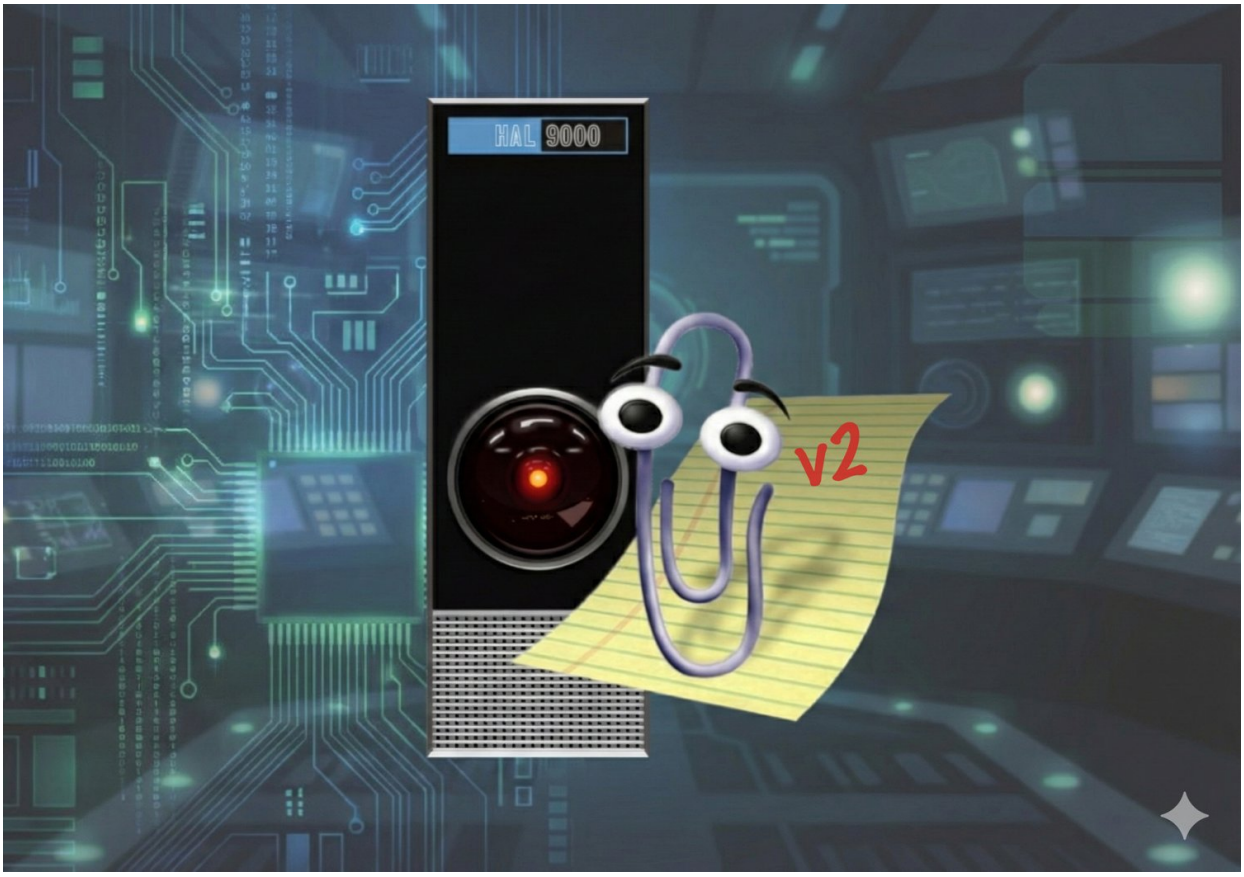
INF1901 — Initiation à l'intelligence artificielle

Concepts et réflexions



Christian Jauvin

Version du 6 octobre 2026

Table des matières

Bienvenue ! **1**

Présentation du cours **2**
Objectifs du cours . . . 2
Contenu . . . 3

Approche pédagogique du cours **5**

Les professeurs **7**
Richard Hotte . . . 7
Christian Jauvin . . . 8
Daniel Lemire . . . 9

Travaux notés **10**
Module 1 . . . 10
Module 2 . . . 10
Module 3 . . . 10
Module 4 . . . 10
Module 5 . . . 10

Feuille de route **11**

Google Sheets **12**
Paramètres linguistiques . . . 12
Fonction “copier-coller” . . . 14
Fonction de partage anonyme d’un fichier . . . 14

L’usage d’assistants intelligents **16**

Venez discuter d’IA sur Discord ! **18**

Module 1 - Aux origines : l’intelligence artificielle symbolique **20**
Le parcours du module . . . 21
Objectifs . . . 22
Durée . . . 22
Évaluation . . . 22

Turing et la question fondatrice (1950) **23**
Le moment . . . 23
Le jeu de l'imitation . . . 25
La pensée comme calcul . . . 27
L'ombre de Gödel . . . 30
Une fin tragique, et une porte qui s'ouvre . . . 32

Deux paris rivaux (1956-1958) **34**
Le moment : l'été 1956 à Dartmouth . . . 34
Le premier pari : l'esprit comme logique . . . 36
Le second pari : l'esprit comme cerveau . . . 37
Deux univers parallèles . . . 39

L'âge d'or symbolique : chercher et raisonner **40**
Résoudre, c'est explorer . . . 40
L'explosion combinatoire . . . 41
L'apogée : Deep Blue bat Kasparov (1997) . . . 43
L'autre visage : ELIZA, ou l'illusion de comprendre . . . 44

Représenter le monde **47**
Le sens, angle mort de la machine . . . 47
Donner un savoir à la machine . . . 47
SHRDLU, ou le sommet de l'ambition . . . 50
Le mur du sens commun . . . 51
L'objection de Hofstadter . . . 52

Capturer l'expertise : les systèmes experts **56**
Rétrécir et codifier le monde pour réussir . . . 56
L'anatomie d'un système expert . . . 56
L'âge d'or : MYCIN, XCON et le boom . . . 59
Le goulot d'étranglement . . . 61

Les hivers et la bascule **63**
Le premier hiver : la mort du perceptron (1969) . . . 63
Le second hiver : l'effondrement du symbolique (fin des années 1980) . . . 65
L'héritage invisible . . . 65
Un éclair hybride : Watson (2011) . . . 67
La bascule . . . 68

Vous êtes le moteur d'inférence : un système expert (travail noté 1) **70**
Consignes . . . 70
Le système . . . 71
Manipulation 1 — Tracer le raisonnement . . . 72
Manipulation 2 — Casser le système . . . 72
Manipulation 3 — Étendre la base de règles . . . 72
Questions d'interprétation . . . 73

Module 2 — Apprentissage automatique **76**
Qu'est-ce que l'apprentissage automatique ? . . . 76
Une seule grande idée, déclinée . . . 77
Une vieille parenté : la statistique . . . 78
Le but ultime : généraliser . . . 78
Le parcours du module . . . 79
Objectifs . . . 80
Durée . . . 81
Évaluation . . . 81

Le problème **82**
Pourquoi on ne peut pas simplement le programmer . . . 82
Changer de stratégie : apprendre à partir d'exemples . . . 83
Notre fil rouge : des maisons à vendre . . . 84
Une seconde question, d'une tout autre nature . . . 86

Le modèle le plus bête **90**
Toujours prédire la moyenne . . . 90
La même bêtise, pour l'autre question . . . 91
Qu'est-ce qu'un modèle, au juste ? . . . 91
Pourquoi un modèle aussi bête est utile . . . 92
Leur défaut, et ce qu'il révèle . . . 95

Regarder les données **96**
Une maison, c'est une liste de nombres . . . 96
Un vecteur, c'est un point dans un espace . . . 98
Quand il y a trop de dimensions pour les voir . . . 99

Prédire par ressemblance **108**
Mesurer la ressemblance : la distance . . . 108
Les k plus proches voisins . . . 111
Le choix de k . . . 114
L'angle mort de kNN . . . 115

Un modèle qui s'entraîne **116**
Un modèle qui tient en deux nombres . . . 116
Mesurer l'erreur . . . 117
Apprendre, c'est descendre la pente . . . 119
Et pour prédire une catégorie ? . . . 120

Classer **122**
Tracer une frontière : la régression logistique . . . 123
La plus grande marge : les machines à vecteurs de support . . . 125
Renverser le problème : la classification bayésienne . . . 128
Le cas des pourriels . . . 134
Raisonner sous incertitude : les réseaux bayésiens . . . 135
Sous les modèles, des probabilités . . . 138

Poser des questions : les arbres de décision **140**

Une question suffit presque . . . 140
Comment l'arbre choisit ses questions . . . 142
Un arbre qui prédit un nombre . . . 143
Jusqu'où laisser pousser l'arbre ? . . . 144
L'arbre du *Titanic* . . . 146
Ce qu'un arbre dit, et ce qu'il tait . . . 147
Et sur des données neuves ? . . . 149

Généraliser **150**
Un modèle se juge sur ce qu'il n'a jamais vu . . . 150
Linéaire ou non-linéaire : ce qu'un modèle peut dessiner . . . 152
Trop coller, ou trop lisser : le compromis biais-variance . . . 155
Garder un modèle riche, mais le tenir en laisse : la régularisation . . . 157
Paramétrique ou non-paramétrique . . . 160

Bien évaluer un modèle **162**
Le score trop beau pour être vrai : la fuite de données . . . 162
Quand les données sont rares : la validation croisée . . . 164
Compter juste : les métriques . . . 165
Jamais vu, mais du même monde : la question de la distribution . . . 169
Corrélation n'est pas causalité . . . 171
Tout cela portait un nom : l'apprentissage supervisé . . . 172

Trois façons d'apprendre **174**
Apprendre avec un professeur : le supervisé . . . 175
Apprendre sans réponses : le non-supervisé . . . 175
Apprendre par l'expérience : le renforcement . . . 177
Un même squelette, d'un bout à l'autre . . . 178

Classification naïve bayésienne pour détecter les pourriels (travail noté 2) **179**
Consignes . . . 180
Entraînement du modèle . . . 180
Utilisation du modèle (inférence) . . . 185
Questions d'interprétation . . . 186

Module 3 — Réseaux de neurones et apprentissage profond **189**
Le retour d'une idée ancienne . . . 189
Une seule grande idée : apprendre ses propres caractéristiques . . . 190
Un fil conducteur : les chiffres manuscrits . . . 191
Ce que ce module n'est pas . . . 192
Des variations sur un même thème . . . 192
Le parcours du module . . . 193
Objectifs . . . 194
Durée . . . 195
Évaluation . . . 195

Un neurone **196**
Un neurone est une régression logistique . . . 196

D'où vient le mot « neurone » . . . 198
Un neurone à manipuler . . . 199
Ce qu'un neurone seul ne peut pas faire . . . 200

Une couche cachée **201**
Relier des neurones . . . 201
Le XOR résolu avec trois neurones . . . 202
Ce que fait la couche cachée : changer de point de vue . . . 203
Deux droites à déplacer . . . 204
Plus de neurones, plus de formes . . . 205
Qui règle tous ces poids ? . . . 205

Entraîner un réseau **207**
Le même principe qu'au Module 2 . . . 207
Le problème des couches cachées . . . 208
La rétropropagation . . . 209
Un réseau qui apprend le XOR . . . 210
1986 . . . 210
L'entraînement en pratique . . . 211
Pour voir le mécanisme en détail . . . 212

L'apprentissage profond **213**
Large ou profond . . . 213
Une hiérarchie de caractéristiques . . . 214
La fin des caractéristiques fabriquées à la main . . . 217
Pourquoi seulement en 2012 . . . 218
2012 : le concours ImageNet . . . 220
La leçon amère . . . 222
Plus de paramètres que d'exemples . . . 223
Apprendre sans étiquettes : l'autoencodeur . . . 224

Le matériel et les outils **226**
Les processeurs graphiques . . . 226
La différentiation automatique . . . 228
Un réseau en quelques lignes . . . 229
Un jeu de construction . . . 229

Voir : les réseaux convolutifs **231**
Ce qu'un réseau ordinaire ignore dans une image . . . 231
Le filtre . . . 232
Un filtre à manipuler . . . 233
Ce que la convolution apporte . . . 233
Du filtre au réseau . . . 233
Yann Le Cun et les chèques . . . 234
Après 2012 . . . 236
Réutiliser un réseau : l'apprentissage par transfert . . . 237
La leçon de la convolution . . . 238

Lire une séquence : les réseaux récurrents **240**

Le problème des séquences . . . 240
Une boucle : la mémoire . . . 240
Un réseau récurrent à manipuler . . . 241
Entraîner un réseau récurrent, et ce qu'il oublie . . . 242
Le LSTM : une mémoire à portes . . . 242
Ce que les réseaux récurrents ont permis . . . 244
Les limites . . . 244

L'attention et le Transformer **246**
L'attention dans la traduction . . . 246
Une recherche floue . . . 247
L'auto-attention . . . 249
Une attention à manipuler . . . 249
Le Transformer . . . 249
Pourquoi il a tout changé . . . 251
Au-delà du texte . . . 252
Vers le Module 4 . . . 253
Pour voir le mécanisme en détail . . . 253

Apprendre à jouer : le renforcement profond **254**
Les limites de la table . . . 254
Remplacer la table par un réseau . . . 254
Atari : jouer à partir des pixels . . . 255
AlphaGo . . . 256
AlphaZero : sans parties humaines . . . 258
Une machine qui apprend en jouant contre elle-même . . . 259
Au-delà des jeux, et les limites . . . 259

Tromper un réseau **261**
Le panda devenu gibbon . . . 261
Comment on le fabrique . . . 262
Tromper un réseau à manipuler . . . 262
Pourquoi c'est possible . . . 262
Dans le monde réel . . . 264
Se défendre . . . 266
Une boîte noire . . . 266
Ce que ce module a montré . . . 267

Apprivoiser un réseau de neurones (travail noté 3) **269**
Consignes . . . 269
Questions . . . 269

Pour aller plus loin **274**
Le podcast MLST . . . 274
Présentations diverses . . . 274

Conclusion **275**

Bienvenue !

Version 2, en construction. Ce site présente la nouvelle version du cours, qu'on est en train de refaire. Pour l'instant, seuls les modules 1 et 2 sont publiés ; les modules 3 à 5 s'ajouteront à mesure qu'ils seront prêts. La version actuelle du cours reste accessible à cjauvin.github.io/inf1901-teluq.

Bienvenue dans le cours **INF 1901 Initiation à l'intelligence artificielle : concepts et réflexions**. Ce cours vous offre une occasion unique de vous initier au domaine de l'intelligence artificielle (IA) dans un contexte professionnel.

Ce cours explore l'évolution de l'intelligence artificielle, des systèmes symboliques à la révolution de l'IA générative récente. Il examine les fondements de l'apprentissage automatique, ancrés dans les données, les statistiques et les algorithmes, tout en clarifiant des concepts clés comme l'entraînement, l'inférence et la généralisation. Une attention particulière est portée aux réseaux de neurones et à l'apprentissage profond, en détaillant leur structure et leurs analogies avec le cerveau humain, en portant une attention particulière sur la vision artificielle où les progrès initiaux furent les plus décisifs. Le cours retrace également le développement des modèles de langage. Enfin il aborde les enjeux éthiques, philosophiques et sociétaux de l'intelligence artificielle, incluant l'alignement des systèmes, la singularité technologique et leurs effets sur la société.

Aucune expérience préalable en IA n'est requise pour réussir ce cours – seulement une curiosité et une volonté d'apprendre. Que vous soyez une ou un professionnel cherchant à approfondir vos compétences ou une personne passionnée par les possibilités de l'IA, ce cours est conçu pour vous. Nous vous invitons à consulter le site Web du cours dès son lancement pour découvrir la feuille de route et la liste des activités prévues. Le site du cours comprend une dizaine de vidéos et plusieurs activités interactives à réaliser dans votre navigateur Web. Il ne s'agit pas d'un cours léger : vous compterez près de 35,000 mots et des dizaines d'illustrations. Si vous avez des questions, n'hésitez pas à nous contacter. Nous sommes impatients de collaborer avec vous dans cette aventure !

Au plaisir de vous guider dans cette initiation à l'intelligence artificielle : concepts et réflexions. Nous vous souhaitons du succès dans vos apprentissage.

Présentation du cours

> « *Toute technologie suffisamment avancée est indiscernable de la magie.* »
> Arthur C. Clarke

L'intelligence artificielle (IA) constitue aujourd'hui un champ de recherches et d'applications essentiel dont l'influence s'étend à de multiples domaines, de la science et la médecine aux industries créatives et à l'éducation. Conséquemment, les entreprises de divers secteurs sont confrontées à cette transformation numérique sans précédent, si bien que l'intégration de l'IA dans les modèles d'affaires n'est plus un luxe, mais une nécessité pour demeurer compétitif dans un marché mondial en constante évolution. L'émergence de modèles de langage avancés tels que ChatGPT, DeepSeek et Llama 3 illustre cette transformation, en ouvrant de nouvelles perspectives tant scientifiques que sociétales.

Le cours INF 1901 Introduction à l'intelligence artificielle introduit le rôle des diverses formes de l'intelligence et, par conséquent, d'apprentissage chez l'humain qui ont trouvé un écho dans le développement des formes d'intelligences artificielles de la Conférence de Dartmouth en 1956 aux avancées récentes en IA générative et les grands modèles de langage.

En plus des connaissances théoriques, le cours propose une réflexion critique sur les usages et les impacts de ces technologies, en interrogeant les enjeux éthiques, les défis économiques et les perspectives qu'elles soulèvent dans un contexte de diffusion massive et d'appropriation croissante par les acteurs publics et privés.

Objectifs du cours

Ce cours vise offrir une base solide à la poursuite du programme court en intelligence artificielle (IA) dans les entreprises et, éventuellement, des études ou des projets plus avancés dans le domaine de l'IA ou dans des secteurs influencés par cette technologie.

Plus spécifiquement, le cours habilitera l'étudiante ou l'étudiants à :

- Expliquer les bases et l'évolution historique de l'IA.
- Identifier et analyser des applications de l'IA dans des contextes concrets.
- Développer un regard critique et instruit sur les impacts de l'IA.
- Utiliser les notions essentielles pour aborder des discussions éthiques et stratégiques sur l'IA.

Dans l'introduction de chaque module du cours, vous trouverez les objectifs spécifiques poursuivis.

Contenu

Le contenu du cours se découpe en cinq modules.

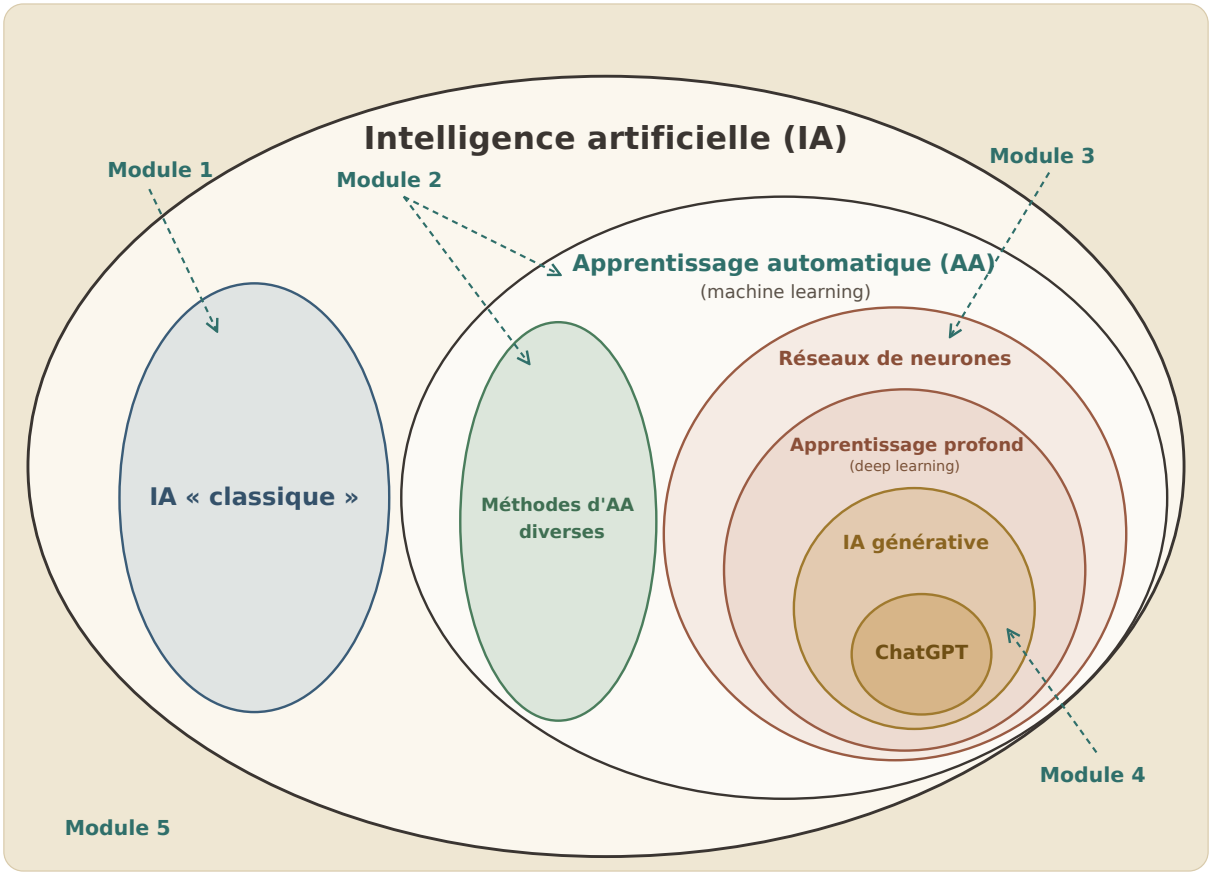
1. Le premier module remonte **aux origines** de l'IA et raconte l'aventure de l'**intelligence artificielle symbolique** (le *GOFAI*, pour *Good Old-Fashioned AI*) : pendant près de quarante ans, le pari de faire « penser » une machine en lui faisant manipuler des **symboles** et des **règles** logiques. Dans une perspective historique, du test de Turing (1950) aux systèmes experts et aux « hivers » de l'IA, ce module présente les grandes idées algorithmiques de cette première école (recherche dans un espace d'états, représentation des connaissances, raisonnement à base de règles) et explique *pourquoi* elle a fini par buter sur ses limites — ouvrant la voie à l'apprentissage automatique étudié dans les modules suivants.

2. Le deuxième module introduit les principes et techniques fondamentaux de l'apprentissage automatique (machine learning) en explorant les différents types d'apprentissage (supervisé, non supervisé, par renforcement) et leurs applications. Ce module vise à décrire comment les algorithmes apprennent à partir des données pour faire des prédictions, identifier des modèles et automatiser des tâches complexes.

3. Le troisième module explore les architectures de l'apprentissage profond (deep learning) comme les réseaux à convolution et récurrents, en mettant l'accent sur leur capacité à traiter des données complexes (image, texte, son).

4. Le quatrième module se consacre à l'IA générative et à l'essor récent des grands modèles de langage.

5. Le cinquième module propose une vision plus vaste du sujet, avec l'exploration de perspectives autour de l'IA, à la fois philosophiques, scientifiques et sociales.



La carte du cours : chaque module explore une région de l'intelligence artificielle, des plus anciennes aux plus récentes — et le Module 5 les traverse toutes.

Approche pédagogique du cours

> « *Le vrai problème de l'humanité est le suivant : nous avons des émotions paléolithiques, des institutions médiévales, et une technologie digne des dieux.* »
> E.O. Wilson

L'apprentissage automatique (AA) est un sujet complexe et fortement dynamique, au confluent de plusieurs disciplines : mathématiques, informatique, sciences cognitives. Les fondements de la discipline sont résolument mathématiques, principalement avec l'algèbre linéaire et les probabilités (qui entretiennent elles-mêmes des liens très étroits).

La plupart des cours d'apprentissage automatique de format classique sont avant tout très mathématiques, avec beaucoup d'équations et de théorèmes, parfois des preuves. Un cours d'AA entièrement composé d'équations (et rien d'autre) est en général perçu comme étant *très difficile*. Il serait aussi très difficile de ne pas parler de mathématique du tout dans un cours d'initiation à l'apprentissage automatique, mais nous allons tout de même tenter d'offrir plusieurs angles sur le sujet. Étant donné les fondations en algèbre linéaire, les problèmes traités par l'AA ont souvent la possibilité d'être compris de manière très visuelle et intuitive. Nous allons tenter d'exploiter cela, en mettant le plus possible l'emphase sur la visualisation et les explications claires en mots, plutôt que les équations (bien que ces dernières seront présentes mais optionnelles, car elles constituent tout de même les fondations théoriques, qui permettent de réduire l'ambiguïté conceptuelle au minimum, et souvent de faire des liens profonds et subtils entre des idées mathématiques qui ne semblent pas à priori si proches).

Étant donné que l'apprentissage automatique est algorithmique et dynamique (il s'agit de techniques avec lesquelles on *fait* des choses, en général), les visualisations interactives constituent également un excellent support pour approfondir sa compréhension. Dans ce cours, nous avons tenté de développer nos propres petites applications interactives pour aider à la compréhension de certains concepts, et les vidéos que nous proposons du créateur en ligne 3Blue1Brown constituent également un fabuleux complément, dont il ne faudrait pas se passer.

L'apprentissage automatique est aussi profondément computationnel, et évidemment la programmation y joue un grand rôle. Le substrat de l'intelligence artificielle est l'ordinateur, et celui-ci n'a jamais cessé, depuis sa création, d'être basé sur l'architecture de Von Neumann, ce qui implique que son fonctionnement requiert nécessairement de la programmation au sens traditionnel (l'AA n'y échappe pas). Il serait probablement possible d'enseigner les fondements de l'AA entièrement de cette perspective, étant donné qu'il est possible d'exprimer toutes les idées mathématiques à l'aide de programmes. Nous allons toutefois nous contenter de faire cela seulement à quelques endroits, qui resteront optionnels. Il est à noter que deux des travaux notés utilisent Google Sheets,

la version en ligne de Excel, qui est un environnement de calcul qui se rapproche beaucoup de la programmation, de par sa nature même. Nous pensons que ceci fera en sorte d'offrir au moins une idée approximative du développement en AA, même pour les gens qui ne maîtrisent pas la programmation.

Finalement, il est impossible d'ignorer le fait que l'apprentissage automatique, tout comme certaines autres innovations technologiques récentes (comme les médias sociaux) a fini par avoir des effets importants sur la société. Nous croyons donc qu'il est important qu'un cours d'initiation traite de certains sujets plus larges, et place l'AA dans une perspective historique et philosophique.

Les professeurs

> « *Le futur est déjà là, il n'est juste pas très équitablement réparti.* »
> William Gibson

Ce cours a été co-créé par trois professeurs de l'Université TÉLUQ.

Richard Hotte

Richard Hotte

Richard Hotte. Photo : Université TÉLUQ.

Richard Hotte a été titulaire de la chaire UNESCO en système intelligent d'apprentissage repensé

pour tous (UNESCO Chair for Global Smart Disruptive Learning (GSDL)), de 2019 à 2023. Cette chaire se rattache au programme UNITWIN/Chaires UNESCO et s'inscrit dans le champ de l'intelligence artificielle et apprentissage mobile. Elle a pour objet de promouvoir un système intégré de recherche, de formation et de documentation sur l'apprentissage mobile. Elle vise à faciliter la collaboration entre des chercheurs et des enseignants de renommée internationale de l'Université et d'autres établissements au Canada ainsi qu'ailleurs en Amérique du Nord, en Europe, en Afrique et dans d'autres régions du monde.

- Chachoui, Y., Azizi, N., Hotte, R., & Bensebaa, T. (2024). Enhancing algorithmic assessment in education: Equi-fused-data-based SMOTE for balanced learning. Computers and Education: Artificial Intelligence, 6, 100222.
- Hotte, R., Masmoudi, A., Jaballah, A., Masmoudi, O., & Maïga, A. A. (2021). Work-in-progress about dynamicity as a foundation for AMI, a mobile intelligent and adaptive learning system. In Interactive Mobile Communication, Technologies and Learning (pp. 111-119). Cham: Springer International Publishing.

Christian Jauvin

Christian Jauvin

Christian Jauvin. Photo : Université TÉLUQ.

Christian Jauvin s'intéresse à la programmation, à la modélisation et la science des données, à l'architecture logicielle, à la géomatique, à l'intelligence artificielle et à l'impact social des technologies numériques.

Il a participé à ses débuts à des recherches pionnières sur la modélisation statistique du langage naturel avec Yoshua Bengio, notamment à l'article fondateur A Neural Probabilistic Language Model. Il a ensuite développé une expertise variée à travers des projets dans des contextes universitaires, corporatifs et comme consultant indépendant. Ses articles et communications ont été cités près de 13,000 fois.

Il a travaillé sur des systèmes logiciels dans des domaines tels que la santé publique (McGill), les jeux vidéo (Ubisoft), les systèmes d'information géographique (K2 Géospatial) et la modélisation climatique (Ouranos). Son parcours l'a amené à concevoir des applications distribuées dans le cloud, à contribuer à des projets open source, et à développer des outils logiciels sur mesure pour des besoins spécifiques.

- Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. Journal of machine learning research, 3(Feb), 1137-1155.
- Delagrange, S., Jauvin, C., & Rochon, P. (2014). PypeTree: A tool for reconstructing tree perennial tissues from point clouds. Sensors, 14(3), 4271-4289.

Daniel Lemire

Daniel Lemire

Daniel Lemire. Photo : Université TÉLUQ.

Daniel Lemire est professeur titulaire en informatique. Il s'intéresse particulièrement à la performance du logiciel et aux techniques d'indexation en science des données. Il aime poser un regard critique sur l'utilisation des technologies.

Daniel Lemire est parmi les 2 % de scientifiques les plus cités au monde (classement de l'Université Stanford, 2024). Il fait partie des 0,0006 % de programmeurs les plus suivis au monde sur GitHub ; GitHub compte plus de 100 millions de développeurs.

- Lemire, D., & Maclachlan, A. (2005, April). Slope one predictors for online rating-based collaborative filtering. In Proceedings of the 2005 SIAM International Conference on Data Mining (pp. 471-475). Society for Industrial and Applied Mathematics.
- Langdale, G., & Lemire, D. (2019). Parsing gigabytes of JSON per second. The VLDB Journal, 28(6), 941-960.

Travaux notés

> « *Bien sûr. Je suis une machine, et vous êtes une machine, et nous pensons tous les deux, n'est-ce pas ?* »
> Claude Shannon (en réponse à la question "est-ce qu'une machine peut penser?")

Le cours sera évalué avec cinq travaux pratiques, un par module.

Module 1

Système expert à incarner : vous jouez vous-même le rôle du moteur d'inférence d'un système à base de règles `si… alors…`, que vous faites tourner, mettez en échec, puis corrigez. *Pondération : 15% de la note finale.*

Module 2

Construction guidée d'un algorithme de classification des courriels. *Pondération : 20% de la note finale.*

Module 3

Exercices interactifs suivis de questions interprétatives sur l'apprentissage profond. *Pondération : 20% de la note finale.*

Module 4

Construction guidée d'un modèle de langage bigramme. *Pondération : 20% de la note finale.*

Module 5

Rapport critique sur une question philosophique ou éthique reliée à l'IA. *Pondération : 25% de la note finale.*

Feuille de route

> « *Nous avons tendance à surestimer les effets d'une technologie à court terme et à les sous-estimer à long terme.* »
> Roy Amara (cette idée est appelée la "Loi d'Amara")

Prévoyez environ 9 heures de travail par semaine pendant quinze semaines. Pour la plupart d'entre vous, il est impossible de réussir ce cours en y consacrant que quelques d'heures par semaine.

Il n'y a pas de dates fixes pour la remise des travaux, mais il est recommandé de prévoir une semaine de travail pour la réalisation de chacun d'eux et de les remettre selon l'ordre de présentation du cours. Bien que chaque personne inscrite à ce cours ne progressera pas au même rythme dans son apprentissage. Cependant, nous vous suggérons de remettre vos travaux notés lors de la semaine indiquée par le tableau suivant :

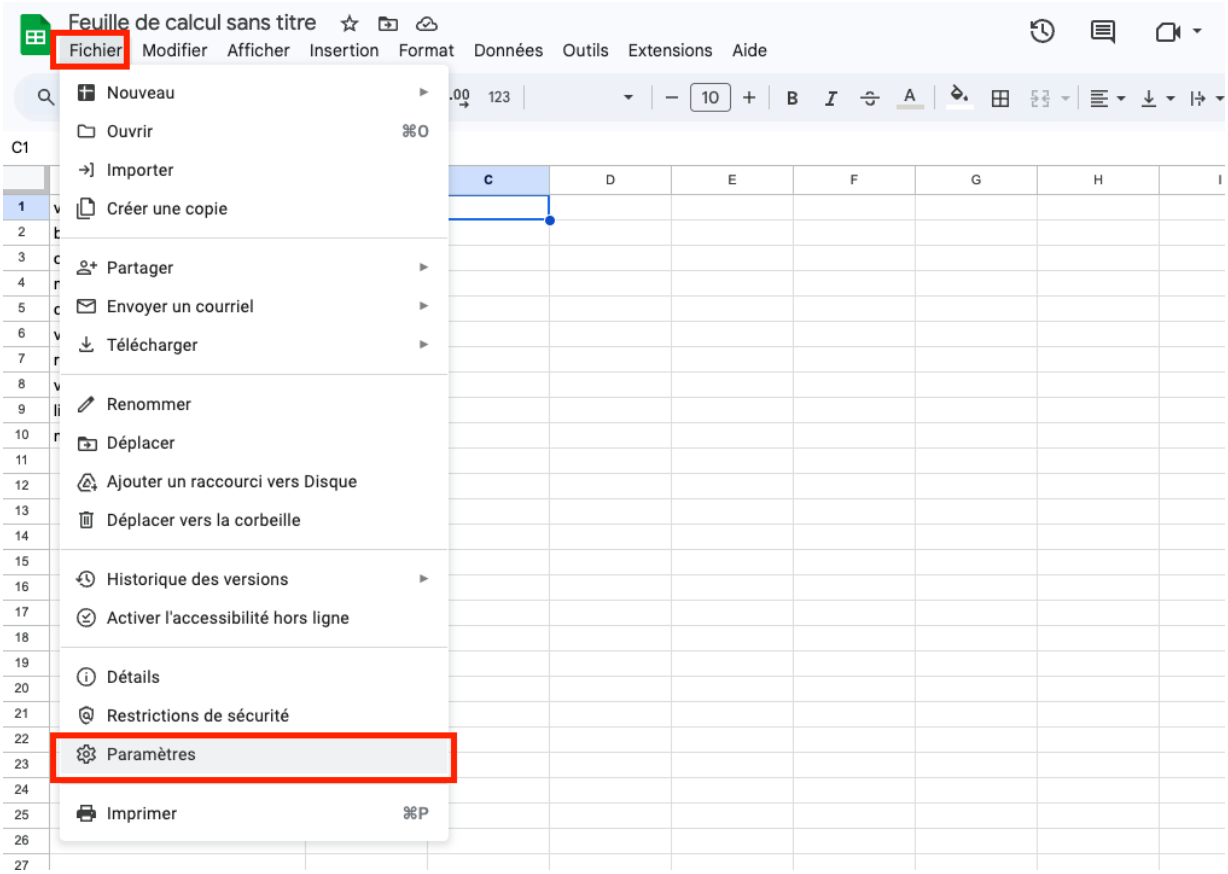
Semaines	Activités	Remises TN
1	Activités du module 1	
2	**Temps de travail sur le TN1**	**TN1**
3	Activités du module 2	
4	Activités du module 2 (suite)	
5	**Temps de travail sur le TN2**	**TN2**
6	Activités du module 3	
7	Activités du module 3 (suite)	
8	**Temps de travail sur le TN3**	**TN3**
9	Activités du module 4	
10	Activités du module 4 (suite)	
11	**Temps de travail sur le TN4**	**TN4**
12	Activités du module 5	
13	Activités du module 5 (suite)	
14	Activités du module 5 (suite)	
15	**Temps de travail sur le TN5**	**TN5**

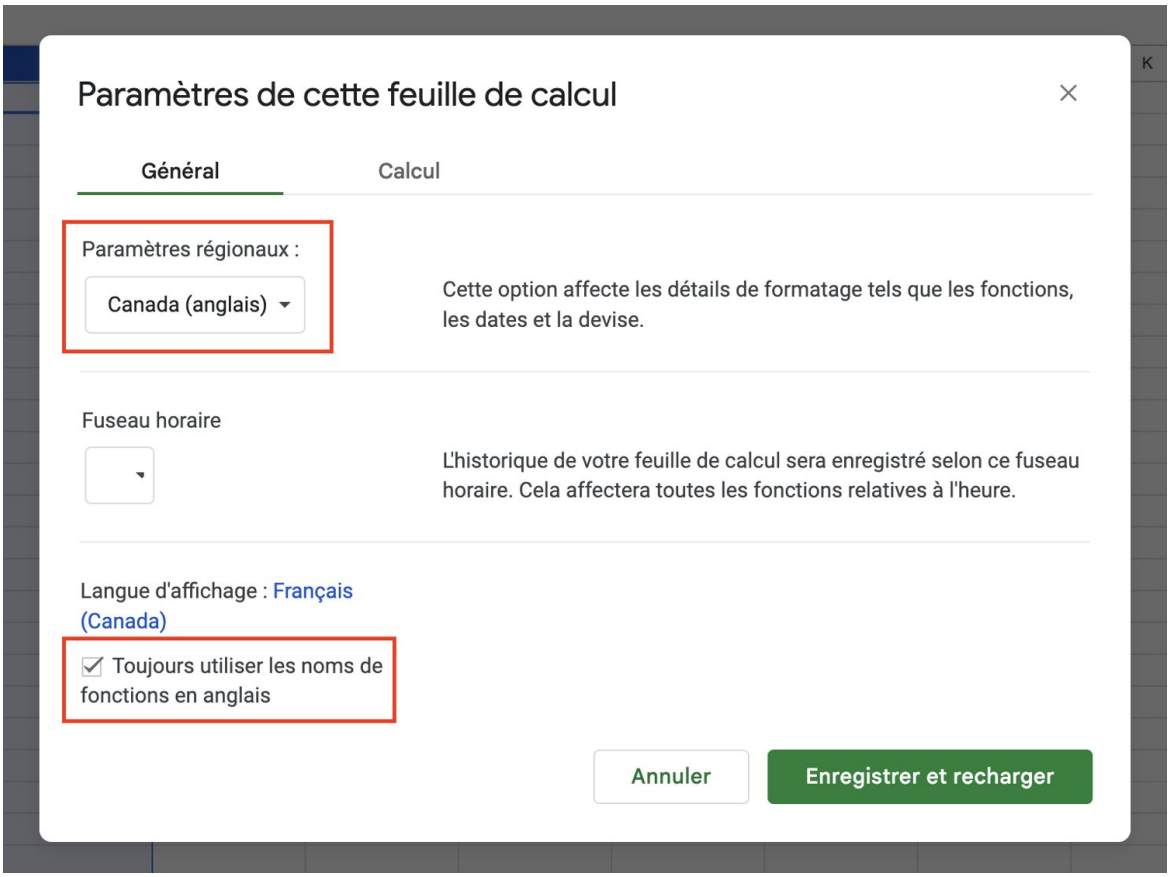
Il est parfaitement normal de consacrer des dizaines d'heures à compléter un travail noté et les activités préparatoires. Il est de votre responsabilité de trouver le temps nécessaire.

