoves += 1
        self.nbPlayedGames += 1
    return nbMoves
```

Cette fonction joue un certain nombre de parties, dépendant des options d'entraînement de l'algorithme. La fonction suit la logique et la séquence classique de tout apprentissage par renforcement : d'abord on génère une instance de jeu aléatoire, puis on entraîne jusqu'à trouver la solution. Ensuite, on génère une nouvelle instance. La phase d'entraînement procède d'abord à la sélection d'une action à faire avec la méthode `selectNewAction(state)` (détaillée ci-après) puis, une fois l'action sélectionnée, elle est exécutée et la table Q est mise à jour dans la méthode `playRound(state, newAction)` (détaillée ci-après).

La sélection de l'action (dans `selectNewAction`) s'opère selon le dilemme classique exploration-exploitation, ici en présence d'une probabilité `epsilon` de prendre une action au hasard (exploration) ou alors de prendre la meilleure action possible connue (exploitation).

```

def selectNewAction(self, currentState):
    """
    Chooses a new action according to the epsilon-greedy selection method.

    :param currentState:    A string corresponding to the current state
                            of the game.
    :return:                An integer corresponding to the new action
                            (i.e. the direction we want to move the empty
                            tile to).
    """
    # Choose a random action with probability epsilon.
    if random() < self.epsilon:
        return randint(0, 3)
    # With probability 1-epsilon, choose the action
    # corresponding to the maximum reward.
    else:
        stateIndex = findRank(currentState) - 1
        maxQValue = max(self.qTable[stateIndex])
        # If multiple actions give the maximum value, we randomly
        # choose one of those maximum actions.
        maxIndexes = []
        for i in range(4):
            if self.qTable[stateIndex][i] == maxQValue:
                maxIndexes.append(i)
        return maxIndexes[randint(0, len(maxIndexes) - 1)]

```

Lorsqu'on choisit l'exploitation, on regarde dans la table Q quelle est l'action qui maximise la récompense future attendue et on la sélectionne. Si plusieurs actions ont les mêmes récompenses attendues, on en choisit une au hasard.

```

def playRound(self, state, action):
    """
    Moves a tile and updates the Q-Table.

    :param state:          A string corresponding to the current state
                            of the game.
    :param action:         An integer corresponding to the direction
                            where to move the empty tile.
    :return:               A string corresponding to the next state
                            of the game (after the move).
    """
    stateIndex = findRank(state) - 1
    nextState = self.makeMove(state, action)
    if nextState == "123456789":
        reward = 1
    else:
        reward = 0

```

```

self.qTable[stateIndex][action] += self.getAlpha() * (
    reward
    + self.gamma * max(self.qTable[findRank(nextState) - 1])
    - self.qTable[stateIndex][action])
return nextState

```

On joue ensuite cette action (dans la méthode `playRound`) et on met à jour la table `Q` en fonction de la récompense obtenue. On définit une récompense de 1 si on est arrivé à la fin et de 0 sinon. La table `Q` est ensuite mise à jour grâce à la formule de Bellman :

$$Q^{nouvelle}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

$Q(s_t, a_t)$	<i>l'ancienne valeur</i>
α	<i>le taux d'apprentissage</i>
r_t	<i>la récompense</i>
γ	<i>le facteur de réduction</i>
$\max_a Q(s_{t+1}, a)$	<i>l'estimation de la valeur future optimale</i>

Ainsi, quelle que soit l'action choisie (exploration ou exploitation), que l'action soit « bonne » ou « mauvaise », l'estimation de sa qualité (le `Q(st, at)`) sera basée sur la meilleure action future possible. L'algorithme du Q-learning est dit *off-policy learning*, dans le sens où l'action future estimée n'est pas basée sur la stratégie actuelle, *policy* π , mais sur la meilleure action suivante possible.

Après un nombre conséquent d'itérations, la table `Q` s'améliore petit à petit et converge vers la stratégie optimale (π). Après l'entraînement, on peut réutiliser la table `Q` fixée et ne plus choisir que la stratégie (π^*) trouvée par l'algorithme (phase de jeu, en pure exploitation).

L'algorithme d'optimisation génétique

Cet algorithme, plus récent dans l'histoire de l'IA, mais que nous esquisserons juste ici pour le redécouvrir plus en profondeur lors de l'étude du sudoku, cherche la solution optimale d'un problème en s'inspirant de la théorie de l'évolution darwinienne et de la sélection des plus adaptés. Dans une population de solutions proposée, chacune de ces dernières est évaluée ; on préserve les meilleures et on génère une nouvelle population en recombinaison et en mutant les solutions les plus prometteuses de la population précédente. On répète ainsi ces quatre opérations – évaluation, sélection, mutation, croisement – jusqu'à obtenir une population dont un des individus est la solution optimale. De manière générale, un algorithme d'optimisation « recherche² » la solution optimale dans un espace de solutions possibles, dans lequel l'algorithme progresse, étape par étape, en espérant les trajectoires les plus courtes pour atteindre l'optimum.

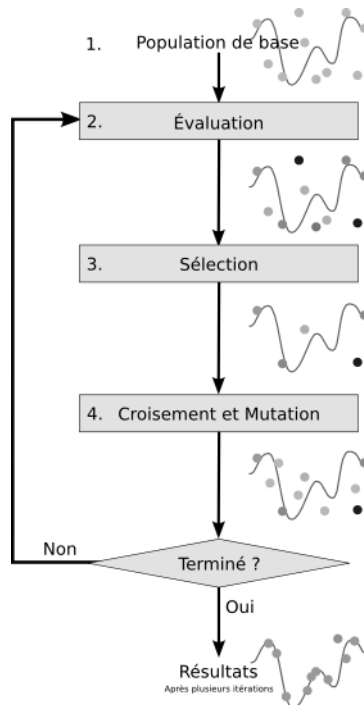
2. Nous discuterons de cette ambiguïté sémantique, car de « recherche » il est ici encore question.

Une bonne image mentale, comme la figure 1–8 extraite de Wikipédia l'illustre, est celle d'une chaîne montagneuse dans laquelle on progresse, étape par étape, en espérant les trajectoires les plus courtes pour atteindre le sommet le plus haut, avec le risque de rencontrer sur le parcours de nombreux sommets moins hauts et qui piègent toute la démarche. Lequel est vraiment le bon ?

Figure 1–8

Illustration des opérations des algorithmes génétiques.

https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique



Dans le cas du taquin, il suffirait de construire les individus-solutions comme une succession d'un certain nombre d'actions possibles, en fait une succession de nombres entre 1 et 4, 1=gauche, 2=droite, 3=haut, 4=bas. La mutation consisterait par exemple à modifier un des nombres tirés au hasard (passer de 1 à 4 par exemple) et la recombinaison à mélanger deux candidats, là aussi tirés au hasard, pour en produire deux autres. Par exemple :

[1,3,4,2,3,1] et [1,4,3,3] pour donner [1,3,4,3,3] et [1,4,2,3,1]

La qualité d'une solution (d'un individu de la population) pourrait se calculer ainsi :

Taille de l'individu (nombre de coups possibles)

+ nombre de carreaux mal positionnés à la fin de la séquence

Il est clair que la solution optimale (ici minimale) serait bien celle cherchée : taille minimum + 0.

Comme beaucoup d'algorithmes d'optimisation dans lesquels une part belle est laissée au hasard, l'algorithme est censé théoriquement converger vers la solution optimale : ici, le nombre minimal de coups pour conduire le taquin dans sa configuration ordonnée. C'est en effet aussi d'une recherche de la solution optimale qu'il est question ici, mais dont le trajet est beaucoup moins balisé et guidé que dans l'exécution de A^* . Pourquoi ? La raison essentielle est la disparition de deux aspects fondamentaux des algorithmes de recherche vus précédemment, les notions d'état et d'arbres. En effet, l'algorithme génétique explore un espace homogène de solutions dans lequel ces deux notions ne sont plus explicitées ; elles sont dissimulées et implicites dans l'exploration de l'espace des solutions et elles ne permettent plus de structurer et parcourir cet espace de manière économe. Ainsi, l'algorithme génétique pourrait conduire à des séquences de coups inutilement longues ou à des cycles dans ces séquences (même s'il est possible de les pénaliser en leur attribuant une très mauvaise évaluation).

Dans A^* , ces deux notions (et on ne parle même pas de l'heuristique qui fait gagner du temps) sont essentiellement présentes et permettent de guider et de baliser la recherche (par exemple, revenir à l'état de départ si une branche partant de celui-ci n'aboutit nulle part, passer d'un nœud à un autre plus prometteur dans le parcours de l'arbre). Il est donc inopportun de négliger, comme le feraient ceux qui choisissent d'affronter le taquin par un tel algorithme d'optimisation, ces éléments connus a priori qui contraignent et balisent les chemins de la recherche algorithmique en apportant un indéniable gain de temps de calcul (même s'il est nécessaire, comme dans la recherche en largeur d'abord, de mémoriser toutes les séquences obtenues, la mémoire étant moins coûteuse que le temps de calcul).

Les adeptes des algorithmes génétiques, dont le champ d'application est infini (on les retrouve un peu partout en IA, pratiquement tous les problèmes de l'IA pouvant se ramener à un problème d'optimisation), mettent souvent en avant cette extraordinaire faculté d'adaptation. Cette simplicité algorithmique (la nature se doit d'être simple, n'est-ce-pas ?) les prédispose à un nombre extraordinaire d'applications et de situations problématiques. Leur mise en œuvre est apparemment possible pour quelque problème que ce soit et pour lequel définir une fonction de coût à maximiser est de l'ordre du possible (de manière générale, résolution de problème et fonction de coût sont intimement liées). Et, en effet, il est bien probable que, après un temps d'exécution de l'algorithme suffisamment long, la solution sera au bout du chemin et on finira par atteindre le sommet des sommets.

Cependant, là se trouve clairement le cœur du problème. À l'origine de l'informatique, les praticiens se sont toujours efforcés de minimiser le temps de calcul d'un programme afin d'aboutir à la solution. Cela leur était vital. À l'époque, les ordinateurs étaient terriblement lents et pauvres en capacité mémoire, cet effort de minimisation des temps de calcul par une écriture savante des algorithmes devenait indispensable. Avec le temps, au vu des accélérations spectaculaires des processeurs et de la massification des mémoires, cette préoccupation fondamentale tend à perdre de sa raison d'être. Les praticiens de l'IA, mais on devrait dire les informaticiens en général, adoptent de manière croissante des solutions de facilité d'écriture algorithmique, laissant à la puissance de calcul de nos ordinateurs le soin de compenser par cette dépense d'énergie leur absence de discernement a priori. On retrouvera cet état de fait très souvent dans la suite de cet écrit, car il est assez caractéristique aussi du succès de certaines voies actuelles empruntées par l'IA, tels les réseaux de neurones profonds qui, malgré leurs succès pratiques indéniables, sont des gouffres énergivores.

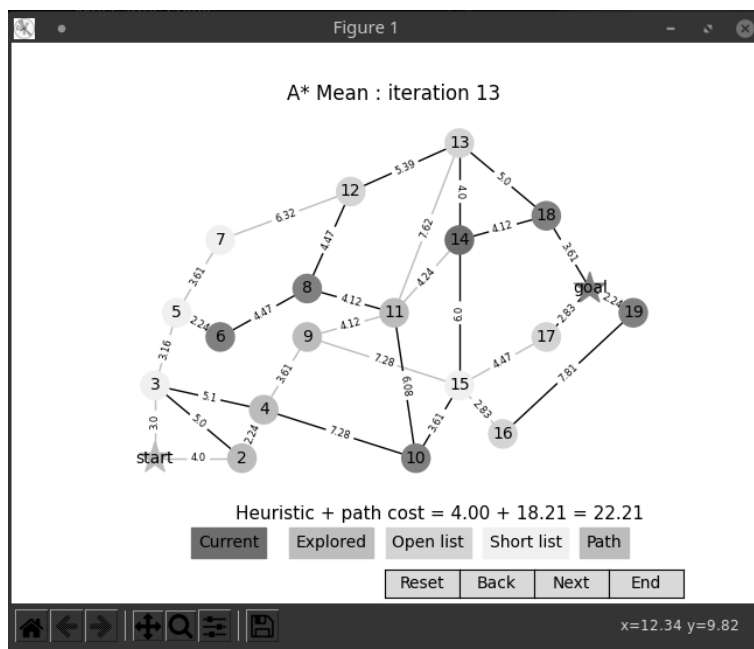
Surtout, ils délèguent à la machine un travail d'analyse préalable des données du problème, que les ingénieurs se réservaient avant, afin de guider et donc d'alléger la démarche algorithmique.

Or, et pour le dire de but en blanc, le numérique aujourd'hui est un des plus importants consommateurs électriques de toutes les activités humaines ; cela ne devrait qu'augmenter avec le temps et la numérisation de nos processus productifs et de ces mêmes activités. Un compromis judicieux deviendra toujours plus d'actualité entre la mise en œuvre du génie humain, pour laborieuse qu'elle soit, en vue d'une écriture subtile d'algorithmes sobres et de faible consommation, et la compensation de notre laxisme intellectuel par la puissance de calcul de nos ordinateurs.

Découvrir le plus court chemin

Figure 2-1

Capture d'écran du logiciel Python mis à votre disposition pour le problème du plus court chemin.



Principe – Choix algorithmique

Recherche du plus court chemin

Vos parents ou grands-parents étaient sans doute, comme les nôtres, amateurs de cartes géographiques qu'ils déplaient fièrement sur le capot de leur voiture afin de rechercher l'itinéraire le plus rapide pour rejoindre leur destination de vacances. Ils comparaient différents trajets possibles en sommant les distances reliant les étapes et écoutaient à la radio ou à la télévision (Internet n'en était qu'à ses premiers balbutiements) les conseils avisés d'un Bison Futé. Le GPS a mis fin à tout cela, à tel point que les psychologues se sont émus de la dégénérescence de certaines capacités cognitives, essentielles par exemple aux chauffeurs de taxi pour se repérer en ville. On comprend, au vu de la puissance et de la capacité calculatoire de nos ordinateurs, qu'il vaille mieux leur confier cette recherche visant à trancher entre une quantité gigantesque d'itinéraires possibles, à l'affût du plus court ou du plus rapide. Cependant, à nouveau, il est aisé de voir combien l'IA des origines, celle essentiellement intéressée par la résolution de problèmes requérant de notre part une démarche des plus rationnelles, s'est inspirée de nos processus cognitifs. L'algorithme de recherche du plus court chemin fonctionne d'une manière très proche de celle de nos anciens penchés sur la carte et traçant d'un doigt fébrile les itinéraires possibles.

L'algorithme qui sous-tend le fonctionnement de ces GPS et qui s'inspire en droite ligne de notre démarche cognitive est de nouveau le fameux A^* du chapitre précédent. On est très proche du taquin, avec ici pour état les lieux géographiques traversés, et non pas les configurations du jeu, et en présence d'un poids (la distance ou le temps de parcours, par exemple) pour chaque arête associant ces lieux (figure 2-2). On voit de nouveau apparaître l'arbre de recherche. L'algorithme part d'un point A donné, énumère à partir de ce point toutes les routes possibles pour rejoindre un certain nombre de points B, relance la même énumération à partir de ces points B, pour rejoindre les points C et ainsi de suite jusqu'à atteindre la destination finale.

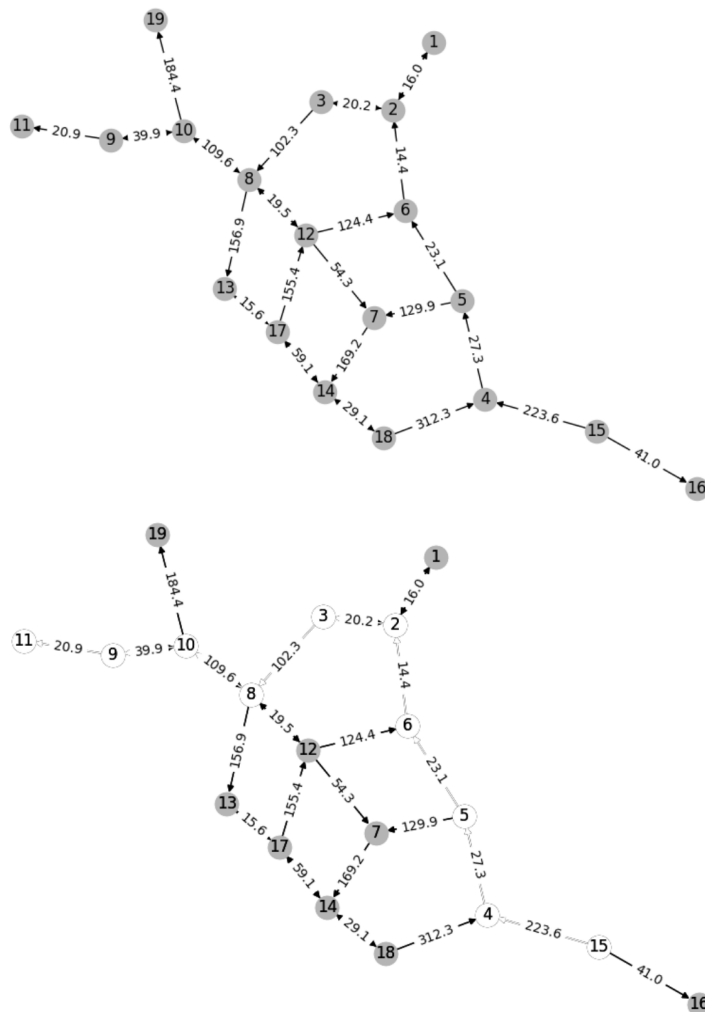
Évidemment, afin de limiter sa recherche et de parer au plus pressé, il peut au choix parcourir ces différentes étapes soit en profondeur (un premier A, suivi d'un premier B, un premier C, ensuite repartir de la base et sonder un deuxième possible B, etc.) ou en largeur d'abord (faire tous les B, puis tous les C, etc.). On a déjà différencié ces deux stratégies précédemment, la recherche en largeur d'abord nous permettant, à l'issue d'une trop large exploration des solutions possibles, d'identifier l'itinéraire optimal. Cependant, on peut y parvenir en nettement moins d'itérations si, au fur et à mesure, on parvient à éliminer des itinéraires candidats devenus beaucoup trop longs et à se concentrer uniquement sur les plus prometteurs. Par exemple, si un parcours A1-B1-C1 est devenu beaucoup plus long qu'un A2-B2 alternatif, on peut reprendre la recherche à partir de cet A2-B2 devenu plus prometteur. C'est ce que permet évidemment l'algorithme A^* , en exploitant comme heuristique la distance parcourue jusqu'ici, additionnée à celle qui reste à accomplir, approchée par sa version dite « à vol d'oiseau ». Mathématiquement, on peut calculer cette heuristique, ce « vol d'oiseau », en mesurant la distance euclidienne entre le lieu géographique dont on parle et la destination finale.

Une amélioration importante apportée par l'IA dans les années 1960 et 1970 est en effet cette idée d'heuristique. On se rappelle que la fonction d'évaluation d'un nœud qui guide la recherche de A^*

est la somme de g , le coût de l'itinéraire jusqu'ici, et de h , la distance à vol d'oiseau pour atteindre la destination finale. Pendant longtemps, les informaticiens se sont contentés du seul g , débouchant sur l'algorithme dit de Dijkstra, très connu et sollicité par eux dès qu'il s'agit de trouver le plus court chemin dans un graphe. Et cela marche ! Car, en repartant dans l'arbre que l'on dessine en parcourant les nœuds, toujours du nœud présentant le plus petit g , on a la garantie que l'itinéraire trouvé à la fin sera bien l'itinéraire optimal, le plus court. Quel est donc l'apport de cette heuristique ? Gagner du temps, sans rien perdre en optimalité de la solution, si l'heuristique sous-évalue la distance finale, comme c'est bien évidemment le cas de celle dite « à vol d'oiseau » (ou euclidienne). L'algorithme de Dijkstra n'est qu'un cas particulier de A^* , pour lequel l'heuristique vaut zéro et ne permet aucune accélération.

Figure 2-2

L'algorithme A^* découvre le plus court chemin, représenté en blanc, du graphe.



Aujourd'hui, des algorithmes comme le célèbre Waze, sont capables d'intégrer l'état du trafic en temps réel en exploitant la durée du trajet effectué pour aller du point A au point B par tous les utilisateurs de ce même Waze. Si plusieurs utilisateurs courants de l'application se trouvent soudainement bloqués par un impondérable, ceux censés emprunter un même chemin s'en trouveront rapidement découragés par des petits embouteillages représentés en rouge sur la carte de leur application. Et l'on comprend comment l'intrusion de cet algorithme dans la vie privée de certains conducteurs permet à beaucoup d'autres de gagner un maximum de temps au volant de leur véhicule ; l'espionnage collectif à des fins d'entraide collective, quelques vies privées dévoilées pour améliorer celle de tous les autres ! Ces problématiques d'atteinte à la vie privée afin, en principe, d'améliorer la vie de tout un chacun, sont devenues un des dilemmes moraux récurrents des développements de l'IA.

Lorsque les graphes en question deviennent trop importants, car présentant de trop nombreux lieux géographiques à traverser, une autre accélération possible (incluse dans le code Python accompagnant ce chapitre) est celle de la recherche dite bidirectionnelle, superposant à celle qui démarre du point de départ une seconde qui démarre, elle, du point d'arrivée. Si le code recourt à une technologie dite de *multi-threading*, capable de faire fonctionner simultanément les deux recherches, le temps de calcul informatique peut se trouver divisé par deux en les faisant se rencontrer. Comme malheureusement bien souvent en informatique, ce que l'on gagne en temps de calcul, on le perd en qualité de la solution, car plus rien n'assure que la recherche bidirectionnelle débouche sur la solution optimale. Ce compromis entre la qualité de la solution et la durée pour l'obtenir est sans doute la leçon cardinale, omniprésente et répétée à l'envi dans tous les chapitres de cet ouvrage. D'autres accélérations de A^* sont tout aussi possibles, comme la suppression de lieux géométriques intermédiaires, en reliant deux lieux extrêmes par la somme des distances qui les relient ; comme si vous choisissiez, quoi qu'il vous en coûte, d'emprunter une autoroute entre deux villes (vous supprimez des nœuds dans l'arbre de recherche), même si une alternative possible, et peut-être moins longue (pas forcément plus rapide), pourrait être de considérer des petits tronçons de routes nationales entre les plus petites villes et de les conserver dans la recherche de l'itinéraire le plus court (une bonne idée en cas de bouchons sur l'autoroute).

Dans la même lignée, certains de nos étudiants ont souvent fait le constat que l'algorithme A^* , malgré la réduction des alternatives explorées, exigeait la même durée d'exécution que son prédécesseur l'algorithme de Dijkstra, sinon plus. La raison en est le temps nécessaire au calcul de l'heuristique. Si, en effet, les quelques secondes gagnées par l'exploitation éventuelle de cette heuristique sont malheureusement perdues par le temps d'exécution nécessaire au calcul de cette même heuristique, autant parer au plus simple et ne plus en faire usage. En général, le recours au génie humain dans l'écriture des algorithmes d'intelligence artificielle a pour vocation d'alléger la force brute de nos ordinateurs et la quantité énorme d'ampères exigée par leur exécution. Mais cela peut ne pas toujours être le cas, nous autorisant une certaine paresse intellectuelle.

Internet aussi est l'objet d'une préoccupation de type plus court chemin afin que votre courriel atteigne son correspondant le plus rapidement possible. Aujourd'hui, et c'est un des derniers vestiges d'une véritable démocratie, tous les messages, les nôtres comme les tweets de Bill Gates, sont traités avec la même urgence. Ce réseau des réseaux connecte entre eux des réseaux locaux, connexion assurée par l'intermédiaire d'ordinateurs capables d'un certain raisonnement et appelés routeurs. Ces derniers ont la tâche stratégique, dès qu'un message leur parvient et en fonction de la

connaissance qu'ils ont de la situation actuelle du trafic sur le réseau, de décider quel chemin ce message doit suivre, c'est-à-dire vers quel prochain routeur il doit être dirigé. Cette décision algorithmique se base sur une table de routage, stockée dans chacun des routeurs et qui associe à chaque destinataire final possible le prochain routeur le plus approprié (le but étant de minimiser la durée du trajet à effectuer afin d'atteindre le destinataire). Il devient donc important de régulièrement mettre à jour cette table au vu de l'évolution constante de la charge sur le réseau. Chaque ordinateur connecté au réseau Internet possède ainsi une adresse et chaque message circulant sur le réseau cherche à atteindre sa destination finale en traversant le moins de routeurs possible. L'algorithme, ici, se charge de mettre à jour la table des routages et d'envoyer le message sur le prochain routeur censé le rapprocher le plus rapidement de son destinataire. Pour autant que l'on préserve son caractère démocratique et égalitaire, ce mécanisme de routage et ses fondements algorithmiques sont en train de se substituer à toute autre forme de protocole de communication, de la veine de ceux qui faisaient fonctionner le téléphone, le télégraphe ou même le courrier postal à l'époque.

Problème du voyageur de commerce

Les problèmes et les solutions de plus court chemin sont un véritable classique du monde algorithmique et il en existe des myriades. Par exemple, le voyageur de commerce, qui est censé relier un certain nombre de villes par l'itinéraire le plus court avant de revenir à sa ville de départ, ressemble au problème précédent et reste toujours un des casse-tête favoris des informaticiens passionnés par les problèmes d'optimisation. En effet, le nombre de parcours possibles est de l'ordre de la factorielle du nombre de villes. Or, la factorielle est une fonction de nombre entier qui part en vrille très rapidement. Elle vaut 24 pour 4, mais 265252859812191058636308480000000 pour 30. Ces mêmes informaticiens essaient donc de mettre au point des algorithmes leur permettant de trouver assez rapidement le parcours le moins long (le meilleur itinéraire dans l'absolu devient très vite hors de portée, même pour le plus puissant des ordinateurs) pour des problèmes de plusieurs milliers de villes. Ils s'affrontent ainsi dans des olympiades informatiques où les organisateurs comparent les propositions algorithmiques en matière de qualité de la solution (la longueur du parcours final obtenu) et/ou la durée d'exécution du programme pour y parvenir. Et, de fait, les algorithmes génétiques sont très prisés pour ce type de problème. Cela semble plutôt une bonne idée de combiner deux itinéraires prometteurs afin d'en générer un troisième qui le serait encore plus.

En quoi le voyageur de commerce est-il différent du problème précédent et, dès lors, rend l'algorithme de recherche A^* inutilisable, obligeant à recourir à des algorithmes d'optimisation devenus fameux en IA tels l'algorithme génétique ou l'algorithme fourmis ? Pourquoi l'optimisation plutôt que la recherche ? Car ce problème exige d'être traité dans sa globalité, il n'a plus rien de Markovien, car toutes les villes doivent être parcourues (ce qui n'est évidemment pas le cas du problème précédent), un choix de villes effectué au début contraignant tout l'itinéraire jusqu'à la fin. Aucune stratégie, progressant dans l'arbre des lieux géographiques, en permettant l'éventuelle sélection de ces lieux au fur et à mesure de la descente dans l'arbre, ne mènera à une solution optimale. Il faut évaluer chaque itinéraire proposé dans sa globalité et parcourir de la manière la plus efficace qui soit un immense espace de recherche dans lequel chaque point devient un de ces itinéraires. Et l'on retrouve l'importance de bien saisir la nature du problème afin d'effectuer le bon choix algorithmique, la recette du bon informaticien. Les algorithmes de recherche qui progressent séquentielle-

ment ne marchent plus ici, car l'optimalité du reste de l'itinéraire peut nous obliger à modifier des choix réalisés précédemment. Ainsi, l'algorithme génétique, envisageant les solutions dans leur globalité, propose des populations d'itinéraires qu'il fait évoluer en sélectionnant les meilleurs, puis en les mutant et les recombinaut afin d'en générer de meilleurs encore. On se répète, mais nous concevons bien qu'en mélangeant deux itinéraires, il est possible d'en découvrir un supérieur qui mélange les intéressantes premières étapes du premier avec les tout aussi intéressantes dernières étapes du deuxième.

Dans la nature aussi, il existe des stratégies algorithmiques de plus court chemin, mais de facture très différente. L'informatique et les systèmes biologiques ont en commun de présenter différents niveaux d'abstraction fonctionnelle. Tous deux peuvent présenter des fonctionnalités nouvelles, parfois complexes, à un niveau supérieur, alors que la réalisation au niveau juste inférieur se fait par interactions et itérations multiples de processus nettement plus simples. Par le dépôt à débit constant d'une substance odorante appelée phéromone, une colonie de fourmis peut, entre deux chemins, se concentrer progressivement sur le plus court. Cette fois, la propriété de « plus court » entre ces deux chemins émerge du comportement de la colonie, car nulle fourmi, prise isolément, ne mesure la longueur des chemins et apprécie l'avantage qu'il y a à emprunter le plus court. Ce que chaque fourmi fait est, quand le choix se présente entre deux chemins, d'emprunter celui qui « sent » le plus. Il est simple de comprendre comment, petit à petit, les fourmis, allant et venant sur les deux chemins, c'est le plus court des deux qui se charge le plus vite de cette phéromone odorante, incitant de plus en plus de fourmis à parcourir celui-là. Cet effet s'amplifie ainsi de lui-même (plus le chemin sent, plus les fourmis l'empruntent et, plus elles l'empruntent, plus le chemin sent), faisant du choix du plus court des deux chemins une fonctionnalité émergente de la colonie de fourmis. Le contraste est en effet saisissant entre la propriété de « plus court chemin » et les deux règles de comportement élémentaires suivies par les fourmis, sans rapport aucun avec cette même propriété : déposer ses phéromones et, entre deux chemins, choisir le plus odorant.

Marco Dorigo, un de nos collaborateurs les plus proches dans le laboratoire qu'Hugues Bersini dirige, a trouvé dans ces colonies de fourmis l'inspiration à la mise au point d'algorithmes très puissants de découverte du plus court chemin, pour aller d'un point à un autre dans des environnements aussi complexes que le trafic en ville (le problème du voyageur de commerce, par exemple) ou le routage sur Internet. L'ordinateur révèle, dans un labyrinthe de chemins que peuvent emprunter des petits points noirs (fourmis virtuelles) se déplaçant sur l'écran, comment, après un certain temps, tous ces points noirs se concentrent sur un même tracé, le plus court. À l'instar des fourmis, les messages circulant dans le réseau Internet laissent derrière eux comme une substance, renseignant les autres messages sur la longueur du chemin parcouru. Petit à petit, et à choisir entre deux itinéraires, les messages sélectionneront celui qui les amènera plus rapidement au but.

Le logiciel Python de la recherche du plus court chemin en pratique

Le projet logiciel du plus court chemin se présente de la manière suivante.

- Le fichier `GUI.py` s'occupe d'illustrer graphiquement les différentes étapes de la recherche.
- Différentes instances de graphes sont proposées dans le dossier `datasets`.
- Le fichier `FileHandler.py` contient le code de chargement des instances de graphes.
- Le fichier `Algorithm.py` contient l'implémentation de la recherche du plus court chemin au sein d'un graphe en utilisant l'algorithme A*, avec les méthodes unidirectionnelle et bidirectionnelle.
- Le fichier `main.py` sert à lancer le programme principal.

La structure de l'algorithme est la suivante : on retrouve une liste ouverte sous forme d'une queue de priorité (de type `heapq`) et un dictionnaire de l'historique des nœuds visités, ainsi que leur coût g associé. Pour le reste, l'implémentation est très similaire à celle du taquin. On récupère la dernière valeur de la queue, on vérifie si ce nœud est valide (non visité ou visité avec un coût supérieur) et on calcule la fonction $f=g+h$, avec g le coût (la distance déjà parcourue pour atteindre le nœud) et h , l'heuristique qui donne l'information sur la distance du nœud à celui d'arrivée.

```
def solve(self):
    """
    Solve the problem using the unidirectional A*
    """
    start = 0
    goal = 1
    # items in priority queue:
    # (heuristic (to minimize), current node number, [list of nodes] )
    q = [(0, start, [start])]
    heapq.heapify(q)
    g_scores = {start: 0}
    iterations = -1

    while len(q) != 0:
        iterations += 1
        self.history.append(copy.deepcopy(q))
        self.history[-1].sort()
        (_, current_node, list_of_nodes) = heapq.heappop(q)
        self.logger.debug(
            f"Iteration {iterations}, current node: {current_node}, list of nodes: {list_of_nodes}"
        )
        if current_node == goal:
            self.path = list_of_nodes
            self.cost = g_scores[current_node]
            break
```

```

# explore neighbours of the current node
for n in list(self.G.adj[current_node]):
    weight = self.G.edges[current_node, n]["weight"]
    g = g_scores[current_node] + weight
    f = g + self.heuristic(n, goal)
    # if this node hasn't been visited yet or the travel cost is
    # smaller, adds it to priority queue
    if n not in g_scores or g < g_scores[n]:
        heapq.heappush(q, (f, n, list_of_nodes + [n]))
        g_scores[n] = g

if len(self.path) == 0:
    self.logger.warn("No solution found")
    return
self.logger.info(
    f"Found path to goal with cost {self.cost:.2f} in {iterations} iterations")
self.logger.info(f"Path: {self.path}")

```

Les différentes heuristiques utilisables sont définies dans la méthode `heuristic` de la classe `Algorithm`:

```

def heuristic(self, a, b):
    """
    Return heuristic between node a and b
    """
    node_a = self.vertices[a]
    node_b = self.vertices[b]
    dx = abs(node_a[0] - node_b[0])
    dy = abs(node_a[1] - node_b[1])
    if(self.heuristic_type == Heuristic.MANHATTAN):
        return (dx + dy)/2
    if(self.heuristic_type == Heuristic.EUCLIDIAN):
        return (dx**2 + dy**2)**(1/2)
    if(self.heuristic_type == Heuristic.CHEBYSHEV):
        return max(dx, dy)
    if(self.heuristic_type == Heuristic.DIJKSTRA):
        return 0
    else:
        return (dx + dy)/2

```

On y retrouve les distances de Manhattan, euclidienne, de Chebyshev et une option Dijkstra qui permet de mettre l'heuristique h à 0 et ainsi de simuler l'utilisation de l'algorithme de Dijkstra à la place de A^* .

La version bidirectionnelle fonctionne de manière similaire, avec deux chemins indépendants commençant l'un du côté du départ et l'autre du côté de l'arrivée, jusqu'à se croiser sur un nœud.

```

def solve_bidirectional(self):
    """
    Solve the problem using the bidirectional A* in a sequential way
    """
    start = 0
    goal = 1
    direction = 0 # 0 or 1

    def other(a):
        return int(not a)

    q = [[(0, start, [start])], [(0, goal, [goal])]]
    heapq.heapify(q[0])
    heapq.heapify(q[1])
    g_scores = [{start: 0}, {goal: 0}]
    save_path = [{start: []}, {goal: []}]
    node_goal = [goal, start]

    keep_searching = True
    iterations = -1

    while len(q) != 0 and keep_searching:
        iterations += 1
        self.history.append(copy.deepcopy(q[direction]))
        self.history[-1].sort()
        (_, current_node, list_of_nodes) = heapq.heappop(q[direction])
        self.logger.debug(
            f"Iteration {iterations}, current node: {current_node}, direction: {direction},
list of nodes: {list_of_nodes}"
        )
        for n in list(self.G.adj[current_node]):
            weight = self.G.edges[current_node, n]["weight"]
            g = g_scores[direction][current_node] + weight
            f = g + self.heuristic(n, node_goal[direction])
            if n not in g_scores[direction] or g < g_scores[direction][n]:
                if n in save_path[other(direction)]:
                    merged_path = copy.deepcopy(save_path[other(direction)][n])
                    merged_path.reverse()
                    self.path = list_of_nodes + merged_path
                    self.cost = g + g_scores[other(direction)][n]
                    if self.path[0] == goal:
                        self.path.reverse()
                    self.history.append(
                        copy.deepcopy(
                            [(self.cost, goal, self.path)] + q[direction]
                        )
                    )
                    keep_searching = False
                    break
            else:
                save_path[direction][n] = list_of_nodes + [n]
                heapq.heappush(q[direction], (f, n, list_of_nodes + [n]))
                g_scores[direction][n] = g

```

```
direction = other(direction)

if len(self.path) == 0:
    self.logger.warn("No solution found")
    return
self.logger.info(
    f"Found path to goal with cost {self.cost:.2f} in {iterations} iterations"
)
self.logger.info(f"Path: {self.path}")
```

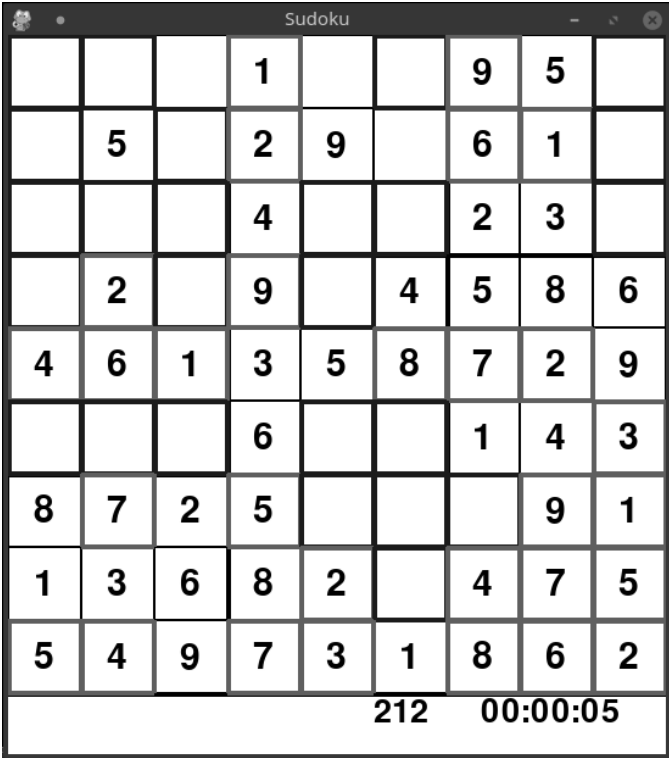
Il est à noter que l'implémentation bidirectionnelle est présentée ici à titre d'information. Elle ne présente pas d'avantage par rapport à son homologue unidirectionnelle pour plusieurs raisons : elle est purement séquentielle, donc ne bénéficie pas d'une éventuelle accélération qu'une version *multi-thread* aurait apportée. De plus, elle ne garantit pas de trouver la solution optimale, puisque la recherche s'arrête lorsque les deux chemins se croisent. On peut donc dire que le chemin trouvé est le plus court entre le départ et le point de rencontre et entre l'arrivée et le point de rencontre, mais pas nécessairement le plus court sur l'ensemble du chemin.

3

Jouons au sudoku

Figure 3–1

Capture d'écran du logiciel Python mis à votre disposition pour le problème du sudoku.



			1			9	5	
	5		2	9		6	1	
			4			2	3	
	2		9		4	5	8	6
4	6	1	3	5	8	7	2	9
			6			1	4	3
8	7	2	5				9	1
1	3	6	8	2		4	7	5
5	4	9	7	3	1	8	6	2
212						00:00:05		

Principe – Choix algorithmique

Ce jeu, comme celui des mots croisés, est devenu un des classiques des distractions de vacances sur la plage quand, après la troisième baignade, l'ennui vous guette. On le trouve dans de nombreux journaux et il distrait tout autant les passagers des métros que des trains bondés. Dans sa version la plus répandue, il consiste en une grille de neuf cases sur neuf, chacune pouvant contenir un seul chiffre de 1 à 9 (figure 3–2). L'objectif consiste à remplir la grille sans qu'un même chiffre se retrouve deux fois sur une même ligne, sur une même colonne, ou à l'intérieur des neuf petites sous-grilles 3×3 qui composent la grille. Le jeu démarre par un ensemble de chiffres déjà correctement préinstallés, la complexité s'accroissant de manière inversement proportionnelle à ce nombre.

Figure 3–2

Une grille de sudoku à remplir par un algorithme d'IA.

							5	
	5			9				
			4			2	3	
					4	5	8	6
			3	5				9
			6			1	4	
8		2						
1	3	6	8					
		9			1			
0					00:00:10			

Le sudoku est à nouveau un parfait problème pour interroger les trois piliers fondamentaux de l'IA et notamment ici comparer les approches « recherche » et « optimisation », jusqu'à même les hybrider comme nous en discuterons à la fin de ce chapitre. À la différence du taquin, ce qui importe dans le cas du sudoku est la solution et nullement le chemin pour y parvenir, c'est-à-dire la séquence des chiffres à ajouter dans les cases pour aboutir à la bonne configuration. Bien sûr, cette séquence a son importance, car les premiers chiffres conditionneront les suivants, mais elle ne fait pas partie en tant que telle de la solution. En substance, toutes les séquences qui conduiraient

à la même configuration désirée se valent. Dès lors, il pourrait sembler qu'une telle définition du problème conduise plus spontanément à choisir un algorithme d'optimisation de type « algorithme génétique », où seul l'aboutissement importe et que nous allons directement présenter. On acceptera aussi qu'une option algorithmique possible, d'une grande banalité mais la moins intéressante qui soit, consisterait juste en la génération de configurations au hasard jusqu'à trouver celle qui satisfait toutes les contraintes. Cela marche mais après un temps infini, prouvant encore s'il le fallait que toute l'intelligence dans le choix de l'algorithme se loge dans la recherche du bon compromis entre la qualité de la solution, le temps mis pour l'obtenir et la complexité de l'algorithme qui y conduit.

Ce jeu va nous permettre d'approfondir quelque peu l'algorithme génétique déjà esquissé dans le cas du taquin. Il se trouve dans le code Python qui accompagne ce chapitre. Il démarre avec une population de X candidats solutions. Comment générer ces X candidats ? Une possibilité est de les générer en respectant la contrainte des chiffres différents dans chaque ligne. Il suffit donc de remplir toutes les lignes en tirant aléatoirement des chiffres entre 1 et 9, à l'exception de ceux déjà préinstallés et de ceux trouvés jusqu'ici. On aura donc une population de X candidats pour lesquels, au moins, toutes les lignes sont admissibles car elles se composent de chiffres différents.

On rappelle que l'algorithme consiste à répéter quatre opérations :

- 1 sélection des meilleurs ;
- 2 recombinaison ;
- 3 mutation ;
- 4 évaluation de la nouvelle population obtenue à l'issue des trois opérations précédentes.

Décortiquons plus précisément ces étapes. Sur les X candidats, on peut garder les $X-N$ meilleurs. Puis, à partir de ces $X-N$, on procède à l'étape de la recombinaison, consistant à choisir deux candidats et à les recombiner afin d'en générer deux autres. L'opération de recombinaison, sans doute l'originalité première des algorithmes génétiques, fait relativement sens ici, car on acceptera que deux candidats puissent s'avérer intéressants pour des raisons différentes et qu'en les recombinant, on en découvrira peut-être un troisième qui prend le meilleur des deux. Cette recombinaison se fait idéalement en échangeant un nombre de lignes tirées au hasard entre les deux candidats. Cet échange de lignes nous assure que les chiffres préinstallés et la contrainte des chiffres différents sur une même ligne sont respectés. La mutation, quant à elle, s'effectue en échangeant deux chiffres tirés d'une même ligne, à l'exception bien sûr des chiffres préinstallés (figure 3-3).

Figure 3-3

Exemple de mutation génétique. On échange deux chiffres d'une même ligne.



Ce faisant, une population de X nouveaux candidats est créée. Dans la stratégie élitiste, on conserve les $X-N$ meilleurs, mais les opérations de recombinaison et de mutation permettent d'en générer N nouveaux. Il est aussi possible, dans une moindre proportion, de régénérer des candidats

à partir de rien, pour conserver une diversité suffisante dans la population, vu que les opérations de recombinaison et de mutation décrites plus haut se contentent de manipuler et brasser un matériel déjà existant.

Il reste maintenant à définir la *fitness*, c'est-à-dire la qualité, de chacun des candidats rencontrés sur le chemin de la solution. Les contraintes sur les lignes étant assurées, la qualité se mesurera uniquement sur les colonnes et sur les blocs internes. La plus simple des possibilités consiste à diminuer la fonction de coût de 1 à chaque fois que le même chiffre se retrouve soit dans une colonne soit dans un bloc.

Lorsque l'on exécute cet algorithme, surtout pour des cas compliqués, c'est-à-dire avec peu de chiffres préinstallés, on se trouve systématiquement confronté à la problématique récurrente des algorithmes d'optimisation : la présence des optima locaux (tous les plus petits sommets dans le relief montagneux). Il s'agit de mauvaises solutions mais qui, par leur pouvoir d'attraction, entraînent l'algorithme sur de mauvaises voies, dont il peut avoir énormément de mal à se dégager. L'unique manière de contrer une telle difficulté est de mieux balancer encore « exploration » (permise par les composantes aléatoires de l'algorithme) et « exploitation » (déterministe cette fois et qui poursuit sur les chemins les plus prometteurs, ce que garantit la sélection des meilleurs). Plus d'exploration entraîne aussi plus de temps de calcul, nous confrontant sans surprise au dilemme le plus omniprésent des algorithmes d'IA : une solution de piètre qualité obtenue très rapidement contre une solution de bien meilleure qualité, mais résultant d'un coût et d'un temps de calcul bien plus importants. Nous verrons qu'une amélioration très classique des algorithmes génétiques consiste à leur joindre un algorithme de recherche locale, preuve à nouveau des vertus de l'hybridation dans le monde de l'IA. Mais comment et pourquoi utiliser un algorithme de recherche apparenté à A^* pour résoudre le sudoku ?

Cet algorithme, que nous avons déjà décrit et testé pour les applications précédentes, résout un problème par l'élaboration d'un arbre dont chaque nœud est un état du jeu et chaque arête, ou transition entre deux nœuds, un coup possible. Ici, le coup est simplement l'ajout d'un chiffre dans une des cases vides, quelle qu'elle soit. À chaque fois qu'une case est sélectionnée, on approfondit donc l'arbre en créant autant de nouveaux nœuds qu'il y a de chiffres possibles pour cette case. Toutefois, à quoi bon un tel algorithme dès lors que le chemin pour parvenir à la solution importe peu ? Car, par l'utilisation d'une heuristique adéquate, cet algorithme a l'avantage de véritablement « séquentialiser » la démarche résolutive, de vraiment prendre en compte la dépendance entre le prochain coup à jouer et tous ceux qui précèdent, ce que l'algorithme génétique réussit beaucoup moins facilement. De nombreuses heuristiques sont ici possibles, la plus simple étant de favoriser la case pour laquelle il reste le moins d'alternatives possible. Une autre plus sophistiquée consiste à sélectionner la case pour laquelle il reste le moins d'alternatives possible pour toutes les cases présentes sur la même ligne, la même colonne et le même bloc. Une autre encore, très proche de la précédente, prend en compte tous les blocs traversés par la ligne et la colonne de la case en question.

Il est important de dire que, bien qu'inspiré de A^* , l'algorithme en question s'en détache quelque peu, car autorisant le backtracking, c'est-à-dire de rebrousser chemin pour repartir d'une autre case promiseuse si la continuation d'une même piste mène à un cul-de-sac, c'est-à-dire un état du sudoku pour lequel plus aucune case ne peut se remplir d'un nouveau chiffre. Surtout, l'algorithme de recherche utilisé ici s'apparente plutôt à un *best-first search*, car l'heuristique (si on prend la plus effi-

cace, celle qui prend le nombre de chiffres possibles restant dans une case) ne donne pas réellement une information sur la « position » du nœud d'arrivée, mais permet une recherche en profondeur plus efficace, en commençant par les nœuds qui ont le minimum de branchements. Cette heuristique minimise donc le nombre d'erreurs dans la recherche dans l'arbre et réduit le recours au backtracking. Nous l'avons dit précédemment, dans un cas comme celui-ci, le chemin importe peu ; seule la solution finale compte. Pour la plupart, les solutions de type recherche appliquées au sudoku parlent alors plus volontiers d'une méthode dite de backtracking plutôt que formellement de A*.

Figure 3-4

Le sudoku par l'algorithme de recherche « best-first search », les cases marquées en gris sont celles qui ont pu être remplies et celles vides marquées en noir ont fait l'objet d'un backtracking.

7	3	6	1	2	4	5	9	
			6					4
					8	6		
		9						
	6	8	9	7	2	3	1	5
1		5					7	9
			7	6	3			
9			8	5	1			
	8		2	4	9			
129						00:00:05		

Les auteurs de la tentative de résolution précédente, celle basée sur l'algorithme génétique, ont fait le constat qu'ils pouvaient considérablement en améliorer les performances s'ils lui adjoignaient une méthode de backtracking locale. À chaque candidat obtenu par l'algorithme génétique, ils permettaient quelques itérations d'une recherche du type précédemment décrit. L'itération successive de l'algorithme génétique démarre alors à partir de cette population améliorée. Cette nouvelle hybridation (on se rappelle que ce type d'hybridation fait les beaux jours de l'IA) est un grand classique de l'optimisation : une recherche exploratoire globale jointe à une micro-recherche afin d'améliorer « localement » une solution. On décrit souvent cette hybridation comme un algorithme lamarckien, du nom du célèbre opposant à Darwin considérant que, de leur vivant, les espèces cherchaient encore à améliorer leur fitness par des ajustements qui pouvaient par la suite se

transmettre génétiquement. Les girafes allongeaient leur cou de leur vivant en tentant d'atteindre les feuilles les plus hautes de l'arbre et, selon Lamarck, leurs descendants héritaient de ces efforts. Là encore, différentes options restent possibles dans la façon de greffer une recherche locale à celle plus globale, mais l'idée a fait son chemin et on la retrouve très souvent mise en œuvre dans de nombreux algorithmes d'IA.

Le logiciel Python du sudoku en pratique

Le projet du sudoku se présente sous la forme suivante.

- Le fichier `sudoku_alg.py` contient les fonctions propres au sudoku comme la validation d'une grille ou la recherche de cases vides.
- Le fichier `main.py` contient la gestion de l'interface graphique du jeu et le programme principal.
- Le fichier `search.py` contient la stratégie par recherche.
- Le fichier `genetic_algorithm.py` contient la stratégie par algorithme génétique.

Recherche best-first

Dans le fichier `search.py`, on trouve la fonction qui permet de résoudre le sudoku grâce à une méthode de recherche qui se base sur un algorithme récursif.

```
def visualSolve(self, wrong):
    """
    Solve the board with Search
    """
    if not isFull(self.game.board):
        possibilities = reducePossibilities(self.game.board, self.possibilities)
        # Adapts the possible numbers for each cell
        self.heuristics = updateHeuristics(
            self.game.board, self.heuristics, possibilities
        )
        cell = getCellToExplore(self.heuristics)
        # This gets the cell with the best heuristic
        for number in possibilities[cell[0]][cell[1]][
            possibilities[cell[0]][cell[1]] >= 1
        ]:
            for event in pygame.event.get():
                # So that touching anything doesn't freeze the screen
                if event.type == pygame.QUIT:
                    sys.exit()
            self.game.tries += 1
            self.game.board[cell[0]][cell[1]] = number
            self.game.tiles[cell[0]][cell[1]].value = number
            self.game.tiles[cell[0]][cell[1]].correct = True
```

```

# pygame.time.delay(63) #show tiles at a slower rate

self.game.redraw({}, wrong, time.time() - self.startTime)
if self.visualSolve(wrong):
    # Then, we try to solve the sudoku starting from the updated board
    return True
self.game.board[cell[0]][cell[1]] = 0
self.game.tiles[cell[0]][cell[1]].value = 0
self.game.tiles[cell[0]][cell[1]].incorrect = True
self.game.tiles[cell[0]][cell[1]].correct = False
# pygame.time.delay(63)
self.game.redraw({}, wrong, time.time() - self.startTime)
return False
return True

```

On démarre en calculant les possibilités de jeu pour chaque cellule, c'est-à-dire les chiffres qui sont des actions valides pour chaque case. On définit ainsi les différents nœuds fils possibles par rapport à l'état actuel du jeu (au début, la grille remplie avec les chiffres de départ). La méthode `updateHeuristics` va alors mettre à jour le tableau des valeurs d'heuristiques pour chaque case. On utilise ici, comme heuristique de base, le nombre de chiffres possibles dans chaque case. La méthode `getCellToExplore` va ensuite sélectionner la meilleure case à explorer suivant la valeur de l'heuristique.

On parle ici de best-first search car on va toujours explorer la case qui présente la meilleure heuristique. La différence avec A^* , vu précédemment, est qu'ici il n'y a pas vraiment de notion de distance, car toutes les solutions auront nécessité le même nombre de coups. De plus, l'heuristique nous informe localement sur le meilleur nœud à visiter, mais n'offre aucune garantie de qualité ou de validité future de la solution. On explore simplement le nœud le moins contraint afin de minimiser la taille de l'arbre à parcourir. Si on commence par les nœuds avec le moins de branches, on aura statistiquement moins de chance de se tromper dès le début.

Après avoir sélectionné la case, on met à jour la grille de jeu et on relance la fonction. Si on ne trouve pas de case à remplir (la fonction retourne `False`), c'est que la grille n'est pas valide et on annule les coups successifs en remontant récursivement la chaîne des coups. On essaie alors d'autres chiffres (grâce à la boucle `for number in`) jusqu'à retrouver une grille valide (qui retourne `True`) et pouvoir à nouveau lancer la fonction (on relance la récursion `if self.visualSolve(wrong)`).

Algorithme génétique

Dans le fichier `genetic_algorithm.py`, on trouve plusieurs classes pour résoudre le problème grâce à un algorithme génétique. La classe `Genetic_Algorithm` instancie le problème ainsi que la population de base de l'algorithme. La classe `Population` définit une population et permet d'évaluer cette dernière. Enfin, la classe `Chromosome` définit un individu d'une population et donc une solution candidate au problème.

Dans le problème du sudoku, on définit une solution candidate comme une grille remplie, mais pas forcément valide. La fitness ou le coût d'une solution candidate est alors défini(e) comme le nombre de chiffres en double dans une ligne/une colonne/un bloc et donc invalides dans cette

solution. Le but de l'algorithme génétique va être de minimiser cette valeur en procédant par sélections-croisements-mutations afin de converger vers une solution valide (avec un coût nul).

```
def visualSolve(self, wrong):
    """
    Solve the board using a genetic algorithm
    """
    genetic_alg = Genetic_Algorithm(
        self.max_nb_gen_with_same_score, self.board.board
    )
    generation = 0
    finished = False
    while not finished:
        for event in pygame.event.get():
            # so that touching anything doesn't freeze the screen
            if event.type == pygame.QUIT:
                sys.exit()
        finished = genetic_alg.generate_next_generation()
        generation += 1
        pygame.display.set_caption("Generation {}".format(generation))
        best_board = genetic_alg.population.chromosomes[0].board
        for i in range(9):
            for j in range(9):
                self.board.tiles[i][j].value = best_board[i][j]
                self.board.tiles[i][j].incorrect = not valid(
                    best_board, (i, j), best_board[i][j]
                )

        self.board.redraw(
            {}, wrong, time.time() - self.startTime, generation
        )

    self.board.redraw(
        {}, wrong, time.time() - self.startTime, generation
    )
```

La fonction `visualSolve` coordonne l'algorithme. On commence par créer une population initiale avec un certain nombre de solutions prises au hasard. Ces dernières sont générées par la fonction `generate_initial_population` (dans le constructeur de la classe `Genetic_Algorithm`), qui va simplement distribuer les chiffres de 1 à 9 dans les lignes, en fonction des chiffres déjà présents dans la grille initiale. Cela aura pour effet de générer des grilles dont les lignes sont valides, mais pas les colonnes ni les blocs. Une solution initiale totalement aléatoire augmenterait encore plus la difficulté de convergence de notre algorithme. Ici, une génération aléatoire avec une des contraintes déjà respectée est plus facile à mettre en œuvre et n'a qu'un faible coût algorithmique.

Dans la boucle `while`, on va ensuite se charger de créer les générations de solutions candidates successives, tester leur validité et sélectionner la meilleure pour l'afficher, jusqu'à convergence sur une solution valide ou épuisement (pas de solution trouvée après un temps long). Lors de la création d'une nouvelle génération, on utilise la fonction `generate_next_generation` suivante :

```

def generate_next_generation(self):
    """
    Generate the future generation by applying elitism, mutation and selection.
    """
    print("best score: ", self.population.chromosomes[0].score)

    # Elitism: Keep the best results from previous generation
    elite_chromosomes = self.population.chromosomes[
        0 : int(self.generation_size * self.elitism_percentage)
    ]
    random.shuffle(elite_chromosomes)
    new_chromosomes_from_crossover = []

    for _ in range(self.generation_size):
        (
            new_chromosome1,
            new_chromosome2,
        ) = copy.deepcopy(random.sample(self.population.chromosomes, k=2))

        self.population.crossover(new_chromosome1, new_chromosome2)

        # 50% chance to apply mutation on a chromosome from the crossover
        if random.randint(0, 9) < 5:
            new_chromosome1.apply_mutation()

        new_chromosome1.evaluate_fitness_score()
        new_chromosome2.evaluate_fitness_score()

        new_chromosomes_from_crossover.append(new_chromosome1)
        new_chromosomes_from_crossover.append(new_chromosome2)

    self.selection(elite_chromosomes, new_chromosomes_from_crossover)

    self.compute_score_and_escape_from_local_optima()
    return self.best_score == 0

```

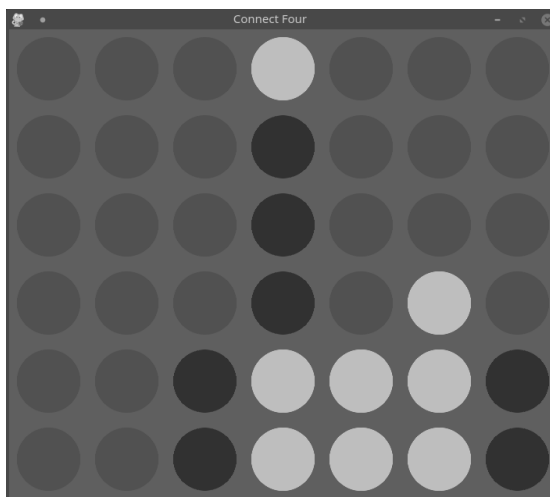
On commence par la phase d'élitisme, pour laquelle on garde seulement une fraction des meilleurs candidats. Ensuite, on crée de nouveaux candidats par croisements et mutations sur ces génomes « élites ». On poursuit par la phase de sélection, pour laquelle on garde une partie des élites, une partie des « enfants » (croisements et mutations des élites) et quelques nouveaux génomes aléatoires, afin de garantir une certaine diversité et éviter les minima locaux. Nous invitons le lecteur à consulter les fonctions `crossover` et `apply_mutation` dans le code source pour en savoir plus.

Au bout d'un certain nombre de générations, on converge vers un minimum. Il n'en demeure pas moins difficile d'atteindre la solution optimale. Le jeu du sudoku a ceci de particulier qu'une erreur dans un chiffre peut entraîner une refonte quasi totale de la grille. En termes d'espace de recherche, cela se caractérise par des minima locaux très profonds desquels il est difficile de sortir sans modifier une grande partie de la grille.

4

Jouons à Puissance 4

Figure 4–1
Capture d'écran du logiciel
Python mis à votre disposition
pour le problème
du Puissance 4.



Principe – Choix algorithmique

Les mêmes hésitations et difficultés de choix algorithmique surgissent dans un autre type d'application qui a fait les beaux jours de l'IA : les jeux de société à deux adversaires, tels le tic-tac-toe (appelé aussi parfois pour d'étranges raisons le morpion), le Puissance 4, les dames, les échecs, le go, les jeux de cartes et tant d'autres... Et, là encore, nos étudiants se plaisent, sans toujours se

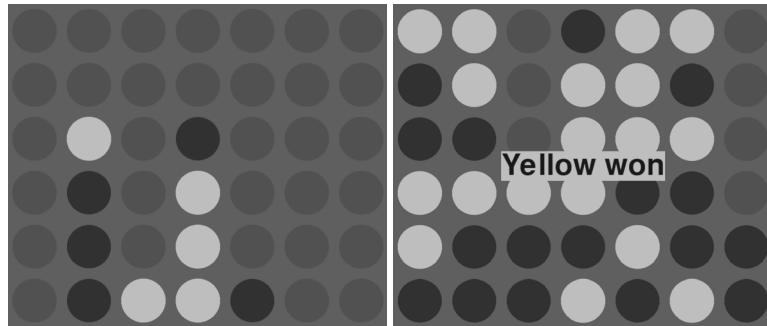
poser les bonnes questions, sans prendre le recul nécessaire, à se lancer dans l'écriture d'algorithmes de type recherche (ici, comme on le verra, l'algorithme Min-Max est le plus prisé), optimisation (comme à nouveau l'algorithme génétique), apprentissage par renforcement (le Deep-Q-learning marche tellement bien pour le jeu de go).

Dans le jeu du Puissance 4, bien plus accessible et facile à comprendre que le jeu de go, chaque joueur, en alternance, place un petit disque de sa couleur dans un emplacement vide se trouvant dans une colonne, dans le but ultime de créer des lignes verticales, horizontales ou diagonales de quatre de ces petits disques de même couleur, tout en empêchant son adversaire d'y parvenir également.

Figure 4-2

Le jeu du Puissance 4.

À droite, les jaunes (en gris)
ont fini par gagner la manche.



L'algorithme Min-Max, un autre ancêtre de l'IA que l'on doit au grand-père de ce domaine scientifique (de l'IA comme de l'informatique en général), John Van Neumann, et qui, dans la tradition première des algorithmes de recherche, fonctionne par l'élaboration d'un arbre de coups, est composé non seulement des coups du programme apprenant, mais également de ceux de son adversaire. À nouveau, la source d'inspiration première est évidente : notre cognition. Nous n'agissons pas différemment lorsqu'il s'agit de jouer : nous anticipons jusqu'à une certaine profondeur nos coups et ceux de l'adversaire.

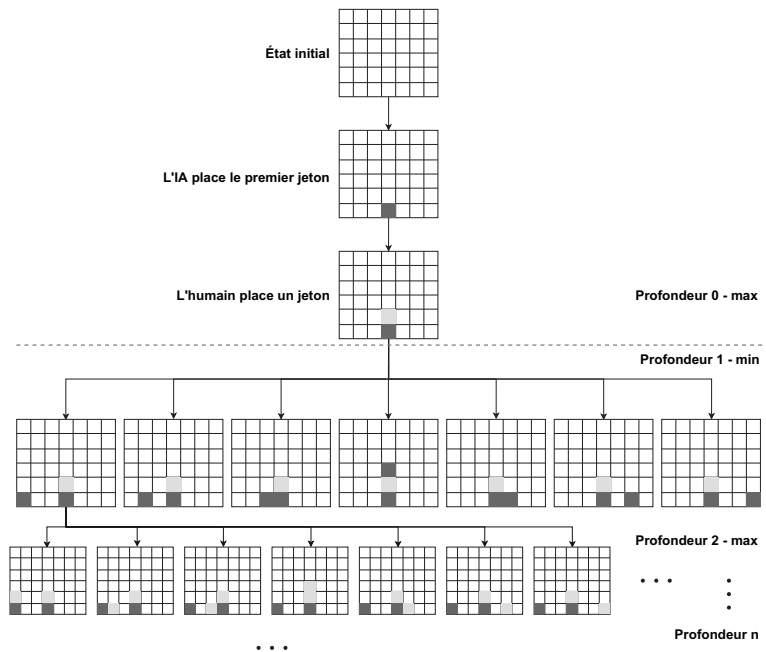
Le nom « Min-Max » vient du fait que, après une certaine profondeur d'anticipation, par exemple quatre coups de l'un et quatre coups de l'autre, les feuilles de l'arbre se voient attribuer à nouveau une évaluation heuristique tentant de prédire au mieux les chances de l'emporter pour le programme IA. Une heuristique possible pourrait être le nombre de lignes que l'IA peut encore réaliser duquel on soustrait le nombre de lignes possibles de son adversaire. Une autre possible heuristique consiste à attribuer une certaine récompense pour le nombre de disques que l'IA a réussi à juxtaposer (2 pour 2 et 3 pour 3) et soustraire ces mêmes nombres pour les juxtapositions des disques de l'adversaire. Sans surprise, de multiples évaluations sont possibles et de la qualité de celle retenue dépendra en grande partie la performance de l'algorithme. Cette évaluation réalisée pour toutes les feuilles terminales de l'arbre, il suffit alors de remonter le long de cet arbre, en prenant la valeur minimum des nœuds de l'adversaire (minimiser les chances de gagner pour l'adversaire) et la valeur maximum des nœuds du joueur IA (maximiser les chances de gagner pour l'IA), jusqu'à arriver au coup à jouer pour ce dernier ; d'où l'appellation « Min-Max ».

Il est clair que, plus l'évaluation heuristique est précise, meilleure est la prédiction de victoire et moins l'arbre se doit d'être profond. Dans le cas contraire, et la puissance de calcul aidant (c'est par

exemple tout à fait possible dans le cas du jeu du tic-tac-toe, le plus élémentaire de tous), il est possible d'imaginer atteindre les extrémités de l'arbre, les fins de partie, pour lesquelles les seules évaluations possibles sont +1 pour gagner, -1 pour perdre et 0 pour égalité. Dans le code Python mis à votre disposition, une possible fonction évaluation est proposée, mais de multiples alternatives sont évidemment possibles.

Figure 4-3

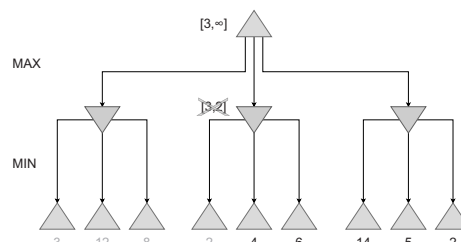
Illustration du fonctionnement de l'algorithme Min-Max pour le jeu de Puissance 4.



Lorsque la fonction d'évaluation est suffisamment précise et facile à déduire à partir de l'expérience humaine, de la connaissance et de la pratique ancestrale de ce jeu, l'algorithme Min-Max demeure imbattable en temps de calcul. Il est même possible et conseillé de l'accélérer en lui préférant sa version dite alpha-bêta, qui permet de ne pas explorer des parties inutiles de l'arbre, débouchant ainsi sur un très précieux gain en temps de calcul (figure 4-4). En effet, il ne vous aura pas échappé que plus on approfondit l'arbre de recherche, plus les calculs sont longs et plus le temps d'exécution s'accroît (exponentiellement avec la profondeur de l'arbre).

Figure 4-4

Exemple d'élagage alpha-bêta de l'arbre des coups. Dans cet exemple, il n'est pas nécessaire d'évaluer les nœuds 4 et 6 puisque 2 est déjà inférieur à 3.



C'est ce même algorithme qui permet au logiciel Deep Blue d'IBM d'avoir le dessus sur Kasparov il y a de nombreuses années, en réalisant des arbres Min-Max (la puissance de calcul aidant) allant jusqu'à une profondeur de 14 coups et des milliards de feuilles terminales à évaluer. De nombreux psychologues à l'époque en ont déduit que Kasparov devait mettre en œuvre des stratégies inconscientes afin de supprimer de son exploration consciente des milliards de coups possibles, coups qu'il avait d'ores et déjà mémorisés (et enfouis dans ses connexions neuronales) comme perdus d'avance (comme lorsque vous reconnaissez une personne chère en un clin d'œil au milieu d'une foule). Les coups gagnants deviennent comme des automatismes. Cela explique aussi le recours aux réseaux de neurones, comme nous le verrons plus loin, censés s'inspirer davantage de nos processus inconscients.

Comme nos étudiants l'ont à nouveau constaté, toute autre approche, de type pur algorithme d'optimisation génétique ou d'apprentissage serait peine perdue. L'approche « recherche » a toujours le dessus. Néanmoins, c'est justement dans ce compromis entre profondeur de l'arbre et précision de la fonction d'évaluation que peut venir se glisser un algorithme de type renforcement en complément et soutien du Min-Max ; par exemple, l'algorithme désigné comme MCTS (*Monte Carlo Tree Search*), en complément mais pas en substitut. Le renforcement vient en effet pallier à la fois l'explosion exponentielle découlant de la profondeur de la recherche et la difficulté d'identifier la bonne heuristique d'évaluation. Cela s'est produit avec le succès que l'on sait dans le cas du jeu de go (illustration de la figure 4-5). Pourquoi ? Car, justement, la découverte de cette évaluation heuristique dans le cas du go s'est avérée vraiment délicate. Pour de nombreux jeux de société (tel le backgammon aussi), elle ne fait pas l'objet d'une expression analytique qu'il soit possible de calculer aisément à chaque feuille de l'arbre. Comme cette valeur heuristique revient en fait à estimer les chances de victoire du joueur IA à partir d'une configuration donnée, il devient tout aussi possible de l'estimer en laissant le joueur IA l'apprendre, exécutant, lui comme son adversaire, des coups au hasard jusqu'à la fin de la partie. Ses chances de victoire peuvent être estimées en compilant le nombre de parties aléatoires gagnées par rapport au nombre de parties perdues. C'est ce que permet l'algorithme MCTS, qui doit à l'omniprésence du hasard dans l'algorithme son doux nom de « Monte-Carlo ».

Cet algorithme se base, avant de jouer chaque coup, sur une succession de quatre opérations (figure 4-6) : sélection, expansion, simulation et rétropropagation. Succession que l'on peut exécuter un certain nombre de fois avant de décider du prochain coup à jouer.

En partant de la configuration actuelle, la sélection se borne à choisir dans l'arbre existant un nœud à analyser davantage ou alors à en créer un nouveau qui permet d'approfondir l'arbre vers le bas. En fait, on descend dans l'arbre en sélectionnant les nœuds selon la mesure suivante qui combine la qualité du nœud (w_i est le nombre de parties gagnées) et la connaissance acquise sur ce même nœud (n_i est le nombre de simulations démarrées à partir de ce même nœud).

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}$$

Cette formule sous-tend le fameux compromis « exploration-exploitation » que l'on retrouve dans pratiquement tous les algorithmes d'apprentissage ou d'optimisation en IA. On se fiera d'autant plus à un nœud qu'il est prometteur (w_i est grand) et que cette connaissance est fiable (n_i est grand, il n'est donc plus nécessaire de s'y attarder). Une fois parvenu au bout de la descente, si tous les coups n'ont pas

Figure 4-5
Illustration du Monte Carlo Tree
Search appliqué au jeu de go.

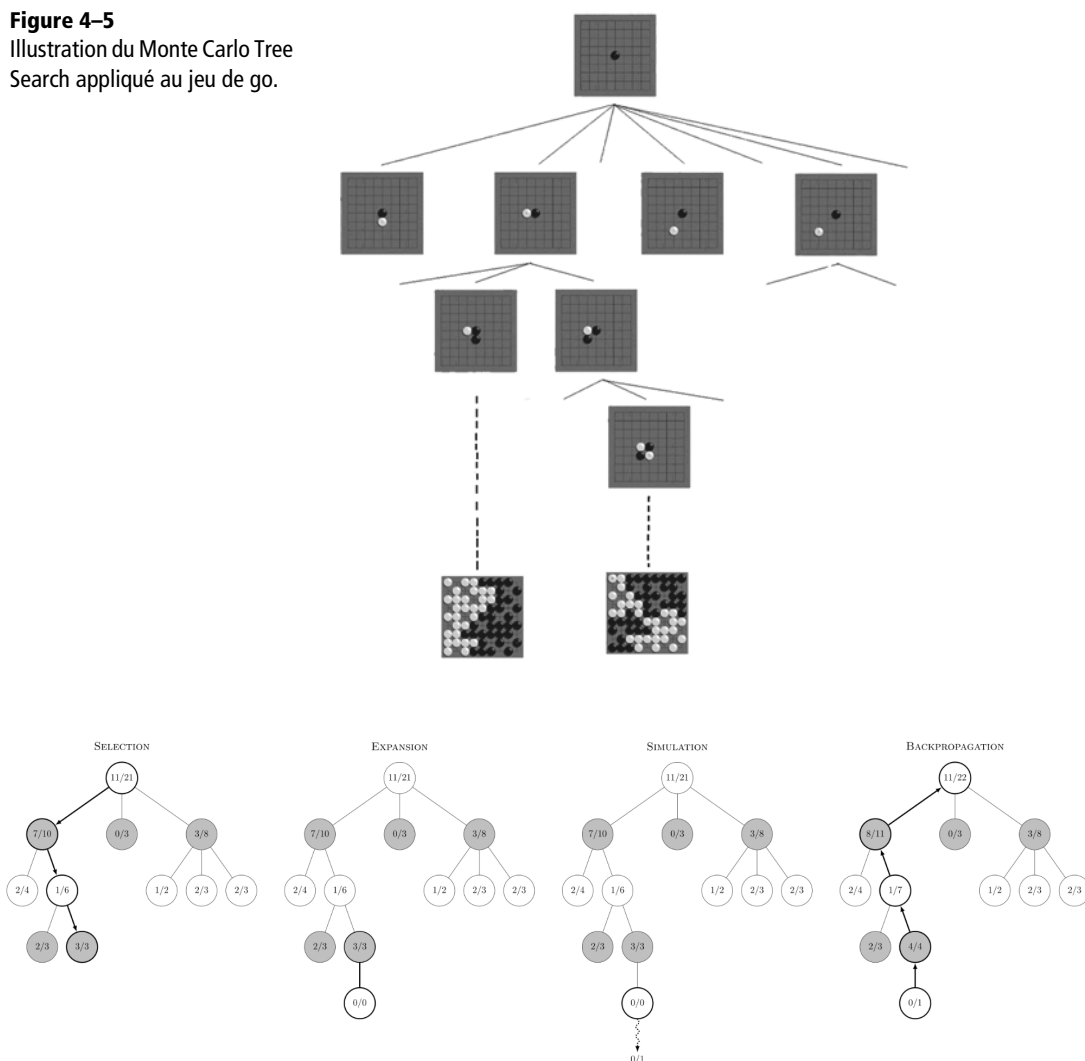


Figure 4-6 Illustration de la succession des quatre opérations qui sont la base de l'algorithme Monte Carlo Tree Search. By Rmoss92 – Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=88889583>

encore été explorés, on en ajoute un. C'est alors la phase d'expansion de l'arbre. On démarre une nouvelle simulation à partir de ce nœud à peine ajouté ou d'un nœud déjà existant et cette nouvelle simulation se déroule jusqu'à la fin d'une partie en faisant jouer les deux joueurs de manière purement aléatoire. Lorsque la partie est terminée et que le résultat est acquis (+1 gagné ou -1 perdu), on remonte cette information tout le long de la simulation jusqu'au nœud de départ, la configuration actuelle, dans laquelle on se trouve quand on démarre toute l'itération et les quatre phases qui la constituent.

Avant de décider du coup à jouer, on peut évidemment effectuer ces itérations un très grand nombre de fois (dépendant du temps imparti pour chaque coup à jouer). Il est aussi important de comprendre que, d'un coup à l'autre, comme dans tout algorithme de renforcement, tout ce qui a été appris est mémorisé et cumulé. On retrouve l'esprit de l'apprentissage par renforcement dans cette succession de coups, avant d'obtenir l'information capitale en fin de partie qu'il convient alors de rétropropager. Néanmoins, l'arbre d'anticipation des coups (venant plutôt des algorithmes de recherche) est aussi très important puisqu'il permet à la fois de choisir les nœuds à explorer et de guider la rétropropagation des résultats. C'est donc bien à une hybridation « recherche/apprentissage » que l'on a affaire ici. Quant à choisir le coup à jouer dans la configuration actuelle, cette sélection se fera au départ de la même formule : choisir le coup le plus prometteur et le moins incertain.

Ce que de nombreuses expériences démontrent lorsque l'on met en compétition les algorithmes Min-Max et MCTS est que, pour un temps d'exécution réduit, souvent le premier l'emporte sur le deuxième. Cependant, si l'on autorise davantage de temps, et pour la simple et bonne raison que l'algorithme Min-Max dépend exponentiellement de la profondeur de jeu là où MCTS n'en dépend que linéairement, MCTS peut prendre le dessus. On se rappelle également que Min-Max dépend grandement de la qualité de l'heuristique alors que MCTS n'en a cure, ce qui libère le développeur d'une démarche additionnelle, parfois difficile et sensible, consistant à identifier la bonne heuristique. On voit bien ici une combinaison gagnante entre deux IA, mélangeant la construction des arbres de coups qui résulte de l'aspect séquentiel du jeu, à l'apprentissage de la valeur heuristique qui permet de limiter la profondeur de ces arbres, mais dont l'explicitation (par exemple, même auprès des meilleurs experts du go) reste un défi majeur. L'apprentissage de cette valeur pour chaque configuration intermédiaire se déroule en répétant de multiples parties jouées au hasard à partir de cette configuration et en compilant la statistique des parties gagnées. Une fois encore, l'apprentissage pallie le manque d'expertise « consciente » des joueurs de go et compense par la puissance de calcul et les vertus du hasard ce que, sans doute, un plus gros effort de réflexion aurait pu produire tout autant de la part de ces mêmes experts.

La puissance de calcul disponible aujourd'hui dans nos ordinateurs, malgré les problèmes écologiques que cela pourrait poser dans le futur proche, réussit parfois à dérouter mêmes les meilleurs de nos experts. On le sait, le Coréen Lee Sedol, l'un des meilleurs joueurs de go au monde, s'est incliné face à AlphaGo (battu depuis par AlphaZero) et s'est dit surpris des coups¹ portés par le logiciel sortant totalement des habitudes des joueurs humains. L'IA a envoyé le jeu de go sur une autre planète non encore explorée même par les meilleurs joueurs humains (les seuls explorateurs sont les logiciels IA). Le Coréen le dirait, par ce déficit de compréhension actuelle, nous vivons aujourd'hui une régression de la pratique de ce jeu, en attendant que les meilleurs joueurs puissent, dans l'avenir, comprendre et se réappropriier ces nouvelles façons de le pratiquer. Nous en sommes là en IA, comme un Lee Sedol décontenancé, comme le démontrent à l'envi tous ces jeunes chercheurs qui s'empressent (trop) de faire tourner, enthousiastes, les logiciels opaques de réseaux de neurones profonds (du moins ceux que les Gafam – Google, Amazon, Facebook, Apple, Microsoft – ont rendus disponibles) pour des traitements du langage, des images, de la musique,

1. On peut d'ailleurs découvrir cette surprise à même le visage de Lee Sedol sur la vidéo disponible sur YouTube qui reprend cette partie.

des vidéos, sans se préoccuper outre mesure des connaissances scientifiques dont tous ces domaines ont fait et continuent de faire l'objet pour et par ceux qui cherchent à mettre des théories sur les faits. Or, c'est souvent ce recours à des algorithmes plus simples et plus compréhensibles qui permet de mieux saisir les tenants et aboutissants des problèmes affrontés et les pourquoi et comment des solutions algorithmiques ; en un mot, de demeurer scientifique avant tout.

À ce jour, le jeu du Puissance 4 a été complètement résolu. Il existe une base de données complète de tous les coups et configurations du plateau possibles, et il a été montré que, en jouant parfaitement, un joueur peut forcer une victoire avant le 41^e coup s'il joue en premier et au milieu. L'algorithme Min-Max avec *pruning* alpha-bêta est donc parfaitement adapté ; même si nos ordinateurs personnels ne parviennent pas à parcourir l'intégralité de l'arbre, un serveur de calcul en est aujourd'hui tout à fait capable.

Le logiciel Python de Puissance 4 en pratique

Le jeu du Puissance 4 est constitué des fichiers suivants :

- `main.py` permettant de lancer le jeu ;
- `connect4game.py` contenant le code de l'interface graphique du jeu ;
- `common.py` et `bot.py` contenant les classes d'abstraction communes pour les différentes IA ;
- `monte_carlo.py` et `minimax.py` contenant les différentes IA pour la recherche par Min-Max et celle par MCTS.

Min-Max

L'algorithme Min-Max est une technique classique de recherche dans le cas des jeux à deux joueurs à somme nulle. Il permet en théorie de trouver le coup optimal à jouer à chaque tour. Dans la pratique, pour la plupart des jeux, il n'est pas possible d'explorer tout l'arbre des possibilités et on fixe une profondeur maximale à atteindre avant d'estimer la qualité des positions et d'en déduire le meilleur coup probable. Pendant cette phase de recherche, on commence par évaluer les nœuds les plus profonds (soit à la profondeur maximale, soit jusqu'à la fin du jeu) et on évalue successivement les niveaux de l'arbre en remontant les valeurs de qualité maximum quand c'est le tour de l'IA ou de qualité minimum quand c'est le tour de l'adversaire. En effet, on essaie toujours de maximiser son propre gain, mais on prend aussi en compte un adversaire essayant de minimiser votre gain quand il joue.

```
def minimax(self, board, depth, alpha, beta, maximizingPlayer, pruning):
```

```
    """
```

```
    Main function of minimax, called whenever a move is needed.  
    Recursive function, depth of the recursion being determined  
    by the parameter depth.
```

```
    :param depth: number of iterations the Minimax algorithm will run for  
                  (the larger the depth the longer the algorithm takes)
```

```

:alpha: used for the pruning, correspond to the lowest value of the
       range values of the node
:beta: used for the pruning, correspond to the highest value of the
       range values of the node
:maximizingPlayer: boolean to specify if the algorithm should
       maximize or minimize the reward
:pruning: boolean to specify if the algorithm uses the pruning
:return: column where to place the piece
"""
valid_locations = self.get_valid_locations(board)
is_terminal = self.is_terminal_node(board)

if depth == 0:
    return (None, self.score_position(board, self._game._turn))
elif is_terminal:
    if self.winning_move(board, self._game._turn):
        return (None, math.inf)
    elif self.winning_move(board, self._game._turn * -1):
        return (None, -math.inf)
    else: # Game is over, no more valid moves
        return (None, 0)

column = valid_locations[0]

if maximizingPlayer:
    value = -math.inf
    turn = 1
else:
    value = math.inf
    turn = -1

for col in valid_locations:
    row = self.get_next_open_row(board, col)
    b_copy = copy.deepcopy(board)

    self.drop_piece(b_copy, row, col, self._game._turn * turn)
    new_score = self.minimax(
        b_copy, depth - 1, alpha, beta, not maximizingPlayer, pruning
    )[1]

    if maximizingPlayer:
        if new_score > value:
            value = new_score
            column = col
        alpha = max(alpha, value)
    else:
        if new_score < value:
            value = new_score
            column = col
        beta = min(beta, value)

```

```

    if pruning:
        if alpha >= beta:
            break

    return column, value

```

La méthode principale `minimax` de la classe `Minimax` qui permet de faire fonctionner l'algorithme est une fonction récursive. En ce sens, l'algorithme du Min-Max ressemble à une recherche en profondeur (*depth first search*). On commence par déterminer les coups possibles depuis la position actuelle du jeu, selon le joueur qui doit jouer. On vérifie ensuite que l'on n'est pas dans un coup terminal et que l'on n'a pas dépassé la profondeur de recherche maximale. Sinon, on initialise la valeur du coup à l'infini, pour le joueur qui maximise (ou $-\infty$, respectivement si on considère que le joueur minimise), puis on parcourt les coups possibles en simulant l'effet du coup sur le score et en rappelant la fonction de manière récursive jusqu'à atteindre un nœud terminal ou la profondeur maximale. On parcourt ainsi l'arbre des coups possibles, en profondeur, en gardant toujours en mémoire le meilleur coup et en minimisant ou maximisant alternativement le score à remonter dans l'arbre.

Revenu à la racine de l'arbre, on récupère une dernière fois le coup maximum, à savoir le coup optimal trouvé dans l'arbre. On trouve aussi dans cette implémentation la recherche alpha-bêta, qui, quand elle est opérationnelle, diminue significativement le nombre de nœuds à visiter dans l'arbre, sans compromettre la qualité de la solution. Ce complément algorithmique évite d'explorer les sous-arbres conduisant à des situations qui ne contribueront pas à maximiser le score à la racine de l'arbre.

Lorsqu'il n'est pas possible d'explorer tout l'arbre, on s'arrête à une profondeur maximale et on évalue le jeu grâce à une heuristique ; ici, la méthode utilisée est `score_position`.