Google Sheets

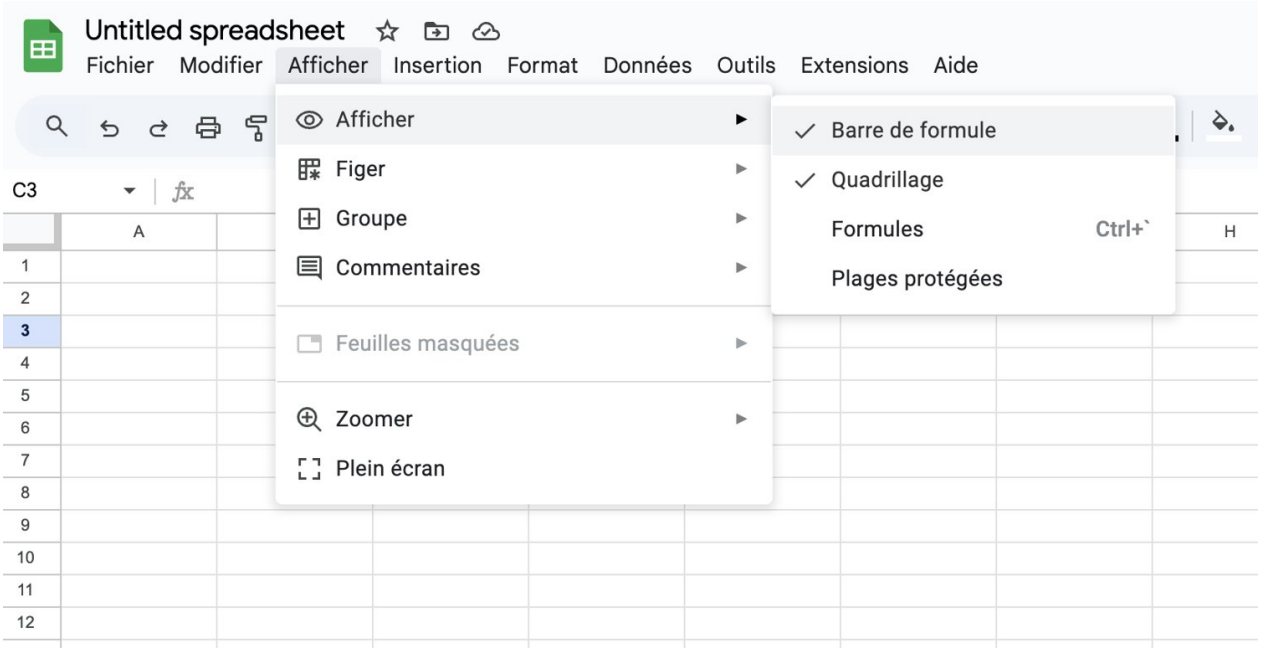
Paramètres linguistiques

Pour les travaux notés 2 et 4, nous allons utiliser le tableur en ligne Google Sheets au lieu du choix plus courant Excel, car Google Sheets est plus accessible, et le langage de ses formules est plus facile à gérer (celui d'Excel dépend de la langue et des paramètres régionaux de votre système d'exploitation). Pour éviter la confusion dans le contexte des travaux notés, nous devons tout d'abord nous assurer que la langue des fonctions et des paramètres régionaux est l'anglais :



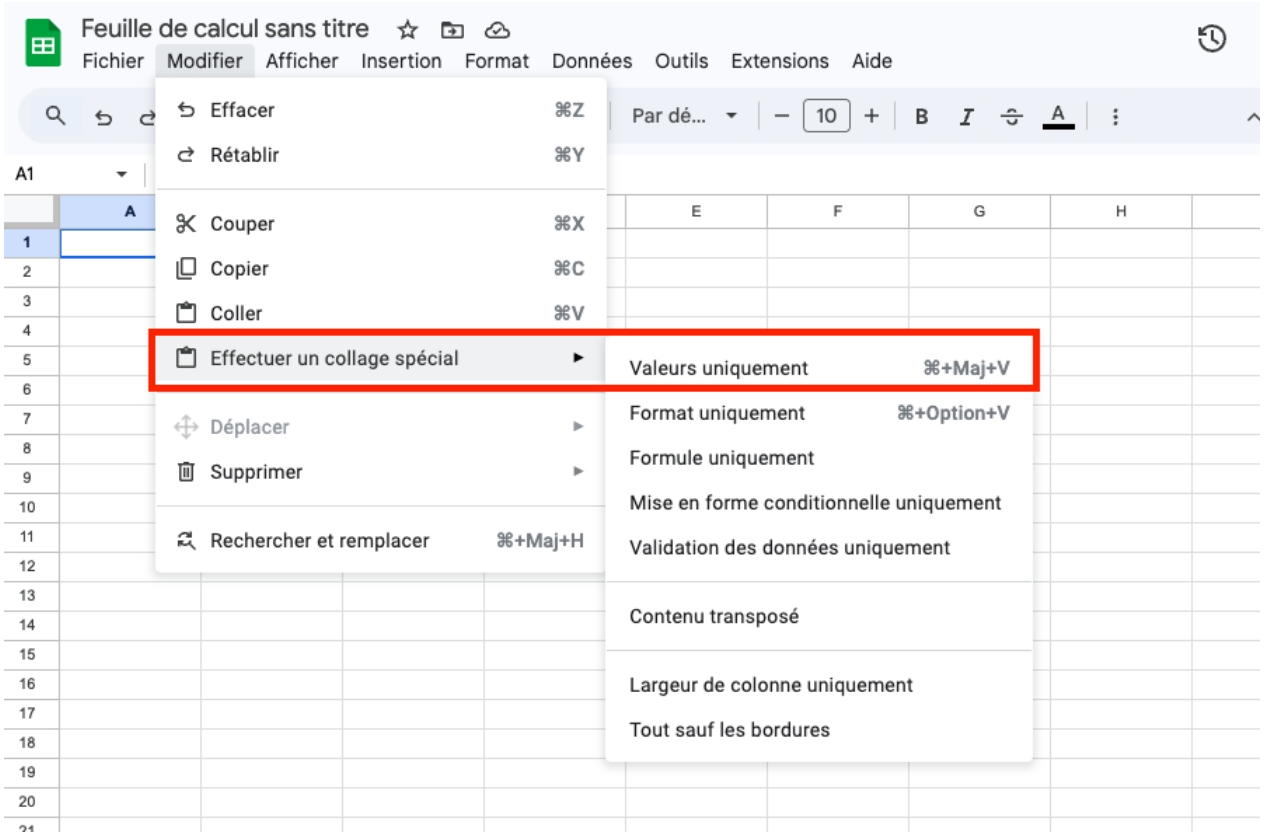


Assurez-vous ensuite que la “barre de formules” soit visible :



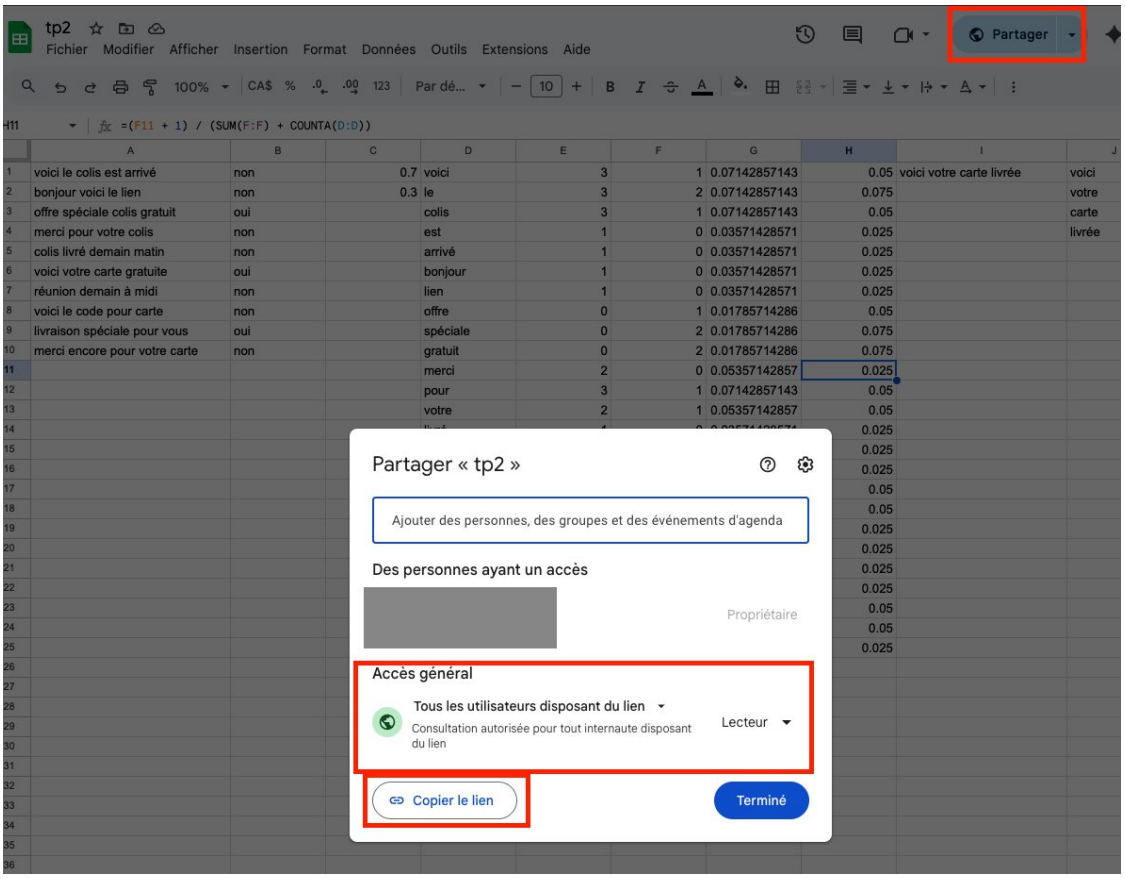
Fonction “copier-coller”

Pour les travaux notés 2 et 4, il est pratique d’utiliser la fonction “copier-coller” de votre système d’exploitation, pour éviter d’avoir à recopier une grande quantité d’information à la main. Si vous utilisez les raccourcis du clavier (`Control + c` pour copier sur Windows, `Command + c` sur Mac) veuillez noter qu’il est préférable d’utiliser cependant `Control + Shift + v` (`Command + Shift + v` sur Mac) pour coller (plutôt que `Control + v` ou `Command + v`), car ceci utilise la fonction “collage sans formatage”, qui fait en sorte de ne copier que l’information textuelle (en ignorant le formatage de la donnée source). Si vous ne désirez pas utiliser les raccourcis clavier et que vous utilisez un navigateur basé sur Chrome, veuillez noter qu’il vous sera nécessaire d’installer l’extension Google Docs hors connexion pour faire en sorte que vous puissiez utiliser la fonction “copier-coller” à partir du menu de Google Sheets.



Fonction de partage anonyme d’un fichier

Il est possible de partager de manière anonyme un fichier Google Sheets (qui ne sera donc pas associé à votre compte Google d’aucune manière) en produisant un hyperlien, en utilisant la fonction “partage” et en cliquant sur le bouton “copier le lien”, après s’être assuré que l’accès est en mode “tous les utilisateurs disposant du lien” :


tp2
Fichier Modifier Afficher Insertion Format Données Outils Extensions Aide
Partager
=(F11 + 1) / (SUM(F:F) + COUNTA(D:D))
Partager « tp2 »
Ajouter des personnes, des groupes et des événements d'agenda
Des personnes ayant un accès
Propriétaire
Accès général
Tous les utilisateurs disposant du lien
Consultation autorisée pour tout internaute disposant du lien
Lecteur
Copier le lien
Terminé

L'usage d'assistants intelligents

Étant donné qu'il s'agit d'un cours sur l'intelligence artificielle, l'usage d'assistants intelligents (ChatGPT, Claude, Gemini, etc) est évidemment encouragé ! Ceci s'applique autant aux travaux notés qu'au reste du contenu du cours.

Par contre si vous utilisez un tel outil, nous vous conseillons de vous assurer d'avoir une compréhension viscérale des réponses qu'il fournira, et de ne pas hésiter à engager un dialogue avec l'agent, dans le but de clarifier des aspects du problème qui pourraient rester obscurs pour vous. N'oubliez pas que le but de ce cours est la construction de bons modèles mentaux pour penser à l'intelligence artificielle, et comprendre son fonctionnement, au-delà de sa surface applicative. L'agent possède déjà ces modèles mentaux, si vous avez développé les vôtres à la fin de ce cours, le but aura clairement été atteint !

Voici un exemple d'interaction : supposons que j'ai de la difficulté à comprendre un aspect des mathématiques de la régression logistique, je peux faire une capture d'écran du texte et la donner à ChatGPT, et poser ma question ("je ne comprends pas pourquoi les deux dernières équations sont équivalentes").. ChatGPT (ou un autre outil) se fera un plaisir de vous expliquer avec une patience infinie :

Cette notation classique en apprentissage automatique utilise donc $\mathbf{x}$ pour dénoter les points en 2D sous forme vectorielle (x_1 et x_2 correspondent aux x et y de la représentation 2D classique). y est utilisé pour dénoter la *vraie* classe d'un point (0 ou 1, correspondant arbitrairement à `bleu` ou `rouge`). Les paramètres seront représentés par le vecteur $\mathbf{w} = [w_1, w2]$. Il est maintenant possible de réécrire notre fonction de décision à l'aide de cette nouvelle notation vectorielle :

$$z = \mathbf{w}^\top \mathbf{x} + b.$$

($\mathbf{w}^\top \mathbf{x}$ est le produit vectoriel de $\mathbf{w}$ et $\mathbf{x}$). Notons tout d'abord qu'il y maintenant 3 paramètres ($w1$, $w2$ et b), alors que dans l'exemple ci-haut seulement 2 sont mentionnés : m et b. On introduit aussi une nouvelle variable z : que veut-elle dire? Pour comprendre cela, on doit faire un peu d'algèbre. Il suffit de noter que notre équation de départ :

$$y = mx + b$$

est en fait équivalente à :

$$x_2 = mx_1 + b$$

je ne comprends pourquoi les deux dernières équations sont équivalentes

Excellente question ! Pour comprendre pourquoi les deux équations suivantes :

1. $y = mx + b$
2. $x_2 = mx_1 + b$

sont **équivalentes**, il faut savoir que :

- L'équation $y = mx + b$ est la forme classique d'une droite en 2D (le plan cartésien).
- En apprentissage automatique, on note souvent les coordonnées d'un point non pas avec x et y, mais avec un **vecteur** $\mathbf{x} = [x_1, x_2]$, où x_1 est l'abscisse et x_2 l'ordonnée.

Donc :

- x_1 joue le rôle de x,
- x_2 joue le rôle de y.

👉 Autrement dit :
$y = mx + b$ devient $x_2 = mx_1 + b$
si on adopte la notation vectorielle $\mathbf{x} = [x_1, x_2]$.

ChatGPT

Venez discuter d'IA sur Discord !

On tente une expérience : si ça vous tente, venez discuter des idées de ce cours sur la plateforme en ligne Discord, à l'adresse suivante : https://discord.gg/QxrtkpgtRS

Venez poser des questions et discuter de questions philosophiques avec au moins un des professeurs de ce cours !

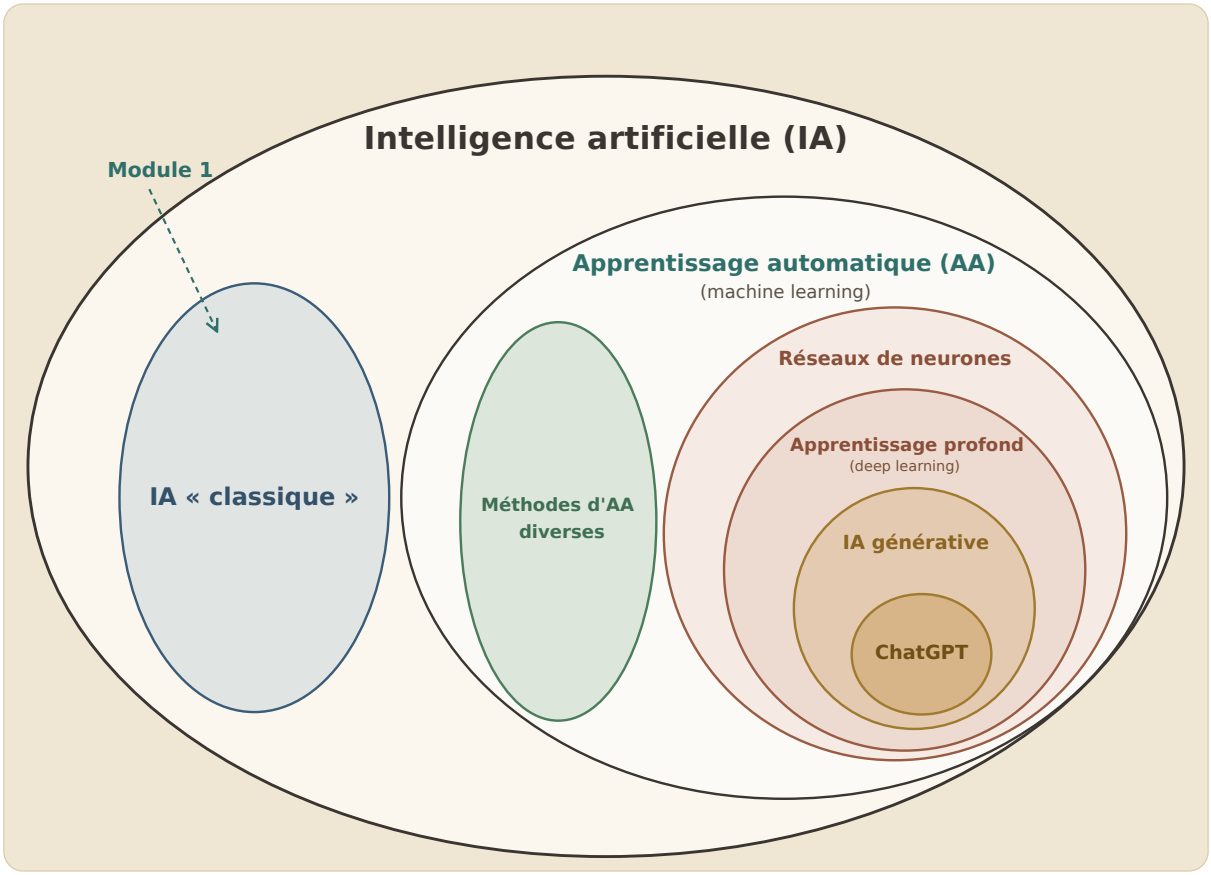
Contenu interactif, à consulter en ligne : https://discord.com/widget?id=1448801083991396450&theme=dark

Module 1 - Aux origines : l’intelligence artificielle symbolique

Avant les réseaux de neurones et avant ChatGPT, l'IA a d'abord cherché à **fabriquer une machine qui pense en manipulant des symboles et des règles**, comme le ferait un logicien ou un joueur d'échecs méthodique. On appelle aujourd'hui cette approche l'**IA symbolique**, ou *GOFAI* (pour *Good Old-Fashioned AI*, la « bonne vieille IA »). Pendant près de quarante ans, elle a dominé le domaine et obtenu des succès importants, puis elle a rencontré des limites qui l'ont fait paraître démodée.

Ce module présente cette histoire en **six chapitres**. Chacun associe un *moment historique* à une *idée technique*. Le but n'est pas de vous rendre capable de programmer ces systèmes, mais de comprendre l'**idée** sur laquelle ils reposent et les **difficultés** qu'ils ont rencontrées. En comprenant pourquoi cette première IA a atteint ses limites, on comprend mieux pourquoi la suite (les modules 2, 3 et 4) est si différente.

Dans l'ensemble du domaine de l'intelligence artificielle, l'IA symbolique correspond à l'**IA « classique »**, qui se distingue de l'apprentissage automatique que le cours étudiera ensuite.



La carte de l'IA : le Module 1 porte sur l'IA « classique » (symbolique), antérieure à l'apprentissage automatique.

Une idée traverse tout le module. Dès les années 1950, **deux grandes hypothèses rivales** sur la nature de l'intelligence apparaissent presque en même temps. L'une considère l'esprit comme de la **logique** (manipuler des symboles, appliquer des règles). L'autre le considère comme un **cerveau** (un réseau qui apprend de ses expériences). Le Module 1 présente cette opposition, et le Module 3 montrera comment elle a évolué.

Le parcours du module

Le module suit l'ordre chronologique. Chaque chapitre présente une idée, puis la limite qui mène au chapitre suivant.

1. *Turing et la question fondatrice (1950)* : une machine peut-elle penser, le jeu de l'imitation, et la pensée vue comme un calcul.
2. *Deux paris rivaux (1956-1958)* : la naissance de l'IA à Dartmouth, l'esprit comme logique et l'esprit comme cerveau.
3. *L'âge d'or symbolique : chercher et raisonner* : résoudre un problème en explorant un espace d'états, l'explosion combinatoire, Deep Blue et ELIZA.
4. *Représenter le monde* : donner un savoir à la machine, SHRDLU et son micro-monde, le problème du sens commun.
5. *Capturer l'expertise : les systèmes experts* : les règles *si… alors…*, le moteur d'inférence, MYCIN, et le goulot d'étranglement de la connaissance.

6. *Les hivers et la bascule* : les deux périodes de recul de l'IA, ce qui reste du GOFAI aujourd'hui, et le passage à l'apprentissage.

Le travail noté 1 clôt le module.

Objectifs

À la fin de ce module, vous devriez être en mesure de :

- Expliquer ce qu'est l'IA symbolique et en quoi elle se distingue de l'apprentissage automatique étudié dans les modules 2 à 4;
- Situer les grandes étapes et les figures marquantes de l'histoire de l'IA, du test de Turing (1950) aux systèmes experts et aux « hivers » de l'IA;
- Décrire les idées algorithmiques centrales du GOFAI (recherche dans un espace d'états, règles, représentation des connaissances) à un niveau intuitif;
- Distinguer les deux hypothèses rivales sur la nature de l'intelligence, l'approche symbolique et l'approche connexionniste, et situer chacune dans l'histoire du domaine;
- Comprendre pourquoi cette approche a fini par atteindre ses limites (problème du sens commun, goulot d'étranglement de la connaissance), ce qui a ouvert la voie à l'apprentissage automatique;
- Reconnaître ce qui reste de l'IA symbolique dans l'informatique d'aujourd'hui (recherche de chemins, moteurs de règles, représentation des connaissances).

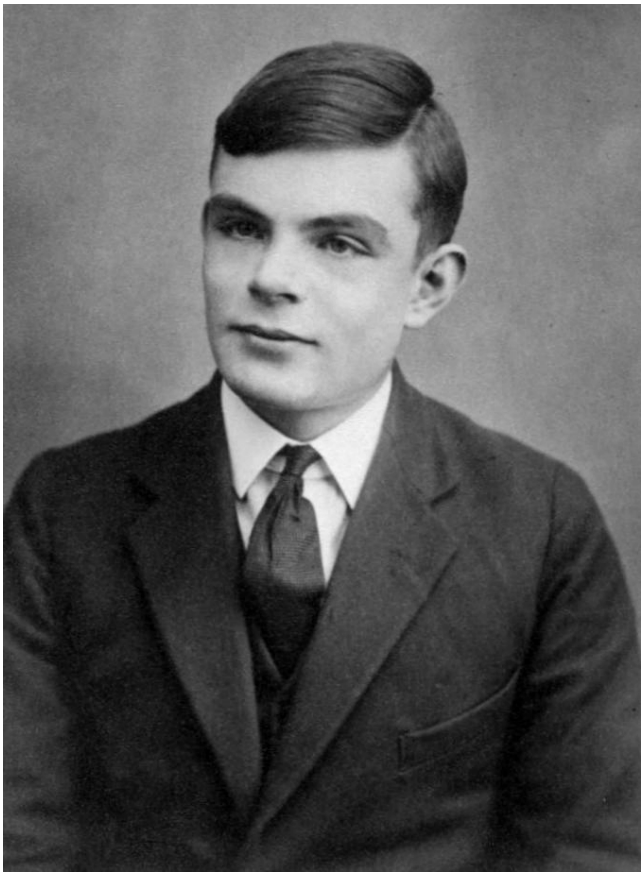
Durée

Deux semaines, soit environ 18 heures.

Évaluation

Le travail noté 1, qui compte pour 15 % de l'évaluation finale, vous fait jouer le rôle d'un **système expert**. À partir d'une base de règles `si… alors…`, vous suivez le raisonnement de la machine sur des cas concrets, vous trouvez un cas qu'elle ne sait pas traiter, puis vous ajoutez une règle pour combler ce manque. Ce travail vous permet de constater par vous-même les forces et la fragilité de l'IA symbolique.

Turing et la question fondatrice (1950)



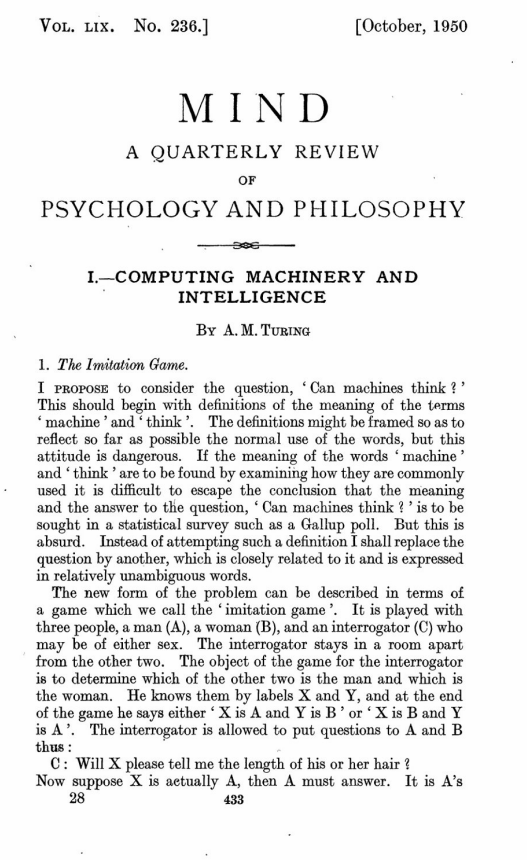
Alan Turing adolescent, vers 1928.

Alan Turing adolescent, vers 1928. Domaine public — Wikimedia Commons.

Le moment

En 1950, peu après la fin de la Seconde Guerre mondiale, le mathématicien britannique **Alan Turing** publie un article intitulé *Computing Machinery and Intelligence*. L'article commence par une question qui occupera tout un domaine pendant les décennies suivantes :

> *« Les machines peuvent-elles penser ? »*



VOL. LIX. No. 236.] [October, 1950

MIND

A QUARTERLY REVIEW

OF

PSYCHOLOGY AND PHILOSOPHY

I.—COMPUTING MACHINERY AND INTELLIGENCE

BY A. M. TURING

1. *The Imitation Game.*

I PROPOSE to consider the question, 'Can machines think?' This should begin with definitions of the meaning of the terms 'machine' and 'think'. The definitions might be framed so as to reflect so far as possible the normal use of the words, but this attitude is dangerous. If the meaning of the words 'machine' and 'think' are to be found by examining how they are commonly used it is difficult to escape the conclusion that the meaning and the answer to the question, 'Can machines think?' is to be sought in a statistical survey such as a Gallup poll. But this is absurd. Instead of attempting such a definition I shall replace the question by another, which is closely related to it and is expressed in relatively unambiguous words.

The new form of the problem can be described in terms of a game which we call the 'imitation game'. It is played with three people, a man (A), a woman (B), and an interrogator (C) who may be of either sex. The interrogator stays in a room apart from the other two. The object of the game for the interrogator is to determine which of the other two is the man and which is the woman. He knows them by labels X and Y, and at the end of the game he says either 'X is A and Y is B' or 'X is B and Y is A'. The interrogator is allowed to put questions to A and B thus:

C: Will X please tell me the length of his or her hair?

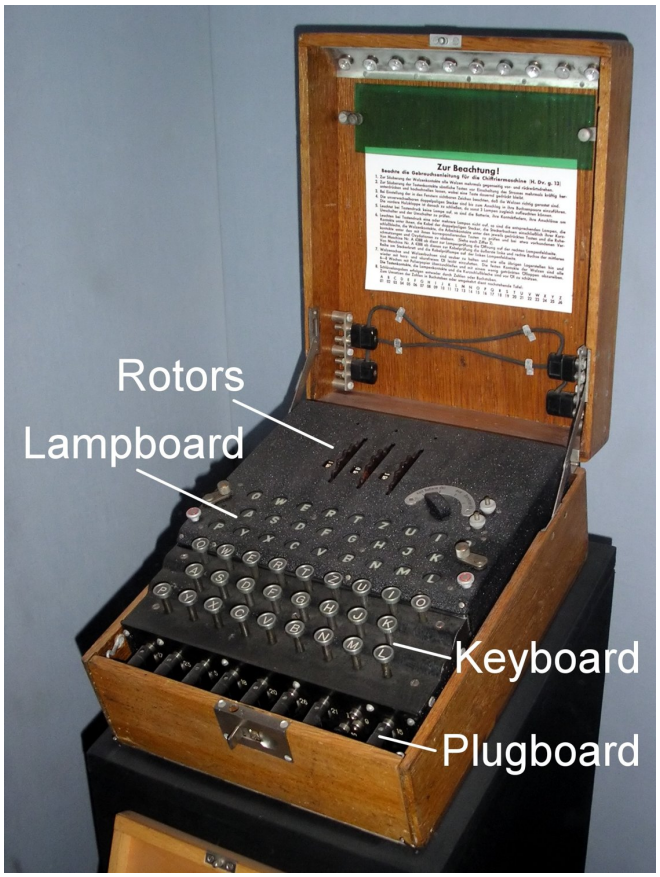
Now suppose X is actually A, then A must answer. It is A's

28 433

La première page de l'article, dans la revue Mind, en octobre 1950. L'article commence par la question « Can machines think? ».

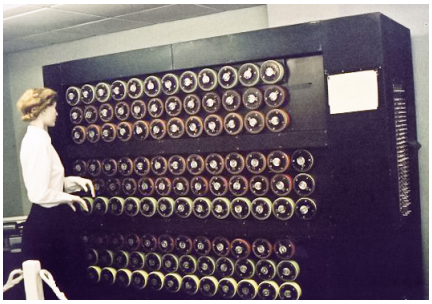
La première page de « Computing Machinery and Intelligence », *Mind*, vol. LIX, nº 236, octobre 1950.

Turing est déjà connu à ce moment. Pendant la guerre, il a joué un rôle décisif dans le décryptage des communications chiffrées par la machine **Enigma** des armées allemandes. Il a conçu des dispositifs électromécaniques capables d'explorer mécaniquement des millions de combinaisons. Ce travail est longtemps resté secret, et les historiens estiment qu'il a pu écourter le conflit de plusieurs années. Turing sait donc mieux que quiconque à son époque qu'une machine peut accomplir des tâches qu'on croyait réservées à l'esprit humain. Son article examine si une machine qui déchiffre des messages peut aussi être considérée comme une machine qui pense.



Une machine Enigma.

Une machine Enigma. Photo : Karsten Sperling, domaine public.



Une « bombe » de décryptage, Bletchley Park.

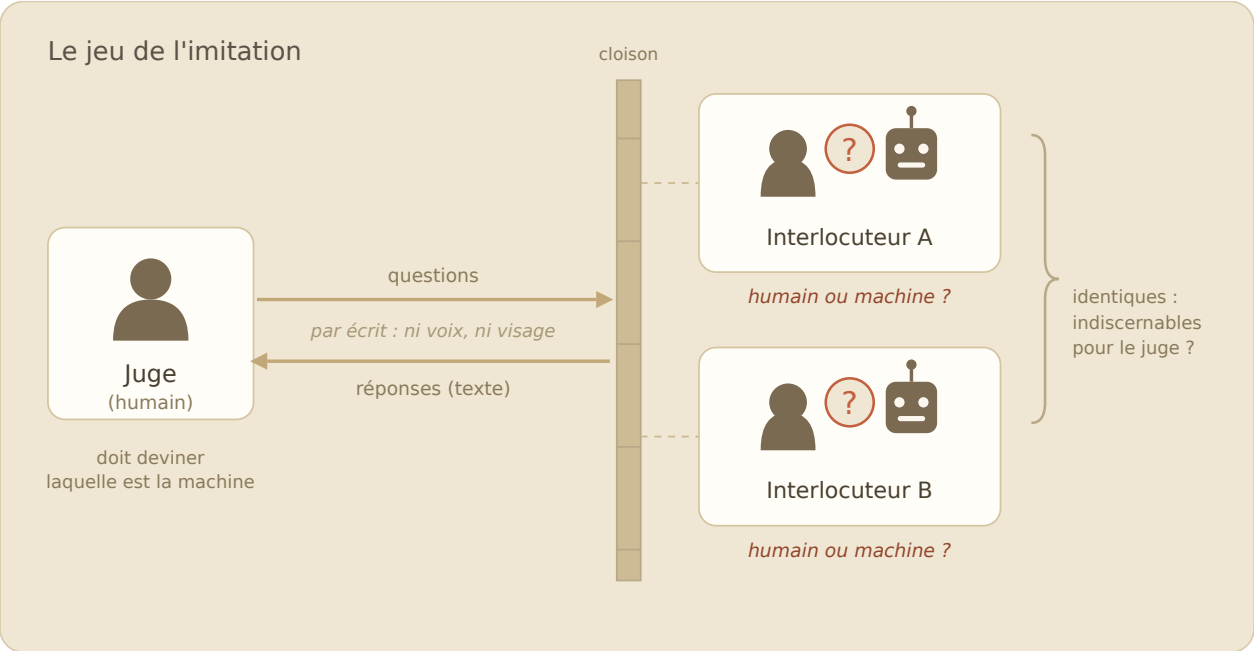
Reconstitution d'une « bombe » de décryptage, Bletchley Park. Photo : Sarah Hartwell, CC BY-SA 3.0, via Wikimedia Commons.

Le jeu de l'imitation

Turing remarque que la question « les machines peuvent-elles penser ? » est mal posée. La réponse dépend entièrement de ce qu'on accepte d'appeler « penser », et on ne sait pas définir clairement ce mot, même pour les humains. Plutôt que d'entrer dans ce débat, Turing **remplace la question par une autre**, à laquelle on peut réellement répondre.

Au lieu de demander si une machine peut penser, il propose une épreuve concrète, le **jeu de l'imitation** (*imitation game*), qu'on appelle aujourd'hui le **test de Turing**. Un juge humain dialogue par écrit avec deux interlocuteurs cachés, un être humain et une machine. Les messages passent par clavier, pour que ni la voix ni le visage ne permettent de savoir qui est qui. Le juge peut poser n'importe quelle question, par exemple sur la météo, la poésie, l'arithmétique ou ses souvenirs d'enfance. Il doit deviner lequel des deux interlocuteurs est la machine. Selon Turing, si la machine parvient, sur un grand nombre de parties, à se faire passer pour un humain aussi souvent qu'un humain y parvient, il n'y a plus de raison sérieuse de refuser de dire qu'elle pense.