```

def score_position(self, board, piece):
    """
    Main function that handles the scoring mechanism.
    Handle the score for the minimax algorithm, the score is computed
    independently of which piece has just been dropped.
    This is a global score that looks at the whole board.

    :param board: board with all the pieces that have been placed
    :param piece: 1 or -1 depending on whose turn it is
    :return: score of the board
    """

    score = 0
    # Score center column
    center_array = [int(i) for i in list(board[COLUMN_COUNT // 2][:])]
    center_count = center_array.count(piece)
    score += center_count * 3

```

```

# Score Horizontal
for r in range(ROW_COUNT):
    row_array = [int(i) for i in list(board[:][r])]
    for c in range(COLUMN_COUNT - 3):
        window = row_array[c : c + WINDOW_LENGTH]
        score += self.evaluate_window(window, piece)

# Score Vertical
for c in range(COLUMN_COUNT):
    col_array = [int(i) for i in list(board[c][:])]
    for r in range(ROW_COUNT - 3):
        window = col_array[r : r + WINDOW_LENGTH]
        score += self.evaluate_window(window, piece)

# Score positive sloped diagonal
for r in range(ROW_COUNT - 3):
    for c in range(COLUMN_COUNT - 3):
        window = [board[c + i][r + i] for i in range(WINDOW_LENGTH)]
        score += self.evaluate_window(window, piece)

for r in range(ROW_COUNT - 3):
    for c in range(COLUMN_COUNT - 3):
        window = [board[c + i][r + 3 - i] for i in range(WINDOW_LENGTH)]
        score += self.evaluate_window(window, piece)

return score

```

Cette méthode va calculer un score pour l'ensemble du plateau en fonction du joueur. Le paramètre `piece` indique si on calcule le score pour le joueur Max (`piece=1`) ou pour le joueur Min (`piece=-1`). Ensuite, on parcourt le plateau de jeu. Le score est alors calculé de la manière suivante :

- deux points pour deux pièces consécutives ;
- cinq points pour trois pièces consécutives ;
- un malus de quatre points si l'adversaire a trois pièces consécutives ;
- trois points pour chaque pièce dans la colonne centrale ;
- $+\infty$ pour un plateau gagnant ;
- $-\infty$ pour un plateau perdant.

Ces valeurs sont totalement arbitraires et ont été déterminées expérimentalement (par exemple, en remarquant qu'avoir des pièces au centre donne un avantage au joueur), la détermination de l'heuristique faisant ainsi partie de l'expertise du problème que le développeur intègre dans son programme.

MCTS

L'arbre de recherche dans le problème du Puissance 4 contient, à chaque coup, sept possibilités (mettre une pièce dans une des sept colonnes). On a donc un arbre de sept coups au premier niveau, puis $7 \times 7 = 49$ coups au niveau 2 de l'arbre, etc. Au final, si on évalue le nombre de coups par

rapport à la profondeur de recherche on trouve 7^p (avec p la profondeur). Si on prend une profondeur de 10 coups pour la recherche, on a déjà $7^{10}=282\,475\,249$ coups à évaluer. Sachant qu'une partie typique de Puissance 4 peut aller au-delà de 30 coups, on en déduit donc qu'il n'est pas raisonnable d'essayer de calculer l'intégralité de l'arbre.

La recherche par la technique de Monte Carlo (MCTS) propose une alternative s'inspirant de l'apprentissage par renforcement. Elle essaie des coups aléatoirement pour essayer de trouver des portions de sous-arbres intéressantes à explorer. Chaque itération de l'algorithme dispose alors d'un « budget » de recherche avec lequel explorer l'arbre avant de prendre une décision.

```
def monte_carlo_tree_search(self, iterations, root, exploration_parameter):
    """
    Main function of MCTS, called whenever a move is needed.

    :param iterations: number of iterations the MCTS algorithm will run for
                       (the more iterations the longer the algorithm takes)
    :param root: root tree, starting point of the algorithm (board of the game
                 at the moment a move is wanted)
    :param exploration_parameter: factor used in the MCTS

    :return: column where to place the piece
    """
    for i in range(iterations):
        node, turn = self.selection(root, 1, exploration_parameter)
        reward = self.simulation(node.state, turn)
        self.backpropagation(node, reward, turn)

    ans = self.best_child(root, 0)
    return ans.state.last_move[0]
```

L'algorithme MCTS, et donc la fonction `monte_carlo_tree_search`, consiste en quatre phases. Tout d'abord vient la sélection, grâce à la méthode `selection`. Depuis la situation de jeu actuelle, prise comme racine de l'arbre de recherche, on parcourt l'arbre en passant par les meilleurs nœuds fils (voir `best_child` plus loin), jusqu'à atteindre un nœud terminal ou non exploré.

```
def selection(self, node, turn, exploration_parameter):
    """
    Expands the root node and takes the best child everytime until
    a winning state is reached. If a node is not fully explored,
    it is expanded and a child is returned.
    If it is fully explored, the best child of that node is taken.

    :param node: starting node
    :param turn: -1 or 1 according to which player plays next
    :param exploration_parameter: factor used for the MCTS algorithm
    """
    while not node.state.last_move or not node.state.check_win(
        node.state.last_move
    ):
        pass
```

```

    if not node.fully_explored():
        return self.expansion(node), -1 * turn
    else:
        node = self.best_child(node, exploration_parameter)
        turn *= -1

    return node, turn

```

Une fois un nœud non exploré sélectionné, on va l'étendre dans la phase d'expansion à l'aide de la méthode `expansion`. On va alors chercher un coup pas encore testé et explorer le nœud de l'arbre représentant ce nouveau coup.

```

def expansion(self, node):
    """
    Add a child state to the node. Concretely, plays a move and adds
    it to the board of a newly created state, which is the child of the
    current node.

    :param node: current node to expand
    :return: a newly created child of the current node
    """
    free_cols = node.state.get_valid_locations()

    for col in free_cols:
        if col not in node.children_moves:
            new_state = node.state.copy_state()
            new_state.place(col)
            break

    node.add_child(new_state, col)
    return node.children[-1]

```

Depuis ce nouveau nœud, on va simuler le jeu avec des coups aléatoires, jusqu'à atteindre un nœud terminal, donc une situation où le jeu est gagné ou perdu. On effectue cette phase grâce à la méthode `simulation`.

```

def simulation(self, state_init, turn):
    """
    Simulates random moves until the game is won by someone and returns
    a reward. Until a winning (or losing) situation is obtained,
    random moves are performed.
    The reward is then simply 1 in case the winner is the actual player,
    and -1 otherwise.

    :param state_init: current state from which we should end up finding a
        winning situation
    :param turn: 1 or -1 depending on whose turn it is
    """

```

```

: return: a reward
"""
state = state_init.copy_state()
while not state.last_move or not state.check_win(state.last_move):
    free_cols = state.get_valid_locations()
    col = random.choice(free_cols)
    state.place(col)
    turn *= -1

reward_bool = state.check_win(state.last_move)
if reward_bool and turn == -1:
    reward = 1
elif reward_bool and turn == 1:
    reward = -1
else:
    reward = 0
return reward

```

Une fois un nœud terminal atteint, on retourne la récompense, qui est soit +1 quand la partie est gagnée, soit -1 quand la partie est perdue, soit 0 pour une égalité. Une fois la récompense obtenue, on l'utilise dans la phase de rétropropagation par la méthode `backpropagation`.

```

def backpropagation(self, node, reward, turn):
    """
    Update the rewards of all the ancestors of a node. The reward is sometimes
    added and sometimes subtracted from the current reward, since it takes
    into account the fact that a winning move from the current player should be
    encouraged, but a winning move from the opponent should be discouraged.

    :param node: current node from which we start the backtracking
    :param reward: reward corresponding to that particular node
    :param turn: 1 or -1 depending on whose turn it is
    """
    while node != None:
        node.visits += 1
        node.reward -= turn * reward
        # node.reward += reward
        node = node.parent
        turn *= -1
    return

```

Dans cette phase, on met à jour l'attribut de récompense de tous les nœuds par lesquels on est passé pour atteindre le nœud terminal, avec la valeur de la récompense acquise à la phase précédente. Lors de la sélection, on utilise la notion de meilleur nœud fils, calculé grâce aux récompenses accumulées lors des simulations avec la méthode `best_child`.

```
def best_child(self, node, exploration_parameter):
    """
    Returns the best child of a node based on a scoring system proposed by
    Auer, Cesa-Bianchi and Fischer. This formula combines a term of
    exploration and a term of exploitation.

    :param node: node from which we want to find the best child
    :param factor: exploration parameter
    """
    best_score = -float("inf")
    best_children = []
    for c in node.children:
        exploitation = c.reward / c.visits
        exploration = math.sqrt(math.log2(node.visits) / c.visits)
        score = exploitation + exploration_parameter * exploration
        if score == best_score:
            best_children.append(c)
        elif score > best_score:
            best_children = [c]
            best_score = score
    res = random.choice(best_children)
    return res
```

On détermine quel nœud doit être visité par la fonction suivante :

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}$$

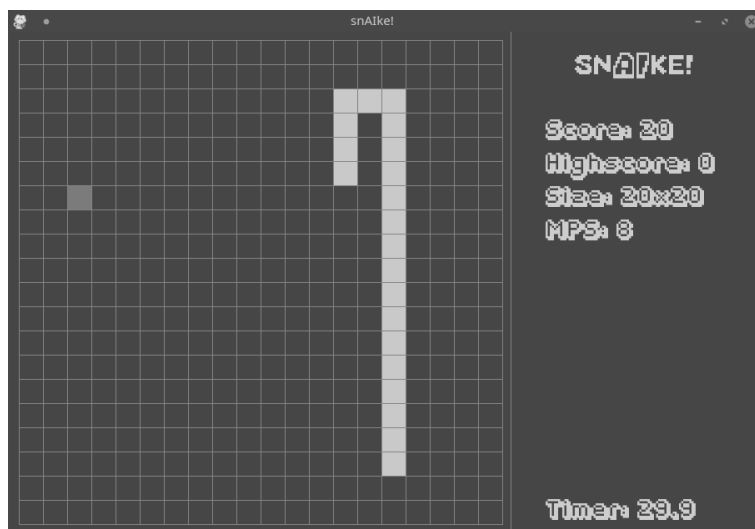
où w_i est le nombre de victoires du nœud fils considéré (donc la valeur totale de récompense), n_i le nombre total de visites du nœud fils et N_i le nombre de visites pour le nœud parent. Ces deux termes sont à la base du compromis exploration-exploitation typique des algorithmes d'apprentissage par renforcement. Le paramètre c (dans le code `exploration_parameter`) est un paramètre de l'algorithme qui calibre le ratio entre exploration et exploitation.

5

Jouons au Snake

Figure 5-1

Capture d'écran du logiciel Python mis à votre disposition pour le jeu vidéo du Snake.



Principe – Choix algorithmique

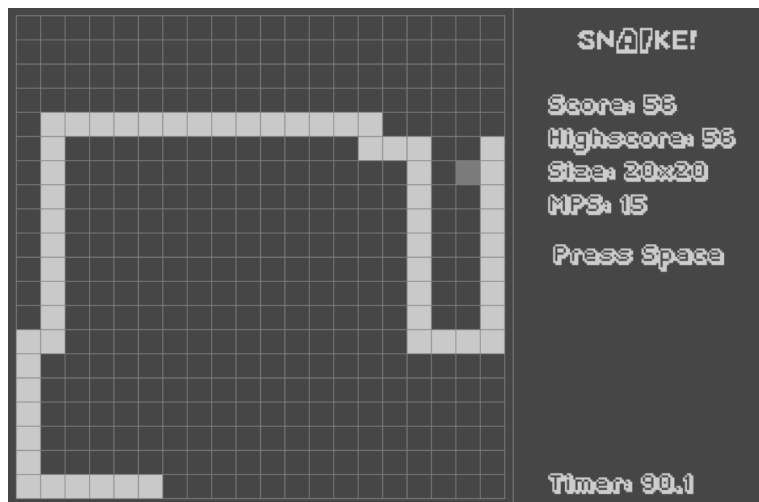
Les petits jeux vidéo tels que Snake ou Tetris ont fait le bonheur des anciens, assis devant leur Commodore 64, et attirent encore très souvent nos étudiants dans la pratique des algorithmes IA. Ils voient en eux l'occasion de tester et comparer différentes approches de l'IA, qu'ils trouvent

a priori et à tort équivalentes. Ils découvrent alors par l'expérience ce qu'ils auraient aisément pu anticiper s'ils s'étaient davantage questionnés sur la véritable nature du problème et les briques algorithmiques plus aptes à l'affronter dans un temps décent. Par ailleurs, des simulateurs de ces jeux sont disponibles sur le Web, ce qui permet aux étudiants d'uniquement se concentrer sur la découverte par l'IA des joueurs optimaux sans se préoccuper de représenter le jeu en question et de le visualiser à l'écran.

Le Snake consiste en une ligne se déplaçant continûment dans un espace 2D, en répondant aux quatre commandes possibles correspondant aux quatre directions cardinales. À chaque commande, la tête du serpent change de direction et entraîne dans sa suite toute la ligne (son corps). Des cibles apparaissent au hasard dans cet espace, que le serpent se doit d'intercepter en passant dessus (comme de la nourriture). À chaque cible atteinte, le score augmente, le serpent s'agrandit d'une case et une autre cible surgit. La difficulté s'accroît avec le temps et le serpent grandissant, car il devient de plus en plus difficile de le manœuvrer en évitant que son corps fasse une boucle, ce qui provoque l'interruption et la perte de la partie. Le score est calculé par le nombre de cibles atteintes et le temps de survie du serpent (qui meurt s'il sort du jeu ou s'il fait une boucle).

Figure 5-2

L'écran de simulation du Snake
(en gris clair) ; la cible à
atteindre est en gris foncé.



Ici, à nouveau, les trois concepts fondamentaux semblent relativement faciles à identifier : états, actions et solutions. L'état serait la configuration du jeu (la position complète du serpent dans son environnement), les actions les quatre commandes possibles et l'objectif consiste à obtenir le score le plus important. Un algorithme de recherche style A* reste-t-il aisément applicable ? À y regarder de plus près, la réponse n'est plus aussi évidente que dans le cas du taquin et, de fait, nos étudiants sont généralement plus enclins à y plaquer un apprentissage par renforcement. Commençons par le problème de l'état. Ici, il faudrait coder chaque état différent, c'est-à-dire chaque position du serpent dans son environnement (en sachant que la taille de celui-ci peut changer) et toutes les possibles positions des cibles. Le nombre d'états possibles devient très vite monstrueusement grand. Or, il n'est sans doute pas nécessaire de coder tous ces états de manière exhaustive au

vu de la symétrie du jeu ; une solution plus naturelle et surtout plus économe est de se mettre à la place du serpent et de ne coder comme état que la perception de ce serpent (les quelques cases, en adoptant un champ de vision assez large, qu'il voit devant lui). Le serpent verra les cibles ainsi que les parties de son corps qu'il risque de heurter et c'est ce que l'état se limitera à reproduire.

Le problème des actions est plus particulier encore, car le serpent se meut sans cesse, qu'il y ait commande ou pas, ces dernières changeant simplement sa direction de mouvement. Donc, sauf à coder l'état d'une manière très recherchée, on peut considérer que les modifications de celui-ci ne résultent pas uniquement des commandes. Excepté, évidemment, si le joueur choisit d'appuyer à chaque déplacement du serpent ou considère l'absence de commandes comme une des possibles actions. On considérera que la perception du serpent change continûment, qu'une commande soit effectuée ou pas, rendant la construction de l'arbre de recherche plutôt problématique. On sait aussi que les cibles apparaissent au hasard (un hasard qu'il est très difficile de prendre en compte dans la représentation de l'état). Finalement, jusqu'où faudrait-il déployer l'arbre de recherche lorsque le joueur joue de façon optimale et que la partie dure longtemps ?

Autre exigence des algorithmes de type A^* , la définition d'un état final cible. On pourrait par exemple trouver la succession de coups qui conduit le serpent sur les cibles, car l'état final dans ce cas est bien défini : se rendre sur la cible. Toutefois, ce n'est pas la seule obligation du serpent, car il lui est aussi demandé de s'éviter et d'éviter les obstacles, ce qui est beaucoup plus difficile à représenter comme état final. En revanche, une évaluation de la solution basée sur les renforcements ou sur une fonction coût s'avère bien moins contraignante et permet de bien évaluer les performances du serpent tout au long du jeu. Renforcements et coûts calculés sur une longue période semblent une bien meilleure manière d'évaluer les commandes effectuées sur le serpent et donc la qualité du joueur.

Néanmoins, et malgré les réserves énoncées plus haut, nous vous proposons une première approche assez performante de type A^* dans le code Python mis à votre disposition. L'algorithme en question va surtout se pencher sur l'accès à la cible, et cela au plus rapide. On sait qu'une solution possible pour le serpent consiste à effectuer des grands « S » au départ des parois, ce qui lui permet de s'éviter au mieux, mais met bien davantage de temps pour atteindre les cibles. Ici, nous allons calculer, pour chaque apparition de cible, la trajectoire la plus rapide pour y accéder. Sans surprise, nous retrouvons notre fameuse fonction d'évaluation $f=g+h$, où h est ici la distance de Manhattan pour atteindre la cible. Une fois le parcours obtenu, on effectue les actions sur le serpent jusqu'à atteindre la cible et on recommence le tout pour la nouvelle cible qui se substitue à la précédente. Afin d'accélérer encore l'algorithme, nous nous basons toujours sur le premier successeur trouvé, car de nombreux nœuds successeurs pourraient se montrer équivalents.

Si cet algorithme nous garantit d'atteindre la cible au plus vite, rien ne préserve en revanche le serpent de se mordre la queue. Une amélioration devient donc nécessaire. Il faut que A^* évite également au serpent de heurter les murs ou son propre corps. Chaque itération de A^* va trouver le chemin le plus court vers la nourriture, tout en prenant en compte, pour chaque case (chaque nœud de l'arbre de recherche), que certaines autres cases sont interdites. Le serpent va donc éviter les cases obstacles et trouver un chemin vers son objectif. En revanche, on ne peut pas garantir que la position dans laquelle il va se retrouver à la fin sera telle qu'il puisse se sortir de sa situation. En effet, le serpent pourrait se trouver coincé dans sa queue ou encore enclavé contre un mur, incapable de trouver un chemin vers la prochaine nourriture.

Dès lors, bien souvent, toutes les difficultés énoncées et toutes les subtilités que A^* réclame mènent naturellement nos étudiants à privilégier une approche de type renforcement ou apprentissage, moins exigeante en représentation du problème : l'état reste le même (la perception du serpent), cependant il n'est plus du tout question d'arbre de recherche, mais de simplement trouver la meilleure action possible pour chaque perception du serpent. Dans le code Python qui est mis à votre disposition, les quatre actions du serpent sont les sorties d'un réseau de neurones composé de quatre couches : la couche d'entrée avec 26 neurones que nous allons détailler plus loin, deux couches intermédiaires (dites couches cachées avec un nombre variable de neurones) et la couche de sortie composée de quatre neurones (les quatre actions du serpent – on choisira toujours l'action correspondant au neurone le plus actif).

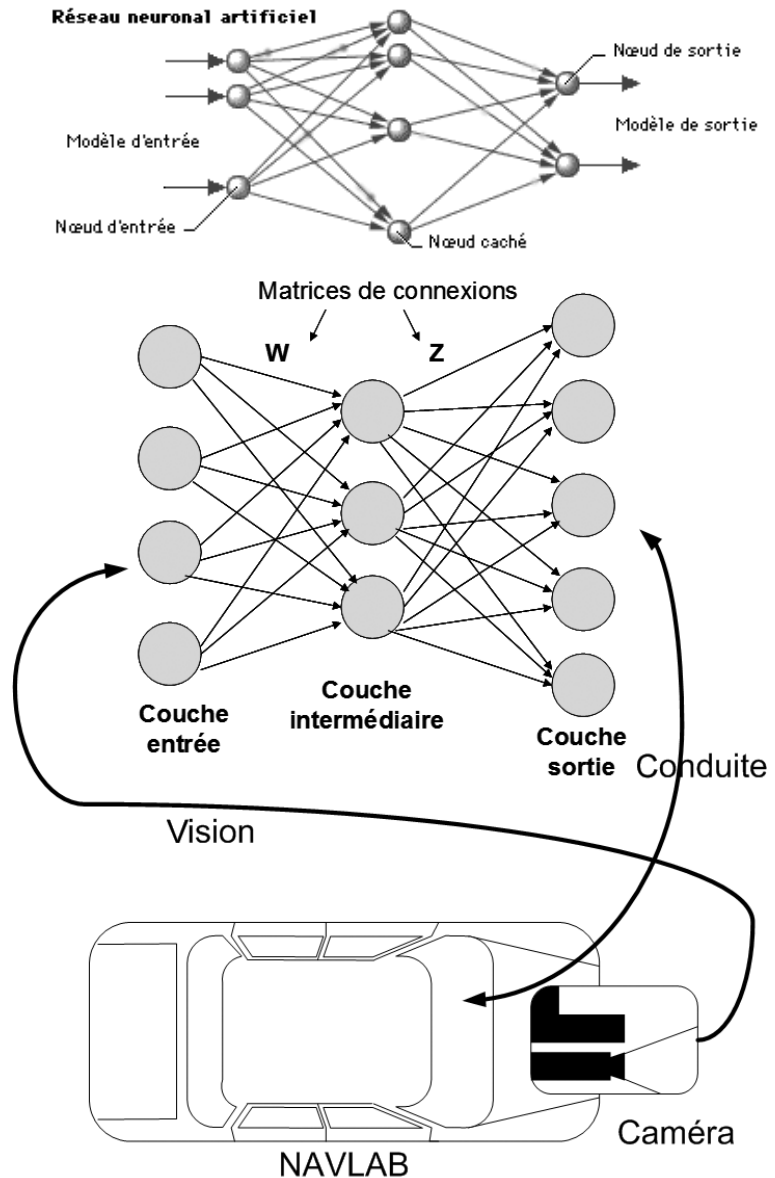
Réseaux de neurones artificiels

Rappelons-le, malgré l'engouement récent pour l'apprentissage dit profond basé sur les réseaux de neurones, il s'agit bien là d'une technologie qui fête son soixante-cinquième anniversaire : la première conférence d'intelligence artificielle, à l'université de Dartmouth, aux États-Unis, date de 1956. Elle faisait déjà la part belle aux réseaux de neurones. Les premières publications sur ces mêmes réseaux datent des années quarante. Lors de cette fameuse conférence de Dartmouth, qui a donné naissance au nouveau champ disciplinaire appelé depuis intelligence artificielle (IA), on parla abondamment d'apprentissage machine (apprentissage automatique) et de réseaux de neurones. Lorsque Hugues Bersini donne son cours sur les réseaux de neurones, il fait toujours allusion à ses travaux de doctorat (la préhistoire) pour lesquels il entraînait un réseau de neurones à conduire une voiture sans chauffeur (figure 5–3). Il collectionnait un nombre déjà gigantesque pour l'époque de données issues de la conduite humaine, sous forme de « perception/action ». Ce que voyait le conducteur humain était enregistré par une caméra placée à l'avant de la voiture et ce que faisait ce même conducteur l'était tout autant (action sur le volant et les pédales). Était ainsi constituée une immense base de données « perception/action » que le réseau de neurones était censé apprendre et reproduire afin d'apprendre et de généraliser la conduite jusqu'à remplacer les conducteurs humains.

Le réseau de neurones apprenait à reproduire les perceptions en entrée et les actions en sortie, en ajustant les poids synaptiques qui relient les couches entre elles (figure 5–3 en bas : les matrices w et z). Le projet est simple. On donne la perception des chauffeurs en entrée du réseau et on entraîne celui-ci à reproduire en sortie les actions de ces mêmes chauffeurs. Au début, les poids synaptiques qui relient tous les neurones entre eux sont aléatoires et, bien sûr, le réseau est incapable de reproduire les entrées/sorties désirées. On va alors lui apprendre à le faire par un algorithme d'apprentissage de type supervisé, car on connaît exactement les sorties que le réseau est censé reproduire. Évidemment, ce qui est recherché ici n'est pas tant sa capacité à reproduire exactement les données qu'on lui apprend, mais bien sa capacité à extrapoler à partir de ces données, c'est-à-dire à exécuter les bonnes actions face à de nouvelles situations qui, bien que jamais rencontrées auparavant, ne devraient pas être trop éloignées de celles qui ont servi à constituer les données d'apprentissage. Le réseau de neurones n'est pas censé conduire une voiture sur la planète Mars.

Figure 5-3

Un réseau de neurones (en haut) qui conduit une voiture (en bas).



Pour être un tant soit peu plus technique, commençons par le réseau de neurones le plus simple qui soit (figure 5-4), composé juste d'une couche d'entrée et d'une couche de sortie, sans couche cachée donc. L'activation de chaque neurone de sortie est donnée par une fonction appliquée à la somme pondérée (par les poids synaptiques) de la couche d'entrée. À l'époque, la fonction Sigmoid était la plus prisée ; depuis, et surtout pour les versions les plus profondes de ces réseaux, on

lui préfère la fonction ReLU (car sa dérivée ne tend pas vers zéro pour les valeurs d'activation importantes, diminuant ainsi l'effet d'estompage du gradient lorsque l'on rétropropage l'erreur de couche en couche).

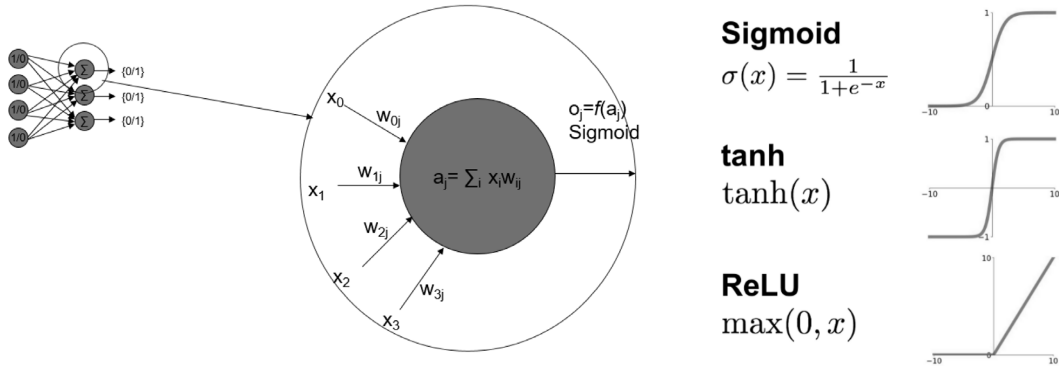


Figure 5-4 À gauche, architecture d'un réseau simple de neurones, sans couche cachée.

À droite, les trois fonctions d'activation neuronale les plus utilisées.

<https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092>

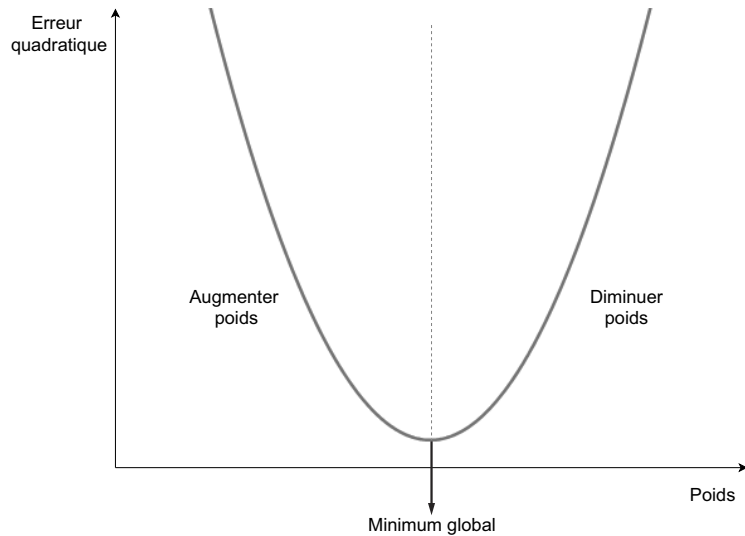
L'algorithme d'apprentissage est basé sur une méthodologie d'optimisation (et on retrouve encore l'hybridation déjà souvent rencontrée dans les chapitres précédents) dite méthode de gradient. Elle part de l'erreur mesurée en sortie par rapport aux sorties effectivement souhaitées et va graduellement modifier les poids synaptiques de manière à diminuer cette erreur. Une erreur possible est le moindre carré, qui consiste à soustraire les sorties désirées à celles obtenues et à porter cette différence au carré : $(y_i^d - y_i)^2$. Lorsque l'on prend le gradient pour ajuster les poids, on obtient l'ajustement du poids w_{ij} suivant (entre le neurone i de la première couche et le neurone j de la deuxième), avec un taux d'apprentissage (qui ajuste la vitesse de l'apprentissage) η :

$$\Delta W_{ij} = \eta \times (Y_i^d - Y_i) \times X_j$$

Les poids s'ajustent petit à petit dans la meilleure direction et la meilleure intensité qu'il faut pour minimiser cette erreur (figure 5-5). De multiples améliorations simples de l'algorithme de gradient ont été proposées, telle la méthode Adam pour laquelle les moyennes courantes des gradients et des seconds moments des gradients sont utilisées. Cette méthode est utilisée de manière à rendre les taux d'apprentissage (tel le η dans l'équation précédente) adaptatifs pour chaque paramètre. Comme on part des poids de la dernière couche (celle responsable de la sortie désirée) pour progressivement ajuster les poids de celles qui précèdent (et elles peuvent être très nombreuses, jusqu'à des centaines dans le cas des réseaux de neurones dits profonds), l'algorithme porte le sobriquet de « rétropropagation ». Il étend la dérivée de l'erreur obtenue sur la dernière couche vers les couches qui précèdent (on enchaîne les dérivations de couche en couche – figure 5-6).

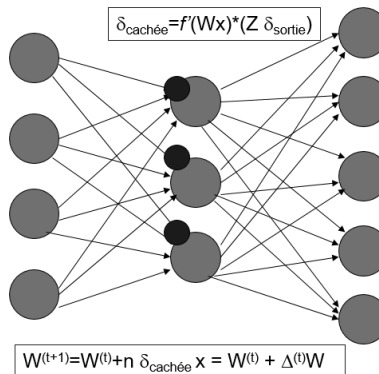
Figure 5-5

Ajustement graduel des poids synaptiques pour atteindre le minimum de l'erreur en sortie.

**Figure 5-6**

Description de l'algorithme de rétropropagation.

■ Algorithm



1. Inject an entry
2. Compute the intermediate h
3. Compute the output o
4. Compute the error output
5. Adjust Z on the basis of the error
6. Compute the error on the hidden layer
7. Adjust W on the basis of this error

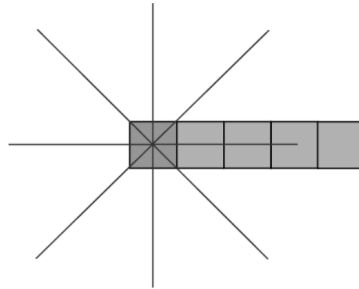
On discutera encore abondamment de tout cela dans le dernier chapitre consacré à la reconnaissance d'images de chiens et de chats (l'application rêvée pour ces réseaux de neurones).

Retour au Snake

Les 24 premiers neurones de la couche d'entrée codent pour la perception relative de la tête du serpent (figure 5-7). Pour chacune des trois directions possibles pour le déplacement, on mesure la distance à la nourriture, aux murs et au corps du serpent. Les deux derniers neurones jouent le rôle d'une petite mémoire bien utile pour savoir d'où vient le serpent et se rappeler les deux derniers mouvements effectués : $(x_t - x_{t-2}, y_t - y_{t-2})$.

Figure 5-7

Perception relative du serpent.



Pour apprendre les poids du réseau de neurones, il est impossible ici d'appliquer un algorithme de gradient à partir des sorties désirées car, justement, ce sont ces mêmes sorties que l'on cherche. La présence de ce réseau reste intéressante, car elle permet de généraliser les mêmes actions du serpent à différentes perceptions de celui-ci. Nouvelle hybridation donc, on applique à la place du gradient un algorithme génétique qui va optimiser les poids du réseau en cherchant à maximiser la fonction de coût suivante :

$$f(\text{score}, \text{moves}) = \text{score}^2 \times \text{moves}$$

Le `score` est le nombre de cibles atteintes et `moves` est le nombre de coups joués. On voit bien, par la définition de cette fonction, que l'on cherche à maximiser à la fois le nombre de cibles englouties et la durée de vie du serpent. Ce dernier est aussi caractérisé par un attribut « faim », plus ou moins proportionnel au temps qui lui reste à vivre s'il ne trouve aucune nourriture. Cet attribut évite que l'apprentissage cesse de progresser à cause d'un serpent se mettant à circuler indéfiniment. Chaque réseau de neurones est donc un joueur possible du Snake et l'algorithme génétique va essayer de découvrir le meilleur d'entre eux par la succession du bouclage sur les quatre opérations classiques : évaluation, sélection, recombinaison, mutation. Chaque joueur est représenté par un vecteur contenant les poids synaptiques du réseau, codés comme des réels compris entre -1 et +1 (figure 5-8).

Figure 5-8

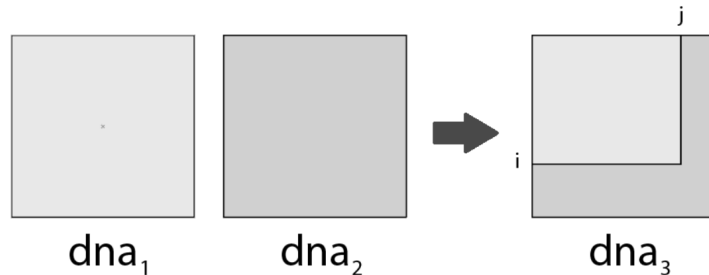
-0.51	0.72	-0.39	0.05	0.25		-0.31	-0.14	-0.92	0.78	0.49
-------	------	-------	------	------	-------	-------	-------	-------	------	------

La population sur laquelle itère l'algorithme génétique comprend mille joueurs, donc mille réseaux de neurones. Pour la sélection, l'algorithme qui vous est présenté utilise le rang plutôt que la valeur réelle de fitness (c'est conseillé), favorisant donc relativement les meilleurs joueurs. La recombinaison

naison agit sur les deux matrices qui constituent les poids du réseau et les recombine (figure 5–9) à partir de deux indices i et j tirés au hasard.

Figure 5–9

Recombinaison des deux ensembles de poids synaptiques, à gauche, pour en générer un troisième, à droite.



La mutation s'effectue en ajoutant un nombre tiré au hasard à certains poids du réseau, tout en maintenant la contrainte des limites $(-1, 1)$.

Figure 5–10

	+0.62		+0.90			-0.40					
Avant	-0.51	0.72	-0.39	0.05	0.25		-0.31	-0.14	-0.92	0.78	0.49
Après	0.11	0.72	-0.39	0.05	1.00		-0.31	-0.14	-1.00	0.78	0.49

Une extension possible, mais non prévue dans ce projet, est de permettre aussi à l'algorithme génétique d'optimiser l'architecture du réseau. On rencontre très souvent cette utilisation des opérations de recombinaison et mutation pour façonner automatiquement la configuration des réseaux de neurones à défaut d'une approche plus formelle (une méthode de type gradient ne convient en rien pour cette architecture). Associer les algorithmes génétiques pour l'architecture du réseau à une méthode de gradient pour l'optimisation des poids nous rapproche étonnamment de la réalité biologique d'un cerveau qui évolue au cours des générations par hérédité génétique et s'améliore de son vivant (durant une génération) par ajustement synaptique.

On voit donc bien que, selon la facilité avec laquelle les notions d'état, d'action, d'arbre de recherche, d'heuristique et de solutions se déclinent, le choix se portera sur une version algorithmique très contrainte et très structurée, type A^* , ou vers une version beaucoup plus laxiste et moins exigeante a priori, type génétique ou Q-learning. Ces approches dites par renforcement sont aussi parfaitement adaptées à la création de logiciels capables d'exceller dans la pratique des jeux vidéo tels Super Mario ou d'autres (surtout quand les yeux des joueurs humains sont de surcroît remplacés par des réseaux de neurones). Le logiciel apprenant joue pendant un certain temps en accumulant récompenses et punitions, il améliore graduellement son jeu en associant des coups de plus en plus efficaces aux différentes situations du jeu. Souvent, ce sont essentiellement des images des états du jeu encodées en entrée d'un réseau de neurones. Ce dernier cherche alors à dériver les

coups gagnants en sortie et effectue cet apprentissage en appliquant un algorithme de type Q-learning comme alternative à l'algorithme génétique. Il n'est pas des plus faciles, dans des cas semblables, de décider a priori de la meilleure manière d'optimiser le réseau de neurones en question, selon l'algorithme du Q-learning ou un algorithme de type génétique ; seuls les résultats pratiques d'expérience (difficulté du codage, qualité de la solution et temps de calcul pour l'obtenir) permettent de trancher.

Le Snake, tel qu'il est présenté ici, ne parvient pas à une solution optimale où le serpent remplirait tout l'écran et finirait par se mordre la queue faute de tout autre possible chemin. Une des options est bien sûr de produire la forme de « S » caractéristique (visible dans le code) et permettant, sans aucune intelligence, en suivant un chemin prédéfini, de parvenir à exécuter une partie parfaite, après un temps particulièrement long. Dans le cas où l'on voudrait réaliser une telle partie, tout en minimisant le temps de récupération de la nourriture, donc sans suivre un chemin prédéfini, il faudrait construire un chemin qui passe par toutes les cases du plateau, tout en allant directement chercher la nourriture. Pour cela, on peut imaginer un mélange entre l'algorithme A* et la recherche de cycle hamiltonien. En effet, un tel cycle est défini par le fait qu'il passe par tous les nœuds d'un graphe exactement une fois. On aura donc la garantie de toujours gagner le jeu et remplir l'écran avec le serpent ! Il existe plusieurs algorithmes afin de rechercher des cycles hamiltoniens, mais cela reste un problème très complexe qui ne pourra être résolu efficacement dans le cas de grands graphes.

Le logiciel Python du Snake en pratique

Le projet du Snake se présente sous la forme suivante :

- un fichier `main.py` permet de lancer le projet avec les différentes options et `gameModule.py` contient l'interface graphique du jeu ;
- les différentes IA sont contenues dans `Astar_snake.py` pour la partie utilisant A*, `genetic_snake.py` pour le module global du Snake avec algorithme génétique, `snakeTrainer.py` qui contient le code d'entraînement du réseau de neurones avec l'algorithme génétique et `dna.py` contenant la définition du réseau de neurones.

Recherche A*

Commençons par la recherche classique avec A*. Le but de cette méthode consiste à déplacer le serpent le plus vite possible vers la prochaine cible. On va donc programmer le serpent pour trouver le chemin le plus court entre lui et la prochaine source de nourriture en évitant les obstacles (notamment son propre corps). Ici, on se trouve en présence d'un cas très classique de recherche de plus court chemin dans un monde « grille » (*grid world*). En effet, on retrouve une grille dans laquelle le serpent peut se déplacer suivant quatre directions. Chacun des mouvements est de distance 1 et toutes les cases sont de la même taille.

Pour trouver la distance jusqu'à la prochaine source de nourriture, on peut utiliser la distance de Manhattan (qui, grâce à cette propriété du monde grille, sera exactement la distance minimale à par-

courir) ou bien la distance euclidienne. Notez que cette dernière ne sera jamais atteinte par le serpent, puisqu'il ne peut se déplacer en diagonale. Cette distance sous-estime donc celle restant à parcourir. Cette heuristique n'en demeure pas moins admissible car elle ne surestime pas cette distance.

Ci-après se trouve la fonction de base qui cherche un chemin entre deux points de la grille (la partie permettant l'affichage interactif de l'algorithme ayant été volontairement omise pour plus de clarté) :

```
def astar(self, state, goal_pos, interactive=False):
    """
    This function is an implementation of the A* algorithm
    :param state: The current state of the game
    :param goal_pos: The position where the snake has to go
    :param interactive: Display the execution of the A* algorithm
    :return: The path to the goal
    """

    grid, score, alive, snake = state
    head = snake[0]
    closed_list = set()
    open_list = []
    head_node = Node(head)
    food_node = Node(goal_pos)

    heapq.heappush(open_list, head_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_list.add(current_node)

        if current_node == food_node:
            path = []
            while current_node.parent is not None:
                path.append(current_node)
                current_node = current_node.parent
            return path

        for new_position in self.moves:
            node_position = (
                current_node.position[0] + new_position[0],
                current_node.position[1] + new_position[1],
            )
            # Make sure within range
            if not self.is_in_grid(node_position, grid):
                continue
            # Make sure walkable terrain
            if (
                grid[node_position[0]][node_position[1]] == SNAKE_CHAR
                or grid[node_position[0]][node_position[1]] == WALL_CHAR
            ):
                continue
            # Create new node
            child = Node(node_position, current_node)
```

```

# Child is on the closed list
if child in closed_list:
    continue

# Child is in the openlist with smaller cost
if (
    child in open_list
    and open_list[open_list.index(child)].g <= current_node.g + 1
):
    continue
# Create the f, g, and h values
child.g = current_node.g + 1
child.h = self.h_cost(child, food_node)
child.f = child.g + child.h
child.parent = current_node

# Add the child to the open list
heapq.heappush(open_list, child)

return "No path"

```

On a choisi ici de présenter l'algorithme A* de manière classique. On commence par définir une liste ouverte, `open_list`, contenant les cases à visiter (contenant initialement le nœud de départ du Snake), une liste fermée, `closed_list`, contenant les cases ayant déjà été visitées, `head_node` étant le nœud de la tête du serpent et `food_node` celui où se trouve la nourriture. On procède ensuite de manière analogue aux autres recherches par A*. Tant que la liste ouverte n'est pas vide, on va prendre l'élément le plus petit de la liste. Celui-ci est déterminé grâce à l'utilisation d'une `heapq` de Python, qui implémente une liste prioritaire automatiquement triée. Dans notre cas, la `heapq` doit trier des objets de type `Node`. Pour cela, il suffit que la classe `Node` implémente la méthode `__lt__`. Voici l'implémentation de notre classe `Node` permettant de sauver la position, le nœud parent ainsi que les valeurs de coût (g), d'heuristique (h) et de coût total (f).

```

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent

        self.g = 0
        self.h = 0
        self.f = self.g + self.h

    def __repr__(self):
        return f"({self.position[0]}, {self.position[1]})"

    def __lt__(self, other):
        return self.f < other.f

    def __key(self):
        return self.position

```

```
def __hash__(self):
    return hash(self.__key())

def __eq__(self, other):
    if isinstance(other, Node):
        return self.__key() == other.__key()
    return NotImplemented
```

On remarque la méthode `__lt__` permettant de trier les objets `Node` entre eux ; elle utilise bien la valeur de `f` comme critère de tri.

De retour dans notre algorithme, à la ligne 24, on vérifie si le nœud que l'on vient de prendre dans la liste est le nœud final d'arrivée, auquel cas, on remonte la chaîne des nœuds fils en parents jusqu'au nœud initial et on retourne le chemin.

Dans le cas contraire, on va générer tous les nœuds fils du nœud courant, c'est-à-dire les nœuds qui correspondent à un déplacement du serpent d'une case (tout en restant dans la grille et sans heurter un mur ou son corps). Si ces nœuds fils ne sont ni dans la liste fermée, ni dans la liste ouverte avec un coût inférieur (car, dans ce cas, il serait inutile ou contre-productif de les visiter), on les ajoute à la liste ouverte, puis on recommence à chercher le chemin en prenant dans la liste ouverte le nœud avec la plus petite valeur de `f`.

L'interface principale du Snake va simplement lancer l'algorithme A^* au travers de la fonction `choose_next_move`, d'abord au début, puis à chaque fois qu'il trouve une source de nourriture afin de trouver le chemin vers la prochaine.

Réseau de neurones et algorithme génétique

L'architecture du projet utilisant les réseaux de neurones est bien plus complexe que la précédente. La solution présentée ici tient de la configuration d'algorithme. En effet, le réseau de neurones contrôlant le serpent va être entraîné avec un algorithme génétique se basant sur la capacité du serpent à survivre et trouver de la nourriture. Cette approche se différencie de l'apprentissage par renforcement classique. Les améliorations apportées au réseau de neurones ne sont pas, comme avec le Q-learning, réalisées à chaque action et en se basant sur la récompense que cette action aurait donnée, mais sur le score que le serpent arrive à atteindre à l'issue d'une partie complète du jeu. Ainsi, lors de l'apprentissage, on doit attendre la fin d'une partie avant de faire évoluer le réseau de neurones. On introduit donc une notion de « faim » lors de l'entraînement du Snake, pour éviter que les réseaux de neurones que nous allons entraîner ne bloquent l'apprentissage par d'éventuelles boucles infinies du serpent sur lui-même.

Découvrons tout d'abord la classe `Snake` qui contrôlera notre serpent lors de l'entraînement ou lors d'une partie normale.

```
class Snake:
    """
    Represents the AI that plays the Snake game.
    """
```

```
def __init__(self, dna, hunger=100):
    """
    Constructor.
    dna (Dna): The DNA of the snake
    hunger (int): The starting hunger of the snake that decreases at each
    movement
    """
    self.dna = dna
    self.fitness = None
    self.hunger = hunger
    self.maxHunger = hunger
    self.nbrMove = 0
    self.previous_moves = []
```

Cette classe contient un certain nombre de méthodes, volontairement omises à ce stade, et un certain nombre d'attributs, ici visibles :

- dna, l'objet du réseau de neurones détaillé un peu plus tard ;
- fitness, le score de l'individu ;
- hunger, la « faim » du serpent, grossièrement proportionnelle au temps qu'il lui reste à vivre s'il ne trouve pas de nourriture ;
- maxHunger, la « faim » initiale lors de la création du serpent ;
- nbrMove, le nombre de mouvements que le serpent a déjà faits ;
- previous_moves, une liste ordonnée de l'historique des mouvements du serpent.

Lors d'une partie normale, contrôlée par un réseau de neurones, l'interface du jeu va appeler la fonction `choose_next_move` suivante :

```
def choose_next_move(self, state):
    """
    Choose a new move based on its vision.
    If the hunger of the snake is nul then return a string
    to indicate that the snakes is dead by starvation.
    vision (list): List containing distances between the head
    of the snake and other elements in the game
    Return the movement choice of the snake (tuple)
    """
    vision = self.get_simplified_state(state)
    if self.hunger > 0:
        self.hunger -= 1
        self.nbrMove += 1
        movesValues = self.dna.predict(vision)
        choice = 0

    movesValues = movesValues.tolist()
```

```

    # Chooses the best move (the move with the highest value)
    for i in range(1, len(movesvalues[0])):
        if movesvalues[0][i] > movesvalues[0][choice]:
            choice = i

    MOVEMENT = (RIGHT, LEFT, UP, DOWN)
    self.previous_moves.append(MOVEMENT[choice])
    if len(self.previous_moves) >= 3:
        self.previous_moves.pop(0)
    return MOVEMENT[choice]
return "starve"

```

Cette fonction récupère l'état simplifié du jeu (tel que décrit dans la partie précédente), contenant la vue du serpent en ligne droite sur huit axes, ainsi que l'historique des deux derniers mouvements. L'état simplifié est généré grâce à la fonction `get_simplified_state` suivante :

```

def get_simplified_state(self, state):
    """
    Returns a matrix of elements surrounding the snake and the previous two
    moves, this serves as the input for the neural network.
    """
    res = self.get_line_elem(RIGHT, state)
    res += self.get_line_elem((DOWN[0], RIGHT[1]), state)
    res += self.get_line_elem(DOWN, state)
    res += self.get_line_elem((DOWN[0], LEFT[1]), state)
    res += self.get_line_elem(LEFT, state)
    res += self.get_line_elem((UP[0], LEFT[1]), state)
    res += self.get_line_elem(UP, state)
    res += self.get_line_elem((UP[0], RIGHT[1]), state)

    if len(self.previous_moves) == 0:
        res += [0, 0]
    elif len(self.previous_moves) == 1: # previous previous move
        res += [self.previous_moves[0][0] / 2, self.previous_moves[0][1] / 2]
    else:
        res += [
            self.previous_moves[0][0] + self.previous_moves[1][0] / 2,
            self.previous_moves[0][1] + self.previous_moves[1][1] / 2,
        ]

    return res

```

Une fois cet état local récupéré, on va d'abord vérifier si le serpent n'est pas mort de faim. Si oui, la partie s'arrête ; sinon, on décrémente la « faim », on augmente le total de mouvements et on tente de prédire le prochain coup grâce à la fonction `predict` du réseau de neurones. Les valeurs de sortie sont évaluées et la plus grande est sélectionnée afin d'être le prochain mouvement à effectuer.

Penchons-nous sur la classe du réseau de neurones Dna :

```
class Dna:
    """
    Represents the DNA of a snake by using a list of weights and
    a list of biases. It is possible to modify the DNA by mutating
    it or by performing a crossover.
    """

    def __init__(self, weights=None, biases=None, layersSize=None):
        """
        Constructor.
        weights (list): A list of weights
        biases (list): A list of biases
        layersSize (list): A list containing the number of hidden
                           neurons used by the neural network
        """
        self.weights = cp.deepcopy(weights)
        self.bias = cp.deepcopy(biases)

        if self.weights is None:
            self.initialize_rd_weights(cp.copy(layersSize))
            self.initialize_rd_bias()
        elif self.bias is None:
            self.initialize_rd_bias()
```

Le constructeur de cette classe permet de prédire les actions du serpent. On commence par initialiser les poids synaptiques et les poids synaptiques des nœuds des biais du réseau. Dans le cas où l'on crée un nouveau réseau, on leur assigne des valeurs aléatoires grâce aux fonctions `initialize_rd_weights` et `initialize_rd_bias`, sinon on utilise les valeurs de poids et de biais fournies lors de la création de la classe.

La méthode la plus importante de cette classe est `predict`, qui va prendre les valeurs d'entrée correspondant à l'état simplifié de la vision du serpent et transformer cela en prédiction sur la meilleure action à exécuter.

```
def predict(self, inputs):
    """
    Predict the next movement of the snake.
    inputs (list): The vision of the snake
    outputs (list): Outputs of the neural network (one for each direction)
    """
    weights = []
    for layer in range(len(self.bias)):
        weights.append(np.vstack((self.weights[layer], self.bias[layer])))

    outputs = np.matrix([inputs])

    for layerWeights in weights:
        outputs = self.addBias(outputs)
```

```
    outputs = outputs.dot(layerweights)
    outputs[outputs < 0] = 0 # ReLU function

    return outputs
```

Cette méthode va prendre les valeurs des poids synaptiques et des poids sur les biais, et effectuer le produit matriciel entre les poids des différentes couches et les entrées. Elle va ensuite y appliquer la fonction d'activation (la non-linéarité) ReLU (*Rectified Linear Unit*) afin de déterminer les valeurs de sorties dans la liste `outputs`.

Les poids synaptiques et les poids des biais sont des paramètres du réseau à trouver. Leur valeur va directement déterminer le comportement du serpent en fonction de sa vision à un moment donné du jeu (un état). Il nous reste donc à essayer d'optimiser les poids, initialement tirés aléatoirement, pour essayer de converger sur une configuration donnant le meilleur comportement possible du serpent.

Cette phase d'entraînement est prise en charge par un algorithme génétique. Lors du lancement du programme en mode « entraînement », c'est la classe `SnakesManager` du fichier `SnakeTrainer.py` qui s'occupera de gérer l'entraînement des générations de serpents.

Dans le fichier `train.py`, lors de la sélection de l'option d'entraînement du serpent, on trouve la définition des hyperparamètres utilisés dans le `SnakesManager` :

```
population = args.population
layers = [20, 10]
mutation = args.mutation
hunger = 150
elitism = args.elitism
snakesManager = SnakesManager(
    game,
    population,
    layersSize=layers,
    mutationRate=mutation,
    hunger=hunger,
    survivalProportion=elitism,
)
snakesManager.train()
```

On utilise ici un réseau avec une couche d'entrée de 26 neurones, puis deux couches cachées de 20 neurones et 10 neurones respectivement et enfin une couche de sortie de quatre neurones (en accord avec les entrées/sorties définies précédemment). Les autres hyperparamètres sont définis lors du lancement du programme. Les valeurs par défaut sont de 1 000 pour la taille de population, 0.01 pour la probabilité de mutation et 0.12 pour le taux d'élitisme. Ils sont tous passés au constructeur de la classe `SnakesManager` et serviront à configurer le comportement de l'algorithme génétique. La méthode `train` de cette classe est ensuite appelée :

```
def train(self):
    # Best game score and fitness so far
    bestScore = -1
```

```

bestFitness = -1
itEnd = 0

# Create new generations until the stop condition is satisfied
while itEnd < 150:
    print(
        f"Generation {self.generation}, best: {bestScore}, bestfit: {bestFitness}"
    )
    self.eval_gen()

    currentScore = self.get_best_gen_score()
    currentFitness = self.get_best_gen_fitness()
    self.change_generation()

    # Check if the game score or fitness for this generation improved
    if currentScore <= bestScore and currentFitness <= bestFitness:
        itEnd += 1
    else:
        # Improvement + reset counter
        bestScore = max(bestScore, currentScore)
        bestFitness = max(bestFitness, currentFitness)
        itEnd = 0

```

Cette fonction va évaluer la génération en cours, puis en créer une nouvelle par le processus d'élitisme-croisement-mutation-sélection et recommencer. Tout cela se répète tant que la nouvelle génération améliore les performances du serpent au moins une fois toutes les 150 générations. La méthode `eval_gen` évalue les différents réseaux de neurones dans une génération lors d'un test de performance. La performance d'un serpent est mesurée à l'aide de la méthode suivante (de la classe `Snake`) :

```

def compute_fitness(self, gameScore):
    """
    Compute the fitness of the snake based on the number of moves done
    and its game score.
    Return the snake's fitness (float)
    """
    bonus = self.get_nbr_move()
    self.fitness = gameScore ** 2 * bonus

    return self.fitness

```

Le score représente le nombre d'unités de nourriture consommées par le serpent. Le bonus correspond au nombre de mouvements qui ont été nécessaires pour atteindre ce score. On calcule ainsi la performance d'un serpent par la fonction suivante :

$$f(score, moves) = score^2 \times moves$$

```
def change_generation(self):
    """
    Creates a new generation of snakes.
    """
    # Sort the snakes by their fitness (decreasing)
    newSnakes = sorted(
        self.snakes, key=lambda x: x.get_fitness(), reverse=True
    )

    # Select best snakes
    newSnakes = newSnakes[: int(self.nbrSnakes * self.survivalProportion)]

    # Generate new snakes
    while len(newSnakes) < self.nbrSnakes:
        # Create a new snake and add it to the next generation
        parents = self.pick_parents_rank(newSnakes)
        baby = parents[0].mate(parents[1], mutationRate=self.mutationRate)
        newSnakes.append(baby)

    # Update
    self.snakes = newSnakes
    self.bestGenFitness = 0
    self.bestGenScore = 0
    self.totalGenScore = 0
    self.generation += 1
```

Lors du passage d'une génération à l'autre, on sélectionne les meilleurs serpents en se basant sur leur performance et on crée de nouveaux serpents par la méthode `mate` de la classe `Snake` (ci-après). Pour ce faire, deux serpents sont sélectionnés en fonction de leur rang. Par la méthode `pick_parents_rank`, on crée une nouvelle génération basée sur le croisement et la mutation entre les serpents de la génération précédente.

```
def mate(self, other, mutationRate=0.01):
    """
    Mate with another state to create a new snake.
    other (Snake): The other snake that mates with this one
    mutationRate (float): The probability for the DNA to mutate
    Returns the newly created snake (Snake)
    """
    newDna = self.dna.mix(other.dna, mutationRate)
    return Snake(newDna)
```

La fonction `mate` fait appel à la méthode `mix` de la classe `Dna`, qui utilise à son tour les fonctions `cross_layer`, `crossover`, `mutate_layer` et `mutate` suivantes :

```
def mix(self, other, mutationRate=0.01):
    """
    Mix the copy of this DNA with the copy of another one to create a new one.
```

```

A crossover is first performed on the two DNA, giving a new one,
then the new DNA is mutated.
other (Dna): The other DNA used for the mixing
mutationRate (float): The probability for a weight or bias to be mutated
"""
newWeights = self.crossover(self.weights, other.weights)
newBias = self.crossover(self.bias, other.bias)
newDna = Dna(newWeights, newBias)
newDna.mutate(mutationRate)
return newDna

def cross_layer(self, layer1, layer2):
    """
    Performs a crossover on two layers.
    layer1 (list): The first layer used to do the crossover
    layer2 (list): The second layer used to do the crossover
    Returns a copy of the result (list)
    """
    lineCut = rd.randint(0, np.size(layer1, axis=0) - 1)
    if len(layer1.shape) == 1: # the layer is only one dimension
        return np.hstack((layer1[:lineCut], layer2[lineCut:]))

    columnCut = rd.randint(0, np.size(layer1, axis=1) - 1)
    res = np.vstack(
        (
            layer1[:lineCut],
            np.hstack(
                (layer1[lineCut, :columnCut], layer2[lineCut, columnCut:])
            ),
            layer2[lineCut + 1 :],
        )
    )
    return res

def crossover(self, dna1, dna2):
    """
    Performs a crossover on the layers (weights and biases).
    dna1 (Dna): The first DNA on which the crossover is performed
    dna2 (Dna): The second DNA on which the crossover is performed
    Returns the crossover of the two DNA (list)
    """
    res = []

    for layer in range(len(dna1)):
        newLayer = self.cross_layer(dna1[layer], dna2[layer])
        res.append(newLayer)

    return res

```

```
def mutate_layer(self, layer, mutationRate=0.01):
    """
    Mutate a layer by adding a value from a gaussian distribution of mean 0
    and standard deviation of 0.5
    layer (list): The layer that is mutated
    mutationRate(float): The probability for a value of the layer to be mutated
    """
    with np.nditer(layer, op_flags=["readwrite"]) as it:
        for x in it:
            if rd.random() < mutationRate:
                x[...] += min(max(rd.gauss(0, 0.5), -1), 1)

def mutate(self, mutationRate=0.01):
    """
    Mutate the DNA.
    mutationRate (float): The probability for a value of the layer to be mutated
    """
    # Mutation of the weights
    for layer in self.weights:
        self.mutate_layer(layer, mutationRate)

    # Mutation of the bias
    for layer in self.bias:
        self.mutate_layer(layer, mutationRate)
```

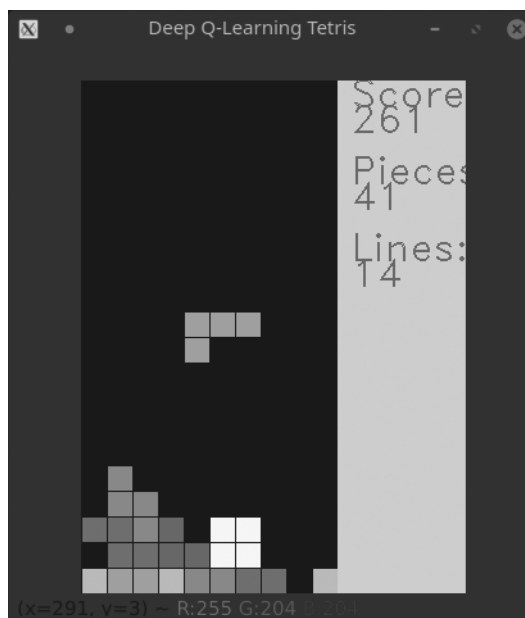
Ces fonctions sont utilisées pour effectuer les opérations de croisement et de mutation sur les différents serpents qui seront sélectionnées pour la prochaine génération. Les paramètres des réseaux de neurones de nos serpents convergent ainsi, au bout d'environ 200 générations, vers un score d'environ 50 à 70 unités de nourriture avalées avant de mourir (il est possible d'obtenir un score encore meilleur après suffisamment de générations).

6

Jouons à Tetris

Figure 6–1

Capture d'écran du logiciel Python mis à votre disposition pour le jeu vidéo du Tetris.



Principe – Choix algorithmique

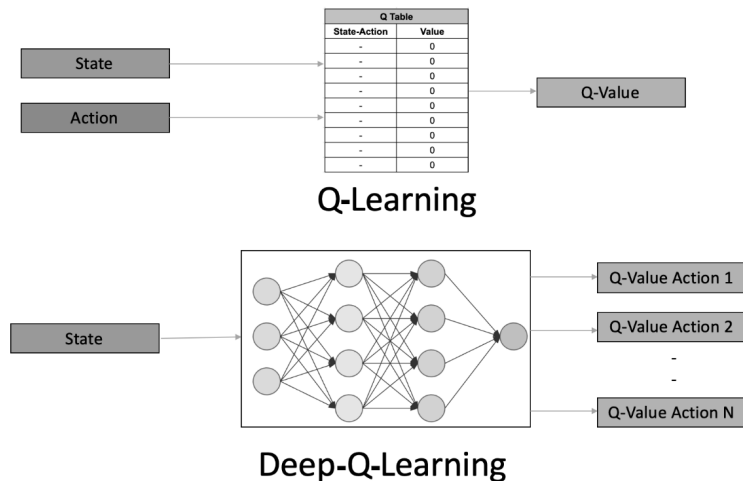
Tetris est un petit jeu vidéo solo très populaire créé en 1984. Le but du joueur est de rester en vie le plus longtemps possible en agissant sur des pièces appelées *tétrominos* qui tombent progressivement afin de se loger dans un plateau. Chaque fois qu'une pièce est placée sur le plateau, une nouvelle est lancée qui se met à tomber également. Le but est de rester dans les limites du plateau et, pour y parvenir, le joueur doit remplir les lignes en agissant sur les tétrominos. Lorsqu'une ligne horizontale est remplie, elle est effacée (ce qui permet au plateau de ne pas monter et remplir l'écran, menant le joueur à sa perte) et le joueur se voit attribuer un certain nombre de points. Il existe de nombreuses manières pour l'IA de jouer au Tetris. Et comme à chaque fois, confusion et hybridation obligent, on retrouve parmi ces tentatives les algorithmes génétiques, ceux de recherche et ceux d'apprentissage par renforcement.

Dans ce chapitre, nous avons choisi d'en utiliser et d'en comparer deux qui ont tout pour se confondre : l'apprentissage par renforcement, plus particulièrement l'apprentissage par renforcement profond, (appelé aussi Deep-Q-learning, la mode étant ces temps-ci d'associer le mot *deep* à tout ce que l'on fait en IA) et l'algorithme génétique. Le Deep-Q-learning fait simplement usage d'un réseau de neurones (qui n'a rien de vraiment *deep*, contrairement à celui du dernier chapitre) pour approcher les Q-valeurs (figure 6-2), car c'est une approche devenue particulièrement populaire pour les jeux vidéo d'un certain niveau de complexité (comme Super Mario ou autre). Il est à noter que, à l'heure actuelle et pour le jeu du Tetris, les approches génétiques semblent également fournir d'excellents résultats parmi toutes celles tentées jusqu'ici. C'est la raison pour laquelle, nous vous en proposons une version possible à la fin du chapitre.

Figure 6-2

Comparaison théorique
entre le Q-learning
et le Deep Q-learning.

<https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>



Nous l'avions déjà évoqué dans le chapitre précédent traitant du Taquin, il n'est pas si évident de différencier le renforcement et le génétique, dans la mesure où tous les deux cherchent à optimiser une séquence d'action, sans réel guidage humain et sur la base d'informations assez succinctes,

comme le résultat obtenu à la suite de cette séquence. Sans doute, la différence la plus importante réside dans l'utilisation ou non de la notion d'état auquel, dans le Q-Learning, se trouvent associées les valeurs des actions possibles qui permettent de passer de cet état à un état suivant. En général, l'algorithme génétique va directement à l'essentiel et s'essaie à optimiser la séquence d'actions sans passer par l'entremise des états. Dans le cas présent, l'algorithme génétique va juste tenter de trouver la bonne pondération des variables décrivant l'état du jeu afin de découvrir les meilleurs coups à effectuer, en fonction de ces variables.

Au vu de la ressemblance évidente entre les deux stratégies, notamment dans l'identification des variables clés décrivant le Tetris, nous allons surtout nous attarder sur l'apprentissage par renforcement laissant, afin d'éviter des redites, la partie pratique du chapitre reprendre plus en détail l'alternative génétique.

L'apprentissage par renforcement pour le Tetris

Il existe généralement deux façons d'appliquer l'apprentissage par renforcement, selon la description que l'on fait de l'état du jeu. La première, la plus directe, est de se baser sur l'image elle-même (approche devenue très populaire, nous l'avons vu, depuis les performances du Deep-Q-learning sur Super Mario et d'autres). La seconde est d'utiliser des paramètres sous-jacents au jeu qui décrivent l'état du plateau et non l'image directement. Le déficit de puissance des processeurs rend l'apprentissage à partir de l'image brute bien trop lent et toute sorte de convergence du Q-learning (qui est extrêmement lent à converger) impossible dans un délai raisonnable. Dans le Tetris, les actions qu'un joueur peut effectuer sont soit de déplacer, soit de faire pivoter une pièce. Le nombre total des états du plateau de jeu est de l'ordre de 2^{200} , puisque chaque cellule peut contenir une pièce ou être vide. Le gigantisme d'un tel espace peut prohiber l'utilisation de l'apprentissage par renforcement, mais il existe des moyens drastiques de simplifier tout cela. Dans un jeu de Tetris, tous les mouvements ne changent pas le tableau ou le score du joueur. Ceux-ci ne sont mis à jour que lorsque la pièce qui tombe cesse de bouger. De plus, étant donné une configuration de tétramino et de plateau de chute spécifique, il n'y a qu'un nombre fini d'états que le jeu peut atteindre après l'ajout du tétramino sur le plateau. Cet ensemble reste inchangé, quelle que soit la séquence des actions effectuées, ce qui simplifie grandement l'environnement initial.

Nous avons déjà rencontré le Q-learning, un algorithme d'apprentissage simple, puissant et surtout peu exigeant en connaissance a priori du problème, juste des récompenses et/ou des punitions. Justement, il peut de ce fait être assez délicat à mettre en œuvre dans des environnements qui se caractérisent par de très nombreux états et actions. Étant donné que la Q-valeur de chaque combinaison état-action est conservée dans une table, la quantité de mémoire nécessaire pour stocker une telle table devient rapidement astronomique et ingérable pour la plupart des ordinateurs personnels. De plus, le temps nécessaire pour explorer tout l'état s'accroît d'autant et pourrait occuper votre processeur pendant des semaines. Une solution possible consiste dès lors à approcher ces mêmes Q-valeurs avec un réseau de neurones recevant l'état du jeu en entrée et produisant la Q-valeur associée aux différentes actions. La prochaine action à exécuter est ainsi déterminée par la sortie maximale du réseau de neurones.

Dans le contexte de ce Q-learning neuronal, la cible du réseau de neurones est en général la valeur optimale de la fonction Q . Ici, la technique utilisée est plutôt celle de la *policy iteration*, qui consiste à prévoir la Q -valeur des meilleurs états successifs et de favoriser parmi ceux-là en entrée celui avec la plus grande de ces valeurs en sortie. Cette méthode présente plusieurs avantages par rapport à l'approche classique, car on réduit la dimension à l'entrée du réseau de neurones, améliorant ainsi le déroulement de l'apprentissage. De fait, plusieurs états précédents peuvent mener au même état successif. Par exemple, imaginez un plateau de Tetris avec quatre lignes totalement pleines, il est possible d'arriver dans cet état depuis de nombreux états dans lesquels on aurait un trou à n'importe quel endroit faisant exactement la taille d'une pièce donnée. Utiliser les états successifs en lieu et place des actions en entrée réduit considérablement l'espace de recherche.

L'apprentissage nécessite un grand nombre d'itérations désignées comme « lots ». Il est aussi conseillé de baser l'apprentissage sur la technique dite de l'*experience replay* : plutôt que d'utiliser l'action la plus récente pour adapter le réseau, on se sert d'un échantillon aléatoire d'expériences antérieures.

Algorithme du Deep-Q-learning

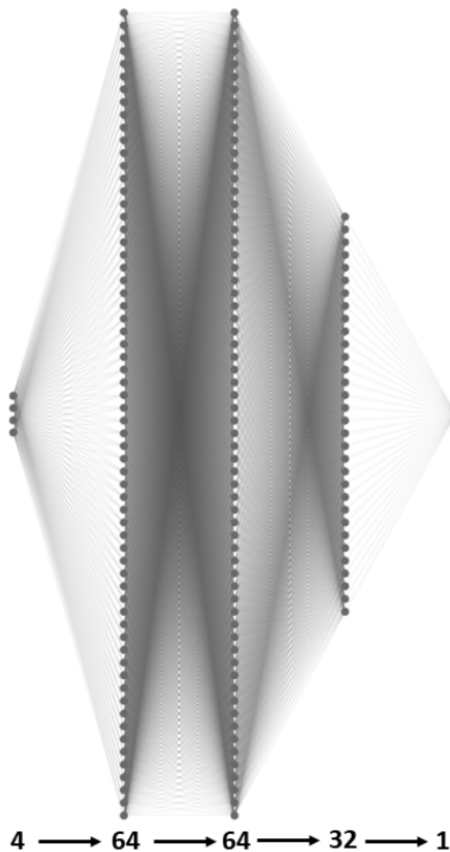
```
Initialiser  $Q(s,a)$  arbitrairement
Initialiser  $s$ 
répéter
    Choisir  $a$  dans  $s$  suivant la stratégie dérivée de  $Q$  (ex.  $\epsilon$ -greedy)
    Appliquer l'action  $a$ 
    Observer  $r$  et  $s'$ 
    si  $s'$  est terminal alors
         $cible \leftarrow r$ 
        Prendre  $s'$  comme nouvel état initial
    sinon
         $cible \leftarrow r + \gamma \max_{a'} Q(s', a')$ 
    Mettre à jour les poids du réseau par rétropropagation avec  $target$ 
jusqu'à ce que le nombre maximum d'itérations soit atteint
```

Dans le code Python mis à votre disposition, le réseau de neurones utilisé est multicouche et entièrement connecté. Il utilise ReLU comme fonction d'activation. Comme la figure 6-3 l'illustre, la topologie du réseau de neurones est 4-64-64-32-1 (on expliquera le premier 4 par la suite). L'optimiseur choisi est du type Adam, qui est devenu le plus populaire pour les réseaux multicouches. Un taux d'apprentissage par défaut a été choisi (0,001) et la sortie du réseau à optimiser est donnée par l'erreur quadratique moyenne.

La couche de sortie produisant la Q -valeur en fonction de l'état, pour déterminer l'état suivant, on les teste tous en les donnant au réseau qui attribue une valeur Q à chacun d'eux. L'état suivant est choisi avec une stratégie favorisant l'exploration au début et l'exploitation à la fin. Les 2 000 expériences les plus récentes sont stockées dans une mémoire tampon. Des lots de 64 éléments sont uniformément choisis pour mettre à jour les poids du réseau. Les scores des expériences passées stockées en mémoire sont prédits par le réseau de neurones. L'IA met à jour les poids du réseau en utilisant l'optimiseur Adam, avec la valeur d'état suivante comme valeur cible et les dernières expériences comme données d'apprentissage.

Figure 6-3

Architecture du réseau de neurones utilisé pour jouer au Tetris.



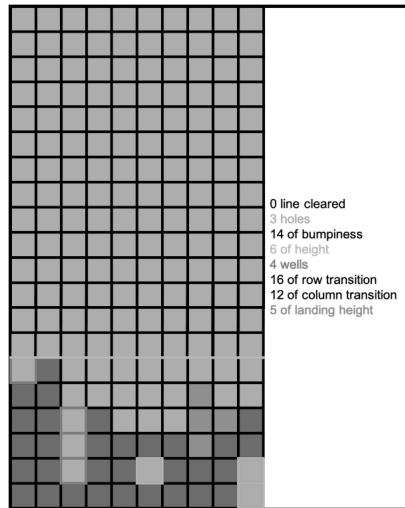
Les différents éléments considérés au départ pour la définition d'un état sont les suivants (figure 6-4).

- 1 Les lignes effacées : nombre de lignes remplies à supprimer à chaque fois qu'un tétramino atterrit sur le plateau.
- 2 Les trous : nombre de trous dans le plateau. Chaque case non remplie avec au moins une case remplie au-dessus est considérée comme un trou.
- 3 La bosse : elle provient de la bosse du plateau à chaque étape du jeu. C'est la somme des différences de hauteur entre chaque colonne et sa voisine de droite.
- 4 La hauteur : hauteur de la case remplie la plus haute sur le plateau.
- 5 Le puits : nombre de cases ayant des voisins à gauche et à droite remplis mais aucune case remplie au-dessus.

- 6 Les transitions de ligne : nombre de transitions entre les cases remplies et vides en parcourant le plateau par ligne.
- 7 Les transitions de colonne : nombre de transitions entre les cases remplies et vides en parcourant le plateau par colonne.
- 8 La hauteur d'atterrissage : hauteur de la case la plus élevée dès l'arrivée d'un tétramino.

Figure 6–4

Illustration des huit variables possibles à utiliser pour décrire l'état du Tetris.



Deux versions de la définition de l'état ont été testées, toutes deux résultant d'expérimentations publiées sur GitHub. La première n'utilisait que les quatre premières variables ; la seconde les utilisait toutes. D'autres définitions d'état ont été testées, mais n'ont donné aucun résultat intéressant et ont été rapidement abandonnées. Finalement, c'est la version la plus simple, retenant les quatre premières variables, qui a donné les meilleurs résultats et justifie donc l'architecture du réseau utilisé. La récompense qui a été retenue, car donnant là aussi les meilleurs résultats, est la suivante : $1 + (\text{largeur du plateau} / 2) \times (\text{nombre de lignes effacées})$. Pour accélérer l'apprentissage, une des astuces a été de le démarrer sur un plateau diminué de moitié, puis de l'accroître graduellement. À l'issue de ces différentes options, le programme a réussi à effacer 21 956,55 lignes en moyenne, un score plutôt impressionnant.

Dans le jeu du Tetris, à l'heure actuelle, la version de l'IA la plus performante est détenue par Gabillon et al.¹. Elle utilise une technique dérivée de l'apprentissage par renforcement, appelée CBMPI (*classification-based modified policy iteration*) avec un record de 51 millions de lignes de Tetris complétées.

1. Gabillon, Victor & Ghavamzadeh, Mohammad & Scherrer, Bruno. (2013). *Approximate dynamic programming finally performs well in the game of Tetris*. Advances in Neural Information Processing Systems.

Le logiciel Python du Tetris en pratique

Le jeu du Tetris est composé des fichiers suivants :

- `tetris.py` contenant le programme de l'interface du jeu ;
- `agent.py` contenant l'architecture et les fonctions associées au réseau de neurones ;
- `main.py` pour tester un réseau de neurones entraîné ;
- `train.py` pour lancer une phase d'entraînement d'un agent sur le jeu utilisant l'apprentissage par renforcement neuronal (Q-learning neuronal ou Deep-Q-learning).

Apprentissage par renforcement neuronal

Ce projet se démarque des précédents par son utilisation de la bibliothèque `tensorflow/keras` afin d'abstraire l'entraînement ainsi que l'utilisation du réseau de neurones.

```
def main(render=True):
    parser = argparse.ArgumentParser(description="The Tetris game")
    parser.add_argument(
        "-w",
        "--weights",
        type=str,
        help="Path to weights file to load.",
        default="weights.h5",
    )

    args = parser.parse_args()
    if not Path(args.weights).is_file():
        parser.print_help()
        print()
        print(f"Invalid path for weight file: {args.weights}")
        sys.exit()

    # launch the environment
    game = Tetris()
    # initialize the agent
    agent = Agent(input_size=4)
    # load previous neural weights into the agents
    agent.load(Path(args.weights))

    running = True
    # while the game is not done, keep doing a step
    while running:
        states = game.get_next_states() # fetch all the next possible states
        action, state = agent.act_best(
            states
        ) # the agent then decides the next action
        score, done = game.step(action, render=render)
        # informs the environment of the move and gives the score
        # and if the game ended
```

```

if done: # if the game ended end the loop and print the score
    running = False
    print(
        f"End Score = {game.tetris_score} \nCleared Lines = {game.total_cleared_lines}"
    )

```

La fonction principale d'utilisation et de démonstration du réseau entraîné fait usage d'un réseau de neurones tel que défini dans la classe `Agent` de `agent.py` et charge les poids synaptiques pré-entraînés depuis le fichier `weights.h5` (par défaut, vous pouvez sélectionner d'autres poids avec l'option `"-w"`). Le jeu se lance alors et demande au réseau de neurones une prédiction de la meilleure action à effectuer à chaque itération du jeu, quand un nouveau tétramino doit être placé.

La classe `Agent` qui contient le réseau de neurones est initialisée de la manière suivante :

```

class Agent:
    """
    The Deep-Q-learning agent.
    It will interact with the Tetris environment
    """

    def __init__(
        self,
        input_size=4,
        epsilon=0.9,
        decay=0.9995,
        gamma=0.95,
        loss_fct="mse",
        opt_fct="adam",
        mem=2000,
        metrics=None,
        epsilon_min=0.1,
    ):
        """
        :param input_size: Number of features given to the NN.
        :param epsilon: Parameter controlling the exploration/exploitation balance.
        :param decay: The epsilon value will decay after each episode by a certain value.
        This parameter defines the rate.
        :param gamma: This is the discount factor in the Bellman equation.
        :param loss_fct: Functions which will calculate the error obtained for the NN
        predictions.
        :param opt_fct: Optimization function for the NN.
        :param mem: Memory size of the past experiences. By default 2 000.
        :param metrics: Those are the metrics monitored during the training phase of the
        neural networks.
        :param epsilon_min: This is the lowest value possible for the epsilon parameter.
        """
        if metrics is None:
            metrics = ["mean_squared_error"]
        self.input_size = input_size

```

```

self.epsilon = epsilon
self.gamma = gamma
self.loss_fct = loss_fct
self.opt_fct = opt_fct
self.memory = deque(maxlen=mem)
self.decay = decay
self.metrics = metrics
self.epsilon_min = epsilon_min
# build the neural network
self.model = Sequential()
self.model.add(Dense(64, activation="relu", input_shape=(input_size,)))
self.model.add(Dense(64, activation="relu"))
self.model.add(Dense(32, activation="relu"))
self.model.add(Dense(1, activation="linear"))
self.model.compile(
    optimizer=self.opt_fct, loss=self.loss_fct, metrics=self.metrics
)

```

On y retrouve notamment l'architecture du réseau de neurones, avec une couche d'entrée, puis deux couches cachées de 64 neurones, suivies d'une de 32 neurones et finalement une sortie unique. Toutes les couches sont entièrement connectées aux suivantes dans un modèle séquentiel classique.

La différence majeure par rapport à l'utilisation du Q-learning pour le taquin vient de l'utilisation d'un réseau de neurones à la place d'une table pour déterminer l'espérance de score (Q-valeurs). Ce changement est bien utile dans le cas où la table Q serait beaucoup trop grande pour être stockée en mémoire et pour être entraînée efficacement. Dans le cas du Deep-Q-learning, on utilise donc un réseau de neurones pour approcher cette table.

Par ailleurs, la différence majeure par rapport au Snake réside dans le mode d'apprentissage du réseau de neurones et les sorties de ce même réseau. Là où le Snake utilisait un algorithme génétique opérant à chaque fin de partie par croisement-mutation, ici, on utilise l'apprentissage par renforcement pour améliorer le réseau de neurones à chaque action (chaque nouveau tétromino).

La particularité de ce programme est qu'il recourt à une variante du Q-learning basée sur les états ultérieurs (*afterstates*). Dans ce cas, on va prendre en entrée du réseau de neurones non pas un couple état-action s, a mais seulement un état ultérieur s' , c'est-à-dire un possible état futur du système, une fois notre action effectuée. On va donc générer, depuis l'état actuel de notre système, les possibles états ultérieurs du système, en appliquant toutes les rotations et translations possibles sur la pièce à placer, et prédire la Q-valeur, grâce à notre réseau de neurones pour chaque état ultérieur s' afin de déterminer le meilleur $Q_{s'}$ (celui qui maximise l'espérance de gain).

Cette approche basée sur les états ultérieurs est possible grâce à la nature déterministe du problème. Lorsqu'on décide d'une action, notre système va évoluer dans le sens de l'action et il n'y aura aucune incertitude sur l'état futur (notre problème est dit non stochastique). Cette méthode présente plusieurs avantages par rapport à l'approche classique : tout d'abord, on réduit la dimension à l'entrée du réseau de neurones, ensuite on rend l'apprentissage un peu plus efficace. En effet, pour le Tetris, il est possible d'arriver dans un état donné depuis plusieurs états précédents. Imaginez un plateau avec quatre lignes totalement pleines ; il est possible d'arriver dans cet état depuis de nombreux états

dans lesquels on aurait un trou à n'importe quel endroit faisant exactement la taille d'une pièce donnée. Cette approche réduit donc considérablement la dimension du problème.

Voici la fonction d'entraînement :

```
def training(render=False):
    """
    The training of the agent with the tetris environment
    """
    parser = argparse.ArgumentParser(
        description="The Tetris game trainer for RL."
    )
    parser.add_argument(
        "-w",
        "--weights",
        type=str,
        help="Path to weights file to save to (default=weights.h5).",
        default="weights.h5",
    )
    parser.add_argument(
        "-e",
        "--episodes",
        type=int,
        help="Number of episodes to train on (default=10000).",
        default=10000,
    )

    args = parser.parse_args()
    if Path(args.weights).is_file():
        parser.print_help()
        print()
        print(
            f"File {args.weights}, already exists, do you want to overwrite ?"
        )
    )
    y = input("Type yes or no: ")
    if y != "yes":
        print("Aborting.")
        sys.exit()

    # --- Initialisation --- #
    game = tetris.Tetris(height=10)
    agent = Agent(input_size=4, decay=0.9995)
    saving_weights_each_steps = 1000
    print("\n >>> Begin Epsilon = " + str(agent.epsilon))
    print(" >>> Decay = " + str(agent.decay))

    # -- Episode LOOP -- #
    for i in tqdm(range(args.episodes)):
        # - Game and Board reset - #
        done = False
        game.reset()
```

```

board = game.get_current_board_state()
previous_state = game.get_state_properties(board)

while not done:
    # Fetch all the next possible states
    possible_future_states = game.get_next_states()
    # the agent then decide the next action
    action, actual_state = agent.act_train(possible_future_states)

    # Performs the action
    reward, done = game.step(action, render=render)

    # Saves the move in memory
    agent.fill_memory(previous_state, actual_state, reward, done)

    # Resets iteration for the next move
    previous_state = actual_state

# Train the weights of the NN after the episode
agent.training_montage()

if i % saving_weights_each_steps == 0:
    agent.save(f"weights_temp_{i}.h5")
agent.save(f"{args.weights}.h5")

print("\n >>> End Epsilon = " + str(agent.epsilon))

```

Notre agent reçoit donc ses quatre entrées : le nombre de lignes retirées, les « trous » dans le plateau, la hauteur totale des pièces et, pour finir, la *bosse*, c'est-à-dire la somme des différences de hauteur entre les colonnes successives.

On calcule alors dans la boucle les états ultérieurs possibles avec la méthode `get_next_states`. On détermine l'action à réaliser dans la méthode `act_train` (basée sur le réseau de neurones actuel ou aléatoire, suivant le compromis exploration-exploitation). On sauvegarde les états et les récompenses, puis on met à jour le réseau de neurones dans la méthode `training_montage`.