Dans l'article, le jeu est d'abord présenté avec un homme et une femme, que l'interrogateur doit distinguer. Turing demande ensuite ce qui arriverait si une machine prenait la place de l'un des deux. C'est cette seconde version qui a été retenue par la suite, et vous pouvez le vérifier sur la page de *Mind* reproduite plus haut.



Le jeu de l'imitation : le juge ne dispose que du texte échangé pour reconnaître la machine.

Cette proposition est à la fois philosophique et technique. Turing **remplace la question de l'intériorité par celle du comportement**. Ce qui se passe « à l'intérieur » de la machine, ou le fait qu'elle « ressente » quelque chose, n'a pas d'importance. Ce qui compte, c'est ce qu'elle est capable de faire. Ce changement de point de vue soulève une question de fond, que les philosophes débattent encore aujourd'hui :

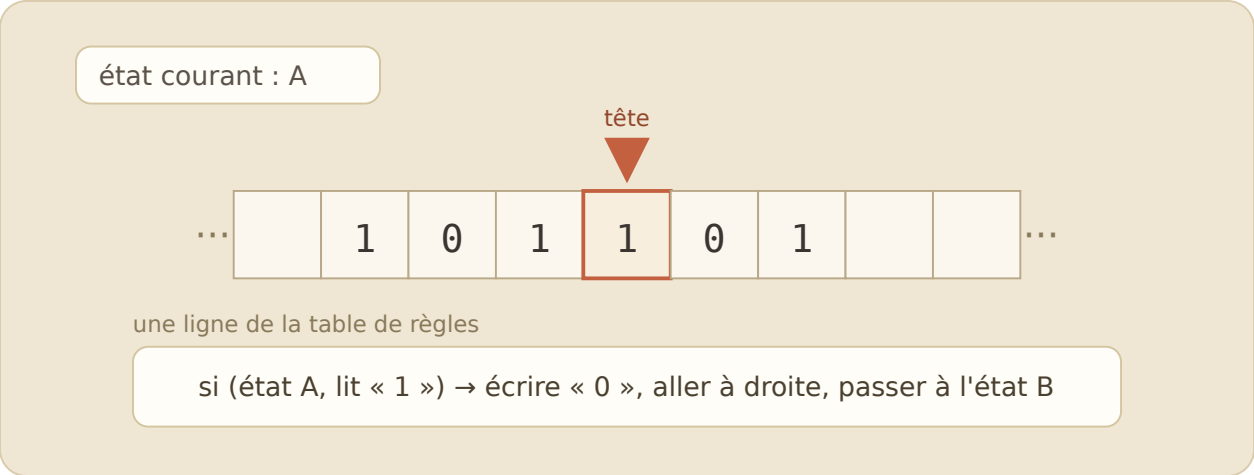
Qu'est-ce qui ferait qu'une machine « pense » vraiment ?
La proposition de Turing correspond aux idées dominantes de 1950, en particulier au **behaviorisme**. Ce courant de la psychologie soutenait qu'on ne peut pas faire de science avec ce qui se passe « dans la tête » (invisible et invérifiable), mais seulement avec le **comportement observable**. Juger la machine sur ses réponses plutôt que sur sa vie intérieure est conforme à cette approche. Une autre thèse, plus précise, allait cependant donner à l'IA son principal

fondement philosophique, le **fonctionnalisme**. Selon cette thèse, un état mental n'est pas défini par la *matière* qui le produit, mais par le *rôle* qu'il joue, c'est-à-dire par ses causes et ses effets. La douleur, par exemple, est « ce qui est causé par une blessure et qui cause le retrait et la plainte », qu'elle soit réalisée par des neurones humains, par le système nerveux d'un poulpe ou par les circuits d'un ordinateur. De ce point de vue, la frontière entre l'esprit et la machine est moins nette dans les deux sens. Un ordinateur pourrait penser, et **notre propre cerveau peut être vu comme une sorte d'« ordinateur biologique »**, un support matériel qui réalise lui aussi une certaine organisation. Si l'esprit est une *organisation* et non une *substance*, rien n'empêche qu'il soit réalisé sur du silicium aussi bien que sur des neurones. C'est ce qui permet de considérer l'IA comme un projet sérieux. Nous reviendrons sur cette idée, et sur sa critique la plus célèbre, la *Chambre chinoise* (*Chinese room*) du philosophe John Searle, au Module 5.

Le test de Turing est souvent mal compris. Ce n'est **pas** un test de conscience, ni une définition de l'intelligence au sens large. C'est une proposition *pragmatique* : *si* on ne peut plus distinguer la machine de l'humain en conversation, *alors* chercher à savoir si elle « pense vraiment » devient un débat sans issue. Le test contourne une question insoluble, il ne la résout pas.

La pensée comme calcul

Le test repose sur une idée plus fondamentale, qui constitue le principal apport de Turing à l'IA. Dès 1936, bien avant son article sur le jeu de l'imitation, Turing avait inventé un objet mathématique qu'on appelle aujourd'hui la **machine de Turing**. C'est un dispositif imaginaire très simple, qui se contente de lire et d'écrire des symboles sur un ruban en suivant un ensemble de règles fixes.



Une machine de Turing : lire un symbole, l'écrire, se déplacer, changer d'état.

Turing montre que ce mécanisme élémentaire peut, en principe, **calculer tout ce qui est calculable**. La machine de Turing donne ainsi une définition précise d'une notion qu'on emploie couramment sans la définir, celle d'**algorithme**. Un algorithme est une suite finie d'étapes sans ambiguïté. On admet que tout ce qu'un algorithme peut accomplir, une machine de Turing peut l'accomplir aussi, et réciproquement. Les expressions « algorithme », « machine de Turing » et « ce qui est calculable » désignent donc la même chose. De plus, il existe une *machine universelle*, capable d'imiter

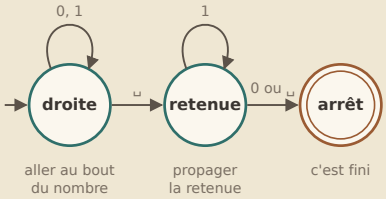
n'importe quelle autre machine à partir de sa description, qu'elle reçoit en entrée. C'est le principe de l'ordinateur moderne : un seul appareil physique qui, selon le programme qu'on lui fournit, sert de traitement de texte, de jeu vidéo ou de calculatrice. Votre téléphone est, en ce sens, une machine universelle de Turing.

Turing laisse ainsi aux fondateurs de l'IA un **pari implicite** : *si* penser n'est qu'une forme de calcul (manipuler des symboles selon des règles), *et si* une machine peut tout calculer, *alors* une machine devrait pouvoir penser. Tout le programme de recherche des décennies suivantes, que nous suivrons dans ce module, découle de ce raisonnement. Turing laisse cependant ouverte une question qui sera centrale dans la suite : penser se réduit-il vraiment à calculer ?

Pour aller plus loin : qu'est-ce qu'une machine de Turing, concrètement ?
Une machine de Turing comprend un ruban de papier infini divisé en cases, une tête qui se déplace le long du ruban en lisant et en écrivant un symbole à la fois, et une petite table de règles de la forme « si je suis dans l'état A et que je lis un `0`, alors j'écris un `1`, je me déplace d'une case vers la droite et je passe à l'état B ». Elle ne comprend rien d'autre, ni grande mémoire ni intelligence cachée, seulement des symboles, des règles et un état courant. Ce mécanisme minimal suffit pourtant à exprimer n'importe quel calcul réalisable par n'importe quel ordinateur, aussi puissant soit-il. Voici un exemple complet. La machine ci-dessous calcule une fonction précise, **ajouter 1 à un nombre écrit en binaire**, avec trois états et six règles. Dans l'état *droite*, elle avance sans rien changer jusqu'à dépasser le dernier chiffre. Elle revient ensuite vers la gauche dans l'état *retenue* et fait la même chose que vous lorsque vous posez une addition. Tant qu'elle lit un 1, elle écrit 0 et reporte la retenue à gauche. Dès qu'elle lit un 0 (ou une case vide, si le nombre n'était composé que de 1), elle écrit 1 et s'arrête. Suivez l'exécution ligne par ligne : onze, 1011, devient douze, 1100, en huit pas.

Une machine complète : la fonction n ↦ n + 1, en binaire

Ses trois états



Sa table de règles

État	Lit	Écrit	Va à	Passe à
droite	0	0	droite →	droite
droite	1	1	droite →	droite
droite	␣	␣	← gauche	retenue
retenue	1	0	← gauche	retenue
retenue	0	1	← gauche	arrêt
retenue	␣	1	← gauche	arrêt

Son exécution sur le ruban : 1011 (onze) devient 1100 (douze)

étape 0	**droite**	␣	**1**	0	1	1	␣	␣	lit 1, écrit 1, va à droite
étape 1	**droite**	␣	1	**0**	1	1	␣	␣	lit 0, écrit 0, va à droite
étape 2	**droite**	␣	1	0	**1**	1	␣	␣	lit 1, écrit 1, va à droite
étape 3	**droite**	␣	1	0	1	**1**	␣	␣	lit 1, écrit 1, va à droite
étape 4	**droite**	␣	1	0	1	1	**␣**	␣	lit ␣, écrit ␣, va à gauche
étape 5	**retenue**	␣	1	0	1	**1**	␣	␣	lit 1, écrit 0, va à gauche
étape 6	**retenue**	␣	1	0	**1**	0	␣	␣	lit 1, écrit 0, va à gauche
étape 7	**retenue**	␣	1	**0**	0	0	␣	␣	lit 0, écrit 1, va à gauche
étape 8	**arrêt**	␣	**1**	1	0	0	␣	␣	**résultat : 1100**

☐ la case sous la tête de lecture

Une machine de Turing complète : trois états, six règles, et la fonction $n \mapsto n + 1$ exécutée pas à pas sur le ruban.

La machine ne sait pas qu'il s'agit de nombres et ne connaît ni « onze » ni « douze ». Elle lit un symbole, consulte sa table, écrit, se déplace et change d'état. Le sens existe pour nous, mais pour elle il n'y a que la règle. Des machines un peu plus grandes additionnent, multiplient ou trient. On peut aussi en construire une, dite *universelle*, qui lit sur son ruban la table de n'importe quelle autre machine et l'exécute. C'est exactement l'idée d'un ordinateur et de son programme.

Une fois qu'on connaît une machine à trois états, on peut se poser une question d'apparence simple : parmi toutes les machines à n états qui finissent par s'arrêter, laquelle effectue le plus de pas avant de s'arrêter ? On appelle cette machine le **castor affairé** (*busy beaver*), et le nombre de pas qu'elle effectue est une fonction de n. Les premières valeurs sont petites : 6 pas pour deux états, 21 pour trois, 107 pour quatre. Pour cinq états, la réponse, 47 176 870 pas, n'a été établie qu'en 2024 par une collaboration internationale d'amateurs, le *Busy Beaver Challenge*. Ce groupe a dû examiner une à une des dizaines de millions de machines et faire vérifier sa preuve par ordinateur. Pour six états, on sait seulement que le nombre dépasse tout ce qu'on peut raisonnablement écrire. Au-delà, la difficulté devient une impossibilité, car cette fonction est **non calculable** (*uncomputable*). Aucun algorithme ne peut la donner pour tout n, parce que la connaître permettrait de savoir, pour n'importe quelle machine, si elle s'arrêtera un jour. Il suffirait de laisser tourner la machine pendant le nombre de pas du castor affairé. Si elle tourne encore après ce nombre de pas, elle ne s'arrêtera jamais. Il s'agit en fait du problème de l'arrêt, présenté plus bas, sous une autre forme. On a donc une suite de nombres parfaitement définis,

dont les premiers sont petits, mais dont on ne connaîtra jamais que les tout premiers termes. Elle montre concrètement où se trouve une limite du calcul.
La machine de Turing ne sert pas à calculer rapidement. Elle sert à définir précisément ce que « calculer » veut dire. L'idée que tout procédé de calcul imaginable se ramène à une machine de Turing porte d'ailleurs un nom, la **thèse de Church-Turing**.

L'ombre de Gödel

L'idée que penser revient à calculer comportait pourtant une **limite**, établie presque au même moment. En **1931**, cinq ans avant la machine de Turing, un jeune logicien autrichien, **Kurt Gödel**, avait démontré un résultat qui a profondément remis en question les fondements des mathématiques, son **théorème d'incomplétude** (*incompleteness theorem*).

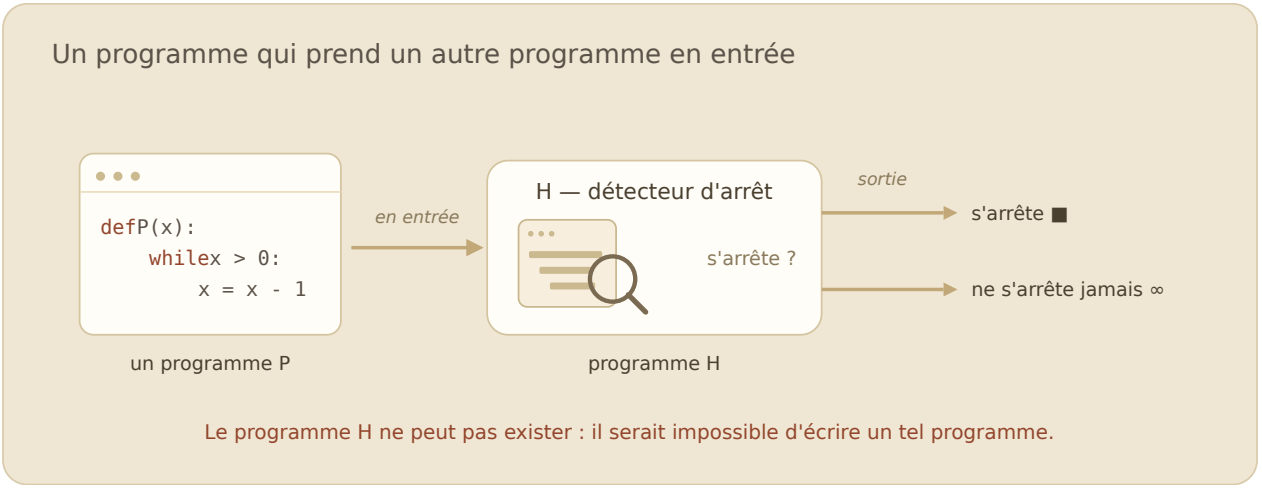
Kurt Gödel étudiant, vers 1925.

Kurt Gödel étudiant, vers 1925. Domaine public — Wikimedia Commons.

Le résultat est simple à énoncer. Tout système de règles formelles assez puissant pour faire de l'arithmétique contient des énoncés qui sont **vrais**, mais que le système est **incapable de prouver**. La vérité dépasse donc toujours la démonstration, et aucun ensemble de règles, aussi complet soit-il, ne peut tout capturer. La preuve repose sur une idée qui sera importante pour nous, l'**auto-référence** (*self-reference*). Gödel construit, à l'intérieur du système, un énoncé qui parle de lui-même et qui affirme en substance : « *Cet énoncé n'est pas démontrable.* » Si le système démontrait cet énoncé, il

prouverait quelque chose de faux. Il ne peut donc pas le démontrer, et c'est précisément pour cette raison que l'énoncé est vrai. Le système ne peut pas sortir de cette boucle.

Turing a lui-même prolongé ce résultat. Sa machine de 1936 servait aussi à montrer qu'il existe des questions qu'**aucun algorithme ne pourra jamais trancher**. La plus connue est le **problème de l'arrêt** (*halting problem*) : déterminer, pour un programme donné, s'il finira par s'arrêter ou s'il tournera à l'infini. Celui qui a proposé l'idée « penser = calculer » a donc aussi établi les **frontières** du calcul. Le projet de l'IA comporte ainsi, dès son origine, la reconnaissance de ses propres limites.



Le problème de l'arrêt (halting problem) : un programme qui prendrait un autre programme en entrée pour prédire s'il s'arrête, et pourquoi il ne peut pas exister.

Pour la question de savoir si une machine peut penser, on a tiré de Gödel **deux conclusions opposées**. Toutes deux reviendront jusqu'à la fin du cours.

Première lecture : le théorème est une limite. Le physicien **Roger Penrose** en a tiré l'argument le plus connu contre l'IA. Selon lui, nous, humains, sommes capables de voir qu'un tel énoncé est vrai, alors que la machine ne le peut pas. Si notre esprit saisit une vérité qu'aucun système formel ne peut prouver, c'est qu'il ne se réduit **pas** à un système formel, autrement dit que penser ne se ramène pas à du calcul. Nous reviendrons en détail sur cette thèse au **Module 5**, avec celle d'un autre critique connu de l'IA, le philosophe John Searle.

Seconde lecture : le théorème est une source. Pour le penseur américain **Douglas Hofstadter** (auteur du livre *Gödel, Escher, Bach* (1979), où il développe précisément cette idée), la boucle auto-référentielle de Gödel n'est pas un défaut, mais le fondement même de l'esprit. Un système assez riche pour se **représenter lui-même** produit une « **boucle étrange** » (*strange loop*). Selon Hofstadter, c'est de ce retour sur soi que naîtraient le sentiment d'un « je », le sens et la conscience. L'auto-référence n'empêcherait donc pas la machine de penser, elle en serait la **source**. Nous retrouverons Hofstadter et ses idées à plusieurs reprises dans ce module.

Le même théorème conduit donc à deux conclusions inverses : l'auto-référence est soit une *limite* infranchissable, soit l'*origine* de l'esprit. Ce désaccord n'est toujours pas tranché. Il faut surtout retenir que, dès son origine, le projet d'une pensée mécanique s'accompagne de questions sur ses limites.

Pour aller plus loin : ce que dit, et ne dit pas, le théorème de Gödel
Le théorème s'applique à tout système formel **cohérent** (sans contradiction) et assez riche pour exprimer l'arithmétique. Gödel établit en fait deux résultats. Un tel système ne peut pas prouver tous les énoncés arithmétiques vrais (**premier théorème**), et il ne peut pas non plus prouver sa **propre cohérence** (**second théorème**). L'outil technique central est le *codage de Gödel* (*Gödel numbering*), qui consiste à numéroter chaque énoncé pour que le système puisse, en parlant de nombres, **parler de lui-même**. Gödel peut ainsi construire la phrase « je ne suis pas démontrable ».
Cette phrase est une variante du **paradoxe du menteur** (*liar paradox* : « cette phrase est fausse »), dont Gödel s'est explicitement inspiré. Le paradoxe du menteur aboutit à une contradiction. Gödel remplace « faux » par « indémontrable », si bien que sa phrase ne se contredit pas. Elle est simplement **vraie sans être démontrable**. Ce remplacement transforme un paradoxe sans issue en théorème.

Le paradoxe du menteur : une phrase qui porte sur elle-même et ne peut être ni vraie ni fausse.

Il faut éviter deux contresens. Le théorème ne dit **pas** que « les mathématiques sont fausses », ni que la vérité serait une affaire d'opinion. C'est une limite **précise** sur ce qu'une démonstration *formelle, à l'intérieur d'un système donné*, peut atteindre. On peut toujours prouver l'énoncé en question dans un système **plus puissant**, mais ce nouveau système aura à son tour ses propres énoncés indémontrables. Quant à l'argument de Penrose, il reste **contesté**. Ses critiques objectent que rien ne garantit qu'un humain puisse réellement « voir » la vérité de cet énoncé sans supposer d'avance, sans justification, la cohérence du système.

Une fin tragique, et une porte qui s'ouvre

La vie de Turing se termine tragiquement. En 1952, il est poursuivi par la justice britannique en raison de son homosexualité, alors illégale, et condamné à subir un traitement hormonal, une castration chimique. Écarté et privé de son habilitation de sécurité, il meurt en 1954, à 41 ans, empoisonné au cyanure, dans des circonstances qu'on a longtemps considérées comme un suicide. Il ne verra pas

les machines pensantes qu'il avait imaginées. Le gouvernement britannique présente des excuses officielles en **2009**, et Turing reçoit une grâce royale posthume en **2013**.

Turing a posé la question et proposé une orientation, *la pensée comme calcul*, mais il n'a pas précisé comment construire concrètement une machine intelligente. Cette tâche revient à la génération suivante. Dès le départ, celle-ci se divise entre **deux réponses rivales**, apparues presque en même temps, qui reposent sur deux conceptions très différentes de l'intelligence. C'est le sujet de « Deux paris rivaux ».

Deux paris rivaux (1956-1958)

Le moment : l'été 1956 à Dartmouth

À l'été 1956, un petit groupe de chercheurs se réunit pendant deux mois sur le campus du **Dartmouth College**, dans le New Hampshire, pour un atelier qui est considéré comme l'**acte de naissance** de la discipline. L'expression « **intelligence artificielle** » apparaît pour la première fois dans la proposition de financement de cet atelier, rédigée l'année précédente par le jeune mathématicien **John McCarthy**. Le mot est choisi en partie pour marquer une rupture avec une étiquette alors dominante, la **cybernétique**.

Dartmouth Hall, sur le campus du Dartmouth College où se tint l'atelier de 1956.

Dartmouth Hall (Dartmouth College). Photo : Kenneth C. Zirkel, CC BY 4.0, via Wikimedia Commons.

La cybernétique. Fondée en 1948 par le mathématicien Norbert Wiener (le mot vient du grec *kubernêtês*, le pilote d'un navire), la cybernétique étudiait ce que les machines et les êtres vivants ont en commun lorsqu'ils se **règlent eux-mêmes**. Un thermostat qui maintient une température, un organisme qui maintient la sienne et un missile qui corrige sa trajectoire obéissent au même principe, la **rétroaction** (*feedback*). L'effet d'une action est mesuré et renvoyé à celui qui agit, pour qu'il se corrige. Une bonne partie de ce qui suit dans ce cours vient de là, en particulier

l'idée d'une machine qui s'ajuste d'après ses erreurs. Cependant, la cybernétique raisonnait en signaux continus, en boucles et en organismes, à la frontière de la biologie et de l'ingénierie. McCarthy voulait une science du raisonnement et des symboles et, selon certains témoignages, ne tenait pas à devoir composer avec l'autorité de Wiener. Il a donc choisi un nom nouveau. *Le même mot grec a aussi donné son nom, en 2014, au logiciel Kubernetes, qui « pilote » des conteneurs dans les centres de données, d'où son logo en forme de barre de navire. Les deux n'ont en commun que l'étymologie.*

Devant Dartmouth Hall, à l'été 1956 : les organisateurs de l'atelier et quelques participants.

Devant Dartmouth Hall, été 1956. À l'arrière, de gauche à droite : Oliver Selfridge, Nathaniel Rochester, Marvin Minsky et John McCarthy ; à l'avant : Ray Solomonoff, Peter Milner et Claude Shannon. Photo : famille Minsky.

L'ambition affichée est très grande. La proposition postule que « *tout aspect de l'apprentissage, ou de toute autre caractéristique de l'intelligence, peut en principe être décrit avec une telle précision qu'une machine peut être construite pour le simuler* » (en version originale : « *every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it* »). Les organisateurs sont McCarthy, **Marvin Minsky**, **Claude Shannon** (le père de la théorie de l'information) et **Nathaniel Rochester**, d'IBM. Ils pensent sincèrement qu'un groupe d'une dizaine de personnes peut faire des progrès significatifs sur ce programme en un seul été.

Ce travail, qu'ils pensaient mener en quelques étés, a occupé plusieurs générations de chercheurs et n'est toujours pas terminé. L'atelier de Dartmouth a cependant lancé le domaine. Très rapidement, deux familles d'idées sur la façon de construire cette intelligence s'y dessinent.

Un mot qui a changé de sens. Si vous abordez ce cours en pensant à ChatGPT, il faut savoir que ce que McCarthy et ses collègues appelaient « intelligence artificielle » en 1956 a très peu de rapport avec ce que ce mot désigne aujourd'hui dans la conversation courante. Pour eux, l'IA consistait à faire **raisonner** une machine : manipuler des symboles, appliquer des règles de logique, explorer méthodiquement des possibilités, comme on le fait pour démontrer un théorème ou jouer aux échecs. Aucune partie de ce programme n'apprenait à partir de grandes quantités de données. Aujourd'hui, le mot « IA » désigne presque toujours les **grands modèles de langage** (*large language models*) et les modèles apparentés qui génèrent des images. Ce sont d'immenses réseaux de neurones (*neural networks*), entraînés sur une bonne partie de ce que l'humanité a écrit, qui ne contiennent aucune règle écrite à la main.
Ce module raconte ce renversement. L'expression a été créée par les partisans du premier pari, celui de la logique. Elle est aujourd'hui associée aux héritiers du second, celui du cerveau et de l'apprentissage, que les premiers ont longtemps considéré comme une impasse. Entre les deux, il y a soixante-dix ans, deux « hivers » (*AI winters*) et un renversement que rien ne laissait prévoir. Gardez donc les deux sens à l'esprit. Dans les pages qui suivent, « IA » a d'abord le sens qu'il avait à Dartmouth.

Le premier pari : l'esprit comme logique

La première famille d'idées prolonge directement l'intuition de Turing. Si penser consiste à calculer, alors **l'intelligence consiste à manipuler des symboles selon des règles logiques**. Un symbole est ici un jeton qui représente quelque chose (un mot, un objet, une idée). Raisonner consiste à combiner ces jetons d'après des règles précises, comme un mathématicien enchaîne les étapes d'une démonstration. Dans cette vision, il importe peu que la machine ressemble ou non à un cerveau. Ce qui compte, c'est qu'elle possède les bons symboles et les bonnes règles. On parle d'approche **descendante** (*top-down*), parce qu'on programme le raisonnement explicitement, « par le haut ».

Cette approche a rapidement donné des résultats concrets. Dès 1956, un programme nommé **Logic Theorist**, conçu par **Allen Newell** et **Herbert Simon** et présenté à Dartmouth, démontre des théorèmes de logique mathématique tirés d'un ouvrage de référence, les *Principia Mathematica* de Russell et Whitehead. Pour l'une de ces démonstrations, il trouve même une solution plus élégante que celle des auteurs. On le considère souvent comme le **premier programme d'intelligence artificielle**, c'est-à-dire une machine qui ne calcule pas des nombres, mais qui raisonne, du moins en apparence.

Soixante-dix ans plus tard. La démonstration de théorèmes par machine (*automated theorem proving*) a progressé d'une manière que Newell et Simon n'avaient pas prévue. En 2025, des grands modèles de langage entraînés par renforcement ont atteint le niveau d'une médaille d'or aux Olympiades internationales de mathématiques. En septembre 2026, l'un d'eux a traduit toute la démonstration du dernier théorème de Fermat dans **Lean**, un langage où chaque étape d'une preuve est vérifiée par la machine. Les spécialistes estimaient ce travail à plusieurs années. Un autre système a annoncé avoir résolu une version de l'un des sept problèmes du millénaire, celui des équations de Navier-Stokes, un résultat que les mathématiciens sont encore en train de vérifier. Ces machines raisonnent elles aussi, mais elles sont issues de l'autre pari, celui de l'apprentissage, et ne contiennent aucune règle de logique écrite à la main. Nous y reviendrons au Module 4.

Lean, un vérificateur de preuves. Le langage **Lean**, dans lequel la démonstration de Fermat a été traduite, a été créé en 2013 par Leonardo de Moura chez Microsoft Research. Dans Lean, chaque étape d'une démonstration doit être justifiée par une règle de logique précise. Un petit programme central, le *noyau*, vérifie ensuite que chaque étape respecte ces règles. Si le noyau accepte la preuve, le théorème est établi avec une certitude qu'aucune relecture humaine ne peut garantir pour des démonstrations de centaines de pages. Par exemple, l'énoncé suivant est vérifié par Lean :

```
theorem deux_plus_deux : 2 + 2 = 4 := by norm_num
```

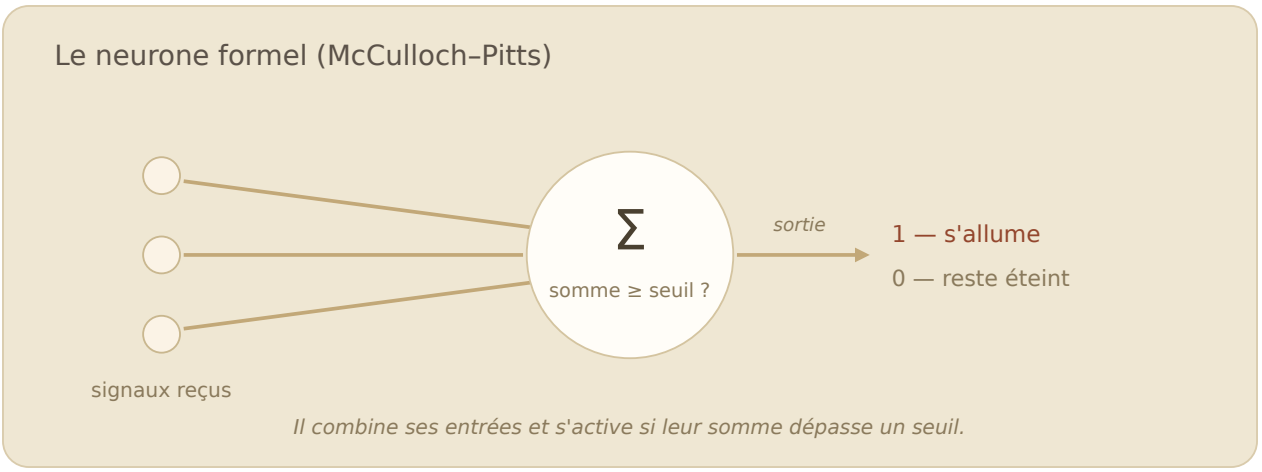
Des mathématiciens du monde entier alimentent une bibliothèque commune, **Mathlib**, qui contient déjà une grande partie des mathématiques enseignées à l'université.
Lean appartient à la tradition symbolique présentée dans ce chapitre : des règles explicites, appliquées mécaniquement. Il est aujourd'hui au centre d'une combinaison des deux paris. Un modèle de langage, issu de l'apprentissage, propose des étapes de démonstration, et Lean vérifie qu'elles sont correctes. Le modèle peut se tromper, mais une erreur ne passe pas la vérification. C'est ainsi qu'ont été obtenus plusieurs des résultats mentionnés ci-dessus. Pour en savoir plus : lean-lang.org.

Ces succès suscitent un grand enthousiasme. Newell et Simon formulent une hypothèse forte : un système qui manipule des symboles de la bonne manière posséderait tout ce qu'il faut pour être intelligent. Pendant les décennies suivantes, cette voie **symbolique** domine la recherche et obtient l'essentiel des financements. Elle fait l'objet des chapitres suivants : la recherche, la représentation des connaissances (*knowledge representation*), les systèmes experts (*expert systems*).

Le second pari : l'esprit comme cerveau

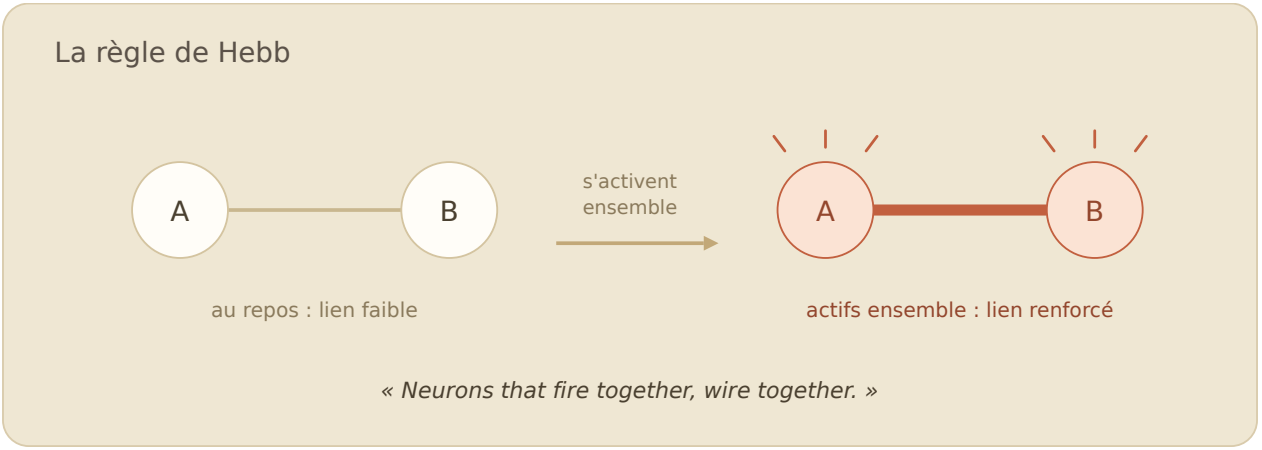
À la même époque, une idée très différente apparaît. Au lieu de programmer le raisonnement explicitement, on pourrait construire une machine qui **apprend seule, à partir d'exemples**, en s'inspirant de l'organe qui est déjà intelligent, le **cerveau**.

Cette idée a une origine précise. En 1943, deux chercheurs, **Warren McCulloch** et **Walter Pitts**, proposent un modèle mathématique très simplifié du neurone. C'est une petite unité qui reçoit des signaux, les combine et « s'allume » ou non selon que leur somme dépasse un certain seuil. Ils montrent qu'en reliant ces unités en réseau, on peut en principe réaliser des opérations logiques. C'est le premier lien établi entre le fonctionnement du cerveau et le calcul.



Le neurone formel : combiner des signaux, puis s'activer si leur somme dépasse un seuil.

Il restait à expliquer comment un tel réseau pourrait apprendre. En 1949, le psychologue **Donald Hebb** propose l'idée suivante : lorsque deux neurones s'activent ensemble de façon répétée, le lien qui les unit se renforce (« *neurons that fire together, wire together* »). Dans cette optique, apprendre ne consiste pas à réécrire des règles, mais à **ajuster la force des connexions**. C'est ce principe que Rosenblatt va appliquer dans sa machine.



La règle de Hebb : deux neurones qui s'activent ensemble voient leur lien se renforcer.

En 1958, le psychologue **Frank Rosenblatt** construit une machine réelle fondée sur cette idée, le **perceptron**. Au lieu de lui donner une règle, on montre au perceptron des exemples, par exemple des images étiquetées « cercle » ou « carré ». À chaque essai, s'il se trompe, il **ajuste légèrement ses réglages internes** pour se rapprocher de la bonne réponse. Exemple après exemple, il s'améliore, sans que personne ne lui ait jamais indiqué ce qui distingue un cercle d'un carré. Il a appris. C'est l'approche **ascendante** (*bottom-up*) : on n'écrit pas le savoir, on le laisse se former à partir des données.

Le perceptron suscite des attentes démesurées. En 1958, le *New York Times*, rapportant les propos de Rosenblatt, annonce que la marine américaine a dévoilé l'embryon d'une machine électronique qui « sera capable de marcher, parler, voir, écrire, se reproduire et avoir conscience de son existence ». On était évidemment très loin de ces capacités. L'idée centrale, celle d'une machine qui apprend de

ses erreurs, s'est cependant révélée très féconde. Elle est à la base de tout ce que nous étudierons aux modules 2, 3 et 4.

Le perceptron est l'**ancêtre direct des réseaux de neurones** actuels. L'apprentissage profond (*deep learning*) d'aujourd'hui est, pour l'essentiel, un empilement de perceptrons perfectionnés, en très grand nombre et sur de nombreuses couches. Nous étudierons cette filiation au **Module 3**.

Deux univers parallèles

On pourrait croire que ces deux paris se sont succédé, d'abord l'un, puis l'autre. Ce n'est pas le cas, et c'est l'une des idées les plus importantes de ce cours. Ils sont nés **presque en même temps** et ont **coexisté en rivaux** pendant plus d'un demi-siècle, chacun avec ses chercheurs, ses revues et ses financements.

Ces deux paris reposent sur **deux conceptions de l'esprit**. Pour le camp symbolique, l'esprit est essentiellement de la **logique**, c'est-à-dire des symboles et des règles, indépendamment du support qui les réalise. Pour le camp connexionniste, l'esprit est avant tout un **cerveau**, c'est-à-dire un réseau qui s'ajuste à l'expérience. La question « qu'est-ce que penser ? » reçoit donc deux réponses presque opposées.

Ces deux approches ne progressent pas en parallèle : elles **dominent tour à tour**. Dès la fin des années 1960, comme nous le verrons dans « Les hivers et la bascule », le camp symbolique met presque fin aux recherches sur le perceptron, et l'approche symbolique domine presque seule les vingt années suivantes. Il faut attendre les années 2010, étudiées au Module 3 de ce cours, pour que la tradition connexionniste revienne au premier plan, sous le nom d'*apprentissage profond*.

> La comparaison avec le cerveau est utile, mais il faut s'en méfier pour cette raison même. Le « neurone » de McCulloch, Pitts et Rosenblatt est une version extrêmement simplifiée du vrai neurone biologique. Nous y reviendrons en détail au Module 3, au moment où la tentation de confondre les deux sera la plus forte.