```

def training_montage(self, batch_size=64, epochs=1):
    """
    Train the model and adapt its weights using the recent experiences
    :param batch_size: Number of samples used for the training
    :param epochs: Number of iterations of the backpropagation
    """
    if len(self.memory) < batch_size:
        return
    # randomly select a batch of experiences
    experiences = random.sample(self.memory, batch_size)

    # compute the target for the neural network
    next_states = [experience[1] for experience in experiences]
    scores = self._predict_scores(next_states)

```

```

dataset = []
target = []
for i in range(batch_size):
    previous_state, _, reward, done = experiences[i]
    if not done:
        next_q = self.gamma * scores[i] + reward
    else:
        next_q = reward
    dataset.append(previous_state)
    target.append(next_q)
self.model.fit(
    dataset, target, batch_size, epochs, verbose=0
) # train the model
self.epsilon = max(
    self.epsilon * self.decay, self.epsilon_min
) # explore less

```

Afin d'entraîner efficacement le réseau de neurones, on sélectionne un lot aléatoire d'expériences dans l'historique des actions/récompenses sur lequel on lance l'entraînement. On calcule alors la Q-valeur pour chaque expérience du lot et on entraîne le réseau de neurones par rétropropagation. La valeur de ϵ qui contrôle le compromis exploration/exploitation est décrétementée petit à petit à chaque partie.

Algorithme génétique

Dans la version utilisant l'algorithme génétique, le jeu du Tetris se présente sous la forme suivante :

- fichiers contenant le programme du jeu, ainsi que la gestion de l'interface graphique : `Tetris.py`, `TetrisSettings.py`, `TetrisUtils.py`, `GUI_Menu.py`, `GUI_RunMenu.py` ;
- fichier permettant de gérer l'exécution de plusieurs IA en parallèle afin de les entraîner : `TetrisParallel.py` ;
- fichier permettant de gérer l'exécution d'une IA entraînée : `TetrisSolo.py` ;
- fichier contenant l'architecture de l'IA : `TetrisAgents.py` ;
- programme principal pour exécuter une IA déjà entraînée : `evaluation.py` ;
- programme principal d'entraînement : `training.py`.

Choix des coefficients

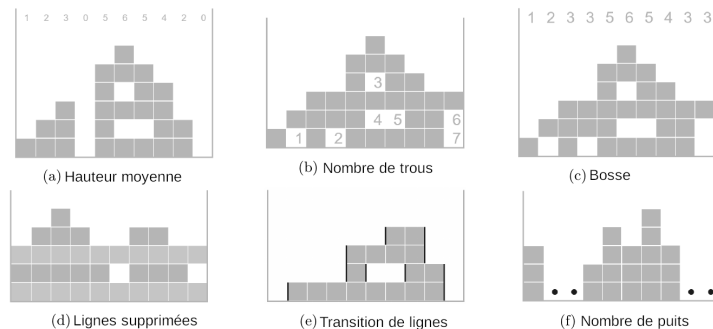
Dans ce projet, la particularité se trouve dans la simplicité de l'architecture du modèle qui décide d'un coup à jouer pour une situation de jeu donnée, lors de l'apparition d'une nouvelle pièce.

En effet, cette architecture se compose d'un simple ensemble de coefficients sur des variables du jeu. Il y a huit variables sur lesquels l'algorithme peut apprendre ; elles sont quasiment les mêmes que pour le Tetris d'apprentissage par renforcement. Certaines sont représentées sur la figure suivante :

- (a) hauteur moyenne des colonnes du plateau ;
- (b) nombre de trous : c'est le nombre de cases vides situées en dessous d'au moins une case remplie ;
- (c) bosse : c'est une mesure de la différence de hauteur des différentes colonnes ;
- (d) nombre de lignes supprimées si un coup spécifique est joué ;
- (e) nombre de colonnes avec au moins un trou, i.e. nombre de transitions en lignes et en colonnes (bords de cases remplies adjacentes à une case vide en ligne ou en colonne) ;
- (f) nombre de puits i.e. de colonnes complètement vides.

Figure 6-5

Six variables d'apprentissage de l'algorithme.



Toutes ces variables sont calculées grâce aux fonctions présentes dans le fichier `tetrisutils.py`, par exemple, la fonction pour trouver le nombre de puits est la suivante :

```
# Get the number of empty column
def get_pit_count(board):
    nb_pit = GRID_COL_COUNT
    for col in range(GRID_COL_COUNT):
        for row in range(GRID_ROW_COUNT):
            if board[row][col] != 0:
                nb_pit -= 1
                break
    return nb_pit
```

Il est possible de sélectionner lesquelles de ces variables seront utilisées par l'algorithme.

Évaluation initiale

Avec un algorithme déjà entraîné, donc avec un ensemble de coefficients déjà optimisés, voici comment fonctionne cette IA.

Lorsqu'une nouvelle pièce apparaît en haut du jeu, les différentes possibilités sont évaluées. En effet, il est possible de simuler toutes les positions finales possibles en plaçant verticalement la pièce dans toutes les colonnes et pour toutes les orientations.

C'est la méthode `calculate_actions` dans la classe `GeneticAgent` de `TetrisAgent.py` qui calcule toutes les possibilités :

```
def calculate_actions(self, board, current_tile, next_tile, offsets) -> List[int]:
    """
    Calculate action sequence based on the agent's prediction

    :param board: the current Tetris board
    :param current_tile: the current Tetris tile
    :param next_tile: the next Tetris tile (swappable)
    :param offsets: the current Tetris tile's coordinates
    :return: list of actions (integers) that should be executed in order
    """

    best_fitness = -9999
    best_tile_index = -1
    best_rotation = -1
    best_x = -1

    tiles = [current_tile, next_tile]
    # 2 tiles: current and next (swappable)
    for tile_index in range(len(tiles)):
        tile = tiles[tile_index]
        # Rotation: 0-3 times (4x is the same as 0x)
        for rotation_count in range(0, 4):
            # X movement
            for x in range(0, GRID_COL_COUNT - len(tile[0]) + 1):
                new_board = TUtils.get_future_board_with_tile(board, tile, (x, offsets[1]),
True)
                fitness = self.get_fitness(new_board)
                if fitness > best_fitness:
                    best_fitness = fitness
                    best_tile_index = tile_index
                    best_rotation = rotation_count
                    best_x = x
            # Rotate tile (prep for next iteration)
            tile = TUtils.get_rotated_tile(tile)

#####
# Obtained best stats, now convert them into sequences of actions
# Action = index of { NOTHING, L, R, 2L, 2R, ROTATE, SWAP, FAST_FALL, INSTA_FALL }
actions = []
if tiles[best_tile_index] != current_tile:
    actions.append(ACTIONS.index("SWAP"))
for _ in range(best_rotation):
    actions.append(ACTIONS.index("ROTATE"))
temp_x = offsets[0]
while temp_x != best_x:
    direction = 1 if temp_x < best_x else -1
    magnitude = 1 if abs(temp_x - best_x) == 1 else 2
```

```

    temp_x += direction * magnitude
    actions.append(ACTIONS.index(("" if magnitude == 1 else "2")
                                + ("R" if direction == 1 else "L")))
    actions.append(ACTIONS.index("INSTA_FALL"))
    return actions

```

Dans cette fonction, on calcule d'abord toutes les situations futures possibles et les nouveaux agencements de plateau qui en résultent, grâce à la première boucle et à la méthode `get_future_board_with_tile`.

Calcul des scores

On évalue ensuite le score de chacun de ces scénarios : les variables décrites précédemment sont multipliées par leur coefficient et le score résulte de la somme de toutes ces multiplications. C'est la position avec le plus grand score qui est choisie pour être jouée.

Ensuite, on convertit la meilleure position finale possible en une liste d'actions à effectuer avec le tetromino (un ensemble de rotations et translations), dans la deuxième partie de la fonction.

De cette façon, cette IA peut se résumer à un ensemble de quatre à huit coefficients, suivant les variables avec lesquelles elle a été entraînée.

Ces paramètres sont stockés dans la classe `GeneticAgent` de `TetrisAgent.py`.

```

class GeneticAgent(BaseAgent):
    """ Agent that uses genetics to predict the best action """

    def __init__(self, weighth_to_consider=[0, 1, 2, 3]):
        super().__init__()
        self.weight_array = []

```

Sélection

Pour entraîner cette IA, l'algorithme génétique (ou algorithme évolutionnaire) entre en jeu. C'est un algorithme qui utilise la notion d'évolution inspirée du génome du vivant.

Dans le processus évolutif, le génome des individus évolue par recombinaison lors de la reproduction et par des mutations aléatoires. Il en va de même ici ; le génome de l'IA est l'ensemble des coefficients.

On va tout d'abord initialiser une population avec des coefficients pris aléatoirement, par exemple une population de 100 individus (paramétrable dans le programme).

On évalue alors tous les individus de cette population initiale. Ils ne seront probablement pas très efficaces, puisque les coefficients sont aléatoires.

Certains, par hasard, le seront plus que d'autres. On va donc classer tous les individus de notre population par le nombre de points gagnés au Tetris avant de perdre la partie.

Les meilleurs individus, par exemple les 50 % de premiers, seront sélectionnés. Les autres sont supprimés et seul le meilleur individu passe sans modification à la génération suivante.

Recombinaison

Tous les autres nouveaux individus sont créés par recombinaison et mutation.

Dans notre programme, cette phase est exécutée dans la méthode `update` de la classe `TetrisParallel` dans `TetrisParallel.py`.

def update(self, screen):

```

    """Called every frame by the runner, handles updates each frame"""
    self.time_elapsed += 1

    # Check if all agents have reached game over state
    if all(tetris.game_over for tetris in self.tetris_games) or (
        self.limit_time != -1 and self.time_elapsed % self.limit_time == 0
    ):
        # Save the generation agents information
        df = save_gen(self.agents, self.tetris_games)
        df.to_csv(f"{self.path}/model_gen_{self.current_gen}.csv", encoding="utf-8",
            index=False)

        # Update the generation
        self.time_elapsed = 0
        # Everyone "died" or time's up, select best one and cross over
        combos = zip(self.agents, self.tetris_games)
        parents = sorted(combos, key=lambda combo: combo[1].score, reverse=True)
        # Update generation information
        self.current_gen += 1
        self.gen_previous_best_score = parents[0][1].score
        if self.gen_previous_best_score > self.gen_top_score:
            self.gen_top_score = self.gen_previous_best_score
        # Undo zipping
        parents = [a[0] for a in parents]
        # Discard 50% of population
        parents = parents[: GAME_COUNT // 2]
        # Keep first place agent
        self.agents = [parents[0]]
        if self.random_run:
            self.agents = []

        # Randomly breed the rest of the agents
        while len(self.agents) < GAME_COUNT:
            parent1, parent2 = random.sample(parents, 2)
            if self.random_run:
                self.agents.append(GeneticAgent(self.heuristics_selected))
            else:
                self.agents.append(parent1.breed(parent2))

        # Reset games
        for tetris in self.tetris_games:
            tetris.reset_game()

```

```
for a in range(GAME_COUNT):
    # If game over, ignore
    if self.tetris_games[a].game_over:
        continue
    self.tetris_games[a].step(self.agents[a].get_action(self.tetris_games[a]))

self.draw(screen)
return pygame.event.get()
```

Lors de cette phase de recombinaison et mutation, on sélectionne les génomes de couples d'individus au hasard parmi les 50 % de meilleurs et on les combine pour donner un nouvel individu. On répète cette opération en sélectionnant des couples aléatoires jusqu'à remplir à nouveau notre population de 100 individus. C'est la méthode `breed` de la classe `GeneticAgent` de `TetrisAgent.py` qui s'en charge :

```
def breed(self, agent):
```

```
    """
    "Breed" with another agent to produce a "child"

    :param agent: the other parent agent
    :return: "child" agent
    """

    child = GeneticAgent(self.weight_to_consider)

    self.crossover(agent, child)
    self.mutate_genes(child)

    return child
```

Dans notre cas, une recombinaison consiste à sélectionner certains coefficients d'un individu et à les remplacer par ceux d'un autre individu. C'est la méthode `crossover` de la classe `GeneticAgent` de `TetrisAgent.py` qui crée le nouveau génome :

```
def crossover(self, agent, child):
```

```
    """
    Crossover the genes of the current agent with another one and modify the genes
    of the child accordingly
    :param agent: the other agent with which the current is breed
    :param child: the child
    """

    for index in self.weight_to_consider:
        if random.getrandbits(1):
            child.weight_array[index] = self.weight_array[index]
        else:
            child.weight_array[index] = agent.weight_array[index]
```

Mutation

On passe ensuite à la phase de mutation, où l'on va appliquer des changements aléatoires aux génomes des individus de la population.

Dans notre cas, cela correspond à changer aléatoirement, sur certains individus, la valeur d'un ou plusieurs coefficient(s) en l'augmentant ou diminuant légèrement. C'est la méthode `mutate_genes` de la classe `GeneticAgent` de `TetrisAgent.py` qui applique les mutations :

```
def mutate_genes(self, child):
```

```
    """
    Apply the mutation of the genes of the child
    :param child: the child on which to apply the mutation
    """
    for index in self.weight_to_consider:
        if np.random.random() < MUTATION_RATE:
            child.weight_array[index] = TUtils.random_weight()
```

Une fois que plusieurs générations (possiblement plusieurs centaines) sont exécutées, les meilleurs génomes convergent vers un maximum de performance.

On ne peut jamais être assuré d'avoir atteint une solution optimale, mais on choisit le nombre de générations de l'algorithme, de manière à avoir une bonne performance, qui ne varie plus beaucoup au cours des dernières générations. On atteint ainsi un optimum local.

La structure du génome peut paraître simple, car elle ne contient que très peu de coefficients (entre quatre et huit) mais, en réalité, une grande partie de l'intelligence est contenue dans la manière dont les différentes variables du jeu sont calculées. En effet, on a vu que les coefficients du génome servent à déterminer le poids donné à chaque variable du jeu ; la qualité de ces variables est donc critique pour que les performances atteintes par l'IA soient les meilleures possibles.

Capture d'écran du logiciel Python mis à votre disposition pour la détection des spams. Il s'agit ici du nuage des mots les plus fréquents dans vos spams.



Principe – Choix algorithmique

Nos boîtes de courriels regorgent de spams¹ (messages indésirables) qui sont souvent identifiés comme tels avant même d'atterrir dans la boîte des messages en effet consultables. Ce rejet n'est pas fiable à 100 % et, malheureusement, il laisse passer beaucoup plus de faux négatifs (spams non reconnus comme tels) que de faux positifs (messages jetés comme spams alors qu'ils n'en sont pas). Tout dépend du problème en question mais, dans les problèmes de reconnaissance binaire ou de détection d'événements (fraude/pas fraude, malade/pas malade...), les faux négatifs sont en général plus dérangeants que les faux positifs, car on omet de prendre les mesures qui s'imposent en ratant le problème. Faux négatifs : on ne détecte pas une fraude, on rate une maladie. Faux positifs : c'est comme une fausse alarme, au pire on prend des mesures inutiles. Ici, évidemment, considérer comme spams des messages qui n'en sont pas est tout aussi enquinant car, pour la plupart, les boîtes de courriel les dissimulent à vos yeux. Ces notions de faux positifs/faux négatifs, propres aux problèmes à deux classes ou à la détection d'événements, sont souvent mal assimilées et prêtent à de nombreux errements. On le sait depuis des siècles, les hommes sont de piètres statisticiens, d'où le coup de main plus que bienvenu des logiciens.

La problématique de reconnaissance de spams que nous allons aborder dans ce chapitre est intéressante à deux titres, car elle mobilise deux domaines très stratégiques de l'IA : l'apprentissage ou classification supervisée (dont ce chapitre constitue une belle porte d'entrée), mais aussi quelques bribes de traitement de langage naturel (l'anglais en l'occurrence ici). La méthode d'apprentissage que nous allons décrire est la plus courante pour ce type de problème et est appelée classification naïve bayésienne. Découvrons-la d'abord de manière très intuitive. Si un courriel vous parvient contenant plusieurs fois les mots « sex » ou « viagra », pour la plupart d'entre vous, il s'agira d'un spam. Si on vous demande de justifier ce jugement, il n'est pas impossible que vous répondiez quelque chose du genre : « les messages qui contiennent ces deux mots sont pour la plupart des spams ». Sans vous en rendre forcément compte, vous venez de raisonner de manière assez bayésienne. Car la célèbre formule de Bayes à la base de l'algorithme d'apprentissage qui suit ne dit rien d'autre.

$$P(\textit{Spam} \mid \textit{Message}) = \frac{P(\textit{Message} \mid \textit{Spam})}{P(\textit{Message})} \cdot P(\textit{Spam})$$

En français : lorsque vous recevez un message, la probabilité que celui-ci soit un spam est égale à la probabilité que la catégorie des spams produise ce message, multipliée par la probabilité d'avoir un spam en général, finalement divisée par la probabilité de recevoir un message. La probabilité que la catégorie des spams produise un message contenant les mots « sex » et « viagra » est plutôt élevée, d'où la probabilité inverse également très élevée qu'un message contenant ces mêmes mots soit un spam (accrue encore par la simple probabilité d'être spammé en général).

1. Nous ne résistons pas à l'envie de vous indiquer le mot canadien utilisé pour spam : « pourriel ». Génial, non ?

On parle aussi de classifieur naïf, car on va baser ce dernier sur l'apparition et la fréquence des mots, pris indépendamment les uns des autres. Ainsi, pour le dire plus mathématiquement, dans notre exemple avec les simples deux mots, on multiplie les deux probabilités :

$$P(\text{Spam} \mid \text{sex}, \text{viagra}) = \frac{P(\text{sex} \mid \text{Spam}).P(\text{viagra} \mid \text{Spam})}{P(\text{sex}).P(\text{viagra})}.P(\text{Spam})$$

On conçoit que cette simplification, naïve en effet, soit problématique, car « sex » et « viagra » n'ont rien d'indépendants et l'apparition de l'un est souvent conditionnée par l'apparition de l'autre. Il existe d'ailleurs bien d'autres méthodes d'apprentissage qui n'exigent pas cette indépendance statistique et peuvent obtenir de meilleurs résultats. Cette méthode de classification automatique a aussi ceci de très intéressant qu'elle met idéalement en relief les deux mécanismes fondamentaux qui sous-tendent quelque classification que ce soit : « similarité » et « fréquence ». Pour classer toute nouvelle donnée, on va d'abord se baser sur la ressemblance (ou similarité) entre cette donnée et toutes celles qui existent dans la base de données d'apprentissage. De manière générale, qui se ressemble s'assemble et la nouvelle donnée a de grandes chances d'appartenir à la classe contenant les données qui lui ressemblent le plus (ici, le message contenant « sex » et « viagra » ressemblant le plus à tous les messages contenant ces mêmes deux mots, on peut étendre cette similarité sémantique à des mots comme « orgasme » ou « verge »). Cette chance sera également d'autant plus grande que cette classe est surreprésentée par ses données. Imaginons que « sex » et « viagra » se retrouvent également dans des messages de type « médicaux » (une troisième catégorie éventuelle, au-delà des spams et non-spams), mais que cette catégorie comprenne très peu de messages, on conçoit que la « fréquence » d'apparition des messages d'une classe influe aussi considérablement sur la classification finale.

$P(\text{sex} \mid \text{Spam})$ se calcule en divisant le nombre de fois que « sex » apparaît dans les spams par le nombre de mots présents dans tous les spams. Et de façon encore plus générale :

$$\frac{P(\text{Spam} \mid \text{Message})}{P(\text{nonSpam} \mid \text{Message})} = \frac{P(\text{Spam})}{P(\text{nonSpam})} \prod_i \frac{P(\text{mot}_i \mid \text{Spam})}{P(\text{mot}_i \mid \text{nonSpam})}$$

Dès lors, pour qu'un message soit déclaré comme spam, il suffit donc que la fraction à gauche de l'égalité soit supérieure à 1. Les mot_i sont tous les mots contenus dans les messages. On voit bien que cet afflux massif de mots peut constituer un problème pour ce type de classification, à cause de toutes ces multiplications pas forcément justifiées. Nous verrons comment parer à ce problème dans la suite. Tout d'abord, comment transformer ce raisonnement bayésien en un véritable algorithme d'apprentissage ?

Dans la plupart des algorithmes d'IA d'apprentissage, tout démarre avec un ensemble de données d'entraînement, le plus souvent déjà prélabellisées (ce n'est pas le cas de l'apprentissage par renforcement) par les classes auxquelles on essaie de prédire que de nouvelles données leur appartiennent. C'est ce qu'on appelle l'apprentissage ou la classification supervisée, car les données d'entraînement savent à quelle classe elles appartiennent. Chaque donnée endosse le label de sa classe. C'est bien le cas ici avec un ensemble de messages dont on sait a priori si ce sont des spams ou pas. À partir de cet ensemble d'entraînement, on va essayer d'élaborer un classifieur automatique de spams, qui sera censé jouer son rôle pour tous les messages à venir.

Imaginons donc un ensemble de 10 000 messages dont une partie n'est que des spams. La démarche classique est de scinder cet ensemble en deux, l'un pour entraîner et l'autre pour tester, c'est-à-dire se faire une première idée des performances de ce classifieur pour tous les prochains messages à venir. Et cette idée, on se la fait en confrontant ce classifieur avec des messages dont on sait bien s'il s'agit de spams ou pas, mais en procédant comme si on l'ignorait. On se fait donc une première idée de la capacité de ce classifieur à traiter les messages en général. Par exemple, on prend 9 000 messages pour l'entraînement et 1 000 pour le test. Bien sûr, dans ces deux sous-ensembles, on essaie de conserver la même proportion de spams et de non-spams.

Une fois l'apprentissage effectué, on évalue le résultat de ce sous-ensemble test par une série de métriques : par exemple, la valeur prédictive positive (*precision*), le rappel (*recall*) ou la précision (*accuracy*) (autant de façons de combiner les faux et vrais positifs et négatifs).

```
precision = true_pos / (true_pos + false_pos)
recall = true_pos / (true_pos + false_neg)
accuracy = (true_pos+true_neg) / (true_pos+true_neg+false_pos+false_neg)
```

Mais comment procède l'apprentissage ? On va se servir de l'ensemble d'entraînement pour calculer la probabilité qu'un mot apparaisse dans les spams :

```
P(mot|spam) = nombre d'occurrences de ce mot dans les spams divisé par le nombre de mots
au total dans les spams
```

Et les non-spams, $P(\text{mot}/\text{non-spam})$ se calculent de manière similaire.

C'est à ce stade que le traitement du langage naturel va nous aider à améliorer l'algorithme et surtout à l'alléger par deux étapes décisives : 1) réduire considérablement le nombre de mots à traiter et 2) donner encore plus de poids aux mots les plus discriminants.

Pour la première étape, nous allons traiter chaque message de trois manières successives : 1) tout mettre en minuscules, 2) supprimer les mots non significatifs ou *stop words* (articles, prépositions, etc.), 3) ramener tous les mots à leur racine (*stemming* ou « racinisation »). La bibliothèque Python NLTK met tous ces outils à notre disposition pour réaliser ces trois étapes, qui ont essentiellement pour résultat de réduire considérablement le nombre de mots à traiter.

Il est possible aussi de travailler sur les paires de mots plutôt que sur les mots simples (on parle alors d'une approche « 2-gram » plutôt que « 1-gram », on peut aussi aller jusqu'à « n-gram ») car, par exemple, « aimer » ou « pas aimer » dans un même texte ne veulent vraiment pas dire la même chose. Et dans pareil cas, il vaudrait mieux conserver la deuxième option. Si on la choisit, on ne recourt plus aux opérations d'élimination des *stop words* et de *stemming*. L'autre étape propre au langage naturel est d'utiliser une mesure complémentaire appelée l'IDF (*Inverse Document Frequency*) et de la multiplier à la première, appelée TF (*Term Frequency*). Cette nouvelle mesure donne plus de poids aux mots vraiment discriminants.

$$IDF(\text{mot}_i) = \log \frac{\text{nombre de messages}}{\text{nombre de messages contenant le mot}_i}$$

Ainsi, l'IDF d'un mot contenu dans tous les messages sera 0. Par exemple, on pourrait imaginer que le mot « vendre » apparaisse dans tous les spams et ait un TF important, mais s'il apparaît aussi dans tous les non-spams et donc finalement dans la totalité des messages, son pouvoir discriminant s'annule.

Cet indice TF×IDF associé à tous les mots d'un document est devenu un grand classique du traitement du langage naturel, notamment lorsqu'il s'agit de rechercher un document associé à des mots-clés (comme lors de la première étape de Google Search pour l'indexation des documents, avant de les hiérarchiser par leur popularité). C'est un type d'approche nommé souvent « sac de mots » (*bag of words*) qui en vient à caractériser chaque document par le TF×IDF de ses mots. Chaque document s'apparente alors à un vecteur dans l'espace à n dimensions constitué par l'ensemble des mots (réduit par les étapes précédentes). Afin de trouver le document le plus proche de quelques mots-clés, il suffit de calculer le sinus de l'angle entre le vecteur que constitue le document dans cet espace et le vecteur associé aux mots-clés.

Une fois les TF×IDF appris pour tous les mots des spams et des non-spams contenus dans l'ensemble d'apprentissage, il suffit, pour classer les messages contenus dans l'ensemble de tests, de comparer $P(\text{Spam}|\text{Message})$ avec $P(\text{non-Spam}|\text{Message})$ (en se basant sur les mots contenus cette fois dans l'ensemble de test) et de prendre le plus grand. Une petite ficelle algorithmique est de mise pour affronter le zéro qui pourrait se produire, si un mot présent dans l'ensemble de test n'a jamais été rencontré avant dans l'ensemble d'apprentissage. Ces deux probabilités se calculent en multipliant les TF et IDF obtenus à partir de l'ensemble d'apprentissage pour tous les mots apparaissant dans les messages de l'ensemble test. Dans le programme Python qui accompagne ce chapitre, on effectue cette classification pour un ensemble de messages (spams et non-spams) disponible sur la plate-forme web Kaggle, un répertoire devenu classique d'ensembles de données pour expérimenter et comparer les algorithmes d'apprentissage. De nombreuses compétitions officielles (récompensées monétairement) sont organisées à partir des données accessibles sur cette plate-forme.

Aujourd'hui, les détecteurs de spams utilisés en production ne se contentent pas d'utiliser les données du texte des courriels pour déterminer si un message est un spam ou non. De nombreuses métadonnées, comme l'adresse IP de l'envoyeur, l'encodage et le comportement global du serveur émetteur vis-à-vis du serveur de réception, sont utilisées pour déterminer si un message est potentiellement indésirable ou pas.

Le logiciel Python du détecteur de spams en pratique

Le détecteur de spams est composé des fichiers suivants :

- `main.py` contenant le programme principal ;
- `spam.csv` contenant des données d'entraînement pour notre algorithme de classification.

Classificateur naïf de Bayes

Avant de pouvoir classer un message en spam ou non-spam, il est nécessaire d'entraîner l'algorithme. Dans ce cas, cela revient à compter les occurrences de mots, les fréquences de mots (*Term Frequency – TF*), ainsi que la fréquence inverse dans le document (*Inverse Document Frequency – IDF*). Cette opération est codée dans la méthode `calc_tf_and_idf` de la classe `SpamClassifier`.

```
def calc_tf_and_idf(self):
    noOfMessages = self.mails.shape[0] # Nombre de messages
    self.spam_mails, self.ham_mails = (
        self.labels.value_counts()[1],
        self.labels.value_counts()[0],
    )
    self.total_mails = self.spam_mails + self.ham_mails
    self.spam_words = 0
    # Nombre de mots dans les spams
    self.ham_words = 0
    # Nombre de mots dans les non-spams
    self.tf_spam = dict()
    # Dictionnaire avec le TF de chaque mot dans les spams
    self.tf_ham = dict()
    # Dictionnaire avec le TF de chaque mot dans les non-spams
    self.idf_spam = dict()
    # Dictionnaire avec l'IDF de chaque mot dans les spams
    self.idf_ham = dict()

    # Dictionnaire avec l'IDF de chaque mot dans les non-spams
    for i in range(noOfMessages):
        # Appelle les bibliothèques nltk
        message_processed = process_message(self.mails.get(i))
        count = list()
        # Pour savoir si un mot est apparu dans le message ou non
        # IDF
        for word in message_processed:
            if self.labels[i]:
                self.tf_spam[word] = self.tf_spam.get(word, 0) + 1
                self.spam_words += 1
                # Calcule le TF d'un mot dans les spams
            else:
                self.tf_ham[word] = self.tf_ham.get(word, 0) + 1
                self.ham_words += 1
                # Calcule le TF d'un mot dans les non-spams
            if word not in count:
                count += [word]
        for word in count:
            if self.labels[i]:
                self.idf_spam[word] = self.idf_spam.get(word, 0) + 1
                # Calcule l'IDF -> le nombre de spams qui contiennent ce mot
            else:
                self.idf_ham[word] = self.idf_ham.get(word, 0) + 1
                # Calcule l'IDF -> le nombre de non-spams qui contiennent ce mot
```

Cette méthode utilise la fonction `process_message` avant de calculer les différentes valeurs de TF et d'IDF et de les ajouter à différents dictionnaires.

```
def process_message(
    message, lower_case=True, stem=True, stop_words=True, gram=1
):
    """
    Cette fonction est très importante car c'est elle qui transforme les messages
    en une liste de mots-clés essentiels : non stop et "stemmés".
    Si gram > 1, ce sont des couples de mots-clés qui sont pris en compte.
    """
    if lower_case:
        message = message.lower()
    words = word_tokenize(message)
    words = [w for w in words if len(w) > 2]
    if gram > 1:
        w = []
        for i in range(len(words) - gram + 1):
            w += [" ".join(words[i : i + gram])]
        return w
    if stop_words:
        sw = stopwords.words("english")
        words = [word for word in words if word not in sw]
    if stem:
        stemmer = PorterStemmer()
        words = [stemmer.stem(word) for word in words]
    return words
```

La fonction `process_message` transforme un message en une liste de mots, en enlevant les mots non essentiels. On associe alors, lors de la phase d'entraînement, une probabilité d'être spam ou non. Pour cela, on utilise soit simplement le nombre d'occurrences du mot dans la classe `BowClassifier`, soit la métrique $TF \times IDF$ telle que présentée dans la méthode qui suit (proposée car étant plus efficace que la première) :

```
def calc_tf_idf(self):
    # Effectue le calcul global avec le tf_idf.
    self.prob_spam = dict()
    self.prob_ham = dict()
    self.sum_tf_idf_spam = 0
    self.sum_tf_idf_ham = 0
    for word in self.tf_spam:
        self.prob_spam[word] = self.tf_spam[word] * log(
            (self.spam_mails + self.ham_mails)
            / (self.idf_spam[word] + self.idf_ham.get(word, 0))
        )
        self.sum_tf_idf_spam += self.prob_spam[word]
```

```

for word in self.tf_spam:
    self.prob_spam[word] = (self.prob_spam[word] + 1) / (
        self.sum_tf_idf_spam + len(self.prob_spam.keys())
    )

for word in self.tf_ham:
    self.prob_ham[word] = (self.tf_ham[word]) * log(
        (self.spam_mails + self.ham_mails)
        / (self.idf_spam.get(word, 0) + self.idf_ham[word])
    )
    self.sum_tf_idf_ham += self.prob_ham[word]

for word in self.tf_ham:
    self.prob_ham[word] = (self.prob_ham[word] + 1) / (
        self.sum_tf_idf_ham + len(self.prob_ham.keys())
    )

self.prob_spam_mail, self.prob_ham_mail = (
    self.spam_mails / self.total_mails,
    self.ham_mails / self.total_mails,
)

```

Une fois les probabilités calculées, on a alors un « sac de mots » (*bag of words*) pour les spams et un pour les non-spams (*ham*), qui associe chaque mot à sa probabilité d'être inclus dans un type de message ou un autre. Il ne reste plus qu'à classer un éventuel message complet grâce à la méthode `classify` de la classe.

```

def classify(self, processed_message):
    # classe les messages du test set
    pSpam, pHam = 0, 0
    for word in processed_message:
        if word in self.prob_spam:
            pSpam += log(self.prob_spam[word])
        else:
            pSpam -= log(self.sum_tf_idf_spam + len(self.prob_spam.keys()))
        if word in self.prob_ham:
            pHam += log(self.prob_ham[word])
        else:
            pHam -= log(self.sum_tf_idf_ham + len(self.prob_ham.keys()))
    pSpam += log(self.prob_spam_mail)
    pHam += log(self.prob_ham_mail)
    return pSpam >= pHam

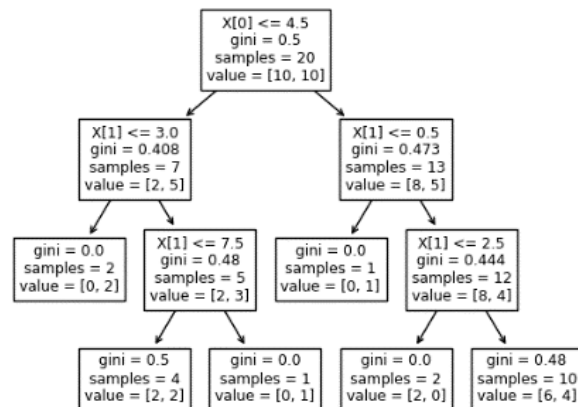
```

On compare alors la probabilité d'être un spam par rapport à celle d'être un non-spam (*ham*) et on en déduit la classification du message.

8

Découvrir les règles d'accès au crédit

Figure 8-1
Arbre de décision
et algorithme d'apprentissage



Principe – Choix algorithmique

Démarrons par un petit récit personnel

L'accès au crédit, algorithme ô combien enquiquinant pour les futurs débiteurs, est un parfait exemple de l'évolution de l'IA depuis sa naissance il y a soixante ans. En tant qu'un des auteurs de

cet ouvrage, nettement plus âgé que mon binôme, permettez-moi de vous relater une expérience personnelle. Il y a de très nombreuses années de cela, je me suis retrouvé « arroseur arrosé », justement lors d'une demande d'obtention d'un crédit pour l'achat d'une voiture dans ma banque d'alors. Un banquier, extrêmement scrupuleux et prudent, me harcela de questions qu'il lisait et répétait depuis son écran d'ordinateur. Les questions portèrent d'abord sur mon état civil, puis se concentrèrent sur ma « bonne santé financière » : situation salariale, possibilité promotionnelle, les emprunts déjà contractés, existence d'autres comptes en banque, biens immobiliers et autres.... À la suite de ces nombreuses questions, le banquier m'informa qu'il lui était impossible de m'accorder cet emprunt. L'ordinateur en avait décidé ainsi. Or, ce programme, particulièrement bien attentionné à mon égard, avait été réalisé par les soins des chercheurs du laboratoire IRIDIA, le même laboratoire des deux auteurs de ce livre à cette époque...

D'antan, on appelait ce type d'algorithme d'aide à la décision un « système expert », de ceux à l'origine des premiers succès applicatifs notables de l'IA. Aujourd'hui, alors qu'il n'y a plus grand-chose à en dire car ils tournent et s'exécutent un peu partout, ils ont jauni et sont relégués dans les oubliettes de l'histoire des technologies. Ils sont devenus, comme tant d'autres avant eux, de simples et utiles logiciels à l'œuvre dans diverses situations : banques, assurances, cabinets médicaux, bureaux d'ingénieurs...

Leur fonctionnement censé reproduire la réflexion menant à la décision d'un expert humain est d'une simplicité déconcertante, car ils doivent tout à Aristote et à ses fameuses règles d'inférence logique de la forme « si A alors B ». Si on apprend au cours du raisonnement que A est vrai, alors il s'ensuit très logiquement que B le devient aussi. Ces règles reproduisent l'état de l'art et sont généralement programmées par l'expert informatique en concertation avec les experts du domaine concerné (ici la banque). Pour le problème de l'emprunt, on pourrait très grossièrement imaginer les règles suivantes :

- « Si le client a moins de 18 ans, alors il n'est pas majeur. »
- « Si le client n'est pas majeur, alors refus de crédit. »
- « Si le client est sans domicile fixe, alors refus de crédit. »
- « Si le client est sans emploi, alors refus de crédit. »
- « Si bien immobilier supérieur à trois fois montant emprunté, alors acceptation de crédit. »
- « Si emploi stable et salaire supérieur à quatre fois remboursement mensuel, alors acceptation de crédit. »
- « Si emprunts contractés inférieurs à salaire mensuel divisé par trois, alors acceptation de crédit. »

La raison à l'enchaînement des règles est que l'exécution d'une première d'entre elles a pour effet de modifier la suite du raisonnement, en rétrécissant et orientant plus précisément le champ d'investigation. Soit les quatre règles suivantes :

- « Si A, alors B. »
- « Si non A, alors C. »
- « Si B et E, alors F. »
- « Si C et G, alors H. »

Supposons que la première question posée porte sur la véracité de A. Dans le cas positif, il est logique pour progresser dans le raisonnement que la prochaine question à poser porte sur E. Dans le cas contraire, elle porterait sur G.

Il y a plus de trente ans de cela, cet emprunt me fut refusé à l'issue, je l'affirme maintenant, d'une décision un peu trop empressée, évitant le moindre risque. En effet, parmi les problèmes rencontrés par ces systèmes experts, figure en bonne place leur trop grande dépendance par rapport à des humains qui, souvent, n'ont pas une connaissance parfaite de leurs processus décisionnels et n'ont pas toute l'expérience requise, surtout face aux situations les plus problématiques, de celles qui justement caractérisent la classe moyenne. Malheureusement pour eux, cette classe est très largement majoritaire et exige une enquête plus approfondie, sinon le recours à la statistique. En effet, deux personnes au profil très semblable peuvent se comporter différemment en matière d'emprunt, ne laissant d'autre choix qu'une approche statistique de la situation, avec la prise de risque qui s'ensuit. Nous sommes de piètres statisticiens, car nos cerveaux sont calibrés pour affronter le vrai et non pas la nuance.

Et l'apprentissage vint à la rescousse

Toujours dans le prolongement de ma malheureuse expérience, on conçoit aisément que, parmi les critères déterminants d'acceptation ou non de l'emprunt, concourent à la fois le salaire et le nombre d'emprunts déjà contractés. S'il est facile pour le banquier de refuser les clients sans revenu fixe et endettés jusqu'au cou, tandis qu'il accepte les milliardaires toujours sans créancier, ces cas extrêmes ne constituent qu'une frange très marginale de la population. L'essentiel est ailleurs et l'incertitude bien présente, avec un salaire et un taux d'endettement moyen, immergeant la statistique dans le processus décisionnel : comme déjà dit, deux clients présentant des profils très similaires pourraient se comporter très différemment dans l'avenir, allez savoir pourquoi !

Les banques capturent dans d'immenses bases de données ce qui est advenu de l'emprunt de leurs clients. Pour le dire le plus simplement qui soit : « cet emprunt fut-il remboursé ou pas ? ». Il existe toute une pratique de l'IA qui ne date pas d'hier mais qui, depuis quelques décennies, les données s'accumulent et les processeurs se musclent, a regagné ses lettres de noblesse. Elle se base sur le traitement de données et l'apprentissage de modèles prédictifs à partir de ces données. C'est une pratique que l'on pourrait résumer par le précepte : « in data veritas » ou mieux encore « la vérité se cache au fond du puits des données ». Elle cherche à automatiser les processus de décision en les recouvrant à partir des très nombreux cas (comme toujours en statistique : le plus le mieux) rencontrés par le passé et qui, très probablement, sont susceptibles de se reproduire encore et de la même manière. Le monde tourne rond, les créanciers des banques également, même si cela n'est pas systématiquement le cas pour leur demande de crédit. Le banquier, quel que soit son QI, n'a pas tout le talent et la rigueur cognitive (surtout si les statistiques s'en mêlent) pour synthétiser au mieux toute la connaissance que l'on peut extraire de ces données brutes. Dès lors, les mêmes règles décisionnelles décrites juste précédemment pourraient non plus résulter de l'explicitation de l'expertise humaine en matière de crédit, mais se trouver découvertes automatiquement à partir de la base de données et d'un algorithme de modélisation exécuté à même ces données.

Ainsi, si seules deux variables, par exemple le salaire et le taux d'endettement, caractérisent tous les clients de la base de données et si chacun se trouve étiqueté positivement ou négativement selon sa

solvabilité, un algorithme de type « arbre de décision » peut induire des règles comme (les quantités n'ont ici aucune importance) : « SI le salaire est inférieur à 100 et le taux d'endettement est supérieur à 300 ALORS le client est peu fiable à 80 % ». Cela signifie que 80 % des clients présents dans la base de données et se trouvant dans cette situation au départ de l'emprunt ne se sont jamais acquittés de leur dette.

D'autres attributs descriptifs des clients (état civil, lieu d'habitation, citoyenneté) sont souvent pris en compte pour améliorer le taux de réussite. Une des forces de ces algorithmes d'apprentissage est que rien ne les empêche d'accumuler toutes ces variables descriptives, même celles qui, comme résultat de l'algorithme, ne seraient d'aucune utilité. L'algorithme le déduira de lui-même. L'existence des erreurs de classification maintient un état d'incertitude qui, pris en compte de façon fréquentielle dans la décision finale, le sera, en fait, de manière optimale, de cette manière qui échappe à l'entendement humain.

Afin de faire mieux encore, les experts en apprentissage machine cherchent à minimiser leur taux d'erreur, non pas sur l'ensemble dit d'apprentissage, mais sur un ensemble à part, non rencontré pendant l'apprentissage, et qui est appelé « ensemble de validation », car le risque existe que le modèle soit surentraîné sur l'ensemble d'apprentissage et perde en pouvoir inductif (la raison première de sa mise en œuvre). Ce prédicteur doit fonctionner au mieux, non plus sur les clients d'hier (ceux-là, on sait comment cela s'est terminé), mais bien sur les clients de demain, présentant leur visage inquiet devant le guichet. Dès lors, tester le comportement de l'algorithme sur un ensemble de validation donne une bonne idée de cette capacité à anticiper l'avenir et la solvabilité des débiteurs en devenir.

Une procédure d'apprentissage automatisé, nourrie par des données expérimentales, peut très avantageusement pallier des défaillances décisionnelles. Ceci explique le succès fulgurant ces dernières années des algorithmes d'apprentissage machine qui cherchent, chaque jour un peu plus, à se débarrasser de l'expertise humaine dans la mise au point d'outils décisionnels.

Retour à mon expérience personnelle : ce nouveau système informatique mit au rebut le système expert jusqu'ici utilisé pour l'attribution des emprunts car, d'après la banque, le petit nouveau semblait en effet fonctionner de manière moins brutale et grossière, surtout pour les personnes aux revenus et à l'endettement moyen, une partie non négligeable de sa clientèle. Ce sont bien des années plus tôt que j'aurais pu exhiber fièrement ma voiture neuve aux copains.

Comment fonctionnent les arbres de décision ?

Après ces propos introductifs et ce récit quelque peu personnel, entrons plus dans le vif de la technique et le codage de ces arbres de décision. Comme nous le savons déjà à ce stade, le langage Python a donné naissance à un ensemble conséquent de bibliothèques extrêmement précieuses pour le développement, la récupération et l'adaptation d'algorithmes classiques de l'IA, surtout ceux impliqués dans le traitement et l'analyse de données. Parmi celles-ci, Panda se repose presque essentiellement sur le concept de *dataframe*, juste un tableau de données dont les colonnes sont les attributs et les lignes les valeurs prises par ces attributs.

Ainsi, installons six anciens clients de la banque dans un tel *dataframe* en y renseignant essentiellement quatre attributs : leur nom, leur solde, leur taux d'endettement et un attribut final booléen

indiquant si oui ou non le remboursement de leur crédit fut satisfait. Les valeurs numériques données aux soldes et à l'endettement n'ont pas la moindre importance ici. Tout ce qui va suivre fonctionnerait de la même manière en présence de valeurs réalistes.

```
# initialisation des valeurs sous forme de listes de listes
data_credit = [["Durant",1,1,0], ["Dupont",1,2,0],
               ["Duschnock",1,3,0], ["DeLon",1,4,1],
               ["Bersini",1,5,1], ["Hasselmann",1,6,1]]

# Création et visualisation du DataFrame pandas

df = pd.DataFrame(data_credit)
columns_name=['Name','Salary', 'Debt','Solvability']
print(df.head())
print(columns_name)
```

Voici le résultat de ce code :

	0	1	2	3
0	Durant	1	1	0
1	Dupont	1	2	0
2	Duschnock	1	3	0
3	DeLon	1	4	1
4	Bersini	1	5	1
5	Hasselmann	1	6	1
['Name', 'Salary', 'Debt', 'Solvability']				

Pratiquement, tous les algorithmes d'apprentissage machine développés en Python s'exécutent à partir d'une telle structure de données, composée très simplement d'un ensemble d'enregistrements sous forme d'attributs/valeurs. En général, la dernière colonne reprend le dernier attribut, celui que l'on cherche à prédire après modélisation (dans le cas présent, la solvabilité du client).

Lorsqu'un tel apprentissage est désigné comme « supervisé », cet attribut-là, précisément la valeur à prédire, prend une importance toute particulière. Dans le cas que nous traiterons ici, d'une classification booléenne, cet attribut ne prend que deux valeurs (0 = solvable, 1 = pas solvable). Pour d'autres types d'applications, il peut bien évidemment prendre davantage de valeurs, comme lorsque nous cherchons à reconnaître des animaux, des objets ou des comportements d'individus plus diversifiés. On parle alors d'une application multi-classes. Le but ici de la classification supervisée est de réussir à prédire la solvabilité d'un nouveau client à venir et cela, à partir de cette base de données, riche de clients dont la banque a enregistré le comportement passé.

Nous avons vu dans le chapitre précédent que les deux mécanismes fondamentaux qui sous-tendent ce type d'algorithme de classification sont la « similarité » et la « fréquence ». En d'autres termes, on prédit qu'un nouveau client devrait en principe se comporter comme la majorité des clients qui lui ressemblent. Ces deux critères, déjà à la base de la classification naïve bayésienne vue dans le chapitre précédent, se retrouvent aussi à la base des arbres de décision, mais mis en pratique d'une manière différente. L'idée cette fois est de construire automatiquement un arbre déci-

sionnel, basé sur des valeurs seuils de ces attributs. Graduellement, les nœuds des arbres ne se composent plus que de clients semblables, jusqu'à idéalement ne retrouver à l'extrémité de l'arbre, c'est-à-dire ses feuilles, que des clients appartenant tous à la même classe, ici tous 0 ou tous 1 pour leur attribut final.

Un concept important dans le fonctionnement de l'algorithme s'avère justement la « pureté » d'un ensemble de clients, c'est-à-dire le degré d'homogénéité de cet ensemble. Pour ce faire, les informaticiens ont récupéré une mesure très populaire en physique appelée « entropie », qui évalue le désordre d'un ensemble (on verra plus loin que l'index de Gini est une autre mesure très utilisée). Supposons qu'un de ces ensembles soit composé de 50 % de clients solvables et 50 % de peu fiables. Dans un tel cas, le désordre est total et l'entropie, maximale, vaudra 1. Dans l'autre cas, idéal, d'un ensemble constitué d'une seule classe de clients, l'entropie vaudra 0. Mathématiquement, la fonction entropique qui réalise cette mesure s'écrit comme suit :

$$\text{entropie} = \sum_i -P_i \log_2 P_i$$

P_i est la probabilité qu'un élément de l'ensemble soit de la classe i (obtenu comme le ratio du nombre des éléments i sur la cardinalité de cet ensemble). Une des lois très importantes de la physique est l'accroissement naturel de cette entropie¹.

L'attribut qui sera favorisé dans l'algorithme sera celui qui, en effet, minimisera cette entropie et qui, lors de la construction de l'arbre, permettra de réaliser la diminution la plus importante en passant d'un niveau à l'autre. Ainsi, dans l'exemple du dataframe donné précédemment, il est clair que cet attribut est le taux d'endettement (*debt*), dont le seuillage a l'effet suivant :

```
if debt < 4 class(client) = 0 else class(client)=1
```

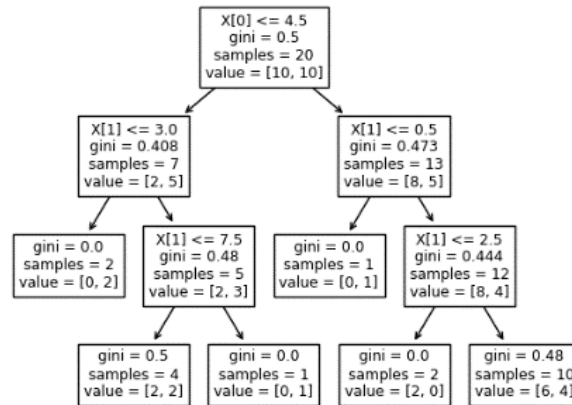
Dans une telle situation, il a suffi d'un seul nœud pour que la classification soit parfaite. C'est évidemment très rarement le cas, la difficulté s'accroissant souvent en présence d'un nombre plus important d'attributs. Le principe est alors de construire l'arbre de manière descendante et récursive, c'est-à-dire en ajoutant graduellement de nouveaux seuillages d'attributs afin de purifier davantage encore les nœuds et en exécutant le même algorithme pour les nœuds parents et les nœuds enfants.

À titre d'exemple, créons un dataframe composé de vingt clients dont le salaire et l'endettement se limitent à des valeurs aléatoires comprises entre 0 et 10. L'algorithme que nous allons faire tourner (et sur lequel se centrera principalement ce chapitre) va nous donner le résultat de la figure 8-2, pour autant qu'on lui fixe au départ le nombre maximum de niveaux que l'on accepte dans l'arbre résultant (trois niveaux ici). Rappelons que $x[0]$ représente le salaire et $x[1]$ l'endettement.

1. Que des parents peuvent vérifier très simplement en observant l'évolution de la chambre de leurs jeunes enfants qui vient juste d'être rangée (les Lego avec les Lego, les poupées avec les poupées, les crayons ensemble, les petites voitures aussi ...), après que ces mêmes enfants y ont joué quelques heures.

Figure 8-2

Arbre de décision
et algorithme d'apprentissage



S'il s'agissait de traduire cet arbre sous forme de règles décisionnelles, voici ce que cela nous donnerait pour chacune des six feuilles de l'arbre (les valeurs numériques ayant été établies de manière aléatoire, les résultats n'ont évidemment aucune vraisemblance d'un point de vue économique et bancaire. Nous importent ici la version et la lecture de l'arbre sous forme de règles) :

- « Si salaire <= 4.5 et endettement <= 3.0, le client n'est pas solvable. »
En effet, les deux clients qui vérifient ces deux conditions de seuil sont tous les deux non solvables.
- « Si salaire <= 4.5 et 3.0 < endettement <= 7.5, le client est solvable à 50 %. »
- « Si salaire <= 4.5 et endettement > 7.5, le client n'est pas solvable. »
- « Si salaire > 4.5 et endettement <= 0.5, le client n'est pas solvable. »
- « Si salaire > 4.5 et 0.5 < endettement <= 2.5, le client est solvable. »
- « Si salaire > 4.5 et endettement > 2.5, le client est solvable à 60 %. »

Car, en effet, sur les 10 clients caractérisant cette feuille, 6 se sont montrés solvables dans le passé.

Observons la figure plus attentivement encore et nous voyons apparaître une valeur nommée Gini. Sans trop entrer dans le détail, cet index joue le même rôle que l'entropie et mesure l'hétérogénéité d'un ensemble de valeurs. C'est la mesure privilégiée par le logiciel que nous avons utilisé car il nous permet aisément une représentation graphique de l'arbre, mais le même arbre aurait été obtenu en utilisant la mesure entropique.

Comment passe-t-on d'un niveau de l'arbre au suivant ? L'algorithme choisit de privilégier l'attribut pour lequel le seuillage adéquat va nous permettre la plus grande décroissance entropique, donc que les nœuds résultant soient les plus homogènes possible. Le gain informationnel ou la décroissance entropique se calcule comme suit (E_i est l'entropie du niveau i , $d(i)$ est la cardinalité de l'ensemble i et les sous-ensembles j sont obtenus comme résultat du seuillage) :

$$Max_j (E_i - \sum_{\text{sous-ensembles } j} \frac{d(j)}{d(i)} E_j)$$

Cette discussion sur le choix de l'attribut, basé sur l'entropie ou le Gini (basés tous deux sur la fréquence relative de chacune des classes), nous permet de comprendre un des avantages capitaux des arbres de décision : ils fonctionnent aussi bien en présence d'attributs qualitatifs que quantitatifs, même si la description essentielle que nous en donnons ici, ainsi que le code détaillé associé à ce chapitre, se limitent aux attributs quantitatifs. De fait, supposons qu'on ajoute à notre base de données bancaire un attribut qualitatif, par exemple le statut civil supposé pouvoir prendre plusieurs valeurs : mineur, majeur, marié, célibataire, divorcé... Il est là aussi possible de séparer les clients en sous-ensembles, selon la valeur prise par cet attribut. Le plus simple reste de sélectionner une valeur contre toutes les autres, mais d'autres manières d'effectuer cette séparation sont possibles, basées sur des valeurs alternatives (par exemple, d'un côté mineur et célibataire et de l'autre majeur et marié), notamment en cas d'un nombre élevé de valeurs. On peut alors imaginer une séparation en plusieurs sous-ensembles ou que chaque sous-ensemble de clients se caractérise, à son tour, par un sous-ensemble de valeurs possibles pour l'attribut en question. Ce célèbre algorithme a donné lieu à de multiples variations selon le type d'attribut en question, mais aussi la manière dont on souhaite concevoir la structure de l'arbre (ex. binaire, ternaire).

Comme le code associé le montrera par la suite, on comprend aussi la nature essentiellement récursive de cet algorithme, car il convient de reproduire la même opération de choix de l'attribut ainsi que la découpe en sous-ensembles les plus homogènes qui soient à chaque niveau. Un autre point très sensible, justement lorsqu'il s'agit de récursivité, est le critère d'arrêt de l'arbre.

On peut décider a priori d'un nombre maximum de niveaux. Cette construction s'arrête évidemment d'elle-même lorsque les sous-ensembles obtenus sont vides ou parfaitement homogènes. Dans la figure 8-2, c'est le cas pour quatre des nœuds dont il n'y a plus lieu de prolonger la découpe. Sans surprise aussi, les informations statistiques que l'on peut extraire dans chacune des feuilles (comme le 60 % de chances que le client soit solvable) sont d'autant plus fiables que l'ensemble des enregistrements est important. Dans notre exemple, il est difficile d'affirmer les 60 % de solvabilité en présence de 10 clients seulement.

On peut aussi décider d'optimiser la structure de l'arbre (par exemple, sa profondeur maximale) en se servant, comme toujours en apprentissage machine, d'un deuxième ensemble, dit de validation, qui nous permettra d'évaluer la structure obtenue à partir de valeurs qui n'ont pas été rencontrées lors de la construction de l'arbre. Par exemple, sur les 20 clients de la banque, on peut choisir de construire les arbres toujours à partir de 15 clients tirés au hasard (en modifiant à chaque fois certains critères comme le nombre de niveaux) et de garder les 5 derniers clients pour évaluer les arbres obtenus. On choisira alors celui qui fournit les meilleures prédictions pour cet ensemble de validation. Ce procédé est un grand classique de la discipline, d'autant plus justifié que l'ensemble d'apprentissage est de taille importante et, là encore, fait l'objet de multiples variations.

On rappellera une dernière fois que cette technologie toujours très populaire dans la communauté d'apprentissage machine doit son succès à sa facilité d'usage, à la possibilité de mélanger différentes catégories d'attributs (quantitatifs et qualitatifs) et, surtout, au résultat présentable sous forme de règles de décision qui sont, jusqu'à un certain niveau de complexité, lisibles et compréhensibles par l'utilisateur. Ses limitations, notamment pour les attributs numériques (à comparer par exemple avec les réseaux de neurones), tiennent à sa manière de garder bien séparés ces attributs et de les tester uniquement en fonction de valeurs seuils. En substance, les séparatrices entre

les classes deviennent automatiquement parallèles aux axes que représentent les attributs numériques. Par exemple, si la solvabilité d'un client devient dépendante d'une relation mathématique particulière reliant le salaire et le taux d'endettement, cette technologie sera mise en difficulté car ayant tendance à les traiter séparément.

De l'arbre à la forêt

Une des techniques devenues les plus populaires en apprentissage machine et plus spécifiquement encore en classification supervisée est de multiplier le nombre d'arbres de décision, en prenant soin de les « affaiblir », et de baser la décision finale sur un traitement d'ensemble (très classiquement, un vote majoritaire). Cet « affaiblissement » consiste par exemple à présenter à chacun des arbres un sous-ensemble des données d'apprentissage ou un sous-ensemble des attributs, ou bien encore la conjonction des deux. La justification intuitive devient que les possibles erreurs dont sera victime chacun des arbres, n'ayant accès qu'à une perception partielle de la réalité, ont de grandes chances d'être très peu corrélées entre elles. La majorité finira toujours par l'emporter et, au total, les chances d'erreurs se verront minimisées par rapport à l'utilisation d'un seul arbre omniscient. Mieux vaut une forêt d'arbres un tant soit peu dégarnis qu'un seul à l'effeuillage imposant.

De l'éthique de l'apprentissage machine

Ce recours de plus en plus important à ces algorithmes d'apprentissage machine, comme les arbres de décision à peine entrevus ou les réseaux de neurones, notamment dans leur version dite d'apprentissage profond, nous oblige à quelques questionnements éthiques fondamentaux. En effet, les algorithmes de ce genre, au vu de la quantité extraordinaire de données aujourd'hui disponible sur les réseaux sociaux ou les sites web interactifs, sont de plus en plus prisés pour épauler nos administrations dans la recherche et l'identification des fraudeurs à l'impôt, ceux qui se jouent de la régulation imposée par les contrats de travail ou qui abusent du droit d'asile. Tout comportement illicite finit toujours par laisser l'une ou l'autre trace sur Internet, que les enquêteurs tentent de repérer et identifier. Des photos d'une croisière sur un yacht, postées fortuitement sur Facebook, s'accordent difficilement avec une déclaration fiscale de type smicard. Rien n'est plus simple pour l'algorithme que de découvrir l'anguille sous roche, l'incompatibilité entre le yacht et le soi-disant revenu, le détail qui finira par crucifier le fraudeur.

En matière d'éthique, ces algorithmes d'accès ou non au crédit, même s'ils se trouvent plus utilisés dans un cadre privé, fournissent malgré tout un excellent cas d'école et d'étude. Il est compréhensible que les banques cherchent à se protéger au mieux des défauts de paiement. Il en va de leur survie et, dès lors, du bien-être des débiteurs diligents. Une faillite bancaire n'arrange pas grand monde. Ce que de nombreux philosophes reprochent néanmoins à la mise en place systématique de ce type de prédicteur est d'amplifier plus qu'il n'en faut une situation déjà défavorable à certains. En substance, la difficulté d'accès au crédit pour les plus démunis ne peut qu'empirer en présence d'un prédicteur froid et rationnel, alors qu'un banquier peut se montrer compréhensif ou se tromper, permettant ainsi à des clients que tout destinait au refus de crédit, d'en bénéficier malgré tout.

Aussi, depuis quelques années, les informaticiens cherchent à comprendre le rôle joué dans les résultats de ces algorithmes prédictifs par des variables a priori très sensibles comme les origines, la

religion, le genre, l'orientation sexuelle, l'état de santé, l'habitat, ou la situation socio-économique. L'usage de ces variables expose ces algorithmes à une critique de nature « morale ». À titre d'exemple, supposons que la variable religion puisse prendre deux valeurs x et y et que l'on souhaite, considérant celle-ci comme sensible et discriminante, supprimer cette variable pour le calcul du risque de crédit. On s'aperçoit alors que les résultats prédits sont beaucoup moins bons, preuve s'il en fallait que cette variable présente un excellent pouvoir discriminant, les x s'avérant beaucoup plus solvables que les y en général. Doit-on malgré tout maintenir son utilisation ? Pour un banquier, cela ne fait pas l'ombre d'un doute et tant pis pour les y ! Pourtant, c'est un choix discutable car, tout en protégeant la banque, on ne concourt pas à faire progresser la société en facilitant l'accès au crédit à des individus y en général déjà largement défavorisés. Vaste dilemme ! Étudier le pouvoir discriminant d'une telle variable sensible n'en demeure pas moins un sujet important et à manipuler avec des pincettes. Et il est plus qu'intéressant de découvrir comment le modèle se comporterait en l'absence de celle-ci, en en neutralisant totalement l'impact (compris sur les autres variables), car il est possible, par exemple, que l'origine ou le nom d'une personne soit un indice très révélateur de sa religion.

De même, les erreurs les plus dommageables, en tous les cas pour les clients, sont celles que les statisticiens taxent de *faux négatifs*, ici le refus d'accorder un crédit à une personne pourtant parfaitement digne de confiance. L'algorithme est alors victime d'un excès de prudence. Une manière de rendre l'algorithme plus éthique est que les catégories de clients que l'on sait défavorisées dès le départ ne soient pas plus maltraitées que les autres, c'est-à-dire ne se prêtent pas à plus de faux négatifs que les autres. En substance, que les clients x et y soient victimes du même taux de faux négatifs. Or, si cette catégorie est majoritaire, le choix d'égaliser le nombre de faux négatifs pour tous les clients, de quelque catégorie qu'ils soient, risque une fois encore de dégrader les performances globales de l'algorithme. En substance, on se trouve en présence d'une opposition de principe entre la qualité des résultats (en termes de nombre d'erreurs) et le souci d'équité de traitement pour la clientèle de la banque. Les praticiens de l'apprentissage machine n'ont généralement comme seule obsession que de minimiser les erreurs de prédiction commises sur les données d'apprentissage et, pour y parvenir, ils s'efforcent de minimiser ces erreurs pour la catégorie majoritaire au détriment de l'autre. Les minorités s'en trouvent affectées, même si rétablir une certaine forme d'équité oblige à dégrader la capacité purement prédictive de l'exercice.

On retrouve exactement le même type de situation paradoxale pour tous les algorithmes de traitement de données dont le résultat risque d'affecter considérablement une catégorie de personnes que la société n'épargne déjà pas. C'est le cas par exemple du très controversé algorithme Compas qui, sur la base de tous les cas précédents de libérations de prisonniers, est capable de prédire si un détenu est susceptible ou non de récidiver et, par là même, d'instruire ou non les requêtes en libération conditionnelle. Sans surprise, aux États-Unis, les prisonniers à la peau noire, vu leur présence majoritaire dans les données traitées, se sont trouvés lésés par l'algorithme malgré des prédictions d'une qualité optimale. Là, le problème est quelque peu différent et il n'est pas du tout évident que l'on puisse aussi rapidement évacuer Compas en le qualifiant de discriminatoire. C'est nettement plus subtil que cela, même si, évidemment, cela n'autorise pas son utilisation tous azimuts. La question essentielle que l'on doit se poser est : « La base de données et sa composition catégorielle est-elle fidèle à la réalité sociale ? ». La présence plus importante de la population noire dans la base de données reflète-t-elle cette même surpopulation dans les prisons ?

Une nouvelle fois, et malgré l'exactitude partielle de ses prédictions, ce n'est pas tant sa qualité prédictive qui doit inquiéter, mais bien davantage l'amplification et la dégradation de la situation qu'il cherche à améliorer. C'est la thèse largement développée dans l'ouvrage de Virginia Eubanks, *Automating inequality*² qui dénonce cette politique algorithmisée qui en vient à réaliser ce qu'elle souhaite justement dénoncer. Ne plus libérer prioritairement les noirs pourrait n'arranger en rien le quotidien des Américains, d'où l'option alternative, par exemple, de mettre tous les faux négatifs sur un pied d'égalité ou, plus audacieux encore, de libérer équitablement toutes les catégories de prisonniers.

Il devient plus qu'urgent de mieux cibler ce qu'on reproche à cet algorithme d'apprentissage, car cela reste l'objet de beaucoup d'incompréhension, d'ambiguïté et de controverses. Pour un informaticien, ce qui compte et doit compter reste avant tout la qualité de la prédiction (que l'on peut aisément évaluer et valider). Pour un juge ou un sociologue, même un algorithme particulièrement précis risque de faire de très gros dégâts et de devenir grandement suspect. Chacun doit garder son rôle et sa partition pour autant qu'on se parle et qu'on se comprenne. Il faut éviter la confusion actuelle entre deux attributions de l'adjectif « mauvais », soit au modèle prédictif soit à la société que ce modèle cherche à décrire et à réguler. Le modèle est « mauvais » si ses prédictions sont « à côté de la plaque » ou si les données qu'il exploite ne sont qu'un très pâle reflet de la réalité sociale (la répartition statistique des variables sensibles n'est pas fidèle à la réalité). La société peut être « mauvaise » si une catégorie d'individus se trouve discriminée, même si le modèle prédit un risque considérable à les adouber. Il est parfaitement absurde d'attendre d'un modèle prédictif qu'il fasse preuve de discrimination positive. Et c'est justement car le modèle nous permet de diagnostiquer l'impact de ces variables sensibles que, et malgré ses prédictions, un décideur (juge, directeur d'école, banquier, responsable d'unité de santé...) peut s'en affranchir et discriminer « positivement » l'une ou l'autre catégorie sociale. Ce modèle est juste un miroir présenté devant la société, aussi cruellement fidèle et précis qu'il soit. Il appartient aux décideurs d'en faire le meilleur usage pour que ses prédictions en matière de criminalité ou autre, ne se stabilisent pas, pire encore, ne s'améliorent pas avec le temps.

Le débat continue de faire rage parmi les informaticiens et les philosophes sur le caractère équitable d'un tel algorithme. Notez que, en sus de l'égalité des faux négatifs, d'autres définitions de l'équité de traitement sont possibles, l'égalité de tous les positifs (vrais et faux) ou, le plus simplement du monde que, parmi les débiteurs, on souhaite autant de religieux $\times$ que de religieux $\forall$, de pauvres que de riches, un point c'est tout. Le choix de l'une ou l'autre manière d'accroître ce caractère équitable s'avère essentiellement arbitraire, ne se résolvant qu'à l'issue d'une discussion entre les principaux concernés, conscients des conséquences de leur choix sur la dégradation des résultats prédictifs mais aussi et surtout sur la société qu'ils souhaitent.

L'arbre de décision en pratique

Pour illustrer la création d'un arbre de décision à l'aide de Python, nous présentons ici une première implémentation originale de l'algorithme, puis on utilisera la bibliothèque `sklearn`, dont l'implémentation standard est aujourd'hui probablement la plus utilisée.

2. Paru en 2018 aux éditions St Martin's Press.

Ces deux implémentations cohabitent dans le fichier `main.py` du projet. En effet, l'élégante simplicité de mise en œuvre d'un arbre de décision fait que la quantité de lignes de code pour ce projet est relativement moins importante que pour les autres projets.

On commence par utiliser `numpy` et `pandas` pour stocker les données d'entraînement. Ces deux bibliothèques sont les plus populaires pour explorer des données et les traiter avec des algorithmes d'apprentissage.

On se basera ici sur un ensemble de données issu de `kaggle`, la plate-forme communautaire d'apprentissage machine la plus populaire. Cet ensemble (accessible sur <https://www.kaggle.com/datasets/sazid28/home-loan>) représente des données de clients qui voudraient faire une demande pour un prêt immobilier.

Le but de la tâche est d'automatiser le processus de décision d'acceptation du prêt, basé sur les données personnelles du client. L'ensemble des données recèle de nombreuses informations : genre, statut marital, éducation, nombre de personnes à charge, salaire, endettement, historique de crédit, etc. (il y a 13 variables en tout). La variable cible (celle à prédire) est le statut du prêt, représenté par la colonne `Loan_Status` dans l'ensemble de données.

On commence par trier la base brute, c'est-à-dire supprimer les données partielles et remplacer les données « catégoriques » par des valeurs numériques. Cette étape de nettoyage est très importante et risque d'affecter considérablement la qualité d'un modèle de prédiction. Ici, nous nous limitons à un traitement basique.

```
# Création et visualisation du DataFrame pandas
df_train = pd.read_csv("HomeLoan_train.csv")

# On élimine toutes les données avec des valeurs manquantes
df_train.dropna(axis=0, inplace=True)

# On remplace toutes les données textuelles par des nombres
df_train.Dependents.replace("3+", 3, inplace=True)
df_train.Dependents = df_train.Dependents.astype("int")

# On encode les données avec des catégories binaires sous forme
# de 0 ou de 1
df_train.Gender = df_train.Gender.apply(lambda x: 0 if x == "Male" else 1)
df_train.Married = df_train.Married.apply(lambda x: 0 if x == "No" else 1)
df_train.Self_Employed = df_train.Self_Employed.apply(lambda x: 0 if x == "No" else 1)
df_train.Education = df_train.Education.apply(lambda x: 0 if x == "Not Graduate" else 1)

# On supprime les colonnes "Loan_ID" et "Property_Area"
df_train.drop("Loan_ID", axis=1, inplace=True)
df_train.drop("Property_Area", axis=1, inplace=True)

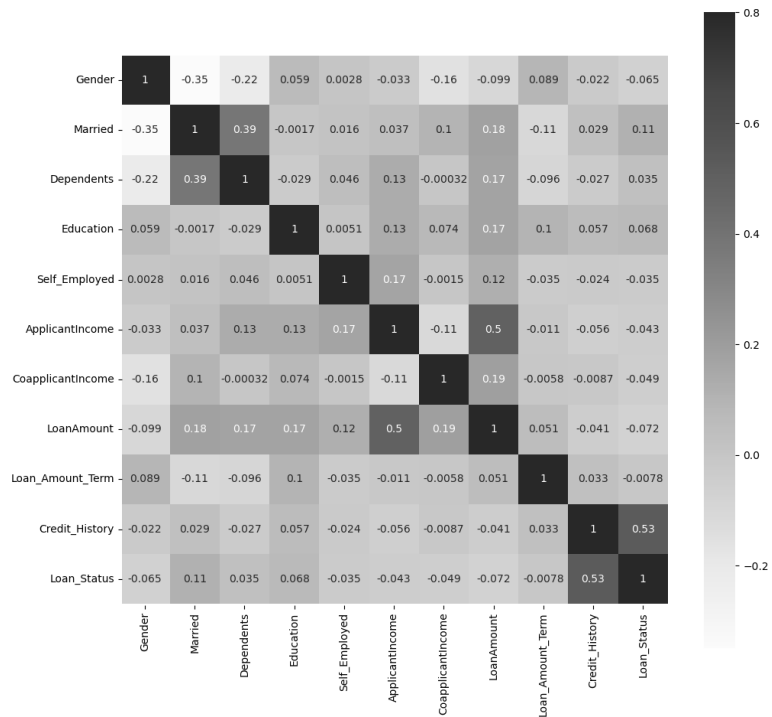
# La variable cible est Loan Status
# On encode la donnée cible à prédire avec des catégories binaires
# sous forme de 0 ou de 1
df_train.Loan_Status = df_train.Loan_Status.apply(lambda x: 0 if x == "N" else 1)
```

On peut ensuite calculer la matrice de corrélation entre les variables, afin de se faire une idée de celles qui influent le plus notre variable cible.

```
# On calcule la matrice de corrélation entre les données
matrix = df_train.corr()
f, ax = plt.subplots(figsize=(10, 12))
sns.heatmap(matrix, vmax=0.8, square=True, cmap="BuPu", annot=True)
plt.show()
```

Figure 8–3

Matrice de données
avec données d'entraînement



On peut également procéder à une étape dite de *feature engineering*, où l'on va essayer de créer de nouvelles variables à partir de celles qui existent (par exemple en les additionnant, ou autre opération mathématique). Cela aide souvent à améliorer le modèle prédictif final, surtout si les relations entre les variables sont plus importantes que ces dernières prises séparément. Nous ne nous attarderons pas sur cette étape et travaillerons directement sur les données brutes.

De manière classique, lorsque l'on définit les données d'entraînement pour un algorithme d'apprentissage supervisé, et notamment en présence de données tabulaires, on appelle x la partie de la matrice des données d'entraînement qui contient les variables d'entrée du modèle et y la partie qui contient la donnée cible, celle que l'on veut réussir à prédire.

On sépare souvent les données d'entraînement de celles de test afin de quantifier la qualité de notre modèle sur des données n'ayant pas servi à son entraînement. Ici, on isole 20 % de l'ensemble des données pour effectuer ces tests.

```
X_train, X_test, y_train, y_test = train_test_split(df_train_x, df_train_y,
test_size=0.3, random_state=42)
```

C'est l'algorithme ID3 (*iterative dichotomiser 3*) qui est implémenté et présenté ici. Il s'agit de l'algorithme classique de création d'arbres de décision, le plus ancien aussi (car de multiples améliorations et alternatives ont été proposées par la suite, tout en respectant l'esprit d'origine).

Le cœur de l'algorithme est la fonction récursive `fit`, qui accepte en paramètre les données d'entraînement (`x` et `y`), l'arbre généré par rapport à ces données et la profondeur de l'arbre.

```
def fit(x, y, par_node={}, depth=0, max_depth=3):
    """
    x: Feature set
    y: target variable
    par_node: will be the tree generated for this x and y.
    depth: the depth of the current layer
    """

    y1 = y.tolist()
    if par_node is None:    # base case 1: tree stops at previous level
        return None
    elif len(y1) == 0:      # base case 2: no data in this group
        return None
    elif all_same(y1):      # base case 3: all y is the same in this group
        return {"val": y1[0]}
    elif depth >= max_depth:    # base case 4: max depth reached
        return None
    else:                      # Recursively generates trees
        # find one split given an information gain
        col, cutoff, entropy = find_best_split_of_all(x, y)

        df_left_idx = x[x[col] < cutoff].index
        df_left = x.loc[df_left_idx]
        df_left_y = y.loc[df_left_idx]

        df_right_idx = x[x[col] >= cutoff].index
        df_right = x.loc[df_right_idx]
        df_right_y = y.loc[df_right_idx]

        par_node = {
            "index_col": col,
            "cutoff": cutoff,
            "val": np.round(np.mean(y)),
        }
        # save the information
        # generate tree for the left hand side data
        par_node["left"] = fit(df_left, df_left_y, {}, depth + 1, max_depth)
```

```
# right hand side trees
par_node["right"] = fit(df_right, df_right_y, {}, depth + 1, max_depth)
depth += 1 # increase the depth since we call fit once
return par_node
```

Cette fonction commence par créer un arbre vide, puis, à chaque itération de l'algorithme, elle parcourt les variables non utilisées des données d'entraînement et calcule l'entropie et le gain d'information de chaque variable. Cette étape est réalisée grâce à la fonction `find_best_split_of_all` suivante :

```
def find_best_split_of_all(x, y):
    """
    Find the best split from all features
    returns: the column to split on, the cutoff value
    and the actual entropy
    """

    col = None
    min_entropy = 1
    cutoff = None
    for c in x.columns:                # iterating through each feature
        entropy, cur_cutoff = find_best_split(x[c], y) # find the best
                                                    split of that feature
        if entropy == 0: # find the first perfect cutoff. Stop Iterating
            return c, cur_cutoff, entropy
        elif entropy <= min_entropy:    # check if it's best so far
            min_entropy = entropy
            col = c
            cutoff = cur_cutoff
    return col, cutoff, min_entropy
```

Cette fonction sélectionne alors la variable avec la plus petite entropie (ou le plus grand gain d'information). Cette variable est donc celle qui sépare le mieux les données d'entraînement. Dans notre cas, on va séparer les données en deux groupes, suivant la variable sélectionnée et avec un seuil qui maximise aussi le gain d'information.

Dans la fonction `fit`, les données d'entraînement sont séparées entre celles qui dépassent le seuil et celles qui ne le dépassent pas. On crée alors deux branches dans notre arbre binaire.

Ensuite, la fonction `fit` se relance, récursivement, sur les deux branches d'arbres nouvellement créées avec les variables restantes et les données triées.

La fonction `fit` cesse de s'appeler récursivement lorsqu'on atteint une condition d'arrêt de création de l'arbre. Ces conditions sont ici au nombre de quatre : soit la branche de l'arbre ne contient plus aucune donnée, soit tous les attributs ont été sélectionnés plus haut dans l'arbre, soit toutes les données d'une branche sont correctement classées (elles partagent la même valeur y), soit on a atteint la profondeur maximale souhaitée pour l'arbre.

Cette stratégie est dite « gloutonne » et ne garantit pas la convergence vers une solution optimale pour la structure de l'arbre. En effet, on va converger vers un optimum local, car on sélectionne à chaque étape la variable qui maximise localement le gain d'information, sans voir plus loin.

Construire un arbre de décision en utilisant `sklearn`

Dans la plupart des projets, le développeur ne va pas se fatiguer à réimplémenter systématiquement tous les algorithmes qu'il doit utiliser. Il ira à l'essentiel et se mâchera la besogne en utilisant probablement une bibliothèque.

L'avantage principal réside dans le fait que l'implémentation est déjà réalisée, ce qui autorise à se concentrer sur d'autres aspects du projet. Un autre bénéfice est que, lorsque l'on utilise une bibliothèque *open-source* ou libre, le code source a potentiellement été testé et vérifié par de nombreux autres utilisateurs. Cela évite les nombreuses erreurs qui arrivent dans le cas où une personne seule réalise toute l'implémentation sans aucun regard extérieur.

Dans le cas présent, nous allons utiliser la bibliothèque `sklearn`, un standard en apprentissage automatique quand on utilise le langage Python. Cette bibliothèque contient une myriade de modèles et algorithmes différents, dont celui qui nous intéresse ici : le modèle de l'arbre de décision. Dans `sklearn`, l'algorithme utilisé est CART, une variante de l'algorithme C4.5, lui-même un successeur de l'algorithme ID3 présenté précédemment.

Son utilisation est très simple : en partant du même jeu de données que précédemment, il suffit d'exécuter la méthode `fit` du modèle en lui passant les données d'entraînement pour entraîner un arbre de décision avec les paramètres par défaut.