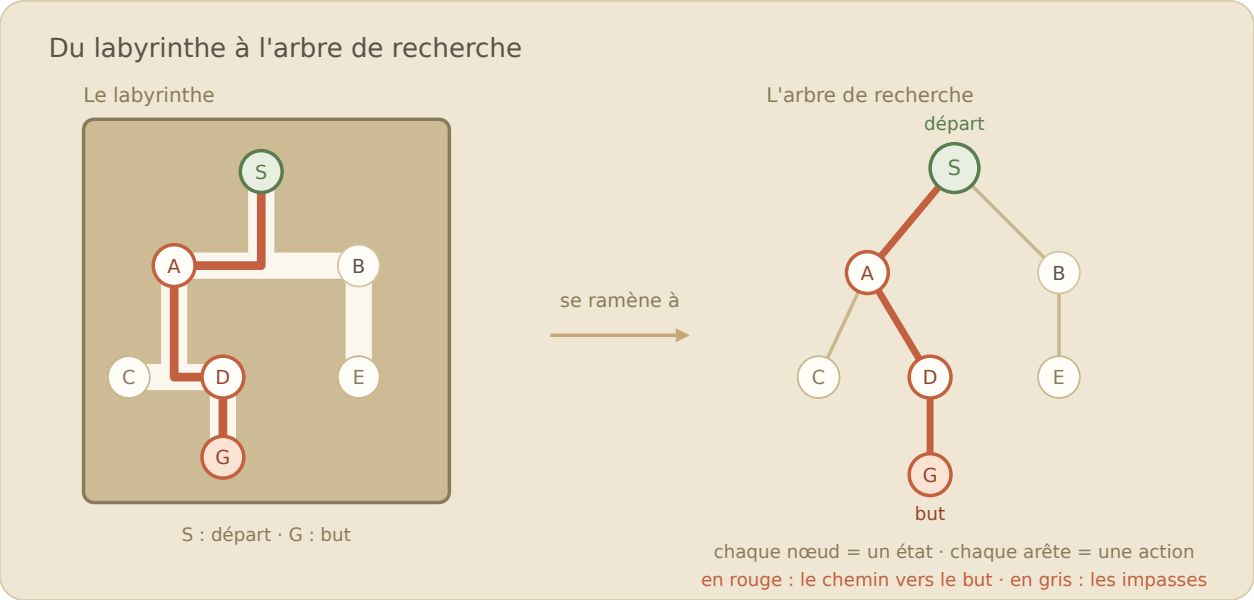
Pour le moment, nous laissons de côté le perceptron pour suivre le camp symbolique, qui est le premier à connaître une période de succès, celle des machines qui cherchent et qui raisonnent. C'est l'objet de « Chercher et raisonner ».

L'âge d'or symbolique : chercher et raisonner

Résoudre, c'est explorer

Une fois le pari symbolique posé, il reste une question pratique : comment faire raisonner une machine ? Les pionniers de l'IA trouvent une réponse très générale, parce qu'elle s'applique à des problèmes très différents. Presque tout problème peut se reformuler comme l'**exploration d'un espace de possibilités**.

Prenons l'exemple d'un labyrinthe. À chaque instant, vous êtes dans une certaine position, qu'on appelle un **état**. À partir de cet état, quelques actions sont possibles (avancer, tourner à gauche, tourner à droite), et chacune mène à un nouvel état. L'ensemble de tous les états atteignables forme une grande arborescence, l'**espace d'états** (*state space*). Résoudre le labyrinthe revient alors à **trouver un chemin** dans cette arborescence, depuis l'état de départ jusqu'à l'état-but (la sortie).



Un labyrinthe est un arbre d'états : résoudre, c'est trouver un chemin de S (départ) à G (but).

Cette idée est utile parce que beaucoup de problèmes qui semblent sans rapport prennent alors la même forme. Le taquin (un jeu de petites tuiles numérotées qu'on fait glisser), une partie d'échecs, la planification d'un itinéraire et la démonstration d'un théorème comportent tous un état de départ,

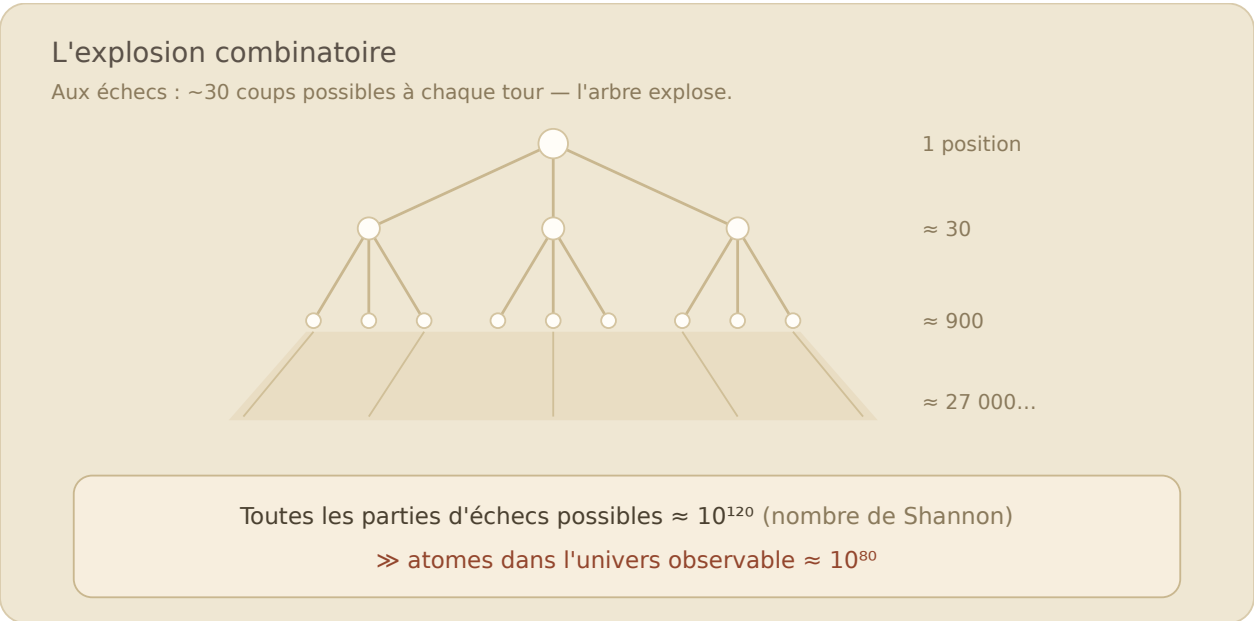
des actions qui font passer d'un état à un autre et un but à atteindre. Dans chaque cas, résoudre le problème revient à **chercher un chemin** vers ce but. Newell et Simon, les auteurs du Logic Theorist, poussent l'idée jusqu'à construire un programme appelé *General Problem Solver* (« solutionneur général de problèmes »), conçu pour traiter n'importe quel problème exprimé sous cette forme.

La recherche n'est pas une technique parmi d'autres. C'est le mécanisme central de l'IA symbolique (*symbolic AI*). Pour démontrer un théorème, planifier un trajet, diagnostiquer une panne ou lever l'ambiguïté d'une phrase, le GOFAI ramène le problème à l'exploration d'un espace de possibilités, jusqu'à y trouver une solution. Nous la retrouverons dans « Représenter le monde » et « Capturer l'expertise ». À la fin du module, dans *Les hivers et la bascule*, c'est aussi elle qui marquera la différence avec l'autre tradition : l'IA symbolique cherche une solution, alors que l'apprentissage automatique (*machine learning*) apprend à partir d'exemples.

L'explosion combinatoire

L'exploration d'un arbre de possibilités a cependant une limite importante. Pour la plupart des problèmes intéressants, cet arbre est **extrêmement grand**.

Les échecs en sont l'exemple le plus connu. À chaque tour, un joueur dispose en moyenne d'une trentaine de coups possibles. Chacun de ces coups permet une trentaine de réponses de l'adversaire, et ainsi de suite. Regarder seulement quelques coups à l'avance multiplie déjà fortement le nombre de branches à examiner. Le nombre de toutes les parties d'échecs possibles, appelé **nombre de Shannon**, vaut environ un 1 suivi de 120 zéros. Il dépasse de très loin le nombre d'atomes dans l'univers observable. Aucune machine, aussi rapide soit-elle, ne pourra jamais explorer un tel espace en entier.

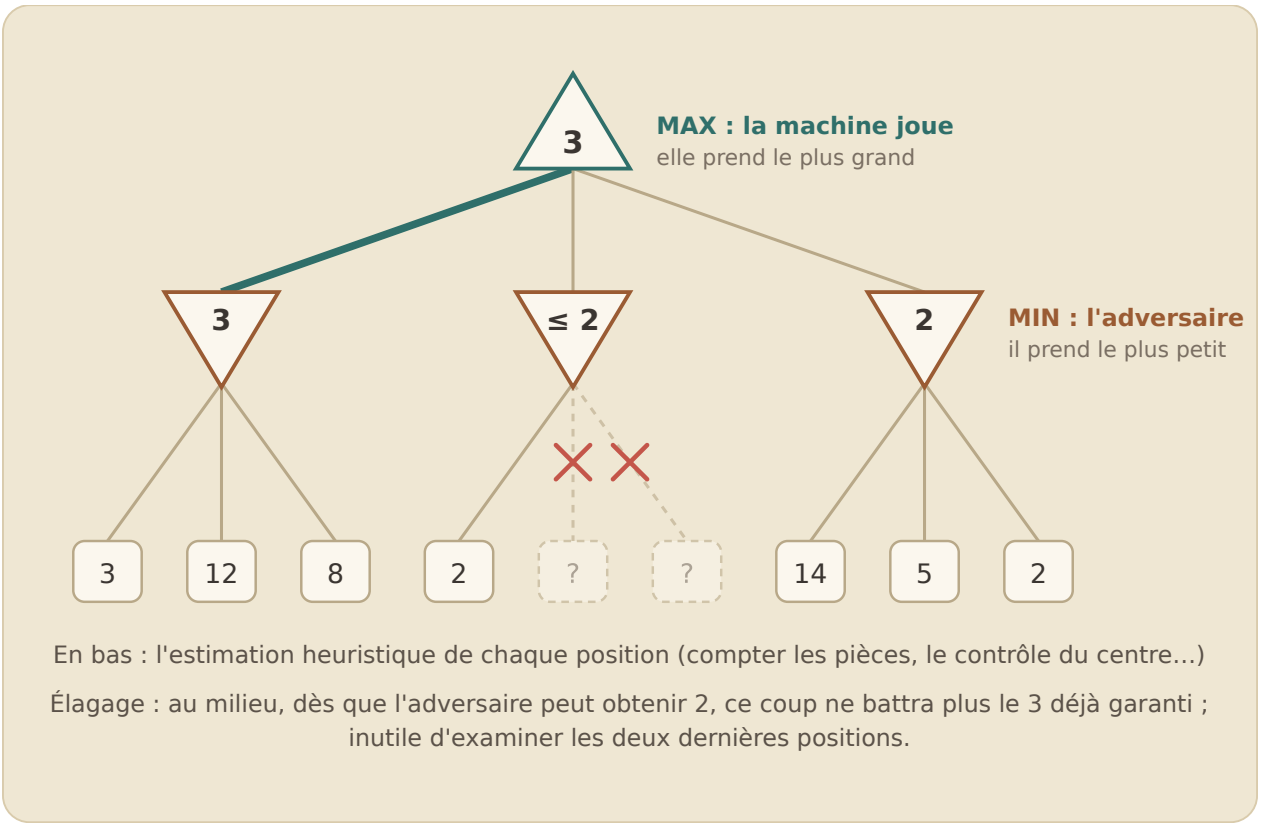


L'explosion combinatoire : à ~30 coups par tour, l'arbre des parties dépasse vite le nombre d'atomes de l'univers.

Pour les jeux à deux adversaires, les chercheurs mettent au point une stratégie appelée **minimax**. La machine explore l'arbre des coups en supposant que son adversaire jouera toujours le mieux possible. À chaque étape, elle cherche à maximiser son avantage, en supposant que l'adversaire

cherchera à le minimiser, d'où le nom. En remontant les conséquences de chaque coup, elle choisit celui qui lui garantit le meilleur résultat dans le pire des cas.

Comme l'arbre reste trop grand pour être exploré jusqu'au bout, il faut des **raccourcis**. Au lieu d'aller jusqu'aux fins de partie, la machine s'arrête à une certaine profondeur et estime la qualité d'une position à l'aide d'une **règle empirique** (une « heuristique »), par exemple en comptant les pièces ou en évaluant le contrôle du centre. D'autres techniques, comme l'**élagage** (*pruning* : ignorer dès le départ les branches qui ne peuvent pas changer la décision), évitent des explorations inutiles.



Minimax sur un petit arbre : les estimations remontent, en alternant le plus petit (l'adversaire) et le plus grand (la machine) ; au milieu, l'élagage évite d'examiner deux positions inutiles.

Le même principe s'applique en dehors des jeux, lorsqu'il faut **trouver un chemin**, par exemple dans le labyrinthe du début ou pour calculer un itinéraire routier. Au lieu d'explorer dans toutes les directions, un algorithme connu nommé **A*** (prononcé « A étoile ») se laisse guider par une heuristique. À chaque embranchement, il privilégie la direction qui semble le rapprocher le plus du but (par exemple, selon la distance à vol d'oiseau jusqu'à la destination). Le GPS qui calcule une route utilise ce genre de stratégie. C'est aussi le cas de l'**IA des jeux vidéo** : les personnages non joueurs (*non-player characters*) qui trouvent leur route sur la carte, ou les ennemis qui poursuivent ou contournent le joueur, s'appuient le plus souvent sur ces mêmes algorithmes de recherche de chemin (*pathfinding*), en particulier A*. La principale leçon de cette période est donc la suivante : un système efficace n'explore pas tout, il explore **au bon endroit**. La qualité des heuristiques est déterminante.

L'applet ci-dessous compare les deux stratégies sur le même labyrinthe. À gauche, une recherche aveugle (*blind search*) s'étend dans toutes les directions à la fois. À droite, A* est guidé par la distance

qui le sépare du but. Lancez les deux recherches, puis comparez le nombre de cases explorées : elles trouvent le même chemin, mais pas avec le même effort. Ajoutez ou retirez des murs en cliquant sur la grille, et observez dans quels cas l'heuristique aide beaucoup et dans quels cas elle mène la recherche dans une mauvaise direction.

Applet interactive : à utiliser en ligne, sur la page « L'âge d'or symbolique : chercher et raisonner » du site du cours.

Nous retrouverons cette limite, sous un autre nom, quand il s'agira d'apprendre à partir de données décrites par des milliers de caractéristiques. Ce sera la malédiction de la dimension (*curse of dimensionality*), au Module 2.

L'apogée : Deep Blue bat Kasparov (1997)

En mai 1997, à New York, se joue un match devenu célèbre. D'un côté, **Garry Kasparov**, champion du monde d'échecs en titre, considéré par beaucoup comme le plus grand joueur de l'histoire. De l'autre, **Deep Blue**, un superordinateur conçu par IBM. Au terme de six parties, la machine l'emporte. C'est la première fois qu'un champion du monde en exercice perd contre un ordinateur dans un match en conditions officielles. L'événement a un retentissement mondial, et la presse y voit le jour où la machine a « dépassé » l'humain.

Deep Blue applique directement les techniques décrites plus haut. Il n'utilise aucun réseau de neurones et aucun apprentissage. Il repose sur de la **recherche par force brute** (la machine évalue jusqu'à 200 millions de positions par seconde), guidée par des **heuristiques** mises au point avec l'aide de grands maîtres, et sur de très grandes bibliothèques d'ouvertures et de fins de partie. C'est du GOFAI typique, rendu très performant par la puissance de calcul.

Le match a été tendu. Au début de la rencontre, un coup subtil et inattendu de la machine, qu'il jugeait trop « humain », a déstabilisé Kasparov. Il en est venu à soupçonner une intervention humaine et a accusé IBM de tricherie. Il a réclamé une revanche, que l'entreprise a refusée, et Deep Blue a été démonté peu après. Ce coup inattendu aurait en réalité résulté d'un simple bogue dans le programme.

La vidéo suivante montre la réaction de Kasparov au moment où il abandonne la sixième et dernière partie, le 11 mai 1997. Elle montre bien ce que représentait cette défaite :

Vidéo : https://www.youtube.com/watch?v=EsMk1Nbcs-s

Au-delà de l'anecdote, cette victoire pose de nouveau la question de l'intelligence des machines. Deep Blue ne « comprend » pas les échecs comme Kasparov les comprend. Il ne sait même pas qu'il joue aux échecs. Il ne perçoit ni la beauté d'une combinaison ni la tension d'une partie, il ne sait rien faire d'autre et il ne peut pas expliquer pourquoi il a joué tel coup. On peut donc se demander s'il s'agit d'*intelligence* ou seulement d'une machine à calculer très puissante qui joue aux échecs.

IBM a relevé un autre défi quatorze ans plus tard, dans un domaine beaucoup plus difficile, le langage. En 2011, son système Watson a battu les meilleurs champions du jeu télévisé *Jeopardy!*, en combinant le savoir structuré de l'IA symbolique et l'apprentissage statistique. Nous le retrouverons à la fin du module, dans *Les hivers et la bascule*.

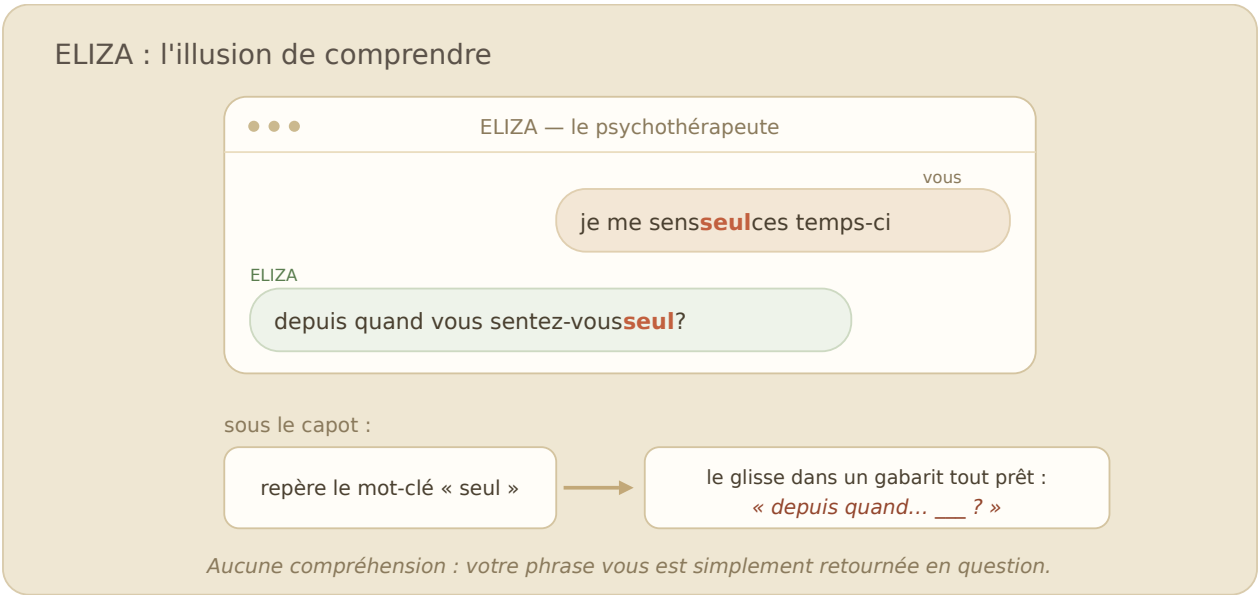
Le cas Deep Blue illustre un constat qui revient dans toute l'histoire de l'IA. Les tâches que nous jugeons les plus « intellectuelles » (jouer aux échecs, démontrer un théorème) se sont révélées **relativement faciles** à mécaniser. À l'inverse, ce qu'un enfant de trois ans fait sans effort (comprendre une phrase, reconnaître une scène, faire preuve de bon sens) a longtemps résisté. C'est le **paradoxe de Moravec**, sur lequel nous reviendrons au Module 2.

Ces difficultés montrent déjà les limites de cette période, sur lesquelles nous reviendrons dans « Les hivers et la bascule ». La même époque a cependant produit un autre type de programme. Il ne calcule pas pour gagner une partie, mais semble parler et écouter, et son cas est encore plus surprenant.

L'autre visage : ELIZA, ou l'illusion de comprendre

L'IA symbolique de cette période ne se limite pas à la recherche et au calcul. L'un de ses épisodes les plus marquants concerne une machine qui semblait non pas jouer, mais parler. En 1966, au MIT, l'informaticien **Joseph Weizenbaum** écrit **ELIZA**, un programme qui imite un psychothérapeute. La conversation paraît étonnamment naturelle. Si vous tapez « je me sens seul ces temps-ci », ELIZA répond « depuis quand vous sentez-vous seul ? ».

Pourtant, le programme ne comprend rien. ELIZA se contente de repérer des mots-clés et de **renvoyer les phrases de l'utilisateur sous forme de questions**, selon quelques règles très simples. Si vous écrivez « ma mère ne m'écoute jamais », le mot « mère » déclenche la réponse « parlez-moi de votre famille ». Le programme ne possède aucun savoir sur le monde, sur la solitude ou sur les mères.



ELIZA : repérer un mot-clé et renvoyer la phrase en question, sans rien comprendre.

La réaction des utilisateurs est ce qui a le plus marqué. Les gens se sont attachés à ELIZA. La secrétaire de Weizenbaum, qui savait pourtant qu'il s'agissait d'un programme, lui a un jour demandé de quitter la pièce pour pouvoir « parler en privé » avec la machine. Des utilisateurs lui ont confié des problèmes personnels, persuadés d'être écoutés. Weizenbaum en a été si troublé qu'il est devenu

l'un des principaux critiques de l'IA. On appelle aujourd'hui « **effet ELIZA** » cette forte tendance à *projeter* de la compréhension, et même des émotions, sur n'importe quelle machine qui manie le langage.

ELIZA est en quelque sorte l'inverse du test de Turing. Elle montre qu'il peut être facile de donner l'illusion de penser sans rien comprendre. Cette mise en garde prendra toute son importance avec les agents conversationnels (module 4) et dans le débat, toujours ouvert, sur ce que « comprendre » veut dire pour une machine (module 5).

Une dernière précision est importante pour la suite. On pourrait voir en ELIZA l'ancêtre direct de ChatGPT, comme s'il s'agissait du même procédé à plus grande échelle. C'est presque l'inverse. ELIZA n'est qu'un petit ensemble de règles écrites à la main (repérer un mot, produire une réponse à partir d'un gabarit), sans aucun apprentissage ni savoir sur le monde. C'est du GOFAI pur, entièrement programmé par son auteur. Les grands **modèles de langage** (les *LLM*) sur lesquels repose ChatGPT relèvent au contraire du **pari adverse**, celui du perceptron et des réseaux de neurones. Personne ne leur a donné de règles. Ils ont appris, à partir de très grandes quantités de textes, un modèle statistique du langage qui comporte des milliards de paramètres, et ils produisent des réponses nouvelles sur presque tous les sujets. La ressemblance n'est donc que superficielle. ELIZA et ChatGPT représentent les **deux paris rivaux** de ce module. Cela ne règle pas la question de fond, qui est de savoir si un LLM comprend ou s'il n'est qu'un imitateur beaucoup plus habile. L'effet ELIZA invite justement à ne pas trancher cette question trop vite.

Un mot sur l'outil. ELIZA, comme presque tous les programmes de cette période, était écrite en **Lisp**, un langage inventé par John McCarthy en 1958, l'année même du perceptron. C'est l'un des plus anciens langages de programmation encore utilisés aujourd'hui, et il a beaucoup influencé l'informatique (la récursion, le ramasse-miettes de mémoire, l'invite interactive et bien d'autres idées aujourd'hui courantes y sont nées). Son nom vient de *LISt Processing*, le « traitement de listes ». La plupart des langages sont d'abord conçus pour calculer des nombres, alors que Lisp est conçu pour **manipuler des symboles** (des mots, des concepts, des relations). C'était donc l'outil idéal pour le pari symbolique.

Pour aller plus loin : à quoi ressemble du Lisp ?
En Lisp, presque tout s'écrit sous forme de listes entre parenthèses, avec l'opération placée en tête. Une addition s'écrit ainsi :

```
(+ 1 2 3)      ; vaut 6
(* 3 (+ 2 4))  ; vaut 18, soit 3 × (2 + 4)
```

Jusqu'ici, rien de particulier. L'idée importante est qu'une liste peut contenir des *symboles* (des mots) aussi bien que des nombres. On peut écrire une liste de concepts :

```
(chien chat oiseau)
```

On peut aussi représenter une connaissance, c'est-à-dire un fait sur le monde :

```
(est-un Socrate humain)   ; « Socrate est un humain »
```

Le point essentiel est qu'en Lisp, **un programme a exactement la même forme que les données qu'il manipule**, puisque dans les deux cas il s'agit de listes. Un programme peut donc lire, transformer et même produire d'autres programmes aussi facilement qu'il manipule une liste d'épicerie. Cette propriété (« le code est une donnée comme une autre ») explique pourquoi Lisp convenait si bien à des systèmes destinés à raisonner sur des symboles.

Pendant des décennies, Lisp est resté le langage principal de l'IA symbolique. On a même construit des ordinateurs spécialisés, les « **machines Lisp** », pour l'exécuter plus efficacement. Nous verrons

dans « Les hivers et la bascule » que leur effondrement, vers 1987, marque l'un des hivers de l'IA.

Deep Blue cherchait et ELIZA manipulait du langage, mais aucun des deux ne connaissait vraiment le monde. Pour aller plus loin, il fallait donner à la machine une façon de **représenter ce qu'elle sait**. C'est le grand chantier de « Représenter le monde », qui s'est soldé par un échec important.

Représenter le monde

Le sens, angle mort de la machine

Les chapitres précédents ont présenté plusieurs programmes. Le Logic Theorist démontrait des théorèmes, Deep Blue gagnait aux échecs et ELIZA tenait une conversation. Tous faisaient cependant la même chose : ils **manipulaient des symboles d'après leur forme**, selon des règles. Par exemple, ELIZA repérait le mot « mère » sans savoir ce qu'est une mère. On parle dans ce cas du niveau de la **syntaxe**, qui consiste à agencer correctement des symboles sans tenir compte de leur sens.

Comprendre le monde demande davantage. Les spécialistes du langage distinguent trois niveaux, qu'on peut illustrer par un exemple simple. À table, quelqu'un vous dit **« Pouvez-vous me passer le sel ? »**

- La **syntaxe** concerne la *forme*. La phrase est une question grammaticalement bien construite. Une machine peut le vérifier sans rien comprendre.
- La **sémantique** concerne le *sens littéral*. La phrase porte sur votre *capacité* à passer le sel. Pour la comprendre à ce niveau, il faut savoir ce que signifient « sel », « passer » et « pouvoir ».
- La **pragmatique** concerne l'*intention réelle en contexte*. Tout le monde comprend qu'il ne s'agit pas d'une question sur vos aptitudes (« oui, je peux passer le sel! »), mais d'une **demande** polie, « passez-moi le sel ». Pour comprendre cela, il faut connaître le contexte et un grand nombre de sous-entendus que nous partageons tous.

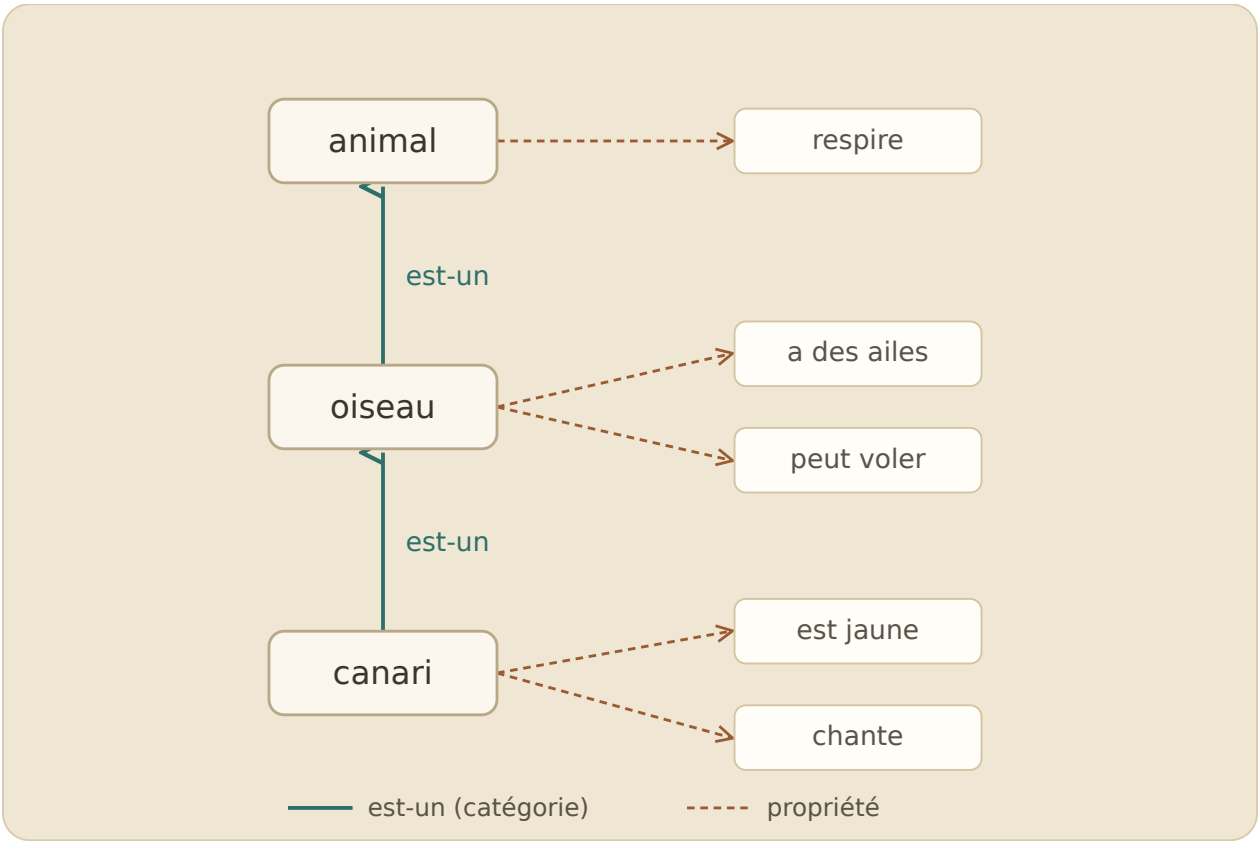
On peut résumer ainsi la situation de l'IA symbolique : elle réussissait bien au niveau de la **syntaxe**, atteignait difficilement la **sémantique** et échouait au niveau de la **pragmatique**. Pour passer de la forme au sens, une machine a besoin de **connaissances** sur le monde, et Deep Blue et ELIZA n'en avaient aucune. Ce chapitre porte sur les tentatives pour fournir ces connaissances aux machines et sur leur échec. La question de départ est la suivante : comment mettre dans une machine ce que tout le monde sait ?

Donner un savoir à la machine

Si l'intelligence exige des connaissances, il faut trouver un moyen de les **inscrire dans la machine** sous une forme qu'elle puisse exploiter. Entre la fin des années 1960 et les années 1970, trois grandes façons de structurer le savoir sont proposées.

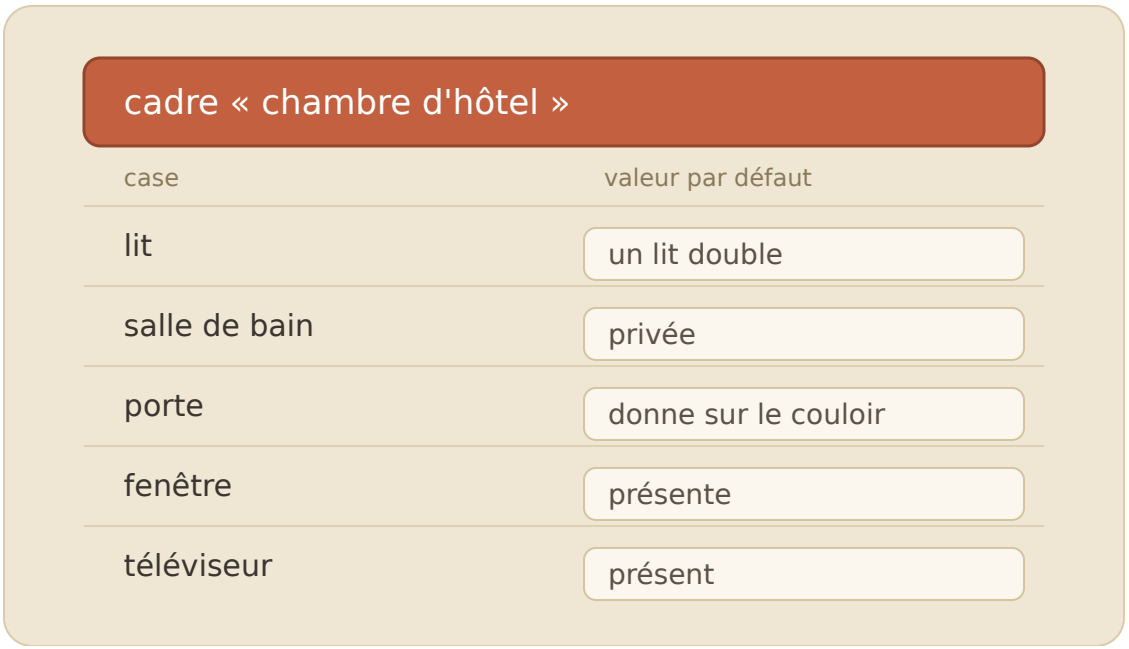
Les réseaux sémantiques (*semantic networks*, Ross Quillian). On représente les connaissances comme un **réseau de concepts reliés** par des relations. « Canari » est relié à « oiseau » par un lien *est-un* (*is-a*), et « oiseau » est relié à « animal » de la même façon. « Oiseau » est relié à « ailes » par un lien *possède*. La machine peut alors **déduire** des faits qu'on ne lui a pas donnés explicitement. Par exemple, pour

savoir si un canari a des ailes, il suffit de suivre les liens (*canari est-un oiseau*, *oiseau possède ailes*) pour conclure que oui. Ces réseaux portent le mot **sémantique** dans leur nom parce que leur objectif est de passer de la forme au sens.



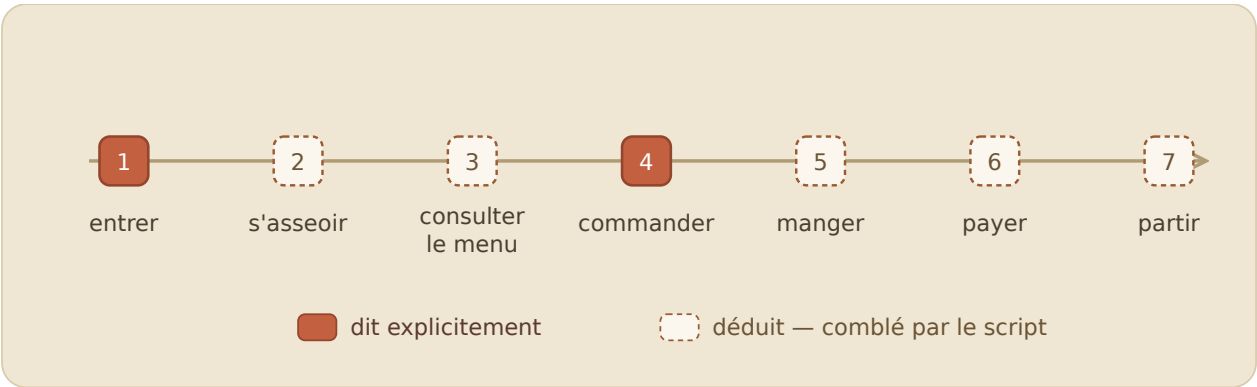
Un réseau sémantique : les propriétés s'héritent en remontant les liens « est-un ».

Les frames, ou « cadres » (Marvin Minsky, 1974). Au lieu de concepts isolés, Minsky propose de regrouper le savoir en **situations types** qui comportent des « cases » à remplir, avec des valeurs par défaut. Le cadre « chambre d'hôtel » comporte des cases pour le lit, la porte et la salle de bain. Par défaut, on s'attend à y trouver un lit. Quand vous entrez dans une chambre d'hôtel inconnue, vous n'analysez pas toute la scène. Vous utilisez ce cadre déjà connu et vous ne corrigez que ce qui ne correspond pas. Les frames permettent donc de représenter nos **attentes**.



Un frame : une situation type dont les cases ont des valeurs par défaut.

Les scripts (Roger Schank et Robert Abelson). L'idée est la même, mais elle s'applique à des **enchaînements d'actions**. Le « script du restaurant » décrit la séquence attendue : entrer, s'asseoir, consulter le menu, commander, manger, payer, partir. Grâce à ce script, une machine peut **compléter** un récit. Si on lui dit « Jean est allé au restaurant et a commandé un steak », elle en déduit qu'il s'est assis, qu'il a mangé et qu'il a payé, même si le récit ne le dit pas.



Un script : la séquence attendue, dont la machine complète les étapes non dites.

Ces trois approches ont en commun une capacité qui manquait à Deep Blue et à ELIZA, celle d'**inférer l'implicite**, c'est-à-dire d'utiliser un savoir de fond pour aller au-delà de ce qui est dit littéralement. Elles représentent un progrès réel vers la sémantique.