```
# fit the decision tree classifier
tree2 = tree.DecisionTreeClassifier(max_depth=max_depth,
                                   random_state=42)
tree2.fit(X_train, y_train)
```

Il est aussi possible de visualiser l'arbre ainsi créé, grâce à la fonction `plot_tree` :

```
tree.plot_tree(tree2)
plt.show()
```

Nous avons évoqué l'usage d'un ensemble d'arbres dans le but d'améliorer l'efficacité de l'algorithme, il s'avère que `sklearn` implémente aussi ce genre de modèle, appelé *random forest*, ou forêt d'arbres décisionnels. Dans `sklearn`, on peut l'utiliser de manière analogue à son homologue à un seul arbre :

```
# fit the random forest classifier
forest = ensemble.RandomForestClassifier(max_depth=max_depth)
forest.fit(X_train, y_train)
```

L'utilisation d'ensembles de modèles offre un avantage par rapport à l'arbre de décision seul, notamment en limitant le surapprentissage dont est victime le modèle de base.

Lors de l'exécution du programme, on sélectionne la profondeur maximale que l'on désire pour nos arbres. Le programme va entraîner les modèles avec l'ensemble des données d'entraînement et calculer la justesse des modèles sur l'ensemble de test. La justesse (*accuracy*) est définie comme le nombre de prédictions correctes sur le nombre total de prédictions faites sur un jeu de données. C'est une métrique classique de quantification des modèles en apprentissage automatique, bien qu'il en existe beaucoup d'autres.

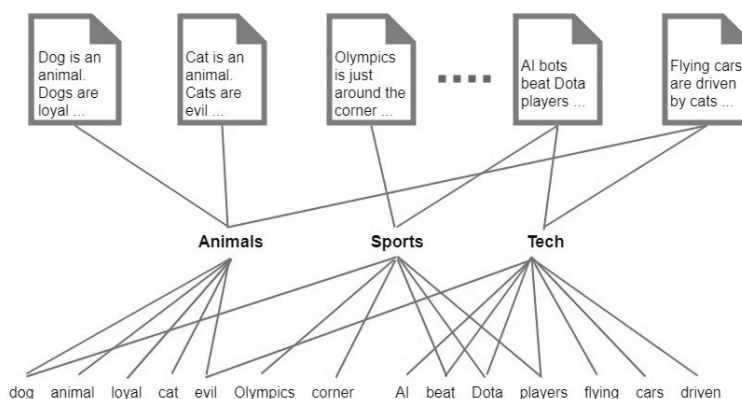
Lorsqu'on observe les résultats sur les différents modèles, on remarque plusieurs choses :

- Avec une profondeur maximale grande, les arbres vont devenir extrêmement précis sur les données d'entraînement et essayer de toutes les prévoir. Toutefois, ils vont surapprendre, ce qui aura pour effet de diminuer la justesse des prédictions sur l'ensemble des données de test.
- Le modèle de forêt d'arbres, devenu extrêmement populaire dans la communauté de l'apprentissage automatique, donne généralement une meilleure justesse que les deux autres modèles d'arbres.

9

Aider à trier la presse ou les avis de clients

Figure 9-1
Résultats de l'algorithme LDA



Principe – Choix algorithmique

Supervisé et non supervisé

Notre laboratoire IRIDIA a souvent été mis à contribution pour des missions consistant à organiser de gros corpus de données textuelles non structurées. Par exemple, un fabricant important, dont

nous taïrons le nom par souci de confidentialité, reçoit chaque jour des centaines de messages en provenance de ses clients, via plusieurs sources : conversations téléphoniques transcrites, courriels, commentaires sur les réseaux sociaux ou directement sur des sites dédiés aux produits en question. À la fin, il s'agit d'un gigantesque corpus textuel dont les origines et les contenus sont globalement semblables : les commentaires clients et la nature de ces commentaires sur les produits du fabricant. Ces messages arrivent en grande quantité et en flux constant. La mission qui nous fut confiée à l'époque était d'aider le fabricant à mieux organiser ces commentaires, c'est-à-dire à les trier par sujet, tout en en répertoriant et en en découvrant les thèmes les plus prégnants. Autre mission assez semblable, un éditeur de presse nous avait également sollicité pour un triage thématique, mais plus efficace, des coupures de presse qui lui parvenaient. Idem encore pour un bureau d'avocats, à partir cette fois de documents juridiques dont il espérait une organisation thématique et une catégorisation automatisées, notamment pour en faciliter la recherche et l'indexation.

C'est le moment propice pour bien saisir la différence entre deux types de traitement de données que l'on désigne souvent comme « supervisé » et « non supervisé ». Pensons par exemple à l'éditeur de presse. Il est plus que probable que celui-ci connaisse déjà les thématiques principales auxquelles se consacrent les coupures en question. À l'heure où nous écrivons ces lignes, nous aurions par exemple : guerre en Ukraine, disparition de la reine d'Angleterre, inflation, flambée des prix de l'énergie, poussée de l'extrême droite en Europe... Il est aussi très possible que quelqu'un dans ce bureau de presse ait pris sur lui dans le passé de pré-classer les documents selon la thématique abordée, voire qu'il continue de le faire. Si un tel corpus pré-classé nous parvient, on parle alors d'un ensemble de données étiquetées. Chaque texte appartient déjà à une catégorie et l'apprentissage qui s'ensuit devient de type supervisé. Il suffit d'apprendre à reproduire le classement de nouveaux documents en se basant sur ceux qui le sont déjà. C'était par exemple le cas d'un des chapitres précédents, automatisant le classement des messages reçus en courriels et pourriels.

En général, non seulement la classification supervisée s'avère plus simple que celle que nous abordons dans ce chapitre mais, de surcroît, il est assez facile de valider la qualité de l'algorithme obtenu à partir d'un ensemble de données à part. Vous proposez cet algorithme de classification automatique à votre commanditaire, tout en l'informant des performances que vous pensez pouvoir lui attribuer. Par exemple, il a fonctionné à 95 % sur les données de test et pourrait faire encore 5 % d'erreur lors de son utilisation prochaine. En substance, il est possible d'objectiver jusqu'à un certain point la qualité ou la fiabilité du logiciel en question. Ce n'est pas le cas en classification non supervisée. On ne sait même pas à combien de catégories on est en droit de s'attendre.

De son côté, il est très probable aussi que, juste en lisant les messages des clients, le fabricant ait une petite idée de leur nature : négatifs ou positifs, sur le prix ou sur un autre aspect propre au produit... Là encore, une pré-classification aura pu être effectuée, ramenant dès lors notre apport à une classification de nouveau supervisée, en général la plus simple à mettre en œuvre et la plus fréquemment demandée dans les entreprises.

Dans ce chapitre, il s'agit de tout autre chose ; nous nous mettons délibérément dans une situation plus délicate, qui se rencontre néanmoins assez fréquemment. Les documents que l'on nous propose de trier ne sont plus pré-étiquetés et c'est justement cela que nous devons découvrir : leur possible étiquette et la manière dont ils se répartissent selon ces catégories.

Nos commanditaires cette fois nous mettent au défi de parvenir à améliorer et même modifier la manière dont jusqu'ici ces documents, ô combien sensibles pour eux, ont été thématisés et répartis. Il peut en être ainsi si le fabricant n'est pas totalement satisfait du classement actuel, ou pour l'éditeur de presse qui se demande si l'on n'a pas eu trop tendance à réunir dans une même catégorie des messages qui, finalement, se distinguent de façon évidente pour qui les relit attentivement. Une autre raison est que la classification utilisée jusqu'ici est devenue obsolète et que de nouveaux sujets sont apparus, qui ont beaucoup de mal à rentrer dans les catégories existantes. Une nouvelle thématique est en train d'éclorre ; il faut la saisir au bond. L'apprentissage non supervisé est délicat à mettre en œuvre, car il est difficile d'évaluer objectivement la qualité du résultat ; il faut, dans le même temps, découvrir les classes et y répartir les documents. On ne peut donc plus se reposer sur un ensemble test pour valider la qualité du résultat. On procède bien davantage à l'aveugle, l'appréciation du résultat final devenant souvent l'objet d'un regard subjectif du connaisseur du domaine.

L'algorithme LDA (Latent Dirichlet Allocation)

Parmi les algorithmes de classification non supervisée de données textuelles, un des plus populaires est le LDA qui, à partir d'un corpus documentaire, débouche sur deux résultats **de nature probabiliste** : un ensemble de sujets, chacun caractérisé par des mots-clés de pertinence décroissante, et, pour chacun des documents, la probabilité qu'il se réfère à l'un de ces sujets. Il découvre donc cet ensemble de sujets dissimulés jusqu'à présent de manière « latente » dans les documents.

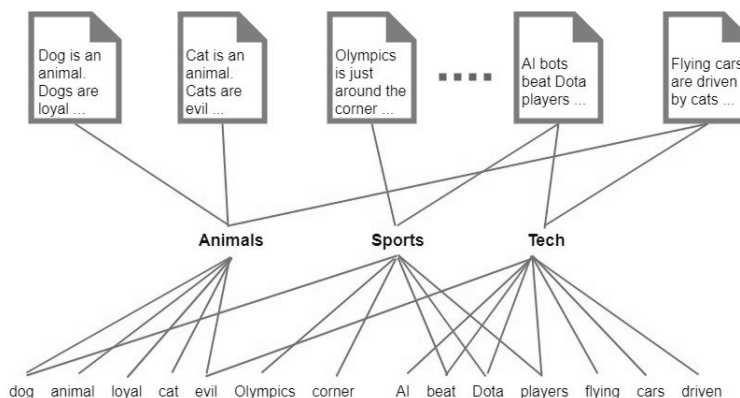
Le tout s'effectue par un tour de passe-passe de nature statistique. La base de l'algorithme est la recherche de deux distributions de probabilités :

- $P(\text{Sujet}(j)/\text{Doc}(k))$ (en substance, l'importance que prend le sujet j pour le document k)
- et $P(\text{Mot}(i)/\text{Sujet}(j))$ (l'importance qu'a le mot i dans la description du sujet j).

De cette première formalisation statistique, nous déduisons déjà deux aspects importants. Premièrement, **chaque document est décrit par un ensemble de mots, chacun avec sa propre pondération**. Il devient donc important de les passer à la même « moulinette » algorithmique, comme nous l'avons fait avant de trier les courriels : supprimer les *stop-words*, ramener les mots à leur racine et d'autres manipulations qui visent à diminuer leur nombre. Deuxièmement, mais comme c'est très souvent le cas dans les algorithmes de classification non supervisée, **l'algorithme LDA de base démarre avec un nombre prédéfini de sujets**. On fixe ce nombre en espérant en découvrir la nature et qu'il soit une espèce d'optimum. Il reste toujours possible de réaliser une succession de tests, en modifiant ce nombre, jusqu'à aboutir à ce qui paraît **subjectivement** comme le découpage le plus satisfaisant.

Une autre structure de données très importante dans le fonctionnement de l'algorithme est la suivante : $P(\text{Mot}(i), \text{Sujet}(j), \text{Doc}(k))$, qui affecte chaque mot i du document k à un sujet j particulier. Les deux résultats de l'algorithme sont représentés par la figure 9-2.

Figure 9–2
Résultats de l'algorithme LDA



L'algorithme fonctionne suivant quatre étapes successives :

- 1 D'abord, on affecte aléatoirement chaque mot dans chaque document à un sujet quelconque. Ce faisant, on initialise aléatoirement les « sujets » pour chaque « mot » dans la structure $P(\text{Mot}(i), \text{Sujet}(j), \text{Doc}(k))$.
- 2 De là, par simple comptage, on peut automatiquement calculer les deux autres distributions de probabilités : $P(\text{Sujet}(j), \text{Doc}(k))$ et $P(\text{Mot}(i), \text{Sujet}(j))$.
- 3 Ensuite, un grand nombre d'itérations démarre (ce nombre est déterminé au départ), qui concerne chaque document et chaque mot dans ce document.
- 4 À chaque itération, on remplace l'affectation d'un mot à un sujet particulier dans un document particulier. Le nouveau sujet pour ce document est décidé en multipliant les deux probabilités conditionnelles : $P(\text{Sujet}(j), \text{Doc}(k))$ et $P(\text{Mot}(i), \text{Sujet}(j))$. En effet, on cherche le sujet j qui maximise le produit de ces deux probabilités. Un mot est d'autant plus représentatif d'un sujet que, à la fois, il est important pour ce dernier (deuxième probabilité) et ce sujet est important pour le document dont est issu le mot (première probabilité).

On converge ainsi vers les deux distributions de probabilités qui nous intéressent vraiment : celle qui nous donne la description des sujets par les mots qui les constituent et celle qui nous informe sur l'affiliation probabiliste des documents dans les sujets en question : le résultat attendu. Il existe ensuite de nombreux traitements (dans de nombreuses bibliothèques Python), qui aident à mieux comprendre et/ou visualiser ces deux distributions de probabilités.

En parcourant le code Python exécutant cet algorithme, vous découvrirez d'autres paramètres : α et β , qui, en substance, évitent d'abord les effets pervers des zéros dans ces deux distributions. Ainsi, le démarrage aléatoire pourrait n'avoir attribué aucun sujet à un document. Quand bien même le sujet est important pour ce document, l'algorithme fonctionnera correctement malgré l'apparition des zéros au dénominateur. Ensuite, ces deux paramètres influencent également l'étalement des deux distributions : plus les valeurs sont petites, plus les distributions seront étalées. C'est la raison de l'appellation « Dirichlet », du nom de ces distributions statistiques qui dépendent de l'utilisation de ce paramétrage. En général, les utilisateurs font confiance aux valeurs

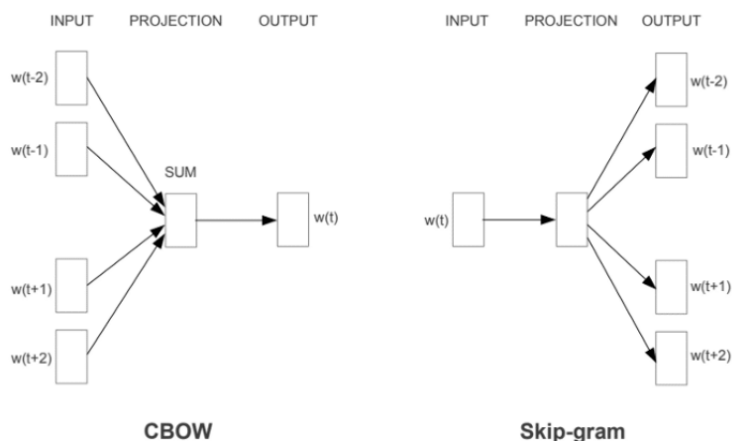
données par défaut. Le code qui vous est fourni décortique une version assez simple et compréhensible de l'algorithme LDA mais, sans surprise, de nombreuses versions de cet algorithme sont disponibles dans les bibliothèques Python.

Word2Vec et Doc2Vec – Catégoriser le sens profond et non plus les mots

Au grand dam des linguistes purs et durs, le traitement du langage naturel a bénéficié de très considérables améliorations par l'entremise de l'apprentissage machine, des manipulations statistiques et des réseaux de neurones. L'une de ces améliorations consiste à transformer les mots en les plongeant dans un espace vectoriel qui en capture beaucoup mieux la signification profonde. Ce plongement lexical est souvent repris en anglais par l'expression *word embedding*. L'idée est de capter la signification profonde de ces mots dans les poids des réseaux de neurones utilisés, en prenant en considération leur contexte d'utilisation, qui se ramène dans la plupart des cas aux mots avoisinants. La signification des mots se ramène dès lors à l'étude statistique de leur voisinage. Il y a deux manières « neuronales » différentes de procéder (reprises dans la figure qui suit par les expressions « CBOW » et « Skip-gram »), selon que l'on se sert du contexte avoisinant pour prédire le mot, ou que l'on se sert du mot pour prédire le contexte avoisinant (figure 9-3).

Figure 9-3

Exemple d'étude de voisinage pour prédire un mot ou le contexte avoisinant



À force de traiter les mots dans leurs différents contextes d'utilisation, cette manière de procéder permet de transformer chaque mot en sa signification reprise par les statistiques de son mode d'emploi. L'important dans les deux cas est que la signification du mot se retrouve dans les poids des réseaux de neurones, un vecteur que l'on peut utiliser pour des manipulations sémantiques intéressantes, se ramenant à des opérations vectorielles. La plus connue et la plus rabâchée de cette transformation des mots en vecteurs s'appelle Word2Vec. Elle permet de réaliser des opérations arithmétiques aussi surprenantes que :

« Roi » - « Homme » + « Femme » = « Reine »

Alors que Word2Vec se limite à vectoriser de simples mots, Doc2Vec va plus loin et permet de vectoriser des documents entiers. Joindre Word2Vec et Doc2Vec à l'opération de catégorisation des documents permet de relancer cette catégorisation, mais en se servant cette fois non plus des mots en tant que tels, mais des vecteurs de leur signification. Il est possible d'obtenir ces vecteurs, par apprentissage, juste en partant des documents en question, comme on le fait dans le code associé à ce chapitre. Toutefois, il y a beaucoup plus intéressant encore : c'est de partir des vecteurs déjà découverts au départ de nettement plus de documents, mis à votre disposition dans les bibliothèques généralement rendues disponibles par les Gafam tels Google¹, un acteur très important, sinon incontournable, dans la mise au point d'algorithmes de langage naturel. Il n'est alors plus nécessaire d'apprendre quoi que ce soit, mais de simplement récupérer les vecteurs associés aux mots et de coupler ces vecteurs afin de vectoriser également les documents. L'idée d'utiliser des structures neuronales qui résultent d'un apprentissage déjà conséquent effectué préalablement à partir de très importantes bases de données est devenue plus que populaire dans la communauté de l'apprentissage profond. On la retrouve mise à l'œuvre autant dans le traitement des images que dans celui du langage naturel, sous la dénomination de *transfer learning*.

À partir de ces nouveaux vecteurs, un des algorithmes de classification non supervisée des plus populaires se nomme *KMeans*, qui découvre, dans un espace vectoriel rempli de points, la manière la plus naturelle de répartir ces derniers en un nombre de groupes (*clusters*), là aussi fixé au départ de l'algorithme. C'est un algorithme plus que classique disponible dans de nombreuses bibliothèques Python.

Le logiciel Python de la catégorisation de documents en pratique

LDA

Le code attaché à ce chapitre traite des documents anglais, mais rien ne vous interdit de procéder aux modifications nécessaires pour traiter des documents français. Il démarre avec un ensemble de huit phrases :

```
documents = [  
    ['The sky is blue and beautiful.'],  
    ['Love this blue and beautiful sky!'],  
    ['The quick brown fox jumps over the lazy dog.'],  
    ["A king's breakfast has sausages, ham, bacon, eggs, toast and beans"],  
    ['I love green eggs, ham, sausages and bacon!'],  
    ['The brown fox is quick and the blue dog is lazy!'],
```

1. Comme ici : <https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz>.

```
[ 'The sky is very blue and the sky is very beautiful today'],
[ 'The dog is lazy but the brown fox is quick!']
]
```

Il prend également un ensemble de sujets. Ici, on les limite à trois (ce nombre doit être fixé au départ) :

```
topic_names = ['food', 'weather', 'animals']
```

Tous ces documents nécessitent une première série de traitements fournis dans les bibliothèques Python correspondantes, qui ne conservent que les racines des mots les plus significatifs associés à chaque document :

```
def pre_process_documents(doc):
    wpt = nltk.WordPunctTokenizer()
    stop_words = nltk.corpus.stopwords.words('english')
    for i in range(len(doc)):
        # lower case and remove special characters\whitespaces
        doc[i][0] = re.sub(r'[\^a-zA-Z\s]', '', doc[i][0], flags=RegexFlag.IGNORECASE | RegexFlag.A)
        doc[i][0] = doc[i][0].lower()
        doc[i][0] = doc[i][0].strip()
        # tokenize document
        tokens = wpt.tokenize(doc[i][0])
        # filter stopwords out of document
        filtered_tokens = [token for token in tokens if token not in stop_words]
        doc[i] = filtered_tokens
    return filtered_tokens
```

Ensuite, nous définissons les deux distributions de probabilités qui sont à la base de l'algorithme :

- d'abord l'importance que prend le sujet `topic` pour le document `d` :

```
def p_topic_given_document(self, topic, d, alpha=0.1):
    """
    P(topic|d,Alpha)
    The fraction of words in document d
    that are assigned to topic (plus some smoothing)
    """
    return ((self.document_topic_counts[d][topic] + alpha) /
            (self.document_lengths[d] + self.K * alpha))
```

- ensuite, l'importance que prend chaque mot `word` pour le sujet `topic` :

```
def p_word_given_topic(self, word, topic, beta=0.1):
    """
    P(word|topic,Beta)
    The fraction of words assigned to topic
```

```

    that equal word (plus some smoothing)
    '''
    return ((self.topic_word_counts[topic][word] + beta) /
            (self.topic_counts[topic] + self.W * beta))

```

Les valeurs de `alpha` et `beta` sont liées à la définition des distributions de Dirichlet qui permettent à la fois de gérer les zéros et d'ajuster l'aplatissement de ces distributions de probabilités. Ici, les valeurs prises sont celles classiquement par défaut.

Le cœur de l'algorithme consiste en une série d'itérations, modifiant l'allocation d'un sujet à un mot précis du document, comme résultat du produit des deux distributions de probabilités sus-mentionnées.

Ici on effectue ce produit :

```

def topic_weight(self, d, word, topic):
    '''
    P(topic|word|d,Alpha,Beta) = P(topic|d,Alpha) * P(word|topic,Beta)
    Given a document and a word in that document,
    return the weight of the word for the k-th topic in that document
    '''
    return self.p_word_given_topic(word, topic) * self.p_topic_given_document(topic, d)

```

Puis, on choisit le meilleur sujet pour le mot `word` du document `d` :

```

def choose_new_topic(self, d, word):
    return self.sample_from_weights([self.topic_weight(d, word, k)
                                     for k in range(self.K)])

```

Enfin, on démarre les itérations et les réallocations des sujets pour chaque mot de chaque document :

```

def gibbs_sample(self, document_topics):
    '''
    Gibbs sampling https://en.wikipedia.org/wiki/Gibbs\_sampling.
    '''
    for _ in range(self.max_iteration):
        for d in range(self.D):
            for i, (word, topic) in enumerate(zip(documents[d],
                                                  document_topics[d])):
                # remove this word / topic from the counts
                # so that it doesn't influence the weights
                self.document_topic_counts[d][topic] -= 1
                self.topic_word_counts[topic][word] -= 1
                self.topic_counts[topic] -= 1
                self.document_lengths[d] -= 1

```

```

        # choose a new topic based on the weights
        new_topic = self.choose_new_topic(d, word)
        document_topics[d][i] = new_topic

        # and now add it back to the counts
        self.document_topic_counts[d][new_topic] += 1
        self.topic_word_counts[new_topic][word] += 1
        self.topic_counts[new_topic] += 1
        self.document_lengths[d] += 1

def run(self, documents):
    # How many times each topic is assigned to each document.
    self.document_topic_counts = [Counter() for _ in documents]
    # How many times each word is assigned to each topic.
    self.topic_word_counts = [Counter() for _ in range(self.K)]
    # The total number of words assigned to each topic.
    self.topic_counts = [0 for _ in range(self.K)]
    # The total number of words contained in each document.
    self.document_lengths = [len(d) for d in documents]
    self.distinct_words = set(word for document in documents
                               for word in document)

    # The number of distinct words
    self.W = len(self.distinct_words)
    # The number of documents
    self.D = len(documents)
    # document_topics is a Collection that assign a topic (number
    # between 0 and K-1) to each word in each document.
    # For example :
    # document_topics[3][4]->[4 document][id of topic assigned to 5 word]
    # This collection defines each document's distribution over topics,
    # and implicitly defines each topic's distribution over words.
    document_topics = [[random.randrange(self.K) for word in document]
                       for document in documents]

    for d in range(self.D):
        for word, topic in zip(documents[d], document_topics[d]):
            self.document_topic_counts[d][topic] += 1
            self.topic_word_counts[topic][word] += 1
            self.topic_counts[topic] += 1

    self.gibbs_sample(document_topics)

    return(self.topic_word_counts, self.document_topic_counts)

```

Une manière d'illustrer les résultats et de visualiser les sujets découverts (c'est loin d'être la seule) consiste à afficher autant de nuages de mots qu'il y a de sujets, montrant l'importance de chaque mot pour le sujet en question (figure 9-4).

Figure 9–4

Nuages de mots
des sujets
découverts



Représenter les documents par un vecteur de « sens »

En partant des mêmes phrases que précédemment, et surtout des vecteurs de mots qui les caractérisent, il est possible cette fois d'essayer d'en capturer un sens plus profond encore par l'entremise des bibliothèques *Word2Vec* (qui vectorise les mots) et *Doc2Vec* (qui vectorise les phrases elles-mêmes).

Ainsi dans le code, la vectorisation des mots s'obtient par :

```
model = Word2Vec(sentences, min_count=1)
```

`min_count` ignore les mots dont le nombre d'apparitions est inférieur à sa valeur.

On perçoit bien que l'on a vectorisé les mots, car il est possible de procéder aux opérations suivantes :

```
print (model.wv.similarity('dog', 'lazy'))
```

qui calcule la distance vectorielle entre ces deux mots, ou :

```
print (model.wv.most_similar(positive=['lazy'], negative=[], topn=2))
```

qui donne les deux mots les plus similaires à 'lazy'.

La vectorisation des huit phrases se fait de la manière suivante :

```
tagged_data = [TaggedDocument(d, [i])
                 for i, d in enumerate(sentences)]
model = Doc2Vec(tagged_data, vector_size = 20, window = 2,
                 min_count = 1, epochs = 100)
```

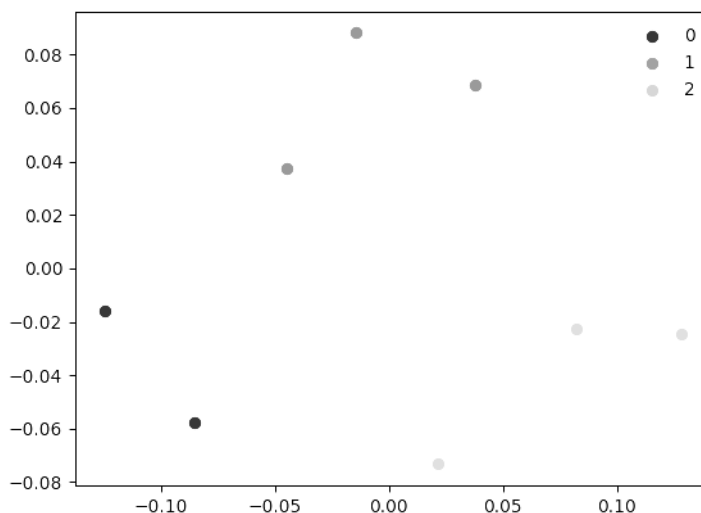
Il est possible ensuite d'exécuter un algorithme de regroupement (*clustering*) de vecteurs classiques (en prenant soin de réduire préalablement les vecteurs à deux dimensions pour visualiser le résultat). On utilise une analyse en composante principale (*PCA*), une méthodologie classique de base que nous ne détaillerons pas ici.

```
print ("Cluster id labels for inputted data")
print (labels)
```

En se limitant à trois catégories pour les huit phrases, on retrouve le même résultat qu'avec LDA : [1 1 2 2 2 0 1 2], chaque nombre indiquant à quel groupe appartient la phrase. Sur la figure 9-5, on voit les huit points représentant les phrases, relativement regroupées selon la signification profonde des mots qui les composent.

Figure 9-5

Représentation de phrases
par un vecteur de « sens »



10

Comment distinguer un chien d'un chat

Figure 10-1

Capture d'écran du logiciel Python mis à votre disposition pour la reconnaissance d'un chien ou d'un chat sur une photo.



Principe – Choix algorithmique

Comment distinguer un chien d'un chat sur une photographie que l'on nous présente devant les yeux ? La distinction est plus délicate encore quand l'animal sur la photo est flou et que, alors qu'il est évident qu'il s'agit bien d'un des deux animaux, plusieurs personnes pourraient se méprendre. En effet, il existe des bases de données de photos de ces deux animaux qui sont soumises à des humains pris au hasard et pour lesquelles, en moyenne, 15 % des photos sont mal identifiées car les sujets d'expérience confondent les deux animaux. Or, depuis quelques années, des réseaux de neurones dits à apprentissage profond sont capables de faire mieux que les humains en moyenne pour ce même ensemble de photos et, par exemple, de ne commettre que 5 % d'erreurs. Ces quinze dernières années, des progrès considérables ont été effectués pour ce type de reconnaissance d'images.

Aux origines de l'IA, la reconnaissance des sujets sur des photos ne semblait en rien un type de tâche cognitive digne d'un grand intérêt. Et pour cause, ce type de processus mental ne semble pas poser de grandes difficultés aux humains, même extrêmement jeunes, ni même à de nombreux animaux, alors que jouer aux échecs, résoudre un théorème mathématique ou résumer un texte... Est-ce à ce point intelligent d'être capable de faire la différence entre un chien et un chat sur une photo ? Parmi les légendes de l'IA, on raconte que Marvin Minsky, un des plus célèbres pionniers de cette discipline, aurait, dès les années 1960, confié comme mission à un de ses étudiants et dans le cadre d'un petit travail de fin d'étude de résoudre le problème de la vision par ordinateur. Près de soixante ans après, cette même vision est une des pierres d'achoppement de l'IA, en témoigne l'immense succès rencontré par les réseaux de neurones profonds qui ont permis d'accomplir des progrès considérables, notamment pour distinguer les chiens des chats dans un grand ensemble de photos.

Si faire mieux que les hommes en moyenne pour distinguer un chien d'un chat ne semble pas être à l'origine d'une révolution conceptuelle majeure, on peut comprendre qu'améliorer les performances humaines puisse être autrement plus sensible dans le cas de l'identification correcte de la malignité ou la gravité d'une tumeur sur une radio, de l'état des poumons d'une personne atteinte du coronavirus, de la bonne reconnaissance d'un tank ennemi dans une forêt ou d'un terroriste potentiel dans un aéroport. Dans des cas comme ceux-là, même 1 % de meilleure reconnaissance va faire toute la différence. Si le réseau de neurones ne fait même qu'1 % de mieux que le cancérologue pour diagnostiquer correctement la gravité d'une tumeur, on sait à qui l'on choisira de se fier. Il en va aussi de la possibilité de faire vieillir ou rajeunir une photo, facilitant parfois la recherche de personnes disparues.

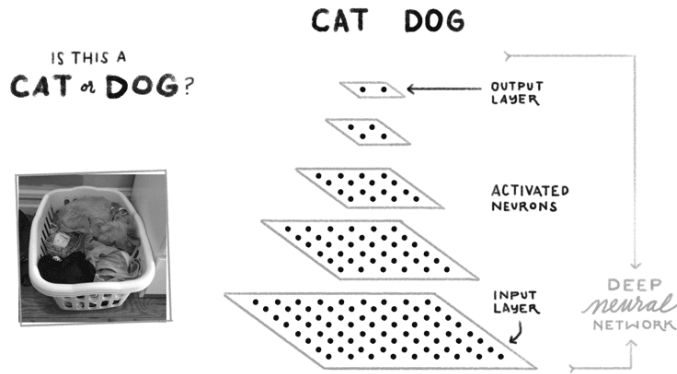
Reprenons la reconnaissance chien/chat qui nous occupe dans ce chapitre. Une possible manière de fonctionner pourrait être la suivante. On essaie d'identifier a priori ce qui fondamentalement constitue la signature visuelle d'un chien et celle d'un chat, peut-être les moustaches du deuxième ou la truffe du premier... Une fois ces éléments graphiques primitifs identifiés, c'est en les retrouvant dans les photos en question que l'on peut les classer comme chien ou chat. Or, c'est justement cette identification et élicitation a priori des primitives visuelles des éléments à reconnaître qui posait des difficultés considérables jusqu'à présent. Elle était l'apanage de quelques rares experts en reconnaissance et traitement des images. Or, c'est à cette même identification de ces primitives graphiques que l'apprentissage des réseaux de neurones profonds semble contribuer de manière

significative. Il est difficile ainsi aujourd'hui de parler d'intelligence artificielle sans faire allusion à ce type d'algorithme occupant depuis quelques années le devant de la scène, l'apprentissage profond, à tel point que l'IA lui est de plus en plus souvent assimilée. Or, comme tout ce qui touche à l'IA depuis de nombreuses années, rien n'est pourtant vraiment neuf et le vieux chercheur qu'Hugues Bersini est devenu a de quoi s'étonner, parfois s'énervé, de cet engouement cyclique (à raison de plus ou moins tous les cinq ans) pour une branche scientifique et technologique ayant pris son envol il y a plus de soixante ans.

Figure 10-2

Manière schématisée dont un réseau de neurones profond arrive à distinguer les chiens des chats sur une photo.

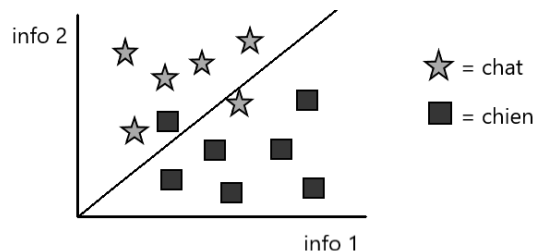
https://about.google/intl/ALL_fr/stories/machine-learning-qa/



L'apprentissage profond est une simple reprise de nos ancestraux réseaux de neurones, proposés pour la première fois, faut-il le rappeler, peu après la fin de la Seconde Guerre mondiale. C'est une technique qui marche plutôt bien pour l'automatisation des processus sensori-moteurs, comme la reconnaissance d'images, la compréhension de scènes, la perception du langage parlé comme écrit ou la conduite automobile. On pourrait même dire que cet apprentissage profond contribue plus qu'il n'en faut à renforcer la nature inconsciente de l'IA dont nous parlerons dans la conclusion. Ces réseaux de neurones sont de très grosses boîtes noires multiparamétrées (on peut aller jusqu'à des millions de paramètres) dont l'apprentissage au départ d'une grande base de données consiste justement en l'optimisation de ces paramètres. In fine, de nombreux systèmes de classification en IA consistent à séparer les classes par une séparatrice géométrique dont la complexité dépend du nombre des paramètres utilisés (figure 10-3).

Figure 10-3

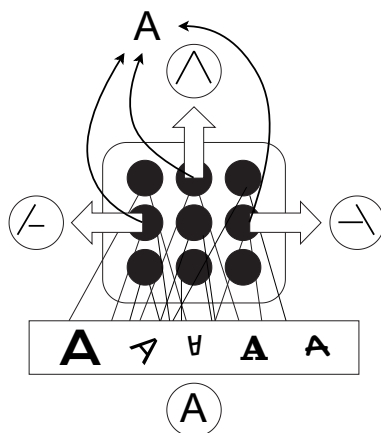
Illustration très simplifiée du problème de classification « chien/chat ».



Dans le cas des réseaux de neurones profonds, ces mêmes paramètres constituent une succession, qui peut être conséquente, de couches qui, en partie, s'inspire de la succession des couches neuronales caractérisant le système visuel humain. Ces couches sont nécessaires chez l'humain comme chez la machine pour extraire les caractéristiques principales des images qui, une fois regroupées, permettent en effet une reconnaissance robuste. Par exemple, imaginons un réseau de neurones en charge de la reconnaissance des lettres majuscules : A, B, ..., Z. Nous, humains, sommes capables de reconnaître facilement ces lettres, quelles que soient leur taille ou orientation et même si elles sont très légèrement effacées. Cette robustesse est due à l'extraction préalable, justement, des caractéristiques essentielles propres à chacun de ces caractères. Le « A » est à la rencontre de trois barres (deux obliques, une horizontale) et trois angles, le « B » d'une barre et de deux demi-cercles, le « Z » de trois barres (deux horizontales, une oblique) et deux angles opposés... Il faut donc d'abord repérer les barres et les ronds avant de reconnaître les lettres. Ces caractéristiques sont essentielles à la catégorisation d'une multitude de « A », de taille, orientation ou dégradation différente, en tant qu'un seul et même « A ».

Figure 10-4

Comment la lettre « A » se reconnaît à la jonction de certaines primitives géométriques.



Pendant de nombreuses années, des experts humains en traitement d'images ou linguistique se sont chargés de faciliter la tâche des algorithmes de reconnaissance en extrayant préalablement ces caractéristiques (les barres, les angles et les ronds). Puis, ils laissaient le réseau de neurones apprendre de lui-même, à partir de multiples exemples de lettres qu'on lui présentait, les meilleures combinaisons de ces caractéristiques facilitant la reconnaissance. Un premier algorithme, codé par les soins d'un ingénieur, extrayait donc quelques caractéristiques géométriques clés présentes dans l'image, telles les traits horizontaux, les limites entre des zones de couleurs différentes ou les angles. Ensuite, le réseau de neurones apprenait par lui-même quelles combinaisons de traits et d'angles coder pour le « A » ou pour le « B ».

Depuis quelques années, vus la taille énorme des bases d'exemples qui sont à leur disposition (des milliards de différents A, B, ..., Z, ainsi que des milliards de photos de chiens et de chats) et l'accroissement extraordinaire (y compris grâce au parallélisme des processeurs) des performances technologiques informatiques (espace mémoire et vitesse de traitement), les ingénieurs ont décidé,

une fois encore, d'écarter ce soi-disant compagnonnage humain du processus de reconnaissance automatisé. Ils ont préféré laisser le réseau de neurones et ses centaines de milliers de paramètres s'occuper lui-même d'extraire ces caractéristiques. Ce faisant, les performances s'en sont trouvées considérablement accrues, dépassant même les humains dans un très grand nombre d'expériences de reconnaissance d'images. Nouveau seuil franchi, nouvelle humiliation du règne humain, nouvelles premières pages de quotidiens ! En substance, là aussi, et comme au jeu de go, les réseaux de neurones ont, de leur plein « gré » mais inconsciemment, découvert de nouvelles voies de résolution qui échappaient aux experts humains. Ces mêmes malheureux experts croyaient pourtant bien faire en leur mâchant la besogne par une extraction manuelle des caractéristiques leur apparaissant discriminantes. Malheureusement, par ces étapes préalables, ils envoyaient les réseaux de neurones sur des voies de traverse, passant à côté de voies résolutives bien plus prometteuses, lorsque ces derniers trouvent d'eux-mêmes ces caractéristiques robustes au pouvoir discriminant. Maintenant que l'on dispose de la puissance de calcul pour cela, autant leur laisser tout faire.

Notez qu'il ne peut y avoir d'apprentissage de réseaux de neurones sans optimisation. Mais qu'optimise-t-on dans ce cas ? Ce sont les paramètres constitutifs de ces réseaux qu'il faut optimiser par une méthode mathématique dont le but ultime est la minimisation des erreurs de classification (classification des lettres ou des chiens et des chats) en partant d'une base de données dite d'apprentissage. L'espoir est que, à partir de cette base de données, le réseau apprendra de lui-même à généraliser cette capacité discriminante pour toutes les lettres et tous les chiens et chats à venir, aussi différents soient-ils de ceux présents dans la base de données d'apprentissage. C'est cette faculté inductive que l'on tente d'optimiser dans ces réseaux de neurones, cette aptitude à distinguer toutes les photos de chiens et de chats à venir, quelle que soit la race, la taille, la luminosité ou l'état de dégradation de l'image.

Dans le cas précis de reconnaissance de photos, l'innovation majeure qui a été apportée sur les réseaux de neurones à l'ancienne est l'adjonction de couches dite de convolution et de *pooling* (figure 10–5).

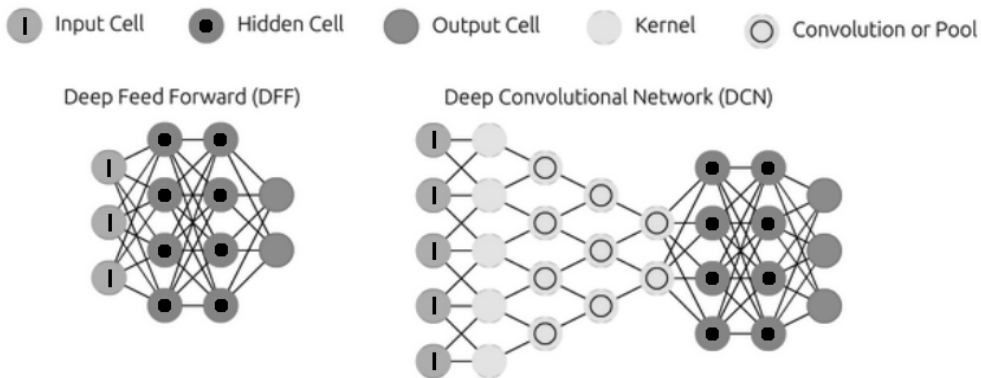


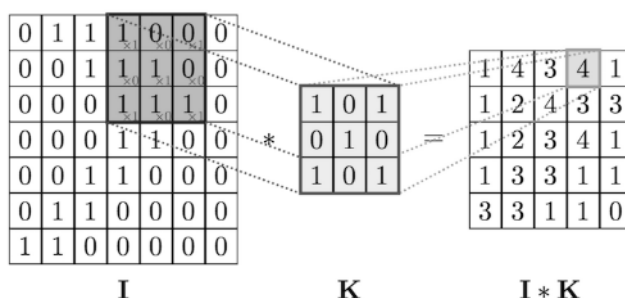
Figure 10–5 À gauche, les anciens types d'architecture neuronale ; à droite, les nouveaux types.

© <https://www.asimovinstitute.org/> Fjodor van Veen & Stefan Leijnen

L'opération de convolution est illustrée par la figure 10-6.

Figure 10-6

Le filtre convolutif en gris traité et simplifié l'image à gauche.
<https://ichi.pro/fr/comment-les-reseaux-de-neurones-a-convolution-interpretent-les-images-34744426328242>

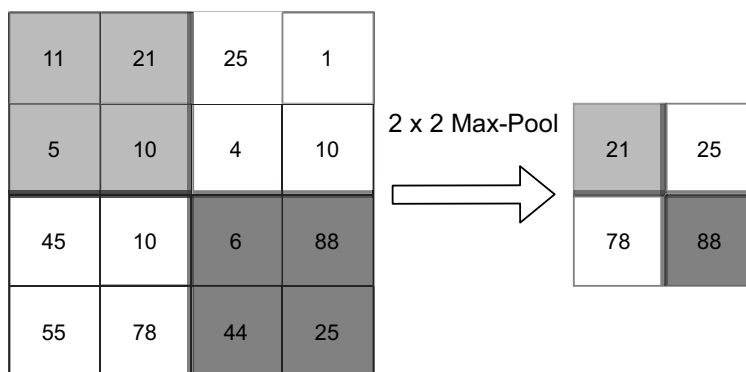


C'est une opération mathématique qui permet d'extraire les caractéristiques locales d'une image et qui réduit ainsi la dimension de l'entrée. Elle utilise un noyau agissant comme un filtre qui va glisser sur l'image. Ce filtre pourra lui aussi faire l'objet d'un apprentissage, tout comme n'importe quel paramètre du réseau, par un algorithme de type rétropropagation. Dans la figure, le noyau est composé des chiffres en gris clair. On prend alors la somme (ou une autre fonction comme la moyenne) qui donne la sortie de la convolution, appelée la carte des caractéristiques.

Cette carte des caractéristiques obtenue après une opération de convolution peut encore être trop massive. Il faut alors agréger davantage encore l'information par une fonction dite de pooling, qui calcule les statistiques de ces cartes caractéristiques à différents endroits. Par exemple, en opérant un *max-pooling*, on prend la valeur maximale d'une partie de la carte des caractéristiques (figure 10-7).

Figure 10-7

Opération de max-pooling.



Outre la réduction de la dimension, cette opération de pooling réduit également le risque de sur-apprentissage. Quand on choisit l'opération de maximisation, on produit gratuitement aussi de l'invariance (en translation, rotation ou agrandissement) par rapport à l'image en entrée. On donne donc beaucoup de robustesse aux réseaux, par rapport à toutes les modifications que peuvent subir nos images de chiens et de chats.

Le projet Python associé à ce chapitre se base, comme pour le chapitre précédent, sur un jeu de données Kaggle composé de 10 000 images, 5 000 chiens et 5 000 chats. Une excellente démarche lorsqu'on pratique l'apprentissage à partir d'un ensemble de données est de diviser ces données d'apprentissage en trois parties : les données d'entraînement, les données de validation et les données de test. Les données d'entraînement servent à faire fonctionner l'optimisation des poids du réseau avec l'algorithme d'optimisation choisi (par exemple, la rétropropagation). Cependant, il reste de nombreuses inconnues structurelles pour ce réseau : combien de couches, de neurones par couche, quels paramètres pour l'algorithme d'optimisation ? On dit souvent que ces choix relèvent plus de l'art, pour ne pas dire du bricolage, que de la science. Toutefois, il y a moyen de se faire un peu aider et c'est le rôle du deuxième ensemble de données.

Il devient possible d'optimiser ces choix par le recours aux données de validation. On teste par exemple, pour chaque architecture de réseau, celle qui donne le meilleur score à l'apprentissage à l'aide de ces mêmes données. On peut aussi les utiliser afin d'éviter le surapprentissage, qui survient lorsque l'on colle trop aux données et qu'on dégrade la capacité du réseau à induire la meilleure généralisation possible. On arrête alors l'algorithme d'apprentissage quand les performances se dégradent, non pas sur les données d'apprentissage mais bien sur l'ensemble de validation. Au fur et à mesure de l'apprentissage, on teste les performances du réseau sur l'ensemble de validation. Finalement, le dernier ensemble de données, les données de test que l'on n'a jamais utilisées durant toute cette phase d'optimisation du réseau, est utilisé juste à la fin, pour le réseau final, afin d'obtenir la performance la plus objective qui soit de ce réseau, vu qu'il n'a jamais rencontré ces données pendant toute l'opération complexe de son optimisation tant structurelle que paramétrique. Dans ce projet, 6 000 images ont été utilisées pour l'apprentissage, 1 500 pour la validation et les 2 500 restantes pour le test.

Au vu de la taille massive et du nombre considérable de paramètres à ajuster dans les réseaux de neurones profonds, une autre pratique qui accompagne très souvent l'apprentissage est d'enrichir l'ensemble des données en en créant de nouvelles à partir de celles existantes. On concevra intuitivement que la taille de l'ensemble d'apprentissage doit augmenter en « proportion » du nombre de paramètres à apprendre du réseau. On obtient ces nouvelles images par les opérations suivantes : retourner l'image horizontalement, déplacer l'image, faire un zoom avant, appliquer une rotation et/ou un cisaillement de l'image. Ces techniques d'augmentation sont opérées à partir des trois ensembles de données et des bibliothèques existent pour réaliser ces manipulations. Dans le projet Python, la construction du réseau de neurones convolutifs a été réalisée empiriquement afin de maximiser la précision sur l'ensemble de validation. Il contient cinq couches convolutives de noyaux 3×3 , avec une quantité décroissante de filtres (128-128-64-64-32). La fonction d'activation ReLU a été utilisée après chaque couche et l'opération de max-pooling a été appliquée également après chaque couche de convolution avec une fenêtre de 2×2 pixels. Le nombre de cinq couches s'est en effet avéré donner le meilleur résultat sur l'ensemble de validation.

L'image obtenue en sortie de ces cinq convolutions a ensuite été aplatie à une dimension et passée à travers une couche entièrement connectée de 128 neurones avec une fonction d'activation ReLU. Afin d'éviter le surapprentissage, la sortie de la couche entièrement connectée a été soumise à une opération dite de *dropout*, qui consiste, durant l'apprentissage, à mettre aléatoirement 20 % des neurones d'entrée à 0. Vu leur gigantisme, l'apprentissage des réseaux de neurones s'est aussi enrichi d'un nombre considérable de trucs et ficelles afin de coller au mieux au problème, tout en

évitant le surapprentissage. Cette opération de dropout y participe. Enfin, le classement final « chat-chien » a été obtenu en passant la sortie dans une couche d'un seul neurone, avec une fonction d'activation sigmoïde. La fonction de coût que nous avons utilisée et que le gradient a minimisé est l'entropie croisée binaire (une alternative au moindre carré assez classique lorsqu'il s'agit d'un problème de classification). Les poids ont été mis à jour à l'aide de l'algorithme RMSprop (propagation de l'erreur moyenne quadratique), une autre adaptation de la méthode de gradient de base, comme la méthode Adam, basée sur l'adaptation du taux d'apprentissage. L'apprentissage a consisté en 100 présentations de 64 images. Pour les étudiants et les chercheurs aujourd'hui, comme dans ce projet effectivement, l'utilisation de toutes ces techniques ne demande aucune véritable programmation, mais juste de faire les bons choix et de combiner ensemble les résultats de ces choix. Cela consiste pour l'essentiel en l'exploitation de bibliothèques telles que Keras, Pytorch ou Tensorflow, que les géants du Net et de l'informatique ont mis à votre disposition.

Le résultat de ce projet est un logiciel que vous pouvez utiliser pour la reconnaissance de nouvelles photos (figure 10-8). Il suffit de lui présenter la photo de votre animal de compagnie et de vous réjouir du résultat. Le réseau de neurones profond obtenu donne une précision de 93 % sur l'ensemble test. Aujourd'hui, il y a moyen de faire encore mieux et d'atteindre des performances de l'ordre de 98 %, en faisant notamment usage de réseaux de neurones ayant déjà appris d'autres tâches de reconnaissance d'images (on parle alors de *transfer learning*) qu'on spécialise à cette nouvelle tâche.

Figure 10-8

Présentez une photo au logiciel et il reconnaîtra soit un chien soit un chat.



Le logiciel Python de la reconnaissance chien/chat en pratique

L'application Chien ou Chat se présente sous la forme suivante :

- un fichier `main.py` permettant de lancer l'interface graphique, de tester un réseau de neurones ou d'essayer une opération de transformation sur une image ;
- un fichier `Conv_operation.py` qui contient les primitives des transformations ;
- un fichier `train.py` pour entraîner automatiquement un réseau de neurones sur les images d'entraînement ;
- un dossier `data` contenant les images d'entraînement et de test utilisées par l'algorithme. Il contient 5 000 images de chats et 5 000 images de chiens.

Apprentissage profond

Ce projet fait un usage conséquent de la bibliothèque `tensorflow/keras` afin d'abstraire l'entraînement ainsi que l'utilisation du réseau de neurones.

```
class DogCatClassifier:
    """
    Image classifier for dog and cat pictures using Deep Learning
    Convolutionnal Neural Network
    """

    IMG_HEIGHT = 256
    IMG_WIDTH = 256
    BATCH_SIZE = 64

    def __init__(self, data_dir="data", epochs=100):
        """
        :param data_dir: directory of the data
        :param epochs: number of epochs for the training
        """
        self.epochs = epochs
        self.data_dir = data_dir

        # Load data and labels
        self.X = sorted(os.listdir(self.data_dir)) # Files names of the images
        self.y = np.empty(len(self.X), dtype=str) # Labels
        self.y[np.char.startswith(self.X, "cat")] = "c"
        self.y[np.char.startswith(self.X, "dog")] = "d"

        self.model = DogCatClassifier._load_model()
```

La classe `DogCatClassifier` dans `train.py` est la classe principale qui va entraîner le réseau de neurones. On y définit simplement les différents attributs nécessaires à son bon fonctionnement.

La méthode `_load_model` sert à initialiser le réseau de neurones :

```
@classmethod
def _load_model(cls):
    """
    Build a CNN model for image classification
    """
    model = Sequential()

    # 2D Convolutional layer
    model.add(
        Conv2D(
            128, # Number of filters
            (3, 3), # Padding size
            input_shape=(
                cls.IMG_HEIGHT,
                cls.IMG_WIDTH,
                3,
            ), # Shape of the input images
            activation="relu", # Output function of the neurons
            padding="same",
        )
    ) # Behaviour of the padding region near the borders
    # 2D Pooling layer to reduce image shape
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(128, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(64, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(64, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(32, (3, 3), activation="relu", padding="same"))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Transform 2D input shape into 1D shape
    model.add(Flatten())
    # Dense layer of fully connected neurons
    model.add(Dense(128, activation="relu"))
    # Dropout layer to reduce overfitting, the argument is the proportion of
    # random neurons ignored in the training
    model.add(Dropout(0.2))
    # Output layer
    model.add(Dense(1, activation="sigmoid"))

    model.compile(
        loss="binary_crossentropy", # Loss function for binary classification
```

```

optimizer=RMSprop(
    lr=1e-3
), # Optimizer function to update weights during the training
metrics=["accuracy", "AUC"],
) # Metrics to monitor during training and testing

# Print model summary
model.summary()

return model

```

On y retrouve toute l'architecture du modèle choisi pour le réseau de neurones, tel que résumé ci-après : cinq couches de convolution, de taille décroissante, la fonction d'activation ReLU et un filtre de max-pooling appliqué après chaque convolution. Le réseau se termine par une couche entièrement connectée (dense), une couche de dropout (technique de régularisation pour minimiser le surapprentissage du réseau) et le neurone de sortie donnant la prédiction, entre 0 et 1 (proche de 0 pour un chat, proche de 1 pour un chien).

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 128)	3584
max_pooling2d (MaxPooling2D)	(None, 128, 128, 128)	0
conv2d_1 (Conv2D)	(None, 128, 128, 128)	147584
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 128)	0
conv2d_2 (Conv2D)	(None, 64, 64, 64)	73792
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 64)	0
conv2d_3 (Conv2D)	(None, 32, 32, 64)	36928
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 64)	0
conv2d_4 (Conv2D)	(None, 16, 16, 32)	18464
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 32)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129

Le réseau totalise ainsi plus de 500 000 paramètres. Il est difficile d'entraîner un tel réseau grâce à la seule puissance de calcul d'un simple processeur. Il est donc nécessaire d'utiliser un bon GPU (une bonne carte graphique) pour entraîner ce réseau de neurones dans un temps raisonnable. Cette substitution de processeur (CPU->GPU) est devenue très courante pour l'utilisation des réseaux de neurones profonds vu le parallélisme de traitement inhérent aux processeurs GPU.

Une fois entraîné, le réseau de neurones est enregistré dans un dossier (dont le dossier `model_Custom` est un exemple) d'où il est facile de le charger et de l'utiliser.

```
def predict(self):
    if len(self.imgPath) > 0 and self.cnn is not None:
        img = transform.resize(
            io.imread(self.imgPath[self.imgIndex]), (256, 256), anti_aliasing=True
        )
        try:
            img = img.reshape(1, 256, 256, 3)

            self.predictions[self.imgIndex] = self.cnn.predict(img, 1)[0][0]
            self.updatePixmap(
                self.imgPath[self.imgIndex], self.predictions[self.imgIndex]
            )
        except:
            QMessageBox(
                QMessageBox.Warning,
                "Error",
                "Cannot convert image, please select a valid image",
            ).exec_()
        else:
            QMessageBox(
                QMessageBox.Warning,
                "Error",
                "Please select an image and a neural network model before reduction",
            ).exec_()
```

La méthode `predict` du fichier `main.py` qui gère l'interface de test du réseau illustre cela. L'utilisation du réseau de neurones se fait à la ligne 9, par le seul appel à la méthode `predict` du modèle préalablement chargé dans l'attribut `self.cnn` de la classe.

Conclusion : les deux IA

Nous sommes en 1950, maître Alan Turing sur son génie perché n'a pas encore subi l'horrible « traitement de faveur » que l'on réservait aux homosexuels à cette époque. Il a trente-huit ans, sa créativité est au zénith et encore inimaginable est le morceau de pomme empoisonnée qui mettra fin à ses jours. Autour de lui, et grâce à lui, de nombreux chercheurs prennent conscience qu'un ordinateur, dénommé par le maître « machine universelle », pourrait, par son câblage et sa programmation, produire le type de processus cognitif dont jusqu'à présent les humains se réservaient la primeur. À l'instar d'une forêt neuronale inextricable d'où jaillissent souvenirs, prévisions, inférences et raisonnements, les mêmes souvenirs, prévisions, inférences et raisonnements pourraient jaillir d'une forêt inextricable, mais cette fois de transistors en silicium et de portes logiques. Ce cerveau-machine, Turing l'ambitionne aussi infiniment programmable que l'est notre cerveau, dont les mêmes neurones se voient sollicités pour des tâches cognitives d'une variété infinie. Entre deux marathons dont les photos d'époque témoignent, il perce le mystère de ces niveaux empilés d'abstraction qui font de l'informatique aujourd'hui la technologie la plus extraordinaire, la plus envahissante et la plus déconcertante jamais produite par l'humain : machine universelle, machine à tout faire, universellement programmable et universellement intelligente.

Quand on l'interroge sur sa définition de l'intelligence, dès lors que c'est d'ordinateur qu'il s'agit (il faudra attendre la conférence de Dartmouth en 1956 pour que le terme « intelligence artificielle » prenne vraiment son envol), il se trouve à ce point dépourvu qu'il s'en tire par une pirouette : plutôt qu'une définition, c'est un test qu'il propose, que dit-on un test, une performance. Bien évidemment, et il en est conscient déjà, c'est la notion d'intelligence qui fait débat. D'ailleurs, aujourd'hui, on ne compte plus les nombreux types d'intelligence qui se disputent le sacre : rationnelle, émotionnelle, sociale, animale, en essaim.... Sans doute, une définition audacieuse et quelque peu immodeste aurait pu être : « moi ». Néanmoins, tout génie qu'il est, Turing entrevoyait déjà l'horizon très lointain et sans doute inatteignable de telles fulgurances pour sa créature. Il choisit alors de se cantonner à une tâche ô combien banale mais à la fois si emblématique de notre humanité : le dialogue. Son test prend la forme suivante : « si, lors d'un dialogue par écran interposé, un ordinateur est capable de vous confondre sur sa nature première, c'est-à-dire de se faire passer pour un interlocuteur humain, l'ordinateur ou l'algorithme en question que ce dernier exécute pourra être qualifié d'intelligent ».

Remarquez combien notre homme fut prévoyant, car il évacua toute la problématique de l'apparence humaine, de celle qui fait tant défaut à nos machines. Il suffit de voir aujourd'hui un ordinateur/robot, aussi humanoïde qu'il tente de l'être, pour ne pas douter un seul instant de sa nature véritable. Turing choisit d'épargner les chercheurs du futur et de faciliter la vie à nos machines. C'est bien de manifestations cognitives et seulement de cela qu'il s'agit ici, les deux interlocuteurs, le cobaye et l'ordinateur, devisant uniquement par écran interposé (on parlerait aujourd'hui, et dans un mauvais français, de *chat* ou de *chatbot*). Ces jours-ci, ce sont les productions prétendument artistiques de l'IA qui nous forcent à nous poser à nouveau le même type de questionnement que Turing. À la simple vision des dessins, à la lecture des vers, à la simple écoute des mélodies, trouve-t-on à l'origine de ceux-ci des artistes authentiques ou des disciples logiciels, rattrapant du terrain, jusqu'à se confondre avec les maîtres humains ?

En effet, en proposant un tel test, Alan Turing, sciemment ou pas, nous confronta dès la genèse de l'informatique à une ambiguïté algorithmique fondamentale. S'il nous proposa une performance en effet, la capacité au dialogue d'une machine à ce point indistinguable de celle d'un être humain banal, il passa complètement sous silence la manière d'y parvenir. Or, il existe de fait plusieurs manières d'y arriver. En voici une première, stupide on vous l'accorde, difficilement imaginable à l'époque de Turing, mais qui devrait pourtant vous sensibiliser à tout ce qui va suivre. Elle est très en phase avec l'IA d'aujourd'hui, mais pas celle de l'époque de Turing évidemment. Imaginons que toutes vos conversations, sur WhatsApp, Messenger, SMS et autres réseaux sociaux, les vôtres et celles de l'humanité tout entière, aient été sauvegardées sur un même serveur informatique. Imaginons ensuite que le programme sommé de réussir le test de Turing procède de la façon suivante : à chaque question reçue, il fouille dans cette gigantesque base de données à la recherche de la réponse qui lui fut le plus souvent donnée. Vu le gigantisme de cette base, on peut tout à fait supposer que chaque extrait du dialogue s'y retrouvera et même de nombreuses fois. Ainsi, « comment ça va ? » sera majoritairement suivi d'un « pas mal et toi », « cela te dit un cinéma ce soir ? » pourrait l'être d'un « chouette idée, quel film ? ». On simplifie à l'extrême (la supercherie pourrait être facilement déjouée en posant « quel jour sommes-nous ? » ou « quel temps fait-il ? »), mais vous devriez saisir l'idée. Qualifieriez-vous vraiment un tel programme « d'intelligent », même si pourtant il parle plutôt bien, jusqu'au point de confondre son interlocuteur ?

Ainsi, il ne fallut pas attendre très longtemps, quinze ans seulement, pour qu'un programme réussisse le test de Turing, mais qu'il le réussisse dans des circonstances quelque peu singulières : le cabinet d'un psychologue. Voici un extrait de dialogue entre le logiciel dénommé Eliza, s'improvisant psychothérapeute, et une patiente.

La patiente : « C'est mon ami qui m'a dit de venir vous parler. »

Eliza : « Votre ami vous a dit de venir me parler ? »

La patiente : « Il dit que je suis souvent déprimée. »

Eliza : « Je suis vraiment navré d'apprendre que vous êtes souvent déprimée. »

La patiente : « C'est vrai. Je suis plutôt malheureuse. »

Eliza : « Pensez-vous que me parler vous aidera à ne plus être malheureuse ? »

La ficelle informatique avait beau être très grosse (le logiciel de quelques centaines de lignes, pas plus, se bornant à associer des bouts de phrase stéréotypés à des réponses construites à partir de ces

bouts de phrase, mais tout aussi stéréotypées), les patients furent pour la plupart convaincus d'avoir affaire à un thérapeute véritable. De vilaines langues pourraient argumenter que cela en dit long sur la profession de psychologue, mais, surtout, cela perturba considérablement Joseph Weizenbaum, le concepteur de ce logiciel idiot, au point qu'il décida par la suite de consacrer l'essentiel de ses réflexions à l'impact sociétal des technologies du numérique et à la crédulité des humains. C'est aussi la preuve qu'un logiciel d'une grande stupidité, comme de nombreuses productions de l'IA aujourd'hui et surtout de l'apprentissage profond, perroquet obèse, obsessionnel du plagiat, peuvent nous bluffer sur leur soi-disant intelligence ou créativité.

Un autre philosophe américain aussi imaginaire qu'incisif, John Searle, dès les années 1980 et alors que de premiers logiciels se montraient capables de traduire très approximativement, de résumer des textes simples et de répondre à quelques questions portant sur ces textes, se révéla très sceptique quant à la définition de l'intelligence artificielle proposée par Alan Turing. Il n'écrivit pas un logiciel, mais présenta une expérience de pensée qui, à l'époque, suscita un formidable émoi dans la communauté des informaticiens et provoqua des milliers de réactions et de critiques. La voici très succinctement relatée. Elle reste toujours d'actualité. Supposez que, lors d'un voyage en Chine, vous soyez emprisonné (rien de trop surprenant jusqu'à présent). Vous ne comprenez bien évidemment rien à la langue chinoise, ni parlée, ni écrite. Votre voisin de cellule, isolé depuis très longtemps, se voit soudain comblé d'aise par votre arrivée et se propose d'échanger avec vous par messages interposés qu'il vous glisse sous la porte. On rappelle que vous ne saisissez rien de l'écriture chinoise. Cependant, ô astuce suprême, secours radical, délivrance, imagination philosophique débridée, vous avez à votre disposition un manuel gigantesque, associant sur sa page de gauche des expressions chinoises avec d'autres à relever sur sa page de droite. Vous en déduisez qu'il vous suffit de feuilleter les pages de ce livre colossal (c'est une expérience de pensée, ne l'oublions pas) afin de retrouver sur la page de gauche le contenu du message transmis par votre voisin et de reproduire aussi adroitement que possible ce que vous retrouvez sur la page de droite. Ensuite, après un laborieux griffonnage, c'est à votre tour de glisser la réponse sous la porte.

Pour votre voisin de cellule, à qui ce compagnonnage apporte soudain quelque réconfort, il ne fait pas l'ombre d'un doute que vous parlez le chinois. Mais vous ? Si, comme le questionnait John Searle avec quelque perfidie, vous affirmez comprendre le chinois, quand bien même vous vous limitez à tourner les pages d'un immense livre et à associer des symboles abstraits avec d'autres symboles abstraits, alors, en effet, les logiciels aujourd'hui, capables de mener un dialogue, traduire ou résumer un texte, répondre à des questions à son sujet, peindre des tableaux et rédiger des poèmes, ces logiciels comprennent ce dont ils parlent, saisissent ce dont ils traitent. Car, dicit John Searle, les algorithmes d'hier et d'aujourd'hui, qui manipulent langages et textes, ne font pas plus pas moins que tourner les pages d'un immense manuel à la recherche de symboles qu'ils reconnaissent, et associer ceux qu'ils reçoivent en entrée à d'autres qu'ils produisent en sortie. Pour le philosophe, cela n'a rien à voir avec le sentiment de compréhension, comme nous le vivons, comme nous le ressentons. À ses yeux, les manipulations syntaxiques ne conduisent pas directement au sens. L'univers sémantique reste partiellement délaissé, inaccessible. Une compréhension hors de portée et qui trouve, selon lui, son origine dans l'expérience et le vécu des choses parlées, puise sa source dans notre incarnation et notre être au monde, tout simplement.

Tant Joseph Weizenbaum que John Searle réussirent à démontrer, chacun à sa manière, que des algorithmes idiots, naïfs, efficaces mais sans finesse, bénéficiant de leur seule force brute, peuvent

produire des comportements d'une grande sophistication. En substance, des algorithmes stupides peuvent apparemment se comporter intelligemment, prouvant que l'intelligence artificielle n'est pas toujours là où on la croit. Ainsi, pour de nombreux types de tâches cognitives que les logiciels aujourd'hui réussissent très décemment et même avec tous les honneurs (comme les jeux de société, la conduite automobile et la traduction), il existe en effet deux manières très différentes d'accomplir ces tâches. La première que nous taxerons « d'inconsciente », c'est-à-dire sans réelle explicitation de l'expertise humaine, sans règle, sans nul besoin de décomposer la tâche pour la simplifier et la soumettre au jugement humain, offre une très large place à l'apprentissage. Cet apprentissage s'effectue soit au départ de myriades de réalisations humaines passées, soit par simples essais/erreurs, conduisant graduellement à des logiciels qui se complexifient et s'opacifient d'autant. On qualifiera la deuxième de « consciente ». Cette fois, l'expertise humaine reste toujours de mise, essentielle, la tâche se trouvant décomposée et découpée en séquences pour y glisser justement cette expertise. Cette IA-là est à l'origine d'algorithmes dont la transparence et la patte humaine permettent encore et dans une large partie d'en comprendre le fonctionnement. C'est la bonne vieille IA d'alors, qui ne pouvait se passer complètement de nous ; mieux encore, qui gagnait en rapidité et en performance lorsqu'elle suivait le guide.

Malgré le succès croissant et l'engouement de plus en plus marqué au vu des performances décoiffantes de l'IA inconsciente, il n'en reste pas moins que cette tension entre les deux IA est vieille comme l'intelligence artificielle elle-même. Dès la conférence de Dartmouth en 1956, son acte de naissance, on trouvait, à côté des premiers réseaux neuronaux et des logiciels apprenant par eux-mêmes à jouer au jeu de dames, des logiciels capables de raisonner (l'un d'entre eux s'appelait GPS pour *General Problem Solver*), de planifier, de résoudre des problèmes mathématiques et de diagnostiquer des situations complexes par la force de la raison humaine et la transposition logicielle de celle-ci. On connaît l'histoire, ce sont les approches les plus cognitives qui l'emportèrent à l'époque et donnèrent véritablement naissance à l'IA et à ses premiers succès pratiques.

Le premier livre d'intelligence artificielle qu'il fut donné de lire à Hugues Bersini il y a plus de quarante ans, et un des premiers livres disponibles en français traitant du sujet, conviait Heidegger, Merleau-Ponty et Wittgenstein pour expliquer et cerner les limitations de l'intelligence artificielle de type « consciente », son incapacité à simplement s'interfacer au monde, faire corps avec le monde et ce, de par sa désincarnation et sa désolidarisation du substrat biologique¹. D'où, de fait, la contre-attaque continue des chercheurs de l'autre IA, voyant dans les réseaux de neurones (et leur version *relookée* ces temps-ci rebaptisée « apprentissage profond »), ainsi que dans la prise en charge informatisée des processus sensori-moteurs, dans leur apprentissage par imitation ou par renforcement, la meilleure manière d'affronter ces critiques. Il n'y a sans doute dans cette distinction rien de très surprenant, ces deux « intelligence artificielle » se bornant à simplement reproduire la distinction bien connue des psychologues cognitifs entre les processus essentiellement conscients, lents, séquentiels, assez laborieux, et ceux inconscients, automatisés, parallèles, quasi-réflexes et pour lesquels nous excellons. Voilà bien une distinction archiconnue et détaillée dans tous les manuels de première année de psychologie.

1. Il s'agit du livre du philosophe de Berkeley Hubert Dreyfus, décédé depuis, et intitulé *Intelligence artificielle : mythes et limites*, paru chez Flammarion en 1992.

Ces deux types de processus cognitifs ont été consacrés par le célèbre prix Nobel d'économie Daniel Kahneman et qualifiés de « système 1 » et « système 2 » de la pensée humaine². Dans les écrits des philosophes phénoménologues, tels Heidegger et Merleau-Ponty, une tentative est faite pour expliquer la transition entre ces deux systèmes, la sortie des processus inconscients et l'activation d'une réflexion plus lente, plus découpée, par l'occurrence de « ruptures » dans ces automatismes. Ces ruptures sont, par exemple, la présence d'alternatives soudaines, d'un blocage imprévu, la survenue d'une attente non satisfaite ou d'un événement totalement inattendu. En réaction à ces ruptures, il nous faut éveiller notre conscience, déclencher les processus délibératifs qu'elle accompagne afin de nous sortir de ce mauvais pas, contourner l'obstacle, aller voir ailleurs.

Actuellement, nous vivons cependant les succès spectaculaires essentiellement de l'IA inconsciente, celle qui s'ingénie à se passer de nous, celle qui met toute la force brute de nos ordinateurs à son service, celle qui se nourrit de la myriade de données que nous lui abandonnons dans les réseaux sociaux et autre carrefours informatiques incontournables sur Internet. Elle gagne chaque jour, chaque heure, chaque microseconde davantage en autonomie. Elle doit une large part de son succès à la puissance de nos ordinateurs, leur parallélisme croissant, tel le recours aux cartes graphiques plutôt qu'aux processeurs classiques, ou l'utilisation de *clusters* (et demain l'ordinateur quantique ?). À même d'éviter de plus en plus les ruptures évoquées précédemment, elle n'a plus jamais recours à l'introspection, à s'interroger sur son comportement, ses productions et à solliciter le cas échéant notre expertise, aussi modeste soit-elle. Nous vivons en ce moment un véritable paradoxe. Chez nous, la complexité et le caractère inhabituel des situations à affronter nous obligent à sortir de nos bien confortables recettes inconscientes, à nous servir péniblement de notre conscience à la recherche de nouvelles solutions (par la simplification ou la découpe du problème, la recherche de situations quelque peu analogues). En IA, ce même type de situation incite à contourner toute explicitation biaisante et peu satisfaisante, afin de s'en remettre au seul apprentissage, l'essai/erreur, et à la compilation de myriades d'expériences humaines ayant dû précédemment affronter ces problèmes.

Restons pourtant vigilants. Malgré ce que beaucoup aimeraient penser, fournir des données brutes à un algorithme pour lui faire apprendre une tâche n'a rien de vraiment objectif. Il est rassurant de penser que le processus d'apprentissage machine pourrait supprimer les biais que le programmeur met naturellement dans un programme qu'il fabriquera « à la main », de par les choix techniques et logiques qu'il doit opérer durant sa conception. En fait, il n'en est rien ; on a plus que souvent simplement déplacé le problème du biais du concepteur aux données d'apprentissage. Le réseau de neurones ainsi entraîné avec un jeu de données biaisé ne sera que l'image de ce dernier : un programme imparfait et biaisé, mais peut-être encore plus pernicieusement, sans qu'on l'ait forcément remarqué et avec la fausse impression du travail accompli.

Il ne faut guère s'étonner que les pionniers de l'IA consciente, comme Marvin Minsky ou Noam Chomsky, se soient émus de l'importance prise par l'IA inconsciente qui, selon eux, donne beaucoup trop d'importance à la qualité des résultats au détriment de la manière d'y parvenir. Or, pour ces pionniers, toute science exige une explicitation de la démarche, des étapes et des modèles mis en œuvre. Si les sciences exactes se détachent de l'ingénierie, c'est bien dans leur exigence de compré-

2. Et très largement décrit dans le best-seller de Daniel Kahneman : *Système 1 et système 2 : les deux vitesses de la pensée*, paru chez Flammarion en 2012.

hension. Elles se doivent de comprendre autant que de prédire. Si la prédiction s'effectue au détriment de la compréhension, il y a tout lieu de faire marche arrière, de revenir aux fondamentaux. Imaginons un physicien dont le seul apport serait de saturer le monde de caméras très précises et d'enregistrer toutes les évolutions de ce même monde. Afin de prédire l'évolution de ce dernier à partir de l'instant t , et ce à partir de ce qu'il fut aux instants précédents, il suffirait de fournir à un réseau de neurones profond toutes les images du monde d'avant et de lui apprendre à prédire le monde d'après. Il pourrait exceller à cette prédiction, mais la physique, elle, que gagnerait-elle à cette nouvelle manière d'exercer sa discipline ? Prédiction parfaite, d'accord, mais compréhension absente !

Pour ces pionniers, un logiciel qui traduit en se nourrissant d'une quantité gigantesque de traductions précédentes, quelle qu'en soit la qualité acquise, n'explique en rien sa démarche de traduction, comme un logiciel qui conduit en singeant de multiples conducteurs. Ces perroquets logiciels n'ont plus grand-chose de scientifique car ils n'expliquent en rien leur performance. Un réseau de neurones, ingurgitant et digérant toutes les chansons des Beatles, peut produire une nouvelle chanson, originale bien sûr, mais sans rien maîtriser du génie créatif des Beatles. Ces réseaux peuvent être à la fois excellents et ignorants. Il n'y a plus aucun objet conceptuel ni chaîne causale explicitée, formalisée, permettant de clarifier la raison de ces comportements, juste une banale performance d'ingénieur, aussi impressionnante soit-elle. Si cela n'est nullement condamnable pour les jeux de société, les linguistes (ce qu'est Chomsky à l'origine), eux, ont raison de regretter que ces nouveaux systèmes se détachent de la manière toute humaine de mener par exemple pareille traduction, rendant le résultat d'une grande sécheresse et dénué de toute nuance.

Pourquoi un réseau de neurones, aussi profond soit-il, devrait-il réapprendre les règles syntaxiques proposées par les linguistes et qui collent à notre maîtrise du langage, comme les accords ou les règles de grammaire ? Pourquoi réapprendre la loi d'Ohm ou celle des gaz parfaits pour un réseau de neurones censé prédire l'intensité d'un courant ou la mesure d'une pression ? Bien sûr, il est de ces règles difficiles à expliciter, comme celles de nature statistique (dans la plupart des cas – mais combien – ce mot est suivi de celui-là), ou celles dont les prémisses sont trop complexes (quand le verbe auxiliaire est précédé du complément, où qu'il se situe dans les mots qui précèdent, alors il s'accorde...) et qui gagnent à être apprises, mais pas toutes... Bien sûr, un réseau de neurones pourrait aussi prédire quelques écarts par rapport à ces lois parfaites de la physique (comme les non-linéarités), mais il le fera d'autant mieux qu'on lui a préinstallé dans ses connexions, à la main, les lois en question. Il est d'ailleurs tout aussi erroné de considérer les réseaux de neurones convolutifs comme de simples boîtes noires, car les filtres convolutifs qui les composent résultent de très longues années d'expérience et de connaissance en matière de traitement d'image. La structuration subtile de ces réseaux de neurones profonds témoigne d'une ingénierie fine de ce champ d'étude et qui ne doit pas grand-chose au hasard. Les ingénieurs sont présents dans la préstructuration de ces réseaux. On tend à penser qu'ils pourraient l'être encore plus.

Avec le temps, il deviendra sans doute possible de mieux comprendre le fonctionnement, tout excellent qu'il soit, de ces opulents réseaux de neurones multiparamétrés, leur partie vraiment boîte noire, et de déceler en leur sein de nouveaux objets de sciences. Cela aiderait quelque peu à la réconciliation de ces deux traditions qui tendent pour l'instant à s'observer en chien de faïence.

Sans doute les conclusions les plus importantes de cet ouvrage, confrontant les approches algorithmiques en IA que sont « la recherche », « l'optimisation » et « l'apprentissage », sont les suivantes.

- Il devient de plus en plus possible de substituer le génie humain par la puissance de nos ordinateurs, comme on l'a vu quand l'apprentissage par renforcement permet de faire l'économie des heuristiques ou des évaluations intermédiaires. Le match AlphaGo contre Lee Sedol en est un exemple saisissant.
- Il est possible aussi de récupérer cette expertise enfouie dans les masses de données existantes grâce aux réseaux sociaux ou à tous ces systèmes de captation des expériences humaines passées (les conducteurs, les traducteurs...) ou à venir, puis de les faire reproduire par de gigantesques boîtes noires multiparamétrées, sans qu'il ait été nécessaire pour des experts humains de chercher à expliciter le pourquoi de l'excellence de ces comportements. Les réseaux de neurones imitent cette excellence sans chercher à la comprendre. On glisse subrepticement de l'inconscient humain à l'inconscient de nos ordinateurs.
- Cependant, ceci a un coût de calcul important (ainsi, les algorithmes de Deep Learning sont des énergivores voraces) et il est très possible que la conscientisation écologique amène les développeurs de demain à une plus grande sobriété, en exploitant plus et mieux nos ressources cognitives qui, comme tous les neurobiologistes le savent, nous surprennent toujours et encore par leur faible consommation énergétique.
- Comme certains chercheurs soi-disant révolutionnaires l'espèrent actuellement, l'avenir de l'IA ne se déploiera pas au détriment de tous les développements passés, nourris de nos connaissances et de nos processus cognitifs les plus rationnels, mais se trouve à la croisée des traditions anciennes et des apports ponctuels de l'apprentissage, apports qu'il reste essentiel de cantonner dans les angles morts de notre cognition.
- Il est donc conseillé, lorsqu'il s'agit de développer un logiciel pour un problème quelconque, de commencer par quelque chose de simple, de sobre, de compréhensible et de maîtrisable (un simple classifieur linéaire avant un Deep Learning), ce que, nous le déplorons, de moins en moins d'étudiants sont enclins à faire quand ils ont à leur disposition cette offre extraordinaire de solutions logicielles, plus complexes et opaques les unes que les autres mais livrées clés en main. Merci les Gafam !
- La science est un processus hypothetico-déductif et, même si le processus inductif (découvrir le modèle à partir des données) peut s'étoffer par l'apprentissage machine, la meilleure manière de guider et limiter ce processus dans l'infini registre des modèles mathématiques ou logiques possibles passe par le raisonnement, les connaissances accumulées depuis des millénaires dans un cerveau humain précâblé depuis l'origine des temps pour mieux s'interfacer au monde. Les machines, elles, n'ont et n'auront jamais aucune volonté de survie.

Vous voici arrivés à la fin du livre. Nous espérons que vous serez maintenant capables de mieux différencier les trois piliers fondamentaux de l'IA que sont la **recherche**, l'**optimisation** et l'**apprentissage**. Nous espérons que, face à un problème pour lequel l'IA a quelque chose à offrir (et il y en a une infinité), vous déduirez assez vite et sans vous égarer quel type de démarche algorithmique s'avère le plus prometteur. Vous allez vous surprendre à découvrir le nombre de problèmes rencontrés dans la vie et dans l'industrie qui s'apparentent au taquin ou à la classification des spams. Et même si ce n'est pas, mot pour mot, le même algorithme que celui présenté dans le livre qui vous permettra de l'affronter, ce que vous aurez assimilé et compris malgré tout dans ces pages transparaîtra sans aucun doute dans l'algorithme idéal que vous écrierez ou réutiliserez.

Le monde est devenu bien trop complexe pour en confier la gestion aux seuls gouvernants en chair et en os. Les lourds nuages noirs chargés d'orage qui nous menacent, comme l'économie mondialisée et les crises qui s'ensuivent régulièrement, l'explosion des inégalités, le réchauffement climatique et la dégradation de l'environnement, la détérioration de l'habitat, la transition énergétique inéluctable, l'agriculture qui s'intensifie en tout sauf en bienfaits pour la santé, les fractures communautaires et la flambée du terrorisme exigent une meilleure compréhension et interprétation des phénomènes, suivies de délibérations approfondies pour en trouver les parades et les issues.

De son côté, l'IA – meilleure joueuse d'échecs, meilleure joueuse de go, championne de poker, du jeu télévisé *Questions pour un champion*, conductrice irréprochable, médecin infailible, meilleure prévisionniste des soubresauts économiques et des variations climatiques, traductrice la plus sollicitée, meilleure décideuse – est capable, par sa boulimie de données multiples à très haute fréquence d'acquisition et son pouvoir de simulation pour en faire sens, d'appréhender ce monde, son évolution et les meilleures façons de le remettre sur les rails. Cette même IA, que l'on qualifie ces jours-ci de super-intelligente et qui, de surcroît, s'immisce dans tous les recoins de nos existences, est capable d'explorer, à partir de ces simulations, ces états des lieux d'une précision encore jamais atteinte, les manières les plus efficaces d'affronter l'ensemble de ces difficultés et d'en découvrir les remèdes les plus appropriés. Et si elle parvient à cela, c'est en exécutant à la vitesse de la lumière de multiples algorithmes dont les fonctions sont nombreuses et variées : planifier et tester des futurs alternatifs, recueillir des myriades de données, les stocker intelligemment pour mieux les exploiter, s'en servir en effet pour la conception de modèles au pouvoir prédictif, exploiter ces modèles jusqu'à plus soif pour tester les options possibles, ajuster, optimiser ces dernières et aller jusqu'à la prise de contrôle de nos vies et le tracé de nos existences vers davantage d'harmonisation sociale dans un monde préservé et pacifié.

Il n'y a plus guère un seul objet de notre quotidien qui ne recèle quelque intelligence algorithmique, qui ne soit capable de percevoir son environnement et de décider de manière autonome de la meilleure action à entreprendre. Il décide en effet tout seul ou de façon coordonnée, dialoguant avec ses partenaires distribués à travers le globe. Nous vivons à la fois l'ère de la super-intelligence informatique, de la puissance décuplée des processeurs, des objets connectés et de l'informatique ubiquitaire. Pourtant, c'est cette même ère qui se trouve menacée d'extinction. Et c'est surtout à vous que nous pensons, les plus jeunes, les programmeurs en herbe, afin de saisir les opportunités extraordinaires que l'IA vous offre pour faire face à tous ces défis, et d'y faire face au plus vite.

Index

A

- A* 2, 11, 13, 15, 17, 19, 24, 28, 40, 62, 63, 70
- accuracy 104, 125
- action 10, 16, 62
- Adam 66, 86, 146
- algorithme
 - fourmis 31
 - génétique 2, 24, 31, 39, 48, 68, 69, 73, 84
 - glouton 124
- AlphaGo 52, 157
- AlphaZero 52
- analyse en composantes principales 136
- apprentissage 1, 11, 50, 112, 156
 - non supervisé 128, 129
 - off-policy 22
 - on-policy 22
 - par renforcement 2, 16, 19, 48, 64, 69, 84, 88
 - profond 3, 64, 132, 140, 153
 - supervisé 64, 102, 103, 113, 117, 128
- arbitrage exploration/exploitation 40, 50
- arbre de décision 112, 113, 116
 - critère d'arrêt 116
 - forêt 117, 124
- arbre de recherche 10, 24, 28, 40, 48

B

- backtracking 10, 40
- bag of words 105
- biais 155

C

- CBMPI – Classification-based modified policy iteration 88
- classification 112, 141
 - binaire 102, 113
 - éthique 117
 - naïve bayésienne 102, 106
 - récurtivité 116
 - supervisée 117
- cognition humaine 14, 19, 48, 152, 156, 157
- compréhension du langage naturel 153

- condition de Markov 18, 31
- conférence de Dartmouth 4, 64, 151, 154
- convolution 145
- coût de calcul 157
- croisement 22
- cycle hamiltonien 70

D

- Deep Blue 50
- Dijkstra 29
- Dirichlet 129, 134
- distance
 - de Manhattan 14, 63, 70
 - euclidienne 28
- données
 - d'entraînement 103, 122, 145
 - de test 122, 145
 - de validation 112, 116, 145
 - enrichissement 145
 - textuelles 129
- dropout 145

E

- Eliza 152
- entropie 114
- entropie croisée binaire 146
- erreur moyenne quadratique 146
- espace de solutions 24
- essai/erreur 3, 17, 19
- état 10, 16, 62
- évaluation 22, 39, 68
- expansion 50

F

- fitness 40
- fonction d'activation
 - ReLU 66, 77, 86, 145
 - Sigmoid 65, 146
- fonction de coût 40, 146
- fourmi 32

G

GPS – General Problem Solver 10, 154
graphe 30

H

heuristique 13, 19, 28, 40, 48
hybridation 41, 52, 68

I

IDF – Inverse Document Frequency 104
index de Gini 114, 115
inférence logique 110
intelligence 151
intelligence artificielle
 consciente 154, 155
 démarche 157
 inconsciente 154, 155

J

justesse 125

K

Kaggle 105, 145
K-moyennes 132

L

LDA 129, 132

M

MCTS – Monte Carlo Tree Search 50, 52
méthode de gradient 66, 69
Min-Max 48, 52
modèle mental 2
multiparamétrage 143
mutation 22, 39, 68, 69

N

n-gram 104
NLTK 104

O

optimisation 1, 38, 48, 66, 143, 156
 génétique 22

P

PCA 136
plus court chemin 28
policy 20
pooling 144
précision 104

programmation dynamique 19

Q

Q-learning 2, 3, 16, 48, 84, 85
 policy iteration 86

R

racinisation 104
rappel 104
recherche 1, 11, 15, 17, 38, 48, 156
 best-first search 40
 bidirectionnelle 30
 largeur d'abord 11, 28
 profondeur d'abord 11
recombinaison 39, 68
réseau de neurones 3, 24, 64, 65, 73, 84, 86, 131, 140, 154, 156
 convolution 143, 144
 pooling 143, 144
 profond 140
rétropropagation 50, 66
RMSprop 146

S

sac de mots 105, 108
sélection 22, 39, 50, 68
similarité 103
simulation 50
solution 10, 17, 62
stemming 104
stratégie 20
 élitiste 39
surapprentissage 125, 144
système expert 110

T

temps de calcul 24
test de Turing 152
TF – Term Frequency 104
traitement du langage naturel 102, 104
transfer learning 146

V

valeur prédictive positive 104
voyageur de commerce 31

W

Waze 30
word embedding 131