Ces représentations ont un ancêtre plus rigoureux, la **logique formelle**. Depuis Boole et Frege, la logique cherche à réduire le raisonnement à un *calcul* sur des symboles : « Socrate est un homme ; tous les hommes sont mortels ; donc Socrate est mortel ». Ce syllogisme semble porter sur le sens, mais on l'obtient par une opération mécanique sur la *forme* des énoncés. Le problème est là. Qu'on

enchaîne des règles logiques ou qu'on suive des liens dans un réseau, on manipule toujours la **forme** des symboles. La question est de savoir si la forme suffit à représenter le **sens**. Comme on va le voir, ce progrès vers la sémantique s'accompagne d'une fragilité importante.

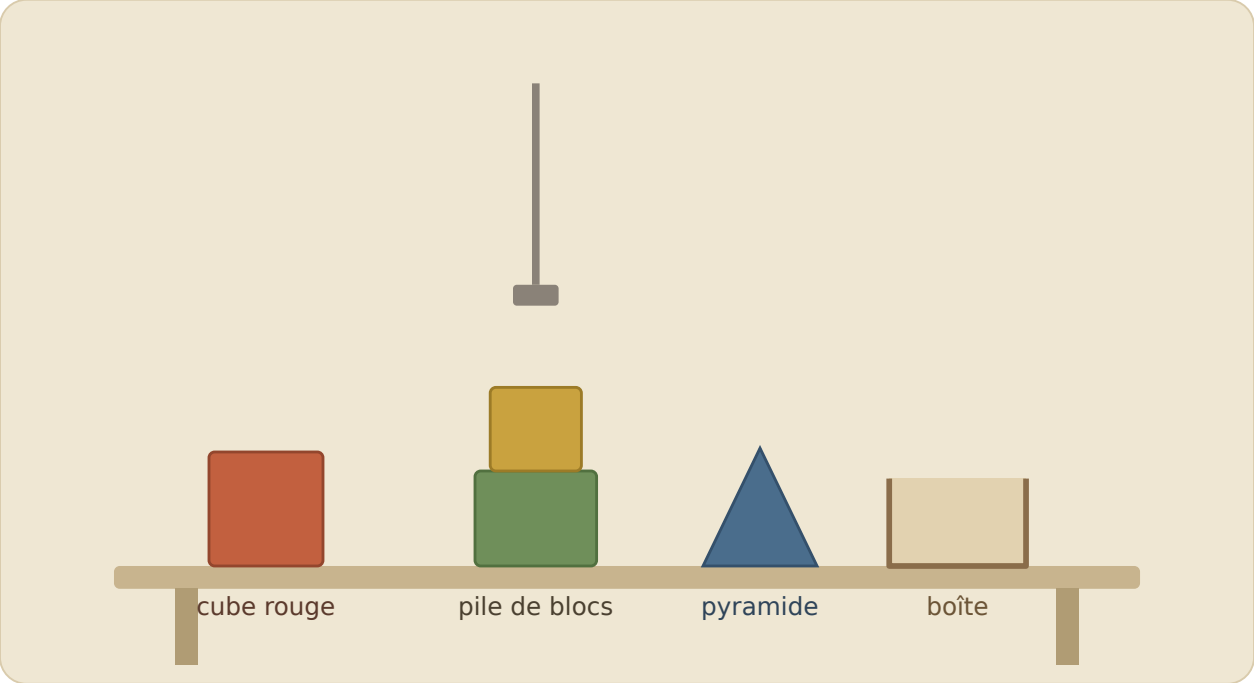
> Ces façons de structurer le savoir (réseaux de concepts, cases à remplir, hiérarchies *est-un*) n'ont pas disparu avec le GOFAI. Elles ont été reprises ailleurs. Le **web sémantique**, les **ontologies** et les **knowledge graphs** utilisés par la recherche de Google ou par Wikidata en sont les descendants directs, et les *cases héritées* des frames ressemblent aux **objets** de la programmation moderne. Nous y reviendrons dans « Les hivers et la bascule », pour évaluer ce que le GOFAI a laissé.

SHRDLU, ou le sommet de l'ambition

C'est dans ce contexte optimiste qu'un jeune chercheur du MIT, **Terry Winograd**, réalise vers **1970** un programme qu'on considère encore comme l'un des résultats les plus avancés de l'IA symbolique, **SHRDLU**. Le programme dialogue en anglais ordinaire avec un humain à propos d'un petit monde de **blocs colorés** (cubes, pyramides, boîtes posés sur une table), qu'il peut déplacer à l'aide d'un bras robotique virtuel. Pour l'époque, la conversation est très impressionnante :

> — *Prends un gros bloc rouge.* — D'accord. — *Saisis la pyramide.* — Je ne sais pas de **quelle** pyramide vous parlez. — *Trouve un bloc plus haut que celui que tu tiens et place-le dans la boîte.* — Par « **le** », je suppose que vous parlez du bloc plus haut que celui que je tiens.

Dans cet échange, SHRDLU **lève une ambiguïté** (quelle pyramide), **résout un pronom** (ce à quoi renvoie « le »), **agit** sur le monde et **se souvient** de ce qu'il vient de faire. On peut aussi lui demander « Pourquoi as-tu fait ça ? », et il **explique** son raisonnement.



Le « micro-monde » de SHRDLU : un univers très petit, entièrement décrit à la main.

Le film de démonstration d'époque ci-dessous (en anglais) montre SHRDLU en fonctionnement : les ordres tapés au clavier, les réponses du programme et le bras virtuel qui déplace les blocs.

Vidéo : https://www.youtube.com/watch?v=bo4RvYJYOzI

SHRDLU y parvient parce qu'il maintient un **modèle du monde** (*world model*), c'est-à-dire une représentation interne de sa petite scène : quel bloc repose sur quel autre, lequel est rouge, lequel est libre, ce que le bras tient à ce moment. À chaque action, il **met ce modèle à jour**, et à chaque question, il le **consulte**. C'est ce modèle interne qui lui permet de résoudre « le », de se rappeler son action précédente et de justifier ce qu'il a fait. SHRDLU ne se contente donc pas de parler des blocs, il en possède une représentation exacte. Cette idée est importante : **un modèle du monde est une représentation interne de la réalité, sur laquelle on peut raisonner**. Nous la retrouverons beaucoup plus loin dans le cours, au centre d'un débat important sur les IA actuelles.

Cette réussite repose cependant sur une simplification. Le modèle du monde de SHRDLU est exact parce qu'il a très peu de choses à représenter : quelques blocs, une table, une boîte, quelques formes et couleurs. Dans un univers aussi petit, on peut tout décrire à la machine, c'est-à-dire la liste complète des objets, des propriétés et des actions possibles. Le « monde » de SHRDLU tient entièrement dans une représentation **codée à la main**. Winograd l'appelait d'ailleurs un **micro-monde** (*blocks world*), et le préfixe *micro* indique bien sa taille. L'intérêt du micro-monde est qu'il rend le problème **traitable** (*tractable*, en anglais). En informatique, un problème est traitable quand une machine peut le résoudre en un temps raisonnable, sans être freinée par l'explosion combinatoire des possibilités. Avec quelques blocs, tout peut être représenté et chaque question obtient une réponse immédiatement. C'est ce qui explique la réussite de SHRDLU, et c'est aussi sa limite, parce que le monde réel n'est pas traitable en ce sens, comme le montre la section suivante.

En dehors de la table à blocs, SHRDLU ne peut rien faire. Son modèle du monde ne contient rien sur la pluie, un mensonge ou un escalier. Il ne peut pas être étendu au monde réel, parce qu'il faudrait alors y représenter tout ce qui existe. Toutes les approches de ce chapitre ont la même limite. Elles fonctionnent tant qu'on reste dans un domaine assez petit pour être entièrement décrit, et elles échouent dès qu'il faut tenir compte de la très grande quantité de choses que les humains considèrent comme évidentes. Cette quantité de savoir évident a un nom, et elle constitue la principale limite du GOFAI.

Le mur du sens commun

Cette limite s'appelle le **sens commun** (*common sense*). Il s'agit de l'ensemble très vaste des choses si évidentes que personne ne prend la peine de les dire. Par exemple, l'eau mouille, un objet qu'on lâche tombe, on ne peut pas pousser une corde, votre mère est plus âgée que vous, et si Jean entre dans un restaurant, il y entre par la porte et non par le plafond. Nous utilisons en permanence des millions de certitudes de ce genre. Comme elles vont **sans dire**, personne ne les a jamais écrites.

Or toute l'IA symbolique repose sur un principe : pour qu'une machine sache quelque chose, il faut lui **inscrire** ce savoir. Les réseaux sémantiques, les frames, les scripts et SHRDLU fonctionnent tant que ce savoir tient dans un domaine assez petit pour être décrit à la main. La machine ne sait que ce qu'on lui a fourni explicitement. Le sens commun, lui, **n'a pas de limite**. Pour le donner à une machine, il faudrait lui décrire le monde entier.

Un chercheur a tenté de le faire. En **1984**, **Douglas Lenat** lance **CYC** (de l'anglais *encyclopedia*), un projet très ambitieux dont l'objectif est d'**encoder à la main**, fait après fait et règle après règle, la

totalité du sens commun humain. Des équipes y ont consacré des **décennies** et des dizaines de millions de dollars, et ont saisi des millions d'assertions, comme « un café chaud refroidit si on le laisse » ou « on ne peut pas être à deux endroits à la fois ». C'est le projet le plus ambitieux de l'histoire du GOFAI.

Le sens commun, écrit à la main : quelques entrées de CYC

dans la langue de CYC (CycL)	*ce que cela veut dire*
`(#$isa #$BillClinton #$UnitedStatesPresident)`	Bill Clinton fait partie des présidents des États-Unis.
`(#$genls #$Tree-ThePlant #$Plant)`	Tous les arbres sont des plantes.
`(#$capitalCity #$France #$Paris)`	Paris est la capitale de la France.
`(#$relationAllExists #$biologicalMother #$ChordataPhylum #$FemaleAnimal)`	Tout animal à colonne vertébrale a une mère biologique, un animal femelle.
`(#$implies (#$and (#$isa ?OBJ ?SUBSET) (#$genls ?SUBSET ?SUPERSET)) (#$isa ?OBJ ?SUPERSET))`	Règle : si un objet appartient à une catégorie, il appartient aussi à toutes celles qui la contiennent.

Des millions d'entrées de ce genre, saisies une à une pendant des décennies.

Le sens commun écrit à la main : quelques entrées de CYC dans son langage, CycL, et leur signification.

CYC n'a jamais atteint son objectif. La raison n'est pas un manque d'argent ou de compétence, mais le fait que la tâche n'a **pas de fin**. Chaque évidence saisie en fait apparaître dix autres, et chacune de celles-ci en suppose cent. On ne peut pas compléter le sens commun par petites quantités, parce qu'il s'étend à mesure qu'on l'écrit. CYC s'est heurté au même problème que le reste de ce chapitre : **le savoir implicite d'un humain ordinaire est trop vaste pour être énuméré**.

Le GOFAI atteint donc ici sa limite. On pensait qu'il suffisait de donner des connaissances à la machine, mais les connaissances les plus importantes sont justement celles que personne ne formule. Cependant, en dehors du courant dominant, un chercheur affirmait depuis longtemps qu'on abordait le problème dans le mauvais sens et qu'on cherchait le sens au mauvais endroit. Avant de terminer sur l'âge d'or symbolique, il faut présenter ce point de vue dissident.

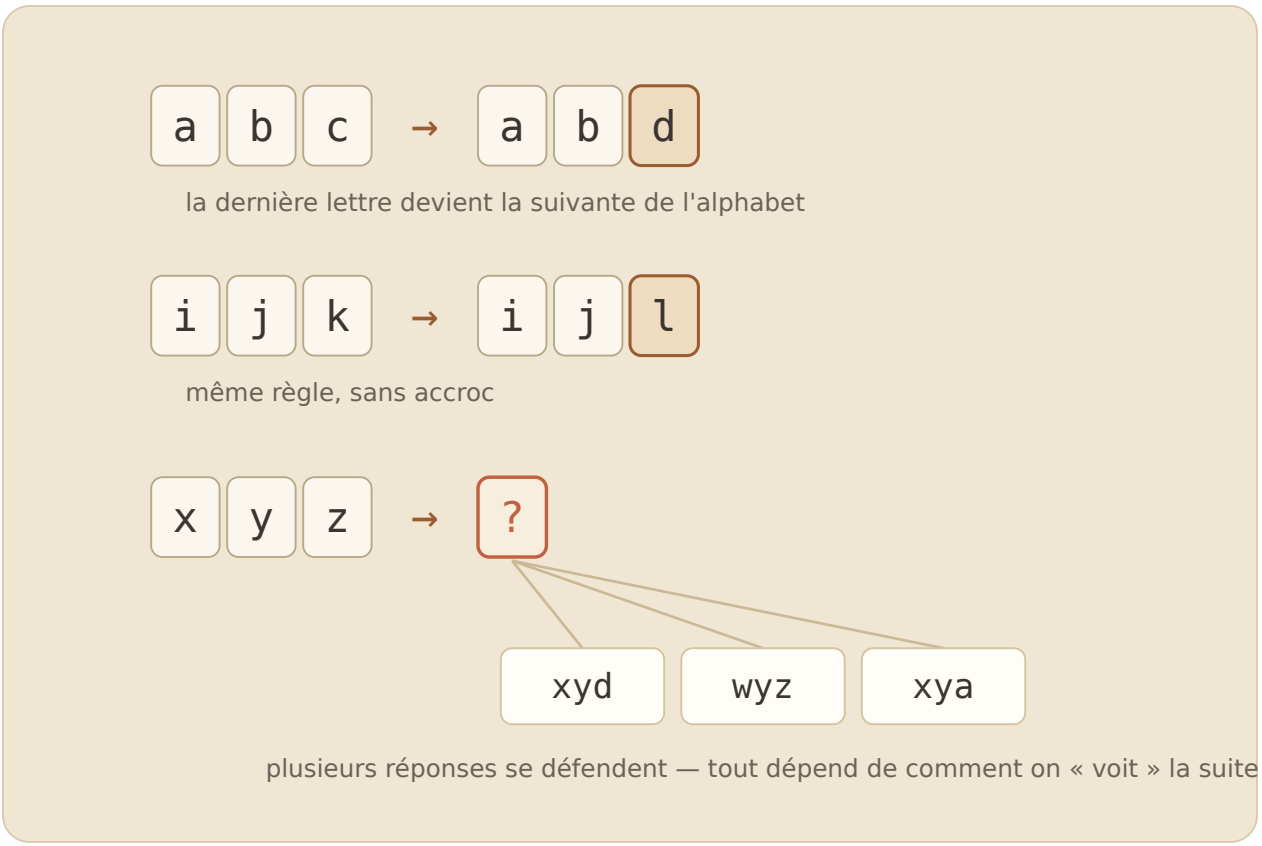
L'objection de Hofstadter

Ce chercheur est **Douglas Hofstadter**, l'auteur de la notion de *boucle étrange*, déjà présenté dans « Turing et la question fondatrice » à propos de Gödel. Pendant que ses collègues construisaient des

moteurs d'échecs et des bases de règles (*rule bases*), il répétait que l'IA dominante ne s'attaquait pas au bon problème. Selon Hofstadter, battre Kasparov par force brute ou accumuler des millions d'assertions comme CYC ne concerne pas l'essentiel de la pensée.

Pour lui, l'essentiel de la pensée est l'**analogie**. Penser ne consiste pas à appliquer des règles, mais à *percevoir des ressemblances*, à adapter des **concepts fluides** (*fluid concepts*) à des situations nouvelles et à comprendre l'inconnu à partir de ce qu'on connaît déjà. Quand vous parlez du *pied* d'une montagne, des *jambes* d'une table ou de la *bouche* d'un fleuve, vous faites de l'analogie sans y penser. C'est l'opération de l'esprit la plus courante et aussi la plus fondamentale. Or une analogie ne peut pas être inscrite d'avance dans une base de connaissances (*knowledge base*). Elle se construit au moment où on en a besoin, selon le contexte. C'est précisément ce qu'un projet comme CYC ne pouvait pas représenter.

Pour le démontrer, Hofstadter et sa collaboratrice **Melanie Mitchell** ont conçu un programme, **Copycat**, qui fonctionne lui aussi dans un micro-monde, mais d'un genre très différent de celui de SHRDLU. Son univers est constitué de simples **chaînes de lettres**. On lui pose des problèmes d'analogie, par exemple : si `abc` devient `abd`, que devient `ijk` ? La réponse naturelle est `ijl`, parce qu'on remplace la dernière lettre par la suivante dans l'alphabet. Cependant, Copycat n'applique pas une règle fixe. Il *perçoit* une structure, et il peut en percevoir plusieurs. Si on lui demande ce que devient **`xyz`** selon la même analogie, le problème est plus difficile, parce que le `z` n'a pas de lettre suivante. Il n'y a plus de réponse unique. On peut proposer `xyd`, ou `wyz`, ou recommencer au début de l'alphabet. La « bonne » réponse dépend de la façon dont on perçoit la situation. C'est ce que Hofstadter veut montrer : l'intelligence n'est pas l'exécution d'une règle, mais une **perception fluide, sensible au contexte**.



L'énigme de Copycat : pour « xyz », la réponse dépend de la façon dont on perçoit la suite.

Les deux micro-mondes ont donc des rôles opposés. SHRDLU utilisait un micro-monde pour **simplifier** le problème, c'est-à-dire pour avoir un univers assez petit pour tout y énumérer d'avance. Hofstadter réduit le sien pour la raison **inverse**. Il écarte tout le savoir encyclopédique afin d'**isoler l'essentiel**, l'analogie elle-même. Ces deux micro-mondes correspondent à deux conceptions opposées de l'intelligence.

Reste la question qui traverse tout ce chapitre, celle de savoir comment le sens peut apparaître à partir de simples symboles. La réponse de Hofstadter prolonge son idée de **boucle étrange**. Selon lui, le sens n'est pas **injecté** de l'extérieur, fait après fait, comme CYC l'espérait. Il **émerge** d'un système assez riche pour s'observer lui-même, percevoir ses propres structures et établir des analogies entre ses propres états. On ne remplit pas un esprit de significations. On met en place un mécanisme dont la signification émerge. C'est l'opposé de la démarche du GOFAI, qui voulait tout écrire à la main.

Hofstadter avait en partie raison. Sa **vision** était juste. Comme on le verra, c'est par l'émergence, et non par l'inscription, que l'IA finira par progresser. Cependant, sa **solution** n'a jamais eu l'ampleur de son ambition. Copycat est resté un petit programme de laboratoire, remarquable mais limité. Hofstadter ne travaillait pas non plus sur les réseaux de neurones. Il a indiqué la bonne direction sans construire l'outil qui permettrait de la suivre.

Cet outil existait déjà, mais il était alors peu développé. Depuis « Deux paris rivaux », nous savons qu'il existe une **autre tradition**, qui ne cherche pas à décrire le monde à la machine, mais qui la laisse l'**apprendre** elle-même à partir d'exemples. Le GOFAI a atteint sa limite, le sens commun lui a résisté, et l'idée d'émergence, proposée par l'un de ses critiques, va déjà dans cette direction. L'idée

de laisser la machine apprendre plutôt que de tout lui dire ne s'imposera toutefois qu'après une longue période de recul de l'IA. Avant ce déclin, l'IA symbolique connaîtra encore sa période de plus grand succès.

Pour aller plus loin : le fonctionnement de Copycat
Copycat ne contient aucune règle du genre « remplace la dernière lettre ». Il explore en parallèle un grand nombre de petits rapprochements possibles (une lettre est-elle un début, une fin, le successeur d'une autre ?), qui se renforcent ou s'inhibent mutuellement. Une mesure interne, que ses auteurs appellent la **« température »**, indique à quel point une interprétation cohérente a émergé. Tant que tout reste flou, la température est élevée et le programme continue d'explorer presque au hasard. Quand une structure d'ensemble se forme, la température baisse et la réponse se stabilise. Le sens n'est donc pas *calculé* en une seule étape, mais **construit** peu à peu, par une compétition entre des perceptions partielles. Selon Hofstadter, ce mécanisme illustre à petite échelle ce qui caractérise la cognition.

Capturer l'expertise : les systèmes experts

Rétrécir et codifier le monde pour réussir

Le chapitre « Représenter le monde » s'est terminé sur un constat : on ne peut pas écrire le sens commun, parce qu'il n'a pas de limite. Les chercheurs des années 1970 en tirent une conclusion pratique. Si le savoir général est hors d'atteinte, il faut viser un savoir plus **étroit**.

L'idée est la suivante. Au lieu de chercher une intelligence générale, on choisit **un seul domaine** bien délimité (diagnostiquer une infection, configurer un ordinateur, prospecter un gisement minier) et on tente d'y reproduire la compétence d'**un seul expert**. Dans un domaine spécialisé, le savoir-faire d'un expert ressemble souvent à un grand ensemble de règles de la forme *si telles conditions, alors telle conclusion*. Le médecin qui raisonne « *si* le patient a de la fièvre **et** telle bactérie dans le sang, *alors* prescrire tel antibiotique » applique une règle. Si on recueille assez de ces règles et qu'on les inscrit dans la machine, celle-ci devrait raisonner comme l'expert, du moins dans ce domaine. C'est le principe des **systèmes experts**, qu'on peut résumer par la formule suivante : **connaissance = règles explicites**.

La démarche est la même que celle de SHRDLU, qui ne « comprenait » son monde de blocs que parce que ce monde était très petit. Les systèmes experts réduisent eux aussi le monde à un domaine restreint, mais il s'agit cette fois de domaines réels, qui ont une valeur commerciale. Cette approche va fonctionner. L'IA symbolique va sortir des universités, produire des revenus et convaincre le monde des affaires que l'intelligence artificielle est utilisable. La section suivante explique comment une machine peut raisonner avec des règles.

L'anatomie d'un système expert

Un système expert comporte **trois composantes**. La première est une **base de règles**, qui contient la connaissance du domaine sous forme d'énoncés *si… alors…*. La deuxième est une **base de faits** (ou « mémoire de travail », *working memory*), qui rassemble ce qu'on sait du cas traité, par exemple les symptômes d'un patient ou l'état d'une voiture. La troisième est un **moteur d'inférence** (*inference engine*), un mécanisme général qui compare les faits aux règles, applique celles dont les conditions sont remplies et en tire de nouveaux faits, jusqu'à une conclusion.

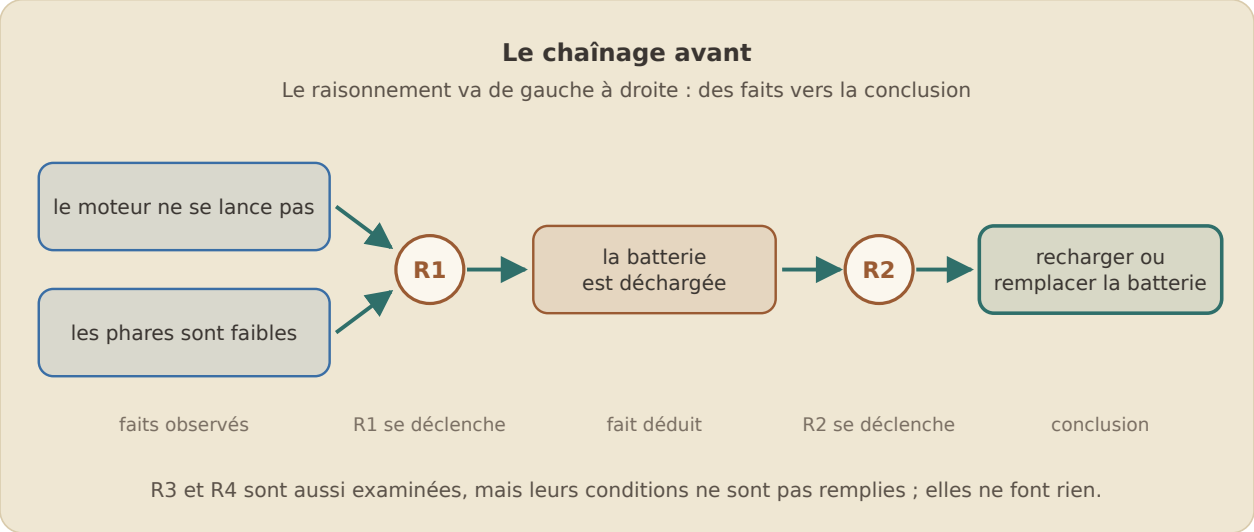
L'intérêt de cette architecture tient à la **séparation** entre ces composantes. Le moteur d'inférence ne contient aucune connaissance de médecine, d'automobile ou de minéralogie. Il se contente d'enchaîner des règles. Toute la compétence se trouve dans la base de règles, qu'on peut remplacer comme une cartouche. Avec d'autres règles, le même moteur devient un système de diagnostic médical, mécanique ou géologique. Comme le savoir est séparé du raisonnement, on peut construire plusieurs systèmes experts sans tout reprogrammer à chaque fois.

Prenons un exemple simple, **une voiture qui refuse de démarrer**. On donne au système les règles suivantes :

> **R1** : *si* le moteur ne se lance pas du tout **et** les phares sont faibles, *alors* la batterie est déchargée.
>
> **R2** : *si* la batterie est déchargée, *alors* recharger ou remplacer la batterie.
>
> **R3** : *si* le moteur se lance normalement mais ne démarre pas **et** le réservoir est vide, *alors* refaire le plein.
>
> **R4** : *si* le moteur se lance normalement **et** le réservoir n'est pas vide, *alors* faire vérifier l'allumage.

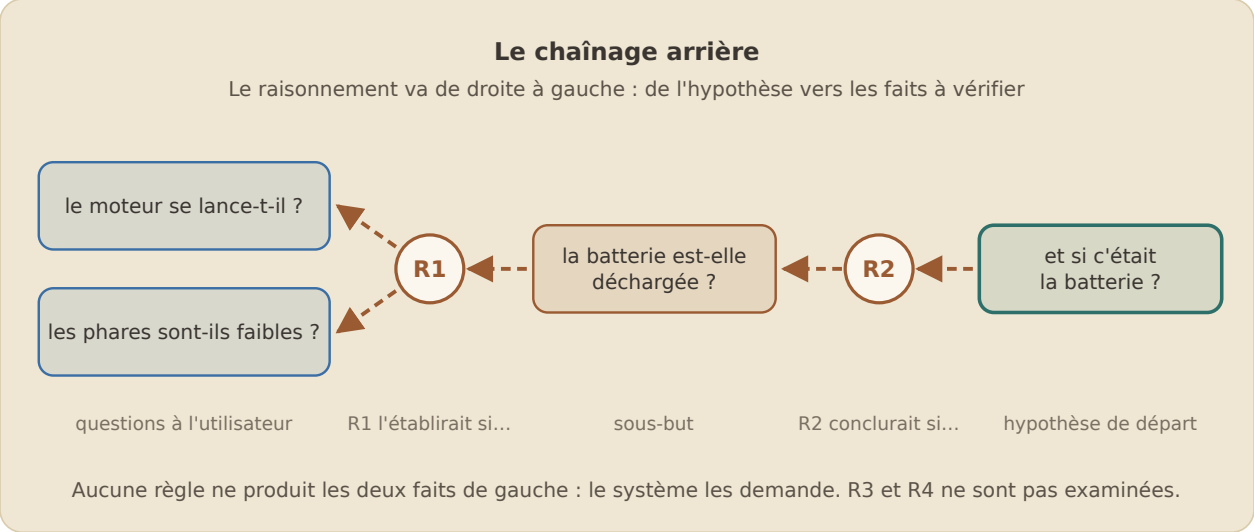
On lui fournit aussi deux faits observés, *le moteur ne se lance pas* et *les phares sont faibles*. Le moteur d'inférence parcourt alors les règles. Les deux conditions de **R1** sont satisfaites, donc R1 se **déclenche** et ajoute un fait nouveau, *la batterie est déchargée*. Ce fait satisfait la condition de **R2**, qui se déclenche à son tour et donne la conclusion *recharger ou remplacer la batterie*. Les conditions de R3 et de R4 ne sont pas remplies, et ces règles ne font rien. En deux étapes, le système a établi un diagnostic à partir des symptômes. Il peut aussi **retracer son raisonnement** : il conclut à la batterie à cause de R1, donc à cause des phares faibles.

Ce mode de raisonnement, qui part des faits pour arriver à une conclusion, s'appelle le **chaînage avant** (*forward chaining*). Il convient bien quand on dispose déjà de nombreux faits et qu'on veut savoir ce qui en découle.



Le chaînage avant : des faits observés vers la conclusion, règle après règle.

On peut aussi procéder dans l'autre sens. Supposons qu'on soupçonne la batterie et qu'on veuille vérifier cette hypothèse. Le moteur d'inférence part alors de l'**hypothèse**, traitée comme un but, et remonte les règles. Pour conclure « recharger la batterie » (R2), il faut que la batterie soit déchargée. Pour cela (R1), il faut que le moteur ne se lance pas **et** que les phares soient faibles. Aucune règle ne produit ces deux derniers faits, qu'il faut donc **observer**. Le système pose alors la question à l'utilisateur (« les phares sont-ils faibles ? ») et n'examine que ce qui concerne l'hypothèse étudiée. C'est le **chaînage arrière** (*backward chaining*).



Le chaînage arrière : de l'hypothèse vers les faits à vérifier, en ne posant que les questions utiles.

Prolog, un langage fondé sur le chaînage arrière. En 1972, on a fait de ce principe (déclarer des faits et des règles *si… alors…*, puis laisser un moteur général les enchaîner) un **langage de programmation**, **Prolog**. Le programmeur n'y écrit que des faits et des règles, et le langage fournit lui-même le moteur d'inférence, un chaînage arrière comme celui qu'on vient de décrire. On n'indique pas au programme comment calculer, mais ce qui est vrai. Cette approche s'appelle la **programmation logique** (*logic programming*). Elle est différente de la programmation fonctionnelle (*functional programming*) du langage Lisp (celui des machines de la photo plus bas). Prolog a longtemps été le langage de référence de l'IA symbolique en Europe. Le projet japonais de **Cinquième Génération**, présenté plus bas, l'a choisi comme langage principal, et ses principes se retrouvent aujourd'hui dans Datalog et dans les moteurs de règles (*rule engines*).

Pour aller plus loin : à quoi ressemble du Prolog ?
Reprenons l'exemple de la voiture. En Prolog, on écrit d'abord les **faits** observés, puis les **règles**. Dans une règle, la conclusion est placée avant le symbole `:-`, qui se lit « *si* », et les conditions sont séparées par des virgules, qui se lisent « *et* » :

```
% Les faits observés :
moteur_ne_se_lance_pas.
phares_faibles.

% Les règles R1 et R2, « si … alors … » :
batterie_dechargee   :- moteur_ne_se_lance_pas, phares_faibles.
recharger_batterie   :- batterie_dechargee.
```

On **interroge** ensuite le programme en lui soumettant un but à prouver (`?-` est l'invite) :

```
?- recharger_batterie.
true.
```

Pour répondre, Prolog effectue le chaînage arrière décrit plus haut. Pour établir `recharger_batterie`, il lui faut `batterie_dechargee`. Pour établir `batterie_dechargee`, il lui

faut les deux faits `moteur_ne_se_lance_pas` et `phares_faibles`, qui sont connus. Le but est donc atteint, et Prolog répond `true`. Le programme n'indique nulle part comment mener cette recherche, puisque le moteur du langage s'en charge.

Une même base de règles peut donc être parcourue dans deux sens, **vers l'avant**, à partir des faits, ou **vers l'arrière**, à partir d'un but. Le chaînage arrière est mieux adapté au diagnostic, parce qu'il ne pose que les questions utiles à l'hypothèse examinée, au lieu de demander toutes les mesures d'avance. C'est pour cette raison que le plus connu des systèmes experts, le système de diagnostic médical **MYCIN**, l'utilisait. Il est présenté dans la section suivante.

Dans l'applet ci-dessous, choisissez ce que vous observez sur la voiture, puis faites avancer le moteur d'inférence règle par règle. Vous le verrez examiner chaque règle, déclencher celles dont les conditions sont réunies et ajouter des faits à la base, jusqu'au diagnostic. Essayez plusieurs combinaisons, et trouvez celle pour laquelle le système ne peut rien conclure.

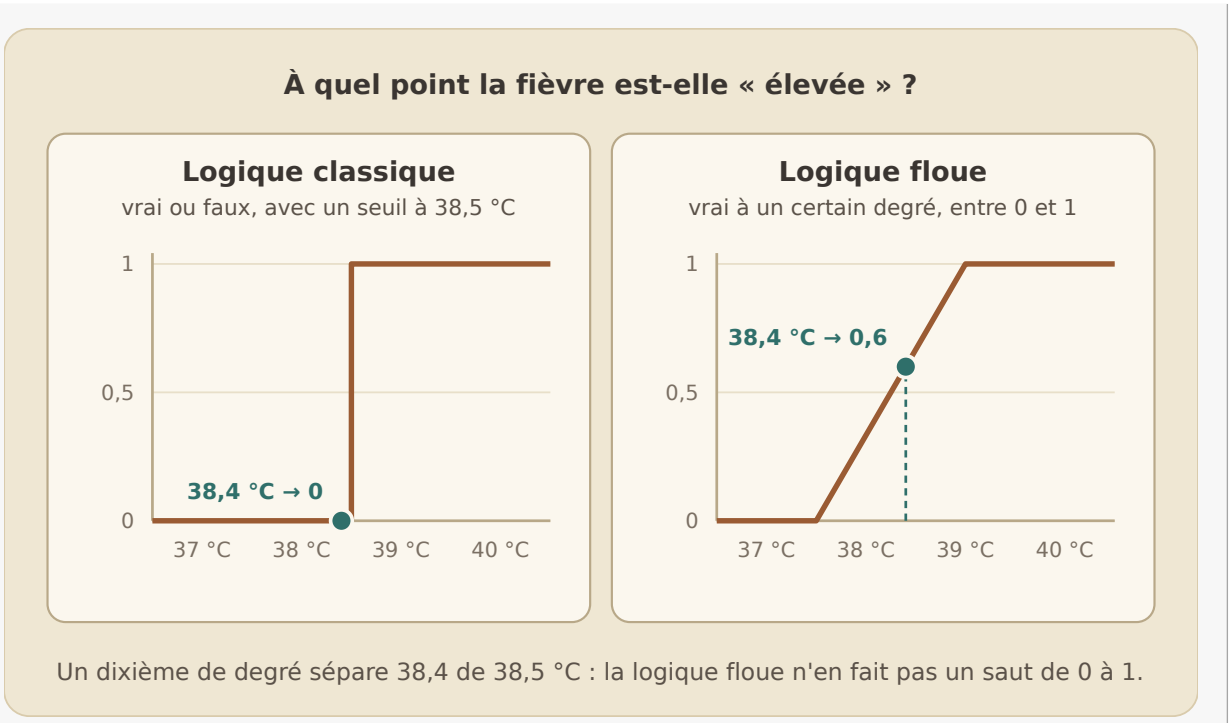
Applet interactive : à utiliser en ligne, sur la page « Capturer l'expertise : les systèmes experts » du site du cours.

L'âge d'or : MYCIN, XCON et le boom

Le premier système expert est généralement considéré comme étant **DENDRAL**, développé à Stanford à partir de 1965. Il identifiait des **molécules** à partir de données de spectrométrie, comme le ferait un chimiste expérimenté. Le plus connu est cependant **MYCIN**, conçu au début des années 1970 par **Edward Shortliffe**. MYCIN diagnostiquait les **infections bactériennes du sang** et recommandait un antibiotique et une dose. Il comptait environ **600 règles** et utilisait le **chaînage arrière** décrit plus haut. Il partait d'une hypothèse sur le germe en cause et posait des questions au médecin jusqu'à sa conclusion. Grâce à ce même mécanisme, il pouvait **justifier** sa démarche. Quand on lui demandait pourquoi il posait une question, il indiquait la règle qu'il cherchait à vérifier.

En médecine, cependant, peu de choses sont certaines. Un symptôme peut suggérer un germe sans le confirmer. MYCIN utilisait donc des **facteurs de certitude** (*certainty factors* : un nombre associé à chaque règle, qui indique à quel point sa conclusion est fiable) et les combinait au cours du raisonnement. Cette méthode était approximative. Plus tard, une théorie plus rigoureuse de l'incertitude, celle des **réseaux bayésiens**, l'a remplacée (voir le Module 2). MYCIN montrait déjà qu'un système qui raisonne doit aussi tenir compte de l'incertitude.

La logique floue. Les règles d'un système expert emploient souvent des mots imprécis : une fièvre « élevée », un moteur « chaud », une vitesse « faible ». En logique classique, un énoncé est vrai ou faux. À partir de quel degré une fièvre devient-elle « élevée » ? Fixer un seuil à 38,5 °C revient à dire que 38,4 °C n'est pas une fièvre élevée du tout. En 1965, le mathématicien **Lotfi Zadeh** propose la **logique floue** (*fuzzy logic*), où un énoncé peut être vrai à un certain degré, entre 0 et 1. Une fièvre de 38,4 °C peut ainsi être « élevée » à 0,6, et une fièvre de 40 °C à 1. Les règles *si… alors…* s'appliquent alors plus ou moins fortement selon ce degré.



« Fièvre élevée » : un seuil en logique classique, un degré en logique floue.

La logique floue ne traite pas le même problème que les facteurs de certitude de MYCIN. Ceux-ci mesurent à quel point on est sûr d'une conclusion. La logique floue mesure à quel point un mot s'applique à une situation. Elle a connu un grand succès industriel, surtout au Japon à partir de la fin des années 1980, dans des systèmes de commande : le métro de Sendai, des machines à laver, des autocuiseurs à riz, des stabilisateurs de caméras. Elle est encore utilisée aujourd'hui en automatique.

MYCIN donnait de bons résultats. En 1979, une évaluation a soumis ses recommandations à un jury d'experts, qui les a comparées à celles de médecins sans savoir lesquelles venaient du programme. Le jury a jugé les recommandations de MYCIN **aussi bonnes, voire meilleures**, que celles des spécialistes. Pourtant, **MYCIN n'a jamais été utilisé pour soigner un patient.** Les obstacles étaient humains et pratiques plutôt que scientifiques. On ne savait pas qui serait **responsable** en cas d'erreur (le médecin, l'hôpital ou le programmeur). Il était difficile de l'**intégrer** au travail clinique, à une époque où il n'y avait pas d'ordinateur au chevet des patients et où tout devait être saisi sur un terminal. Enfin, les médecins n'étaient pas prêts à **déléguer** leur jugement à un programme. Le cas de MYCIN montre que les difficultés des systèmes experts n'étaient pas seulement techniques.

L'industrie a adopté les systèmes experts plus facilement que la médecine. À la fin des années 1970, le constructeur informatique **DEC** devait configurer sur mesure chaque commande de ses ordinateurs **VAX**, ce qui impliquait d'assembler des centaines de composants sans erreur ni oubli. Cette tâche a été confiée à un système expert, **XCON**. XCON configurait les commandes plus rapidement et avec moins d'erreurs que les employés, et il faisait **économiser des dizaines de millions de dollars par an** à DEC. Ce succès a montré que les systèmes experts pouvaient être rentables en dehors des laboratoires. Un problème est cependant apparu : le nombre de règles de XCON augmentait sans cesse, jusqu'à plusieurs milliers, et il fallait constamment les ajuster les unes aux autres. Nous y reviendrons plus bas.

Après XCON, les investissements se sont multipliés. Au début des années 1980, l'IA est devenue pour la première fois une **industrie**. On vendait des **coquilles** (*shells*), c'est-à-dire des moteurs d'inférence vides, prêts à recevoir la base de règles de n'importe quel domaine. Un nouveau métier est apparu, celui d'**ingénieur de la connaissance** (*knowledge engineer*), chargé de recueillir le savoir des experts. Des entreprises ont été créées et ont attiré des capitaux. On a même construit des ordinateurs spécialisés pour ces programmes, les **machines Lisp**. Le Japon a lancé un grand projet national, la **Cinquième Génération**, pour devenir le leader de ce type d'informatique. Les limites des systèmes experts commençaient toutefois à apparaître.

Une « machine Lisp » : le poste de travail Symbolics 3620 (1983), un ordinateur spécialisé conçu pour faire tourner les programmes d'IA symbolique.

Une machine Lisp Symbolics 3620 (1983), Computer History Museum. Photo : leighklotz, CC BY 2.0, via Wikimedia Commons.

Le goulot d'étranglement

La première limite, et la plus importante, concerne l'origine des règles. Quelqu'un doit les écrire. C'est le travail de l'**ingénieur de la connaissance**, qui interroge longuement un expert pour traduire son savoir en règles *si… alors*…. Ce travail s'est révélé très lent. De plus, une grande partie de l'expertise est difficile à exprimer. Le médecin expérimenté qui reconnaît un diagnostic au premier coup d'œil, ou le mécanicien qui identifie une panne au bruit du moteur, ne savent pas toujours expliquer ce qu'ils savent. Leur compétence est un **savoir tacite** (*tacit knowledge*), fondé sur l'intuition et l'expérience, et non une liste de règles qu'il suffirait de noter. On ne peut pas codifier ce qui n'a

jamais été formulé. Ce problème est connu sous le nom de **goulot d'étranglement de l'acquisition des connaissances** (*knowledge acquisition bottleneck*).

La deuxième limite, déjà évoquée dans « Représenter le monde », est la **rigidité**. Un système expert ne connaît que ses règles. Il fonctionne bien à l'intérieur de son domaine, mais dès qu'on en sort, il ne donne plus de résultats utiles, et il ne signale pas qu'il est hors de son domaine. MYCIN diagnostiquait les infections du sang. Si on lui avait décrit une jambe cassée, il aurait cherché un microbe, parce qu'il n'avait aucun moyen de savoir ce qu'il ignorait. Faute de **sens commun**, il pouvait aussi accepter des données absurdes (un patient âgé de moins de zéro an, une dose mille fois trop forte) qu'un étudiant en médecine aurait repérées immédiatement. Réduire le monde à un domaine restreint ne réglait donc pas le problème du sens commun.

La troisième limite est la difficulté de **maintenance**. Un système de quelques dizaines de règles reste gérable. XCON en a accumulé des milliers, et on a constaté qu'au-delà d'un certain nombre, les règles **interagissent** de façon imprévisible. Ajouter une règle pour corriger un cas pouvait en perturber plusieurs autres, sans qu'on s'en aperçoive. Maintenir une grande base de règles demandait plus de temps pour corriger ces effets de bord (*side effects*) que pour ajouter des connaissances. La connaissance explicite, écrite à la main, ne **passait pas à l'échelle**.

La quatrième limite est la plus fondamentale : un système expert n'**apprend rien**. Chacune de ses règles a été écrite par une personne. Le programme peut traiter mille cas sans en tirer une seule règle nouvelle, et il ne corrige pas de lui-même les règles erronées. Toute sa connaissance lui est fournie de l'extérieur, et elle reste figée. C'est à partir de cette limite que des chercheurs ont commencé à poser la question autrement. Puisqu'il est coûteux, voire impossible, d'extraire les règles des experts et de les maintenir à la main, on pourrait confier ce travail à la machine, en la laissant découvrir les règles à partir des données.

Cette approche ne s'imposera que plus tard. À la fin des années 1980, les résultats des systèmes experts restent en deçà des promesses. Les investisseurs se retirent, des entreprises ferment, et les machines Lisp sont abandonnées. L'IA entre dans une longue période de déclin, un **hiver**, décrit dans le chapitre « Les hivers et la bascule ».

Les hivers et la bascule

Pieter Bruegel l'Ancien, Les Chasseurs dans la neige (1565). L'expression « hiver de l'IA » désigne les périodes où l'intérêt et le financement pour la recherche en IA ont fortement diminué.

Pieter Bruegel l'Ancien, *Les Chasseurs dans la neige* (1565), Kunsthistorisches Museum, Vienne. Domaine public, via Wikimedia Commons.

Le premier hiver : la mort du perceptron (1969)

Le chapitre « Deux paris rivaux » a présenté le **perceptron** de Rosenblatt, une machine qui apprend de ses erreurs, que le *New York Times* avait décrit comme le début d'une intelligence électronique. Ce chapitre annonçait aussi que le perceptron allait subir une critique sévère. Cette section présente cette critique.

En **1969**, deux chercheurs importants du camp symbolique, **Marvin Minsky** (l'un des organisateurs de Dartmouth) et son collègue **Seymour Papert**, publient un livre intitulé *Perceptrons*. Il ne s'agit pas d'un pamphlet, mais d'une analyse mathématique rigoureuse. Minsky et Papert y démontrent que le perceptron a une limite de principe : il est incapable d'apprendre certaines fonctions pourtant très simples. L'exemple le plus connu est le **XOR**, le « ou exclusif ».

Le XOR correspond à la règle suivante : l'un ou l'autre, mais pas les deux à la fois. C'est par exemple le principe du **va-et-vient**, où deux interrupteurs commandent une même lampe, souvent aux deux bouts d'un couloir. Quand on bascule l'un ou l'autre interrupteur, la lumière change d'état. Si les deux interrupteurs sont dans la même position, la lampe est éteinte ; s'ils sont dans des positions opposées, elle est allumée. La réponse dépend donc du désaccord entre les deux entrées.

On peut résumer le XOR dans une **table de vérité** (*truth table*), qui donne la réponse pour chaque combinaison possible des deux entrées. On note 0 pour « bas » (ou « faux ») et 1 pour « haut » (ou « vrai ») :

Interrupteur A	Interrupteur B	Lampe (A XOR B)
0	0	0 (éteinte)
0	1	1 (allumée)
1	0	1 (allumée)
1	1	0 (éteinte)

La sortie vaut 1 seulement quand les deux entrées diffèrent. Si l'on place ces quatre cas sur un plan, avec A à l'horizontale et B à la verticale, les deux cas « allumé » occupent deux coins opposés d'un carré, et les deux cas « éteint » les deux autres coins. Cette règle est facile à comprendre pour un humain, mais le perceptron ne peut pas l'apprendre, parce qu'il classe les cas en traçant une seule droite, et qu'aucune droite ne sépare ces deux paires de coins. Le Module 2 reprend cet exemple avec une figure, et montre comment d'autres modèles réussissent là où une droite échoue.

Le livre eut des conséquences importantes. Minsky avait une grande autorité, et il appartenait au camp adverse. Sa démonstration, mathématiquement correcte, fut interprétée comme une condamnation de toute l'approche. Les financements de la recherche sur les réseaux de neurones diminuèrent très rapidement, les revues publièrent moins de travaux sur le sujet et les étudiants choisirent d'autres domaines. Le camp connexionniste connut un **hiver** d'une quinzaine d'années. Rosenblatt n'en vit pas la fin : il mourut en 1971, à quarante-trois ans, dans un accident de bateau.

Le premier hiver de l'IA ne toucha donc pas la tradition dominante, mais sa **rivale**, et c'est un chercheur du camp symbolique qui en fut en partie la cause. Comme le connexionnisme était mis de côté, les vingt années suivantes furent celles d'une **domination symbolique** presque complète : la recherche dans un espace d'états, les systèmes experts et l'ensemble des travaux présentés dans les chapitres « Chercher et raisonner », « Représenter le monde » et « Capturer l'expertise ». Le déclin d'une tradition a ainsi favorisé le développement de l'autre.

La démonstration de Minsky et Papert avait cependant une limite. Elle ne valait que pour le perceptron le plus simple, formé d'une **seule couche**. On pensait qu'en **empilant plusieurs couches**, on pourrait traiter le XOR, mais on ne savait pas comment faire apprendre un tel réseau. La solution n'apparut qu'en 1986, et c'est elle qui permit la reprise des travaux connexionnistes. Avant d'y arriver, il faut d'abord voir comment le camp symbolique connut lui aussi un hiver.

Le second hiver : l'effondrement du symbolique (fin des années 1980)

L'âge d'or symbolique reposait sur une promesse ambitieuse : capturer l'expertise humaine dans des règles. Les limites décrites dans « Capturer l'expertise » (le savoir tacite difficile à extraire, la rigidité sans sens commun, les bases de règles difficiles à maintenir) finirent par affaiblir l'ensemble de l'approche. Comme les systèmes experts livrés décevaient souvent, l'écart grandit entre ce qu'on avait promis aux investisseurs et ce que l'IA réalisait vraiment. Quand cet écart devint trop visible, les investissements diminuèrent.

L'événement le plus représentatif de ce recul fut le **krach des machines Lisp**, vers 1987. Ces ordinateurs spécialisés, conçus pour l'IA, coûtaient très cher. En peu de temps, des stations de travail bon marché, puis les ordinateurs personnels, offrirent des performances comparables pour une fraction du prix. Le marché s'effondra et les entreprises qui en dépendaient disparurent. À la même époque, le projet japonais de **Cinquième Génération**, lancé en 1982 avec beaucoup de publicité, se terminait sans avoir atteint ses principaux objectifs.

Le découragement toucha tout le domaine. Les agences de financement réduisirent leurs crédits, et l'étiquette « intelligence artificielle » devint si mal perçue que les chercheurs évitaient de l'utiliser dans leurs demandes de subvention. On parle, pour cette période, d'un second **hiver de l'IA**. Le terme s'appliquait maintenant au camp symbolique, comme il s'était appliqué vingt ans plus tôt au perceptron. La leçon était la même que dans « Représenter le monde » et « Capturer l'expertise » : le monde réel se laisse difficilement décrire par des règles écrites à la main.

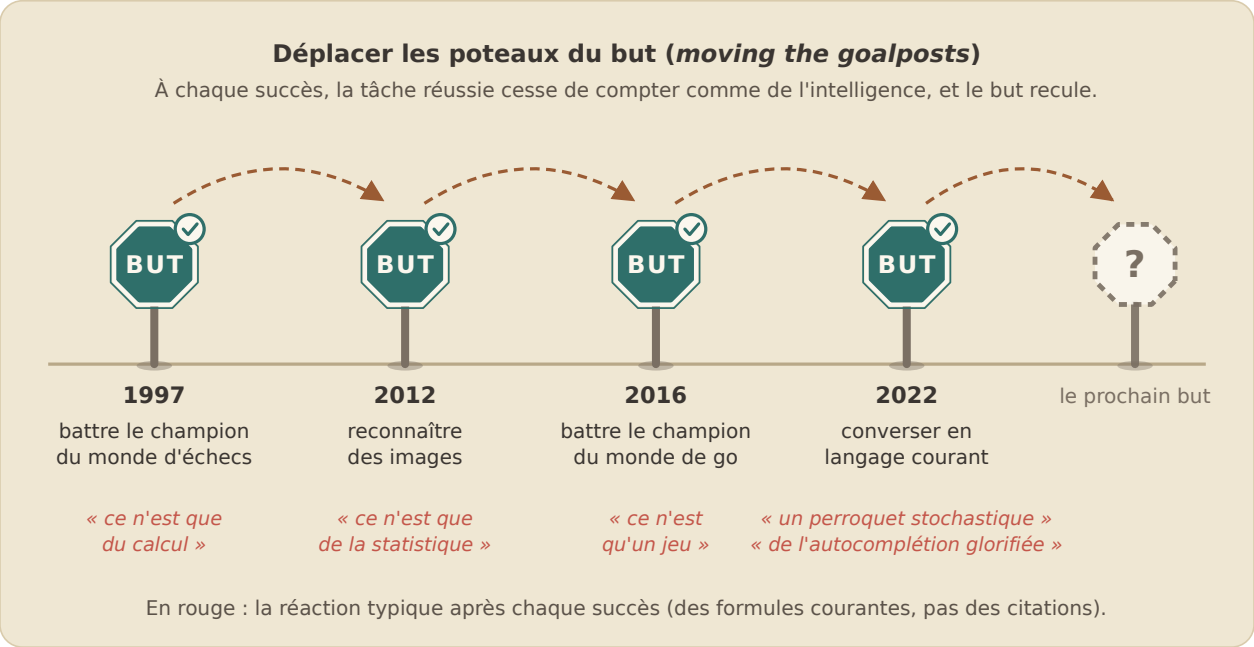
Pendant que le camp symbolique déclinait, le camp connexionniste, qu'on croyait abandonné depuis 1969, **reprenait de l'activité**. En 1986, un petit groupe de chercheurs, dont **Geoffrey Hinton**, avait publié une méthode appelée **rétropropagation** (*backpropagation*). Elle permettait de faire apprendre les réseaux à **plusieurs couches**, c'est-à-dire justement ceux qui, selon les intuitions de l'époque, pouvaient traiter le XOR. L'obstacle signalé dans *Perceptrons* était donc levé. Ce retour resta toutefois discret au début : il fallut attendre les **données** et la **puissance de calcul** des années 2010 pour qu'il devienne largement visible. Les deux traditions échangeaient de nouveau leurs rôles : le déclin de l'une coïncidait avec la reprise de l'autre.

L'héritage invisible

Après deux hivers successifs, on pourrait conclure que le GOFAI a échoué. Cette conclusion serait cependant incomplète, à cause d'un phénomène que les chercheurs appellent l'**« effet IA »** (*AI effect*) : dès qu'une technique fonctionne bien, on cesse de l'appeler « intelligence artificielle » et on la considère comme « juste un algorithme ». La formule la plus courte est attribuée à l'informaticien Larry Tesler, et c'est **Hofstadter** qui l'a popularisée : « *l'IA, c'est tout ce qui n'a pas encore été fait.* »

Les anglophones ont une expression pour ce phénomène : *moving the goalposts*, « déplacer les poteaux du but ». L'image vient du sport. Si l'on recule le but chaque fois qu'un joueur s'apprête à marquer, il ne marquera jamais, quelle que soit sa performance. L'histoire de l'IA suit ce schéma. Avant 1997, on tenait le jeu d'échecs pour un bon test de l'intelligence. Quand Deep Blue a battu Kasparov, on a conclu que les échecs ne demandaient finalement que du calcul. Le même raisonnement a été repris pour la reconnaissance d'images, pour le jeu de go, puis pour la conversation. Les grands modèles de langage ont ainsi été décrits comme des « perroquets stochastiques » (*stochastic parrots*), selon l'expression de la linguiste Emily Bender et de ses coauteures en 2021 : des systèmes qui assemblent des suites de mots probables sans rien comprendre de ce qu'ils disent. On dit aussi

qu'ils ne font que de l'« autocomplétion glorifiée » (*glorified autocomplete*), comme la fonction qui propose le mot suivant sur un téléphone. À chaque succès, la tâche réussie est retirée de la définition de l'intelligence, et le but est placé un peu plus loin. Ce déplacement n'est pas toujours injustifié, parce qu'un succès révèle parfois que la tâche était plus facile qu'on ne le croyait. Mais il explique pourquoi les réussites du GOFAI ont cessé d'être comptées comme de l'IA.



Déplacer les poteaux du but : à chaque succès de l'IA, la tâche réussie cesse de compter comme de l'intelligence.

De ce point de vue, le GOFAI n'a pas disparu : il s'est **intégré** à l'informatique courante. Ses réussites sont devenues si ordinaires et si fiables qu'on a oublié qu'elles venaient des laboratoires d'IA. Les quatre exemples suivants sont des techniques que vous utilisez probablement chaque jour.

La recherche dans un arbre. Les algorithmes d'exploration (minimax, A*) qui permettaient aux machines de jouer aux échecs sont aujourd'hui très répandus. Ce sont eux qui calculent votre itinéraire **GPS** en une fraction de seconde, qui dirigent les personnages des jeux vidéo et qui optimisent les tournées d'un transporteur ou les mouvements d'un robot. Les personnes qui suivent un itinéraire sur leur téléphone ne pensent généralement pas qu'elles utilisent une technique d'« intelligence artificielle » des années 1960.

Les moteurs de règles. Les systèmes experts n'ont pas disparu, mais ils ont changé de nom. On parle aujourd'hui de « règles métier » (*business rules*), et ces systèmes décident automatiquement de l'octroi d'un prêt, du repérage d'une transaction frauduleuse ou du calcul d'une prime d'assurance. Les **configurateurs** qui, sur un site marchand, vérifient que les options d'une voiture ou d'un ordinateur sont compatibles descendent directement de XCON. De même, un logiciel d'**impôts** qui vous guide de question en question jusqu'au bon formulaire fait le même travail que MYCIN.

Les idées de Lisp. Cet héritage est peut-être le plus important. Le langage de McCarthy a introduit dans la programmation courante la **programmation fonctionnelle** : traiter les fonctions comme des valeurs (les *lambdas*), enchaîner des opérations comme `map`, `filter`, `reduce`, utiliser des *closures*. Ces constructions, courantes aujourd'hui en Python, en JavaScript ou en Java, viennent des laboratoires d'IA. Lisp a aussi introduit des outils qu'on considère maintenant comme normaux : le **ramasse-**

miettes (*garbage collection* : la gestion automatique de la mémoire) et le **REPL**, la console qui permet d'essayer du code au fur et à mesure. L'autre grande tradition fonctionnelle, celle des langages typés comme Haskell, descend du langage **ML**, que Robin Milner avait créé pour programmer un **assistant de démonstration de théorèmes** (*proof assistant*). Les deux sources de la programmation fonctionnelle moderne viennent donc du raisonnement symbolique.

La représentation des connaissances. Les réseaux sémantiques (*semantic networks*) et les frames, présentés plus tôt, ont eu des successeurs. Les **ontologies** et les **knowledge graphs** qui structurent le savoir du web en sont les héritiers directs. Quand Google affiche une fiche résumée à côté des résultats, ou quand on interroge Wikidata, c'est la même idée qui est utilisée : relier des concepts par des liens *est-un* ou *possède*. Entre les deux, il y a eu une tentative ambitieuse, qui mérite d'être mentionnée : le **web sémantique**. En 2001, Tim Berners-Lee, l'inventeur du Web, propose d'en faire une grande base de connaissances lisible par les machines. Chaque page contiendrait non seulement du texte pour les humains, mais aussi des faits structurés pour les programmes (*cette personne est née en telle année*, *ce produit coûte tant*), reliés par des vocabulaires communs, c'est-à-dire des ontologies. Des normes sont écrites dans ce but (RDF, OWL). Le projet complet ne s'est pas réalisé, pour une raison déjà vue dans « Représenter le monde » : il supposait que des millions d'auteurs décrivent soigneusement le sens de ce qu'ils publient. C'est le même travail d'inscription à la main qui avait limité CYC, mais à l'échelle de la planète. Certaines de ses composantes ont cependant été conservées : les balises que les sites ajoutent aujourd'hui pour que les moteurs de recherche comprennent une recette ou un horaire, ainsi que Wikidata, en sont les héritiers directs. Quant aux frames, avec leurs cases à valeurs par défaut et leurs hiérarchies d'héritage, elles sont **apparentées** à l'**objet** de la programmation moderne. Elles n'en sont pas l'ancêtre (l'orienté-objet doit davantage à la simulation qu'à l'IA), mais elles reposent sur la même intuition. Une dernière remarque prépare le module 4 : tout ce savoir est **structuré à la main** par des humains. C'est l'opposé de la façon dont les grands modèles de langage acquièrent leur savoir, en traitant d'énormes quantités de texte. Ces deux conceptions du savoir seront comparées plus loin dans le cours.

Ces quatre exemples ne sont pas les seuls. Aucune de ces techniques ne porte plus l'étiquette « IA », parce qu'elles font maintenant partie de l'informatique ordinaire. Le GOFAI n'a donc pas été une impasse : ses idées se sont **dispersées** hors du domaine de l'intelligence artificielle et sont devenues des éléments courants et indispensables de la programmation.

Un éclair hybride : Watson (2011)

Un dernier exemple montre que le GOFAI ne s'est pas seulement intégré à l'informatique courante, mais qu'il est aussi réapparu, combiné à d'autres méthodes. En février **2011**, un système d'IBM nommé **Watson** affronte, au jeu télévisé américain *Jeopardy!*, les deux meilleurs champions de l'histoire de l'émission, Ken Jennings et Brad Rutter, et les bat nettement.

Cette réussite est d'une autre nature que celle de Deep Blue. Les échecs ont des règles précises et un espace de recherche (*search space*) bien défini. *Jeopardy!*, au contraire, repose sur un **langage** difficile, fait de calembours, d'allusions, de jeux de mots et d'indices indirects. Comprendre ce que la question demande est déjà un problème difficile. Watson y répond en quelques secondes, en utilisant une très grande base de connaissances.

Watson n'est **ni du GOFAI pur, ni un réseau de neurones**. C'est un **hybride**. D'une part, il reprend des éléments du symbolique : une grande base de connaissances et des traitements du langage à base de règles. D'autre part, il pondère des centaines d'indices et calcule sa **confiance** avec des

méthodes **statistiques**, apprises sur des milliers de questions passées. Quatorze ans après Deep Blue, ce n'est donc plus seulement la force brute de la recherche qui permet de gagner, mais une combinaison du savoir structuré du GOFAI et de l'apprentissage statistique. Watson illustre ainsi la **charnière** où se trouve maintenant le module, entre deux périodes.

La suite de l'histoire de Watson ressemble à celle des systèmes experts. Après le succès de 2011, IBM annonça que *Watson Health* allait **transformer la médecine** : la machine devait conseiller les cancérologues mieux que leurs collègues. Les résultats furent très différents. Certaines recommandations se révélèrent douteuses, voire dangereuses. De grands hôpitaux partenaires abandonnèrent le projet après y avoir dépensé des dizaines de millions de dollars, et IBM finit par **revendre** Watson Health en 2022. Trente ans après MYCIN, qui avait bien fonctionné en démonstration sans jamais être utilisé auprès de vrais patients, la même leçon se répétait : l'écart reste très grand entre une réussite en laboratoire (ou sur un plateau de télévision) et le monde réel. Une fois de plus, une promesse trop ambitieuse avait précédé la déception.

La bascule

Voici un résumé du chemin parcouru. Le chapitre « Turing et la question fondatrice » est parti d'une hypothèse : penser, c'est calculer. De cette hypothèse est née une famille d'idées (manipuler des symboles selon des règles) qui a produit, pendant trois décennies, des réussites importantes : des machines qui démontrent des théorèmes, qui jouent aux échecs, qui dialoguent, qui représentent le monde et qui établissent des diagnostics comme des médecins. Le programme symbolique n'a donc pas été un échec : il a fondé l'informatique du raisonnement, qui est encore utilisée aujourd'hui.

Il s'est cependant heurté deux fois à la même difficulté. Dans « Représenter le monde », il s'agissait du **sens commun**, l'ensemble des évidences que personne ne pense à formuler et qu'on ne peut donc pas écrire. Dans « Capturer l'expertise », il s'agissait du **goulot d'étranglement** : l'expertise est difficile à mettre en règles, parce qu'une grande partie du savoir humain n'est pas exprimée. Ces deux difficultés relèvent d'une même leçon, que Hofstadter avait peut-être formulée le premier : on ne peut pas inscrire l'intelligence de l'extérieur, fait après fait, parce que le savoir utile est trop vaste, trop tacite et trop changeant pour tenir dans une liste de règles.

Si l'on ne peut pas dicter ce savoir à la machine, il faut qu'elle l'**acquière elle-même**. Cette idée n'était pas nouvelle : elle se trouvait déjà en 1958 dans le perceptron de Rosenblatt, une machine qui apprend de ses erreurs, que le camp symbolique pensait avoir écarté en 1969. C'est ici que se termine le fil suivi depuis « Deux paris rivaux ». Les deux traditions, symbolique et connexionniste, ne se sont jamais simplement succédé : elles ont coexisté en rivales, et chacune a dominé à son tour. L'hiver symbolique de la fin des années 1980 n'est donc pas la fin de l'histoire, mais le moment où l'avantage commence à revenir vers l'autre tradition.

Une question, apparue avec les difficultés du GOFAI, va réorganiser le domaine : au lieu de dicter ses règles à la machine, on peut la laisser les découvrir dans les données. Ce changement d'approche s'appelle l'**apprentissage automatique**, et c'est l'objet du **Module 2**. La tradition connexionniste, relancée par la rétropropagation et la puissance des machines modernes, le mènera ensuite, au **Module 3**, à des résultats que ni Turing, ni Rosenblatt, ni Minsky n'avaient imaginés.

Le module peut se résumer par le passage d'un verbe à un autre. Pendant trois décennies, être intelligent, pour une machine, a surtout voulu dire **chercher** : explorer un espace de possibilités, guidé par des règles fixées à l'avance. La période suivante remplace ce verbe par **apprendre**. Les

deux verbes restent cependant proches, et la suite du cours y reviendra : apprendre, c'est encore chercher, mais dans un autre espace. Au lieu d'explorer les coups d'une partie, on explore l'ensemble des réglages possibles d'un modèle, jusqu'à trouver ceux qui correspondent aux données. Le GOFAI cherchait la solution, alors que l'apprentissage cherche de quoi la produire. La période de la recherche se termine donc, et celle de l'apprentissage, qui est une autre forme de recherche, commence.

Vous êtes le moteur d'inférence : un système expert (travail noté 1)

Au Module 1, nous avons vu qu'avant l'apprentissage automatique (que nous commencerons à étudier en détail au module 2), la grande idée de l'IA « classique » (le GOFAI) était de **capturer la connaissance dans des règles explicites**. Les **systèmes experts** sont l'aboutissement de cette idée. Ils comportent une **base de règles** `si… alors…`, une **base de faits** (le cas qu'on traite) et un **moteur d'inférence**, qui compare les deux, déclenche les règles applicables et en tire de nouvelles conclusions, jusqu'au diagnostic final.

Dans ce travail, vous allez jouer le rôle de ce moteur d'inférence sur un petit système expert qui **identifie des animaux** à partir de leurs caractéristiques. Vous allez le faire fonctionner, trouver un cas où il échoue, puis tenter de l'améliorer. Vous observerez ainsi directement ce qu'on a appelé le **goulot d'étranglement de la connaissance**.

Une règle a la forme suivante :

```
SI  (mammifère)  ET  (mange de la viande)   ALORS  (carnivore)
```

Les conditions d'une règle sont **toujours reliées par des ET, jamais par des OU**. Une règle ne se déclenche donc que si **toutes** ses conditions sont vraies en même temps. Dans l'exemple ci-dessus, `mammifère` **et** `mange de la viande` doivent être vrais tous les deux pour qu'on conclue `carnivore`. Une seule des deux conditions ne suffit pas. Pour exprimer un « OU » (par exemple `a du poil` **ou** `allaite ses petits` → `mammifère`), on écrit **deux règles séparées**. C'est ce que fait la base avec R1 et R2.

Le « savoir » du système n'est qu'un ensemble de règles de ce type. Son « raisonnement » est un mécanisme simple qui vérifie quelles règles ont toutes leurs conditions réunies. Comme nous le verrons, toute l'intelligence du système tient dans une seule formule.

Consignes

1. Ouvrez le fichier Google Sheets fourni pour ce travail et **faites-en une copie** (*Fichier › Créer une copie*). Vous ferez tout votre travail à partir de **votre copie**.
2. Une fois vos manipulations terminées, partagez votre fichier et copiez le lien vers celui-ci dans un document PDF (**Attention : aucun autre format que PDF ne sera accepté**).
3. Répondez aux questions d'interprétation de la dernière section dans le même fichier PDF, en fournissant des réponses claires et précises.

Le système

Le classeur ne comporte que **deux onglets**. Il constitue le système expert réduit à l'essentiel, c'est-à-dire une base de faits et une base de règles.

- **Faits** : la liste de toutes les caractéristiques possibles (les *attributs observables* comme `a du poil`, mais aussi les *catégories* déduites comme `mammifère` et les *espèces*). Chaque fait a une case **Vrai ?** à cocher.
- **Base de règles** : les 11 règles `si… alors…`, sous forme de conditions et d'une conclusion.

L'élément central du système est la colonne « **Activable ?** » de la *Base de règles*. Pour chaque règle, cette formule vérifie que **toutes ses conditions sont vraies** dans l'onglet *Faits* et que sa conclusion n'a pas déjà été établie :

```
=IF($F3="","",AND(
   IF($B3="",TRUE,VLOOKUP($B3,Faits!$A:$B,2,FALSE)=TRUE),
   IF($C3="",TRUE,VLOOKUP($C3,Faits!$A:$B,2,FALSE)=TRUE),
   IF($D3="",TRUE,VLOOKUP($D3,Faits!$A:$B,2,FALSE)=TRUE),
   IF($E3="",TRUE,VLOOKUP($E3,Faits!$A:$B,2,FALSE)=TRUE),
   VLOOKUP($F3,Faits!$A:$B,2,FALSE)<>TRUE))
```

Cette formule est celle de la règle R1, qui occupe la ligne 3 du tableau. Chaque `VLOOKUP` cherche un fait dans l'onglet *Faits* et lit sa case **Vrai ?**. La colonne voisine, « **Déjà conclue ?** », indique si la conclusion de la règle est déjà cochée. Un **bandeau d'état**, sur la première ligne de l'onglet, résume où vous en êtes : il vous invite à continuer tant qu'une règle est verte, il annonce l'animal identifié, ou il signale que le système est bloqué.

Quand une règle est activable, sa ligne est **surlignée en vert**, ce qui signifie qu'elle est prête à se déclencher. C'est vous qui jouez le rôle du moteur. Vous choisissez une règle verte et vous **cochez sa conclusion** dans l'onglet *Faits*. Cette action rend d'autres règles vertes, et ainsi de suite.

Dans un vrai système expert, ce travail est entièrement **automatique**. Un programme, le *moteur d'inférence*, parcourt la base de règles, repère celles dont toutes les conditions sont réunies, les déclenche, ajoute leurs conclusions à la base de faits, puis recommence, jusqu'à ce qu'aucune règle ne s'applique plus. L'humain n'intervient pas. Ici, **pour des raisons pédagogiques, c'est vous qui tenez ce rôle**. Vous exécutez à la main l'algorithme que la machine exécuterait seule. Le but est de vous faire constater à quel point ce raisonnement est mécanique. Il s'agit d'une simple boucle qui applique des règles, sans aucune compréhension de ce qu'est un guépard ou un manchot.

1	▶ Continuez : déclenchez une règle surlignée en vert							
2	Règle	Condition 1	Condition 2	Condition 3	Condition 4	→ Conclusion	Activable ? (calculé)	Déjà conclue ? (calculé)
3	R1	a du poil				mammifère	TRUE	FALSE
4	R2	allaite ses petits				mammifère	FALSE	FALSE
5	R3	a des plumes				oiseau	FALSE	FALSE
6	R4	mammifère	mange de la vianc			carnivore	FALSE	FALSE
7	R5	mammifère	a des dents pointu	a des griffes		carnivore	FALSE	FALSE
8	R6	carnivore	robe fauve	taches sombres		guépard	FALSE	FALSE
9	R7	carnivore	robe fauve	rayures noires		tigre	FALSE	FALSE
10	R8	mammifère	rumine	long cou	taches sombres	girafe	FALSE	FALSE
11	R9	mammifère	a des sabots	rayures noires		zèbre	FALSE	FALSE
12	R10	oiseau	incapable de volei	long cou		autruche	FALSE	FALSE
13	R11	oiseau	incapable de volei	nage	plumage noir et blan	manchot	FALSE	FALSE

Le Cas 1 au départ : seule la règle R1 est activable, et sa ligne est surlignée en vert.

Manipulation 1 — Tracer le raisonnement

Voici le **Cas 1** : cochez `a du poil`, `mange de la viande`, `robe fauve` et `taches sombres` dans l'onglet *Faits* (laissez tout le reste à FAUX).

Une seule règle devient verte, **R1**. Déclenchez-la (cochez sa conclusion, `mammifère`, dans l'onglet *Faits*). Une nouvelle règle s'active. Déclenchez-la à son tour, et continuez jusqu'à ce qu'une **espèce** passe à VRAI.

Au fur et à mesure, **notez la trace** de votre raisonnement directement dans votre document PDF, sous la forme d'un petit tableau, avec une ligne par règle déclenchée :

Étape	Règle déclenchée	Nouveau fait établi	Pourquoi (conditions réunies)
1	R1	`mammifère`	`a du poil` est vrai
2	…	…	…

Recommencez ensuite avec le **Cas 2**, le manchot (`a des plumes`, `incapable de voler`, `nage`, `plumage noir et blanc`), après avoir remis tous les faits à FAUX.

> **Pour remettre les faits à FAUX**, décochez les cases une à une en cliquant dessus. N'utilisez pas la touche Suppr (ou Retour arrière) sur les cases, parce qu'elle supprime la case à cocher au lieu de la décocher. Si cela vous arrive, annulez avec Ctrl+Z (Cmd+Z sur Mac), ou recréez les cases avec *Insertion › Case à cocher*.

Manipulation 2 — Casser le système

Remettez tous les faits à FAUX, puis chargez le **Cas 3** : `a du poil`, `a des sabots`, `rumine`.

Faites fonctionner le moteur. Vous obtenez `mammifère`, puis plus aucune règle ne s'active. Aucune règle d'espèce ne devient verte. Le système est **bloqué**, parce qu'il ne sait pas identifier cet animal (il s'agit, par exemple, d'une **antilope**).

Manipulation 3 — Étendre la base de règles

Vous devez maintenant ajouter le savoir manquant. Sur la première ligne vide de la *Base de règles*, écrivez une nouvelle règle qui conclut `antilope`. Utilisez les **menus déroulants** pour les conditions.

L'espèce `antilope` n'est pas encore dans l'onglet *Faits*. Ajoutez-la sur la première ligne vide, en remplissant les trois colonnes :

- son nom, `antilope`, dans la colonne *Fait* ;
- une case à cocher dans la colonne *Vrai ?* (*Insertion › Case à cocher*) ;
- la mention `espèce (conclusion)` dans la colonne *Catégorie*. Sans cette mention, le bandeau d'état ne reconnaît pas l'antilope comme une espèce, et il continue d'afficher que le système est bloqué.

Procédez de la même façon pour tout nouvel attribut dont vous auriez besoin, avec la mention `observable` dans la colonne *Catégorie*.

Vérifiez que votre règle identifie bien le Cas 3. Ensuite, **testez sa solidité** : chargez un autre animal (par exemple le **zèbre** : `a du poil`, `a des sabots`, `rayures noires`). Votre nouvelle règle se déclenche-t-elle aussi, par erreur ? Modifiez-la si nécessaire.

1	▶ Continuez : déclenchez une règle surlignée en vert							
2	Règle	Condition 1	Condition 2	Condition 3	Condition 4	→ Conclusion	Activable ? (calculé)	Déjà conclue ? (calculé)
3	R1	a du poil				mammifère	FALSE	TRUE
4	R2	allaite ses petits				mammifère	FALSE	TRUE
5	R3	a des plumes				oiseau	FALSE	FALSE
6	R4	mammifère	mange de la viand			carnivore	FALSE	FALSE
7	R5	mammifère	a des dents pointu	a des griffes		carnivore	FALSE	FALSE
8	R6	carnivore	robe fauve	taches sombres		guépard	FALSE	FALSE
9	R7	carnivore	robe fauve	rayures noires		tigre	FALSE	FALSE
10	R8	mammifère	rumine	long cou	taches sombres	girafe	FALSE	FALSE
11	R9	mammifère	a des sabots	rayures noires		zèbre	FALSE	FALSE
12	R10	oiseau	incapable de voler	long cou		autruche	FALSE	FALSE
13	R11	oiseau	incapable de voler	nage	plumage noir et blan	manchot	FALSE	FALSE
14	R12	mammifère	a des sabots	rumine		antilope	TRUE	
15								
16								

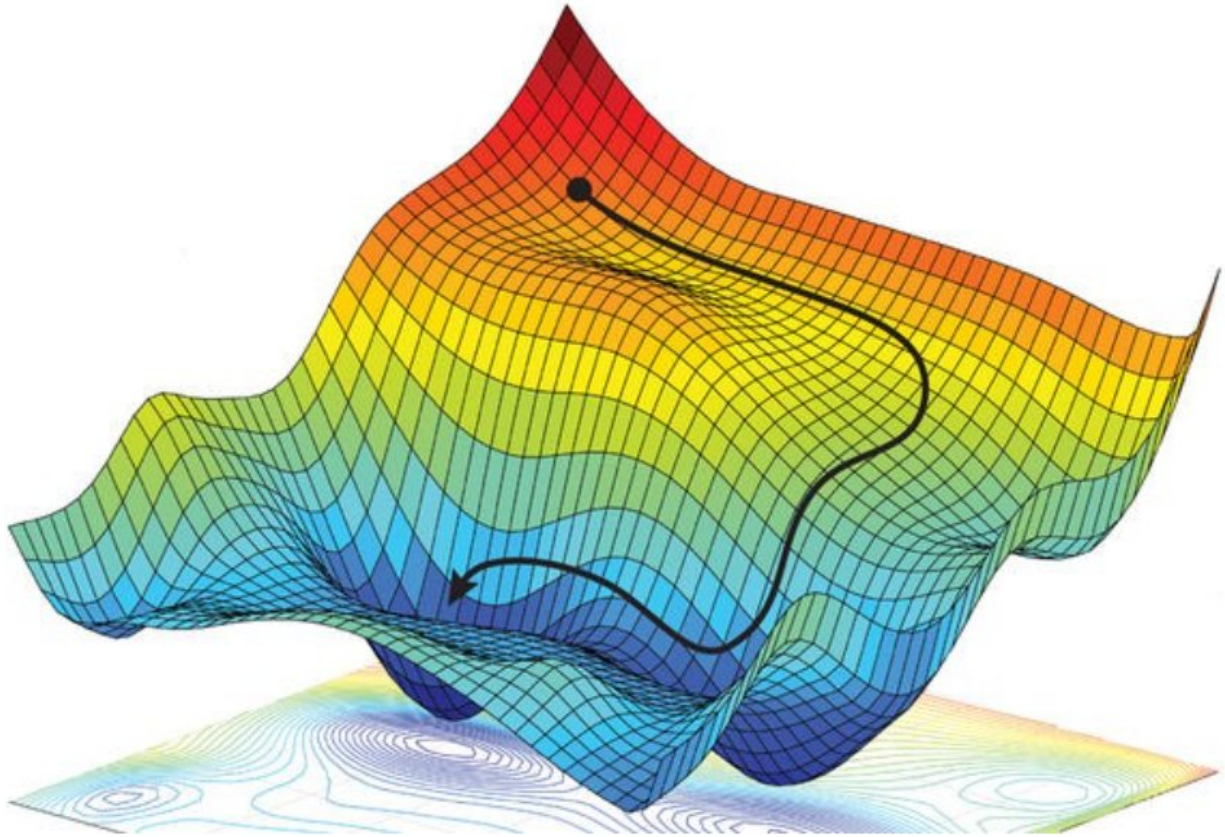
Une nouvelle règle, R12, ajoutée sur la première ligne vide : elle devient activable pour le Cas 3.

Questions d'interprétation

1. Pour le Cas 1, combien de règles se sont déclenchées, et dans quel ordre ? Décrivez la **cascade** : en quoi la conclusion d'une règle sert-elle de condition à une autre ?
2. Dans ce système, le « savoir » (la base de règles) et le « raisonnement » (le moteur) sont **séparés**. Si l'on remplaçait entièrement la base de règles par des règles de diagnostic automobile, qu'est-ce qui changerait, et qu'est-ce qui resterait identique ?
3. Le système peut **expliquer** sa conclusion (le tableau de trace que vous avez rempli le montre). En quoi est-ce une force ? Comparez avec ce que vous anticipez d'un réseau de neurones (Module 3).
4. Vous avez fait fonctionner le moteur « vers l'avant » (des faits vers la conclusion), ce qu'on appelle le **chaînage avant**. Comment auriez-vous procédé pour vérifier une hypothèse précise (« et si c'était un tigre ? ») sans tout cocher d'avance, c'est-à-dire en **chaînage arrière** ?
5. Au Cas 3, pourquoi exactement le système s'est-il bloqué ? Qu'est-ce que cela révèle sur sa capacité à gérer un cas **imprévu** ?
6. La règle que vous avez ajoutée risquait-elle de **mal classer** d'autres animaux ? Décrivez ce que vous avez observé en la testant sur le zèbre, et ce que cela indique sur la difficulté de maintenir une grosse base de règles.
7. Supposons qu'on veuille étendre ce système pour identifier **tous les animaux du monde**. Décrivez concrètement ce qui se passerait pour la base de règles. Quel concept des chapitres « Capturer l'expertise » et « Les hivers et la bascule » cela illustre-t-il ?
8. Un zoologiste reconnaît souvent un animal « d'un coup d'œil », sans pouvoir énoncer la règle exacte qu'il applique. Quel nom donne-t-on à ce type de savoir, et pourquoi pose-t-il problème à un système expert ?

9. Ce système **apprend-il** de son expérience ? Si on lui présentait mille animaux, ses règles s'amélioreraient-elles d'elles-mêmes ? Quelle conséquence cela a-t-il, et vers quel changement de paradigme (Module 2) cela conduit-il ?
10. Toute « l'intelligence » du moteur tient dans la formule de la colonne *Activable ?*. Diriez-vous que ce système pense ? Reliez votre réponse au test de Turing et à l'« effet IA ».
11. Les systèmes experts ont presque disparu sous ce nom, mais leur mécanisme existe toujours (voir « Les hivers et la bascule »). Donnez un exemple, tiré de la vie courante, d'un endroit où une base de règles `si… alors…` décide encore quelque chose à votre sujet.

Module 2 — Apprentissage automatique

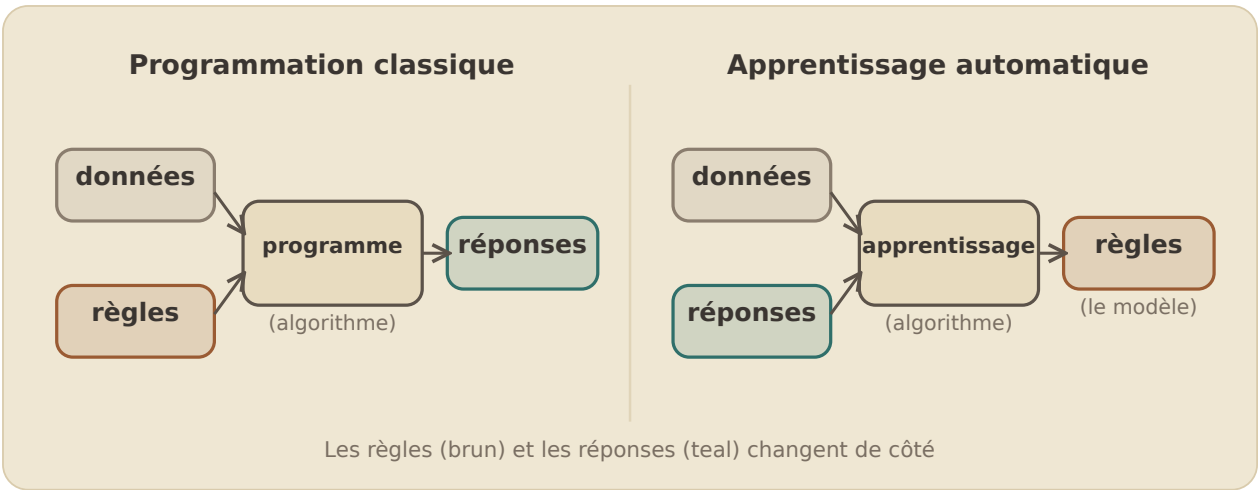


Qu'est-ce que l'apprentissage automatique ?

L'apprentissage automatique (AA, ou *machine learning*) est un ensemble de techniques qui permettent à un ordinateur de résoudre des problèmes qu'il serait très difficile de programmer à la main : reconnaître un chat sur une photo, estimer le prix d'une maison, filtrer les pourriels, jouer aux échecs, converser, etc.

La différence avec la programmation classique est importante. Un programme traditionnel encode une série de **règles** écrites par un humain. Un modèle d'AA, au contraire, **dérive son fonctionnement à partir d'exemples** : on ne lui donne pas la règle, on la lui fait découvrir dans les données. C'est ce passage des règles aux exemples qui rapproche l'AA de ce qu'on appelle l'intelligence. Il le rattache aussi au **connexionnisme**, présenté au module 1 : l'hypothèse, rivale de l'approche symbolique, selon laquelle l'intelligence ne s'écrit pas en règles explicites, mais **émerge d'un réseau de connexions qui s'ajustent à l'expérience**. L'apprentissage automatique reprend cette idée principale, sans nécessairement garder les neurones. Ce qui s'ajuste ici s'appelle des **paramètres**, et c'est en les réglant sur des exemples que le modèle finit par « savoir » quelque chose.

On peut aussi dire que les deux démarches **échangent leurs entrées et leurs sorties**. En programmation classique, un humain écrit les règles, et l'ordinateur les applique aux données pour produire des réponses. En apprentissage automatique, on fournit à l'ordinateur les données **et** les réponses (ce sont les exemples), et c'est l'ordinateur qui en dégage les règles, qu'on appelle alors le **modèle**. Ce qui était fourni par l'humain (les règles) devient le résultat, et ce qui était le résultat (les réponses) devient une donnée d'entrée. Le schéma ci-dessous montre cette permutation : suivez les « règles » (en brun) et les « réponses » (en teal), qui passent d'un côté à l'autre.

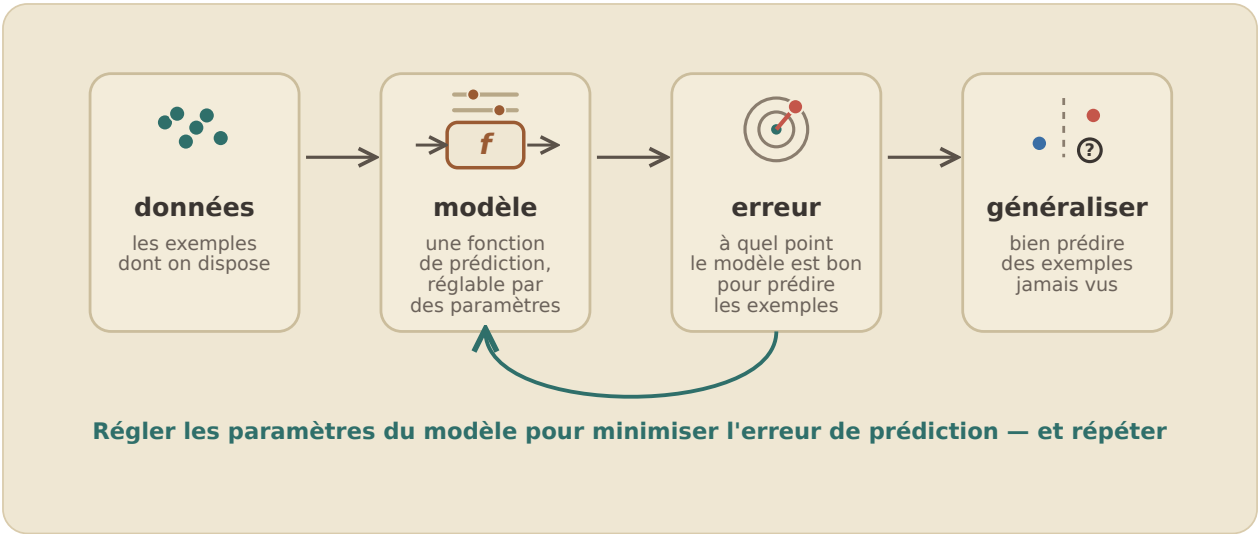


L'inversion au centre de l'apprentissage automatique : ce qu'on fournit et ce que l'ordinateur produit s'inversent. En programmation classique, on écrit les règles ; en AA, c'est l'ordinateur qui les découvre.

Une seule grande idée, déclinée

Au lieu de présenter un catalogue d'algorithmes, ce module construit **un modèle mental unique et transférable**, qui revient de page en page :

> partir de **données** → choisir un **modèle** (une fonction de prédiction, réglable par des paramètres) → mesurer son **erreur** de prédiction → régler les paramètres du modèle pour la **minimiser** → vérifier qu'il **généralise** à des cas nouveaux.



La structure du module : données → modèle → erreur → généraliser, avec au centre la boucle d'entraînement (régler les paramètres du modèle pour minimiser l'erreur de prédiction, de façon répétée).

Chaque algorithme classique présenté dans ce module, du plus simple au plus élaboré, est une **variation** sur ce même schéma. Quand on comprend ce schéma, l'apprentissage automatique n'est plus une boîte noire.

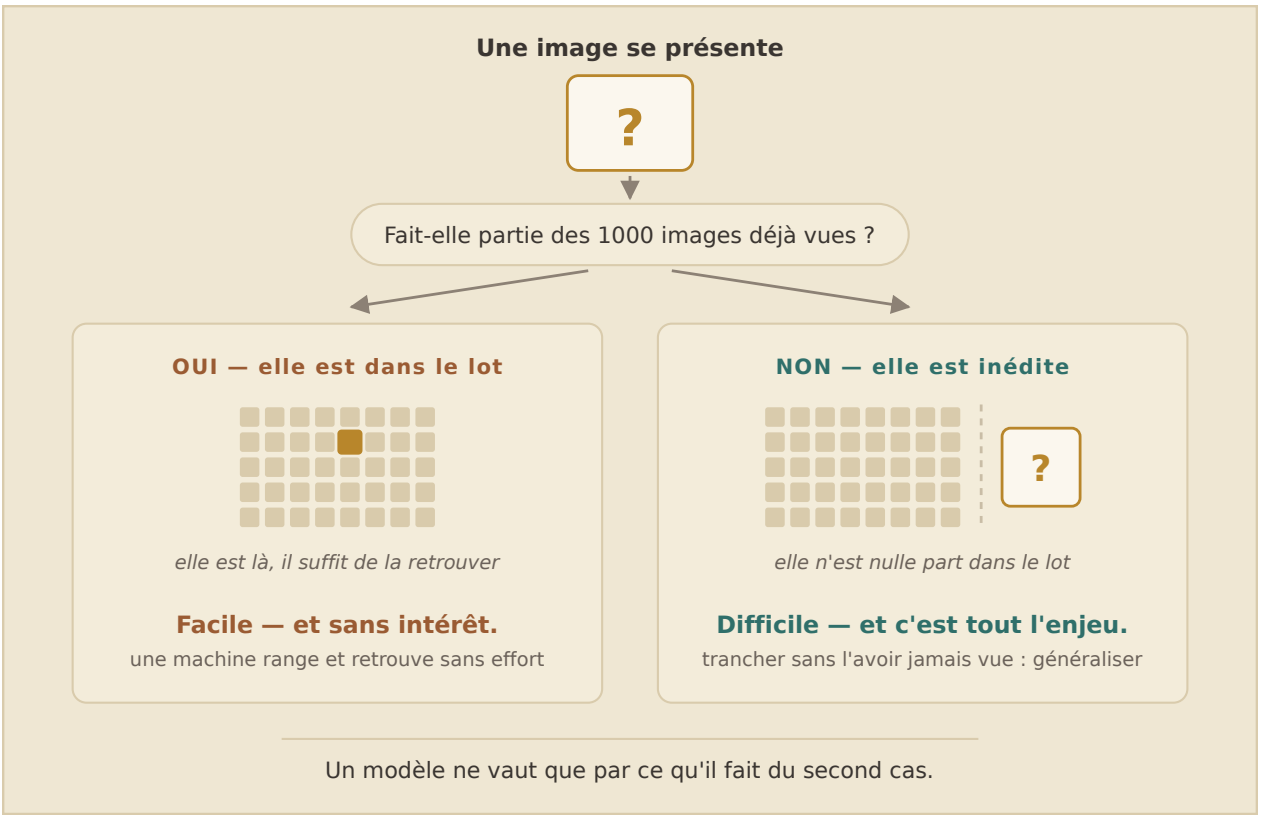
Une vieille parenté : la statistique

Il faut préciser dès le départ un point que le vocabulaire actuel tend à cacher : l'apprentissage automatique n'est pas né avec les ordinateurs, et encore moins avec les années 2010. Presque tout ce que vous verrez dans ce module relève de la **statistique**, parfois depuis deux siècles. La droite ajustée par les moindres carrés date de Legendre et de Gauss, vers 1805. Le théorème de Bayes date de 1763, et le mot *régression* lui-même a été introduit par Galton en 1886. Les plus proches voisins, les arbres de décision et les forêts ont été développés par des statisticiens, et les termes que nous emploierons (échantillon, distribution, biais, variance, test) viennent de la statistique. L'autre discipline d'origine est la **théorie des probabilités** : chaque modèle de ce module repose sur une probabilité, une vraisemblance ou une croyance que l'on révise, et le théorème de Bayes y joue un rôle central.

La différence entre les deux domaines tient moins aux outils qu'à la **question posée**. La statistique classique cherche à expliquer, par exemple quel est l'effet de la superficie sur le prix, et avec quelle certitude. L'apprentissage automatique cherche à prédire, par exemple quel sera le prix d'une maison donnée. Pour bien prédire, il accepte des modèles si complexes qu'on ne peut plus les interpréter, à condition qu'ils fonctionnent bien sur des données nouvelles. Leo Breiman, un statisticien devenu pionnier de l'apprentissage automatique, a décrit cette divergence dans un texte connu, « Statistical Modeling: The Two Cultures » (2001) : il y décrit deux cultures qui partagent les mêmes fondements. Nous signalerons cette parenté chaque fois qu'un modèle viendra directement de la statistique.

Le but ultime : généraliser

La notion la plus importante de l'AA, celle qui permet de l'associer au domaine de l'intelligence, est la capacité à **généraliser**. Si on entraîne un modèle à distinguer un chien d'un chat avec 1000 images, sa performance sur ces 1000 images a peu d'intérêt, puisqu'il devrait les reconnaître par construction. En effet, mémoriser ces 1000 images est **trivial** pour un ordinateur : ranger des données et les restituer à l'identique est exactement ce qu'une machine fait sans effort, et cela ne demande aucune intelligence. Ce qui compte, c'est la **1001e** image, que le modèle n'a jamais vue. Un modèle qui a vraiment appris, au lieu de simplement mémoriser, pourra la classer correctement. La question devient plus intéressante encore si on lui montre une vache au lieu d'un chat ou d'un chien. Bien généraliser est le véritable objectif de l'AA, et c'est l'un des sens les plus concrets qu'on puisse donner au mot « apprendre ».



Les deux cas possibles pour une nouvelle image. Si elle fait partie des 1000 images déjà vues, la retrouver est trivial ; si elle est inédite, il faut décider sans précédent. C'est dans ce second cas qu'on évalue un modèle.

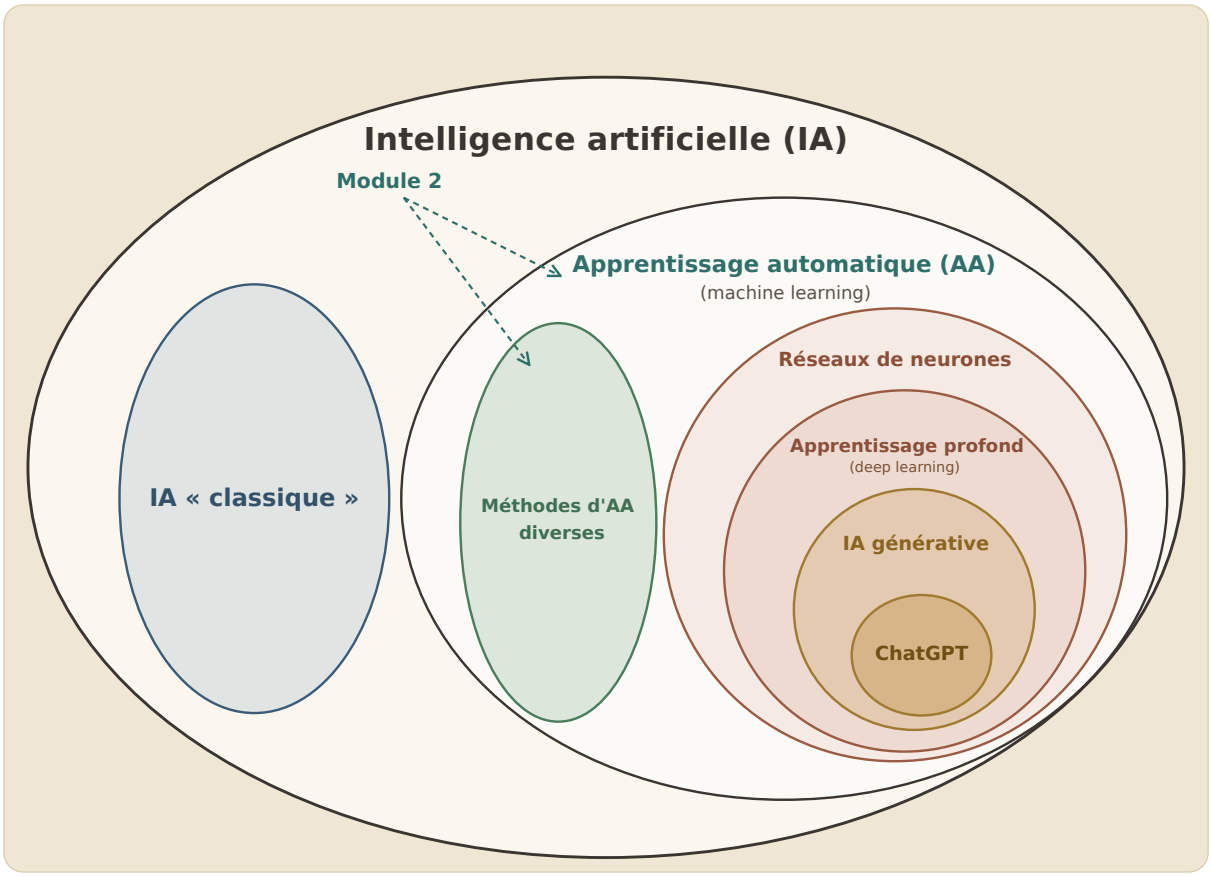
Le parcours du module

Le module part du **modèle le plus simple possible**, puis l'améliore progressivement : chaque étape rencontre une limite qui mène à l'idée suivante.

1. *Le problème* : prédire un prix, prédire une catégorie, et pourquoi c'est difficile.
2. *Le modèle le plus bête* : toujours répondre la moyenne, et ce que ce point de référence nous apprend.
3. *Regarder les données* : une maison, une image, un courriel sont des listes de nombres, donc des points dans un espace.
4. *Prédire par ressemblance* : les plus proches voisins, la distance, et la première frontière de décision.
5. *Un modèle qui s'entraîne* : la droite, la fonction d'erreur, la descente de gradient.
6. *Classer* : la régression logistique, la marge des SVM, Bayes, le filtre anti-pourriel et les réseaux bayésiens.
7. *Poser des questions : les arbres de décision* : un modèle qui cherche au lieu de descendre un gradient, et qui peut expliquer ses décisions.
8. *Généraliser* : le jeu de test, ce qu'un modèle peut dessiner, le compromis biais-variance, la régularisation.
9. *Bien évaluer un modèle* : les fuites, la validation croisée, les métriques, la distribution, la causalité.
10. *Trois façons d'apprendre* : supervisé, non supervisé, par renforcement.

Pour situer ce module dans l'ensemble du cours : l'apprentissage automatique n'est qu'une partie

d'un domaine plus vaste, celui de l'intelligence artificielle, que le cours parcourt module par module.



La carte de l'IA : le Module 2 porte sur l'apprentissage automatique classique, c'est-à-dire l'ensemble « Apprentissage automatique » et, en particulier, les méthodes d'AA diverses.

Objectifs

Au terme de ce module, vous devriez être en mesure de :

- distinguer clairement la programmation traditionnelle de l'apprentissage automatique, et situer celui-ci dans le paysage de l'IA ;
- expliquer le fil conducteur *données* → *modèle* → *erreur* → *minimisation* → *généralisation*, et le reconnaître dans n'importe quel algorithme ;
- décrire de l'intérieur les modèles classiques rencontrés (le modèle bête, les plus proches voisins, la régression linéaire et logistique, les machines à vecteurs de support, Bayes naïf et les réseaux bayésiens, l'arbre de décision et la forêt aléatoire, k-means) et dire ce qui les distingue ;
- expliquer comment un modèle s'entraîne (fonction d'erreur, descente de gradient) et ce qu'est un hyperparamètre ;
- définir le sur-apprentissage (*overfitting*) et le sous-apprentissage, et les relier au compromis biais-variance, à la capacité d'un modèle (linéaire ou non) et à la régularisation ;
- évaluer correctement un modèle : jeu de test, fuites de données, validation croisée, matrice de confusion, précision et rappel, et la question de la distribution ;
- distinguer une corrélation d'une cause, et dire pourquoi un modèle prédictif ne suffit pas pour décider d'une intervention ;

- distinguer les trois grandes façons d'apprendre (supervisé, non supervisé, par renforcement) par la nature du signal dont le modèle apprend ;
- expliquer pourquoi l'explicabilité d'un modèle compte, et quels modèles l'offrent.

Durée

Quatre semaines, soit environ 36 heures.

Évaluation

Un travail noté (20 % de la note finale) où vous construirez, pas à pas, un filtre anti-pourriel par classification bayésienne naïve, avec des questions d'interprétation sur le fonctionnement de l'algorithme.

Le problème

Quand vous regardez votre téléphone, il se déverrouille. Un courriel arrive et il est classé automatiquement dans les indésirables. Sur une chaîne de montage, une caméra repère, parmi des milliers de téléviseurs presque identiques, celui dont l'écran présente un défaut à peine visible. Une application vous suggère un film qui correspond à ce que vous aviez envie de voir ce soir.

Ces tâches font partie du quotidien. Elles ont cependant un point commun : **personne ne sait écrire, ligne par ligne, les règles qui permettraient de les accomplir.** Il est très difficile de décrire, par une suite d'instructions précises, ce qui distingue votre visage de tous les autres, ou ce qui fait qu'un courriel ressemble à un pourriel. On reconnaît ces choses en un instant, mais on ne sait pas les formuler.

C'est la limite à laquelle s'est heurtée l'IA classique du Module 1 : ce savoir est trop vaste et trop tacite pour tenir dans une liste de règles. L'apprentissage automatique adopte une autre stratégie. Au lieu de dicter les règles à la machine, on la laisse les **découvrir**.

Pourquoi on ne peut pas simplement le programmer

Pour comprendre ce qui change, il faut rappeler ce qu'est un programme informatique classique, comme celui qui fait fonctionner un tableur, un site de réservation ou une calculatrice. C'est une **suite d'instructions explicites**, écrites une à une par un programmeur, qui indiquent à l'ordinateur quoi faire, étape par étape : *si* telle condition, *alors* telle action. Quand une tâche peut être décrite par des règles claires, cette approche est très efficace.

Les exemples du début, cependant, ne se réduisent pas à des règles claires. Supposons qu'on veuille écrire une recette qui reconnaît un chat sur une photo. La règle « un chat a deux oreilles pointues » ne s'applique pas quand le chat est de dos, couché ou à demi caché derrière un meuble. La règle « il a le poil gris » ne s'applique pas à un chat roux ou noir, ni à un chat tigré sur un canapé tigré. Chaque règle entraîne de nombreuses exceptions, et la liste n'a pas de fin.

Ce phénomène s'appelle le **paradoxe de Moravec**. Les tâches que nous trouvons difficiles (jouer aux échecs, extraire une racine carrée) sont les plus faciles à programmer, parce qu'elles reposent sur des règles explicites. À l'inverse, ce qu'un enfant de trois ans fait sans effort, comme reconnaître un visage, attraper une balle ou comprendre une phrase, se laisse très difficilement mettre en règles, parce que cette compétence est perceptive et largement inconsciente. Comme nous ne savons pas comment nous faisons, nous ne pouvons pas l'écrire.

Traduction :
- « Quand un usager prend une photo, l'app devrait vérifier s'il est dans un parc national… »
- « Facile : un simple appel cartographique. Donne-moi quelques heures. »
- « … et vérifier si la photo est celle d'un oiseau. »
- « Là, j'ai besoin d'une équipe de recherche et de cinq ans. »

En informatique, la frontière entre le facile et le presque impossible n'est pas là où l'intuition la place.

Cette bande dessinée très connue a plus de dix ans, et son exemple a vieilli : identifier un oiseau dans une photo est devenu facile, justement grâce à l'apprentissage automatique. Son idée principale reste cependant exacte : certaines tâches d'apparence simple sont très difficiles pour un programme classique.

Changer de stratégie : apprendre à partir d'exemples

Comme nous ne savons pas énoncer les règles, on peut aborder le problème autrement. Au lieu de dicter les règles à la machine, on lui **donne des exemples**, c'est-à-dire des cas où la bonne réponse est déjà connue, et on la laisse trouver elle-même la régularité qui les relie.

Cette idée est proche de la façon dont un enfant apprend ce qu'est un chat. Il ne mémorise pas une définition. Il voit des dizaines de chats, jusqu'à ce qu'il sache les reconnaître sans pouvoir expliquer comment. On ne lui a jamais donné la règle ; il l'a extraite des exemples.

C'est le principe de l'**apprentissage automatique** (*machine learning*) : on fournit à un programme un grand nombre d'exemples, ainsi qu'une procédure qui lui permet d'ajuster son comportement jusqu'à reproduire les bonnes réponses. On espère ensuite qu'il répondra bien sur des cas qu'il n'a jamais vus. C'est le changement annoncé à la fin du Module 1 : au lieu de chercher une solution parmi des règles posées d'avance, le programme apprend à en construire une à partir des données.

Pour comprendre comment cela fonctionne concrètement, le plus simple est de partir d'un exemple modeste, qu'on pourra suivre du début à la fin et examiner sous plusieurs angles.

Notre fil rouge : des maisons à vendre

Combien vaut cette maison, et se vendra-t-elle vite ? Ce module montre comment une machine peut apprendre à répondre à ces deux questions.

Photo : Kindel Media, Pexels.

Tout au long de ce module, nous suivrons un seul exemple, volontairement simple pour qu'on puisse en comprendre tous les détails : **un registre de ventes immobilières**. Nous poserons plus d'une question à ce même jeu de données (*dataset*).

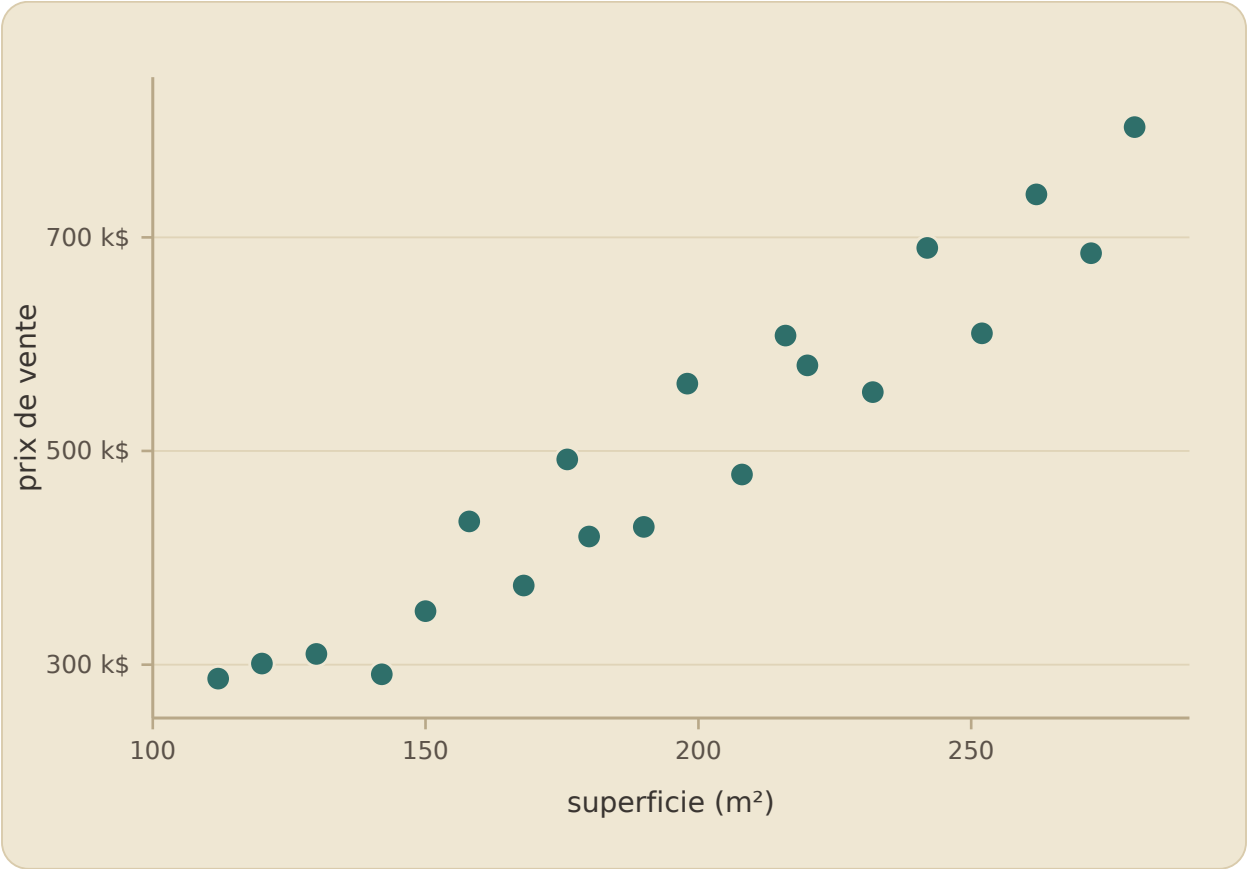
Supposons qu'on dispose d'une liste de maisons récemment vendues, avec pour chacune quelques renseignements (sa superficie, son année de construction, son nombre de chambres) et surtout son **prix de vente** :

Superficie (m²)	Année	Chambres	Prix
180	1995	4	420 000 $
150	1980	3	350 000 $

Superficie (m²)	Année	Chambres	Prix
220	2010	5	580 000 $
130	1972	3	310 000 $
…	…	…	…

La première question est la plus naturelle : **combien vaut une maison ?** Elle s'énonce simplement. Une nouvelle maison se présente, dont on connaît la superficie, l'année et le nombre de chambres, mais pas le prix, et il faut prédire ce prix.

Il y a clairement une régularité à exploiter : une grande maison récente vaut généralement plus cher qu'une petite maison ancienne. Cependant, « généralement » n'est pas une règle. C'est une tendance, qui comporte de nombreuses exceptions.



Chaque maison est un point. Le prix monte avec la superficie, mais pas parfaitement.

Si on ne garde que deux colonnes (la superficie et le prix), on voit déjà cette tendance : les points montent vers la droite, sans être parfaitement alignés. C'est ce genre de savoir approximatif que nous voulons faire émerger des exemples.

Une seconde question, d'une tout autre nature

Un agent immobilier se pose au moins deux questions sur une maison. La première, que nous venons de voir, est **combien vaut-elle ?** La seconde est tout aussi pratique : **va-t-elle partir vite ?** Les registres de ventes contiennent aussi les informations nécessaires, dans deux colonnes supplémentaires. La première donne la réponse. La seconde donne un renseignement qui ne nous avait pas servi jusqu'ici, mais qui va jouer un rôle important : la **distance du centre-ville** de la maison.

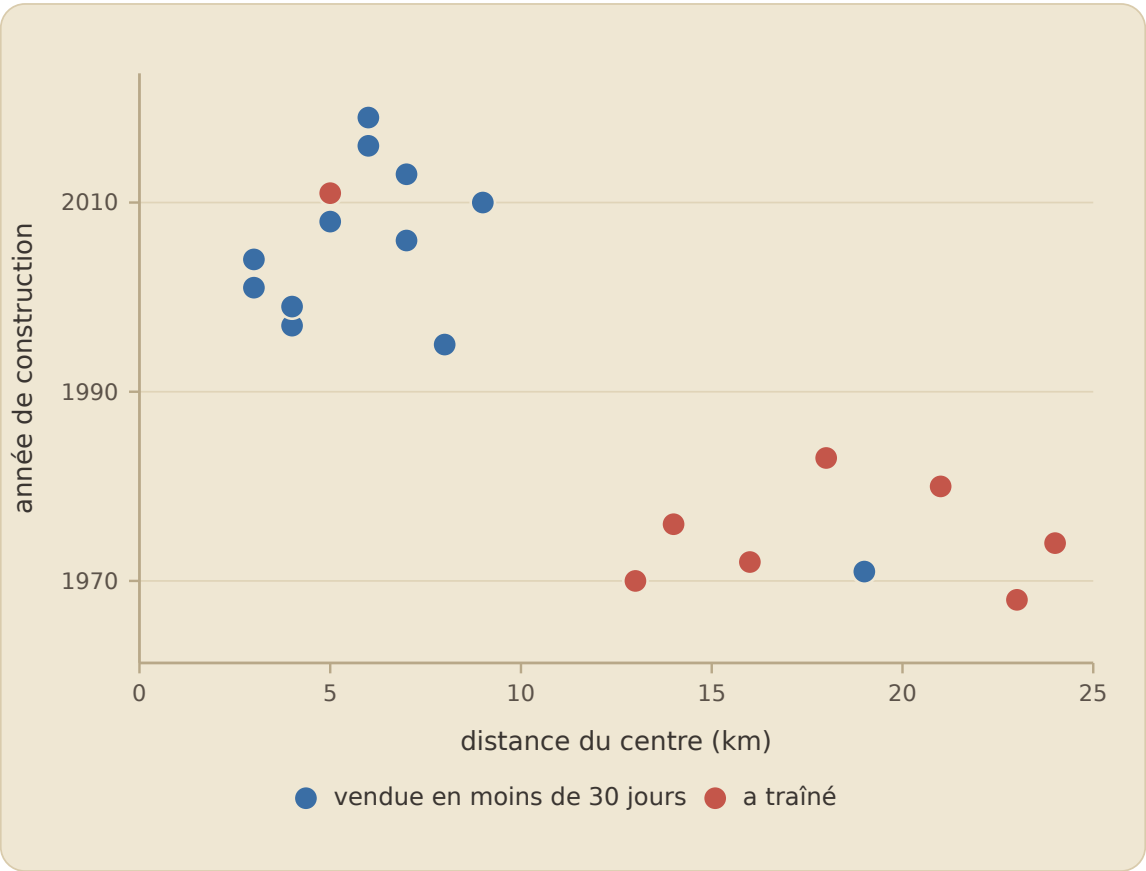
Superficie (m²)	Année	Chambres	Distance du centre (km)	Prix	Vendue en moins de 30 jours ?
180	1995	4	8	420 000 $	oui
150	1980	3	21	350 000 $	non
220	2010	5	9	580 000 $	oui
130	1972	3	16	310 000 $	non
…	…	…	…	…	…

Les deux questions portent sur les **mêmes maisons**, tirées du **même registre**. Cependant, les réponses attendues sont de nature différente. Dans le premier cas, la réponse est un **nombre** (420 000 $, 385 200 $, n'importe quelle valeur sur une échelle continue). Dans le second, c'est un **choix entre deux réponses possibles**, oui ou non. On ne peut pas faire la « moyenne » de *oui* et de *non*.

Cette différence est importante, parce qu'elle change le sens du mot **se tromper**. Prédire 405 000 $ pour une maison vendue 420 000 $ est une erreur, mais une erreur faible, dont on connaît la taille exacte : 15 000 $. Répondre *oui* pour une maison qui s'est vendue lentement est simplement une erreur. Il n'y a pas de « presque oui », pas de demi-erreur, et aucune distance entre les deux réponses possibles. Dans le premier cas, on mesure un écart ; dans le second, la réponse est juste ou fausse. Nous verrons que cette différence a des conséquences sur tout le reste : la manière de prédire bêtement, la manière de mesurer l'erreur et la forme du modèle.

Pour cette seconde question, le **prix devient un renseignement comme un autre**, une colonne parmi les autres, au même titre que l'année de construction ou la distance du centre. Ce qui était la réponse à trouver dans le premier cas devient une donnée de départ dans le second. La table reste la même ; c'est la **question qu'on lui pose** qui change.

On peut aussi visualiser ce changement, à condition de choisir les bons axes. On laisse de côté la superficie et le prix, et on place les maisons dans un autre plan, avec la **distance du centre** en abscisse et l'**année de construction** en ordonnée. On colore ensuite chaque point selon la réponse à la nouvelle question :

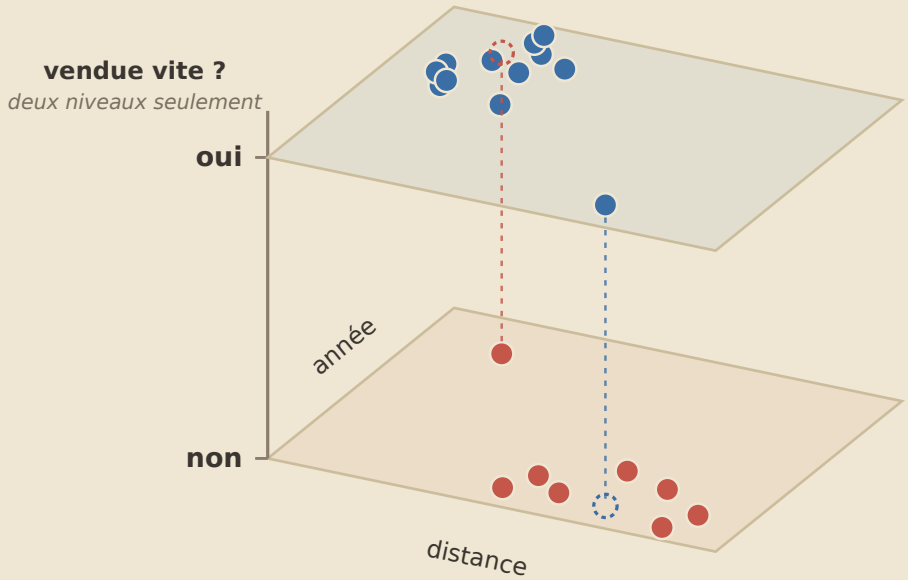


Les mêmes maisons, une autre question, et un autre plan. Ce n'est plus la hauteur du point qu'on cherche à deviner, mais sa couleur.

La différence avec le premier graphique est nette. Dans le premier, on cherchait à deviner la **hauteur** du point, c'est-à-dire sa position sur une échelle continue. Ici, les deux coordonnées sont données, et on cherche la **couleur**. Le motif est clair : les points forment **deux amas** situés dans des coins opposés, séparés par un large espace vide. Les maisons proches du centre et récentes se vendent vite. Les maisons éloignées et anciennes se vendent lentement. Seules deux exceptions se trouvent dans l'espace vide.

On a changé de plan parce que, dans le graphique précédent (la superficie et le prix), les deux couleurs auraient été mêlées et n'auraient rien montré. Le motif était bien présent dans le registre, mais pas dans ces renseignements-là. Ce point annonce une leçon qui reviendra tout au long du module : la difficulté d'un problème tient souvent moins à la méthode qu'on applique qu'au choix de **ce qu'on décide de regarder**.

La couleur sert à régler un problème de place. Nous avons maintenant **trois** renseignements à représenter sur une page plane (la distance, l'année et la réponse). Les deux premiers occupent les axes. Pour le troisième, il ne reste pas d'axe disponible, et on le représente donc autrement. Il s'agit seulement d'une commodité de dessin. On peut le vérifier en donnant à la réponse un troisième axe, dans une vue en perspective :



La couleur du nuage plat était l'ombre portée de ce troisième axe.
Les deux exceptions, vues d'en haut, retombent au milieu de l'autre couleur (points creux).

La réponse a son propre axe, qui ne comporte que deux barreaux, non et oui. Les maisons se trouvent donc sur l'un ou l'autre de deux plans.

Les maisons ne sont plus placées à n'importe quelle hauteur. Elles se trouvent sur l'un ou l'autre de **deux plans**. On passe d'un dessin à l'autre sans perdre d'information. Si on regarde ce relief d'en haut, à la verticale, on retrouve exactement le nuage coloré. La couleur correspondait donc à la projection du troisième axe. Les deux exceptions apparaissent aussi dans ce relief, qui explique pourquoi elles se remarquaient autant. La maison ancienne et éloignée qui s'est vendue vite se trouve sur le plan du haut, mais à la verticale de l'amas rouge. La maison récente et proche qui s'est vendue lentement se trouve sur le plan du bas, à la verticale de l'amas bleu. Le pointillé montre où chacune se place vue d'en haut, c'est-à-dire au milieu de l'autre couleur. Nous reprendrons cette figure au chapitre « Regarder les données », quand nous saurons quoi écrire sur ces deux barreaux.

Dans les **prochains chapitres**, nous suivrons surtout la première question, parce que le prix se prête mieux aux dessins et aux premières explications. La seconde question fera ensuite l'objet d'un chapitre entier, « Classer ». C'est d'ailleurs une question de ce type, *ce courriel est-il un pourriel ?*, que vous traiterez dans le travail noté. Nous verrons alors que presque tout ce qu'on aura appris pour l'une vaut aussi pour l'autre. **Plusieurs** des exemples du début de ce chapitre appartiennent à cette seconde famille. Reconnaître un chat ou repérer un pourriel sont des questions dont la réponse n'est pas un nombre, mais **une catégorie**.

Une catégorie peut avoir plus de deux valeurs, comme dans les questions suivantes. *Quel animal est sur cette photo : un chat, un chien, un cheval ? Quel chiffre est écrit sur cette enveloppe ?* (dix réponses possibles). *Dans quelle langue ce message est-il rédigé ?* Ce qui compte n'est pas le nombre de réponses, mais leur **nature** : une liste de possibilités distinctes, qu'on ne peut ni moyenner ni ranger sur une

échelle. Il n'y a rien entre *chat* et *chien*, pas plus qu'entre *oui* et *non*. La question posée à nos registres, avec ses deux réponses, est le cas le plus simple de cette famille.

Avant de construire un modèle élaboré, il est utile de se demander quelle est la prédiction la plus bête possible, celle en dessous de laquelle il serait absurde de descendre. C'est par cette question que commence la suite du module.

Le modèle le plus bête

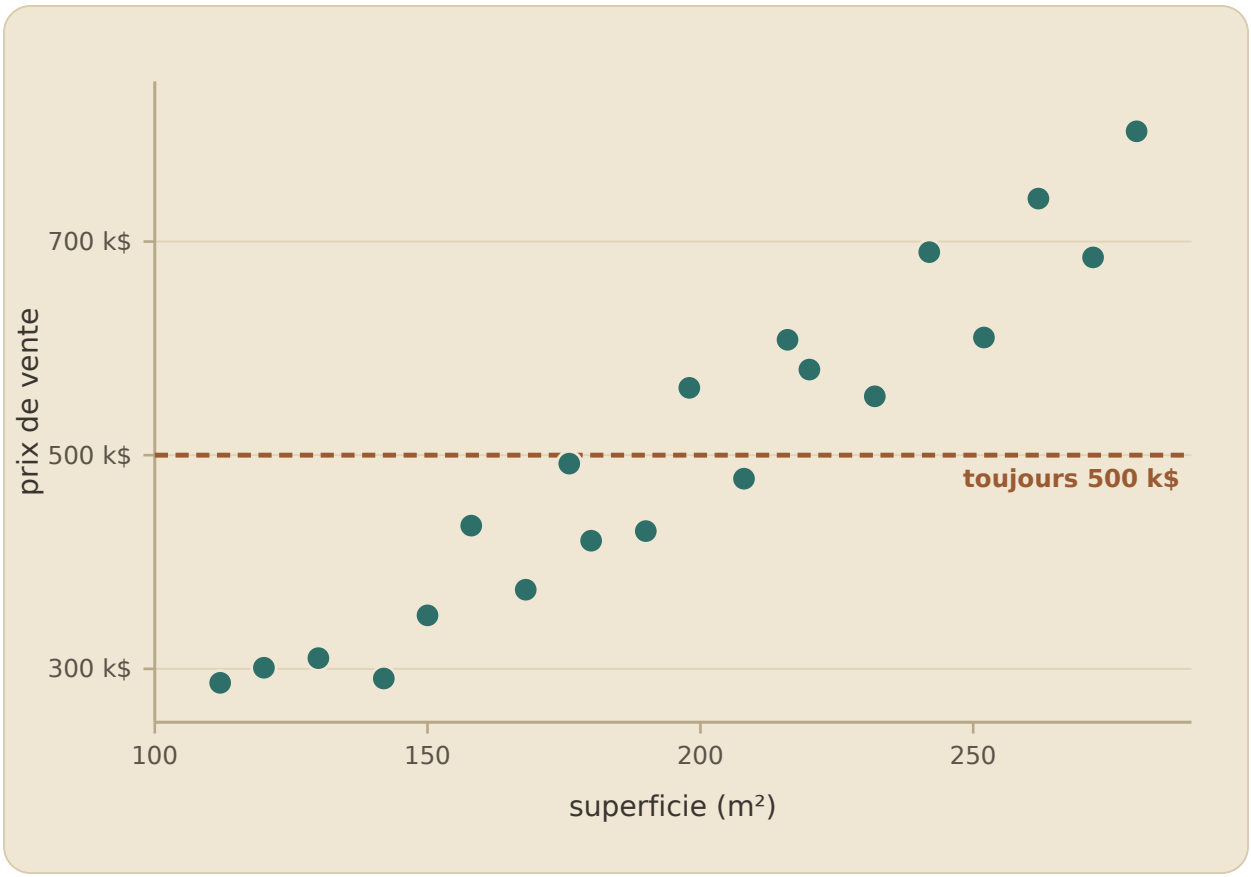
À la fin de la page « Le problème », nous avons posé une question qui peut sembler absurde : quelle est la prédiction la plus bête qu'on puisse imaginer ? Cette question mérite d'être prise au sérieux, parce que la pire réponse possible est instructive. Elle nous obligera aussi à préciser ce qu'est un *modèle*.

Toujours prédire la moyenne

Le pire « prédicteur » qu'on puisse concevoir est le suivant : pour n'importe quelle maison, on ignore toutes ses caractéristiques (sa superficie, son âge, son nombre de chambres) et on annonce toujours le **même** prix, le **prix moyen** de toutes les maisons de notre liste.

Dans notre exemple, ce prix moyen est d'environ **500 000 $**. La prédiction ne dépend donc plus de rien. Elle est de 500 000 $ pour un très petit studio, et de 500 000 $ aussi pour un grand manoir. C'est évidemment absurde.

Ce prédicteur est cependant parfaitement défini, il ne tombe jamais en panne et il donne toujours une réponse. Sur le nuage de points (*scatter plot*) de la page précédente, il correspond à une simple **ligne horizontale**, à la même hauteur (500 000 $) quelle que soit la superficie. Cette ligne traverse le nuage en son milieu, au-dessus des maisons bon marché et en dessous des plus chères.



Le modèle le plus bête : une droite plate à 500 000 $, qui ignore complètement la superficie.

La même bêtise, pour l'autre question

Le prédicteur le plus bête existe aussi pour la **seconde question** du chapitre « Le problème », *cette maison partira-t-elle vite ?* Il suit la même logique : il ignore la maison qu'on lui présente et répond toujours la même chose.

Il ne peut cependant pas répondre la moyenne, parce qu'on ne peut pas faire la moyenne de « oui » et de « non ». L'équivalent consiste à donner **la réponse la plus fréquente**, celle qui revient le plus souvent dans nos registres. Si 60 % des maisons déjà vendues ont trouvé preneur en moins de trente jours, ce prédicteur répondra toujours « oui », aussi bien pour une maison neuve proche du centre que pour une vieille maison des années 1970 située loin en banlieue.

Les deux questions sont de nature très différente, mais les deux prédicteurs ont la **même structure** : une entrée qu'on ignore, une sortie constante et une réponse tirée uniquement des données. Nous retrouverons souvent cette ressemblance entre les deux questions.

Qu'est-ce qu'un modèle, au juste ?

Nous venons d'appeler ce prédicteur un « modèle ». Il faut préciser le sens de ce mot, qui sera central dans tout le module.

> Un **modèle**, c'est une recette qui transforme une description (l'entrée) en une prédiction (la sortie).

Notre prédicteur bête correspond à cette définition : on lui donne une maison en entrée, et il renvoie un prix en sortie. Le fait qu'il ignore cette entrée ne l'empêche pas d'être un modèle. C'est seulement un très mauvais modèle.

Ce modèle est composé d'**un seul nombre**, le prix moyen (500 000 $). Ce nombre représente ce que le modèle a « retenu » des données ; on l'appelle son **paramètre**. Le calcul de ce nombre (la moyenne des prix observés) est déjà une forme simple d'*apprentissage*, puisque le modèle a tiré sa seule connaissance des exemples qu'on lui a fournis.

Ce nombre n'est donc pas arbitraire. On aurait pu annoncer 12 $, ou un milliard. Ce modèle aurait été tout aussi bête, puisqu'il ignorerait lui aussi la maison présentée, mais il se tromperait beaucoup plus. Le prix moyen a été **calculé** à partir des ventes passées, et il a un sens : parmi tous les nombres qu'on pourrait annoncer pour toutes les maisons à la fois, c'est celui qui se trompe le moins, en moyenne. La faiblesse du modèle vient de ce qu'il ignore, et non de ce qu'il retient. Le modèle jumeau fonctionne de la même façon. Son unique paramètre est la réponse majoritaire (« oui »), obtenue en comptant plutôt qu'en calculant une moyenne. Ce choix est lui aussi justifié, puisque c'est la réponse qui a le plus de chances d'être correcte quand on ne sait rien d'autre.

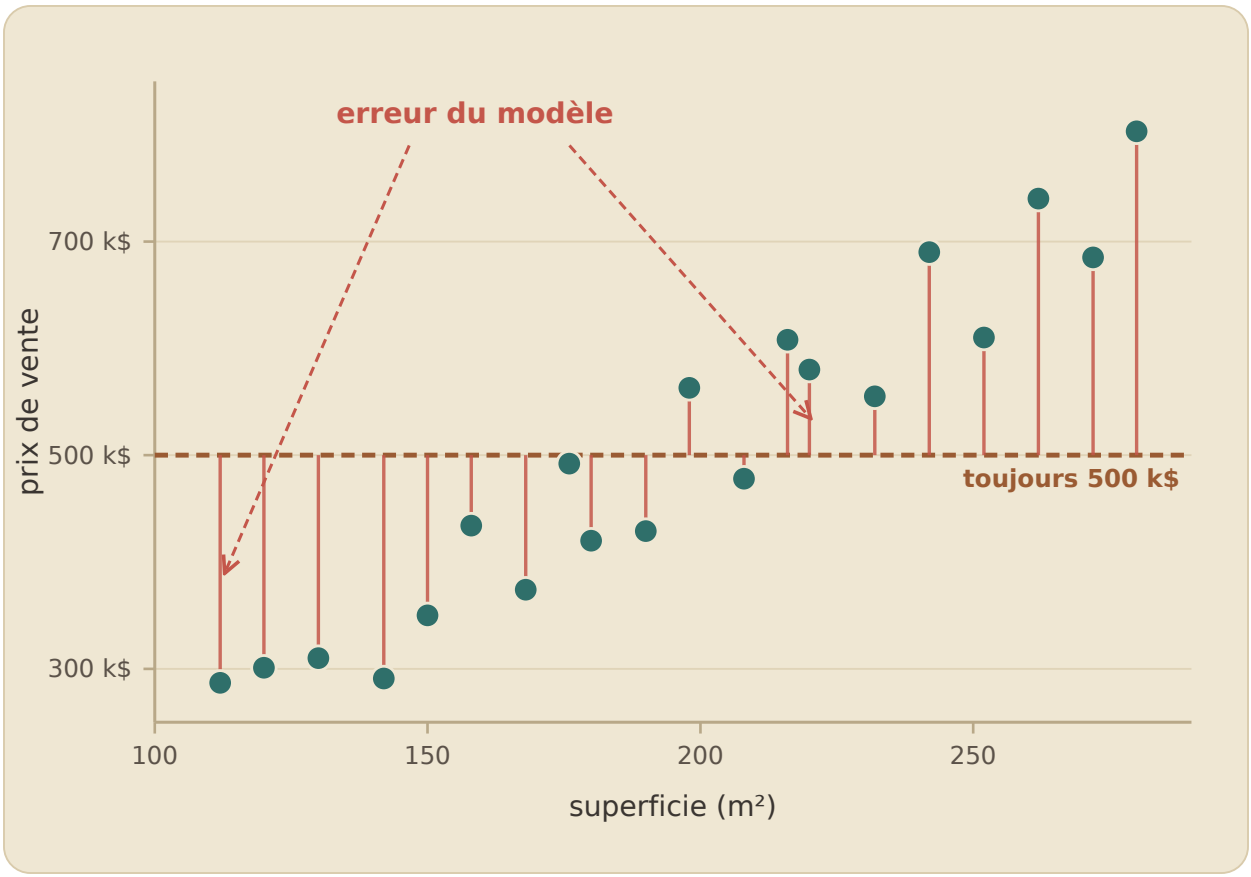
Le reste du module développe cette idée. Les modèles que nous construirons auront davantage de paramètres : un, puis deux, puis des milliers, puis des milliards. La principale difficulté consistera à trouver les bonnes valeurs pour ces paramètres, celles qui correspondent le mieux aux données. La structure restera cependant toujours la même : une entrée, une recette réglée par des paramètres, une sortie.

Pourquoi un modèle aussi bête est utile

Ce modèle est très mauvais, mais il est utile parce qu'il fournit un **étalon** (*baseline*), c'est-à-dire un point de comparaison pour juger tous les modèles suivants.

Les mots « bon » et « mauvais » n'ont pas de sens dans l'absolu. Pour savoir si un modèle a de la valeur, il faut une référence. La référence la plus élémentaire consiste à vérifier s'il fait mieux qu'un modèle qui ne regarde rien du tout. Un modèle, même sophistiqué, qui ne fait pas mieux que « toujours 500 000 $ » n'a rien appris d'utile.

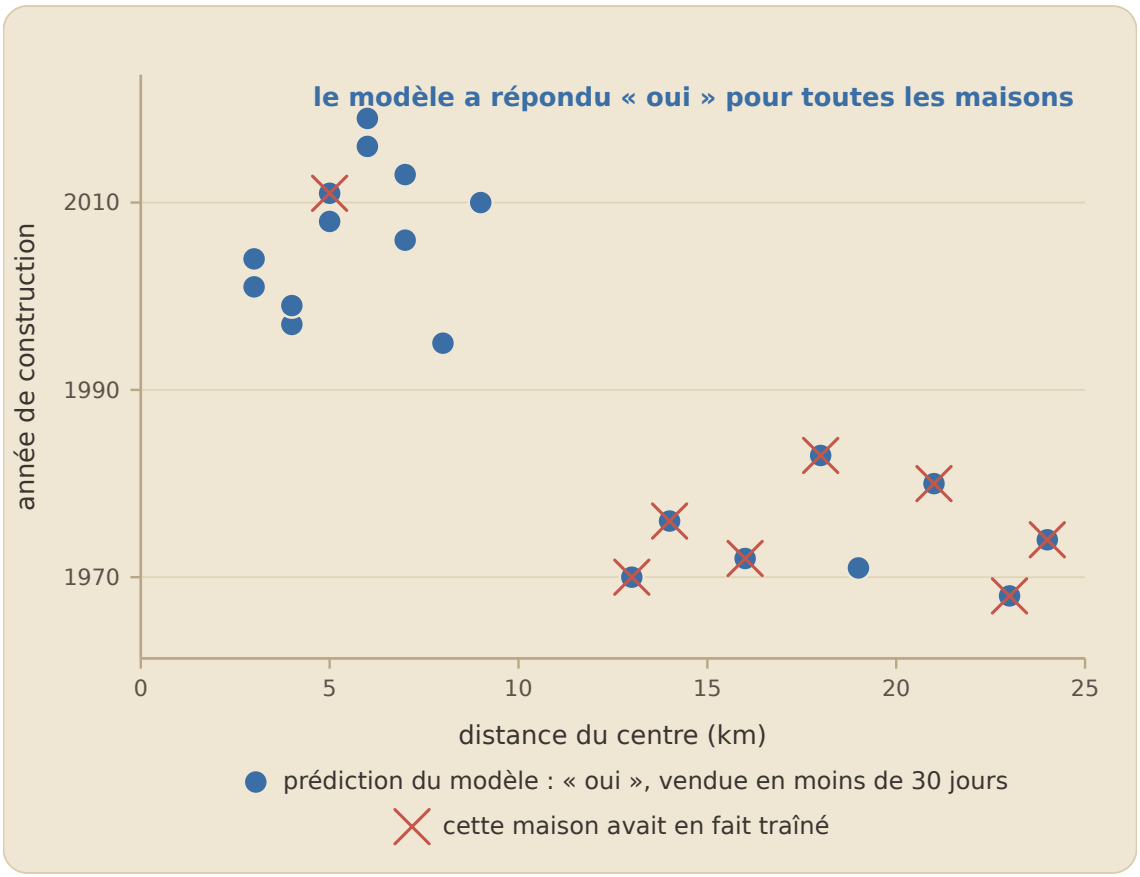
On peut faire cette comparaison concrètement, sans formule, en mesurant **de combien un modèle se trompe, en moyenne**. Pour le prédicteur bête, l'écart entre le prix annoncé (toujours 500 000 $) et le vrai prix dépasse 250 000 $ pour les maisons situées aux extrêmes. C'est cette « distance à la vérité » qu'un meilleur modèle cherchera à réduire. Nous lui donnerons au chapitre « Un modèle qui s'entraîne » un nom et une définition précise, la *fonction d'erreur* (*loss function*), mais l'idée suffit pour l'instant : un bon modèle est un modèle qui se trompe moins.



L'erreur du modèle, maison par maison : l'écart vertical entre le vrai prix et la prédiction.

En pratique, ce rôle d'étalon évite bien des erreurs d'interprétation. Un chiffre de performance ne signifie rien à lui seul. Devant un modèle, la première question à poser est toujours de savoir s'il fait vraiment mieux que de prédire bêtement la moyenne. La réponse est assez souvent non, et c'est l'étalon qui permet de le constater.

Pour la question en oui/non, le principe est le même, mais on compte autrement. Au lieu d'un écart en dollars, on regarde **quelle proportion des réponses le modèle obtient justes**. Le prédicteur jumeau, qui répond « oui » à tout, a donc raison dans 60 % des cas. C'est l'étalon à battre.



Le prédicteur jumeau colore toutes les maisons en « oui », et les × marquent celles où il se trompe. On ne mesure plus un écart en dollars, on compte les réponses justes.

Il s'agit du nuage coloré de la page précédente, sans modification : les **mêmes maisons**, aux mêmes places. Seule leur couleur a changé. Comme le modèle répond « oui » partout, il les colore toutes en bleu, et les × indiquent celles qui étaient rouges. Presque tous les × se trouvent dans l'amas du bas à droite, celui des maisons éloignées et anciennes. Le prédicteur bête se trompe sur presque tout cet amas, ce qui est normal, puisqu'il répond « oui » sans tenir compte de la distance ni de l'année.

On peut comparer cette figure avec le graphique des écarts de prix présenté plus haut. Les axes sont différents (ils ne peuvent pas être les mêmes, parce que les deux questions ne se lisent pas dans le même plan), mais la différence principale est ailleurs. Dans le graphique des prix, chaque maison portait un segment plus ou moins long, et l'erreur se **mesurait**. Ici, chaque réponse est simplement juste ou fausse, et l'erreur se **compte**.

Cet étalon peut être trompeur, et ce cas est instructif. Prenons une question beaucoup plus déséquilibrée, *ce courriel est-il un pourriel ?*, dans une boîte où 99 % des messages sont légitimes. Le prédicteur le plus bête, qui répond « ce n'est jamais un pourriel », obtient **99 % de bonnes réponses**, tout en étant inutile, puisqu'il ne détecte aucun pourriel. Un chiffre élevé peut donc cacher un modèle sans valeur. Nous y reviendrons dans « Bien évaluer un modèle », car bien mesurer la qualité d'un modèle est plus difficile qu'il n'y paraît.

Leur défaut, et ce qu'il révèle

Les deux modèles ont le **même** défaut, qui est évident. Le premier attribue le même prix à un studio et à un manoir. Le second prévoit une vente rapide aussi bien pour une maison neuve près du centre que pour une vieille maison à vingt kilomètres de là. Aucun des deux ne tient compte de la maison qu'on lui présente. Toute l'information utile (la superficie, l'âge, la distance) est disponible, et ils ne l'utilisent pas.

Les deux dernières figures montrent aussi un point important : **les erreurs ne sont pas dispersées au hasard**. Les segments les plus longs se trouvent aux extrémités du nuage, du côté des maisons très bon marché ou très chères, et les × se concentrent dans un seul coin du graphique, celui des maisons éloignées et anciennes. Or une erreur qui se concentre à un endroit est une erreur qu'on peut **prévoir**, donc corriger. Si les modèles se trompaient de façon totalement imprévisible, on ne pourrait rien en tirer. C'est parce que leurs erreurs forment un motif qu'il est possible de les améliorer.

C'est là que se trouve la marge de progression. Si la moyenne (ou la réponse majoritaire) est le meilleur point de départ quand on ne sait rien d'une maison, la seule façon de faire mieux est de **cesser de l'ignorer**, c'est-à-dire de tenir compte de ses caractéristiques. Une grande maison devrait faire monter la prédiction de prix, et une vieille maison devrait la faire baisser. Une maison éloignée du centre devrait faire pencher la réponse du second modèle vers le « non ». Un bon modèle est donc un modèle qui tient compte de l'entrée.

Il faut cependant d'abord préciser ce que signifie concrètement « tenir compte de l'entrée », c'est-à-dire ce qu'est une « donnée » pour une machine, et comment une maison, une image ou un courriel est transformé en quelque chose qu'un modèle peut manipuler. C'est l'objet de la page suivante.

Regarder les données

La page « Le modèle le plus bête » s'est terminée sur une exigence : pour faire mieux que la moyenne, un modèle doit tenir compte des caractéristiques d'une maison (sa superficie, son âge, son nombre de chambres). Il faut donc les lui présenter sous une forme qu'il peut manipuler.

Une machine ne « voit » ni une maison, ni une photo, ni un courriel. Elle ne manipule que des **nombres**. La question est donc de savoir comment transformer un objet du monde réel en nombres sans en perdre l'essentiel. Cette question, plus profonde qu'il n'y paraît, est l'objet de cette page, et elle est un préalable à tout ce qui suit.

Une maison, c'est une liste de nombres

Reprenons notre table de maisons. Chaque ligne décrit une maison par quelques **caractéristiques** (en anglais *features*), c'est-à-dire des grandeurs mesurables :

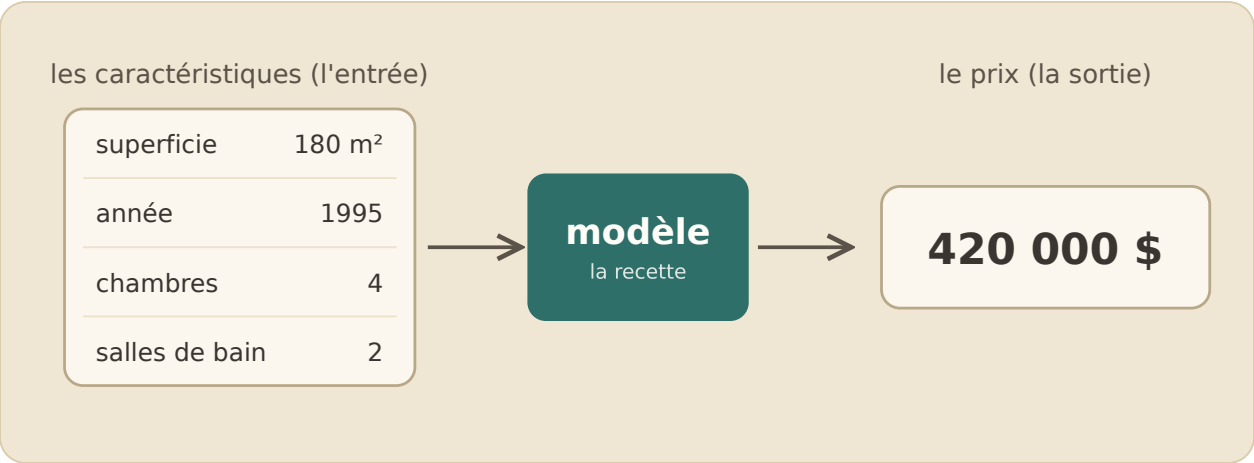
Superficie (m²)	Année	Chambres	Salles de bain	Prix
180	1995	4	2	420 000 $
150	1980	3	1	350 000 $
220	2010	5	3	580 000 $
130	1972	3	1	310 000 $

Pour une machine, décrire la première maison consiste simplement à aligner ses caractéristiques :

$$\text{maison} \rightarrow (180,\ 1995,\ 4,\ 2)$$

Une liste ordonnée de nombres comme celle-ci s'appelle un **vecteur**. L'objet peut être une maison, un client, un patient ou un courriel. Dès qu'on sait le décrire par quelques grandeurs, il devient un vecteur de nombres, et c'est ce vecteur qu'un modèle reçoit en entrée.

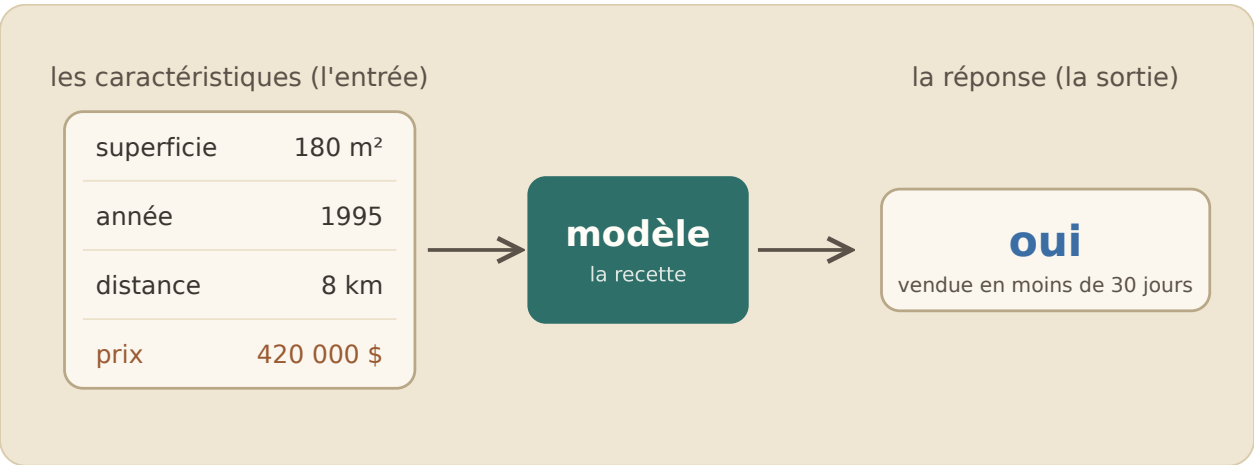
Une précision de vocabulaire servira dans toute la suite. Le **prix** ne fait pas partie de cette description, parce que c'est justement la valeur qu'on cherche à prédire. On l'appelle la **cible** (*target*). On a donc, d'un côté, les caractéristiques (l'entrée du modèle) et, de l'autre, la cible (sa sortie attendue).



Un modèle : les caractéristiques entrent, le prix sort.

Et quand la cible est une catégorie ?

Le prix est un nombre, donc il se prête sans difficulté à cette mise en forme. Notre seconde question était cependant différente : cette maison va-t-elle partir vite ? Le schéma est le même, avec deux différences. La sortie n'est plus un nombre, mais un **oui** ou un **non**. De plus, le prix, qui était la sortie, passe du côté de l'entrée et devient une caractéristique parmi les autres :



Le modèle de classification : les caractéristiques entrent, le prix compris ; une catégorie sort.

Or une machine ne manipule que des nombres. Il faut donc traduire le « oui » en nombre.

La façon la plus simple consiste à décider que **oui vaut 1 et non vaut 0**. Ce choix est arbitraire (on aurait pu prendre l'inverse, ou n'importe quel autre couple de valeurs), mais il a l'avantage de faire de la cible une grandeur comme une autre. Avec cette convention, une catégorie devient un nombre, et tout ce qui suit s'applique sans changement.

Le relief du chapitre « Le problème » ne change pas, sauf pour deux étiquettes : ses deux barreaux, *non* et *oui*, s'appellent maintenant 0 et 1. Ce changement est cependant important, parce que la réponse devient une **grandeur** comme les autres. Un modèle peut la calculer, la comparer et se tromper sur elle d'une quantité mesurable.

Il ne faut toutefois pas confondre cet axe avec les autres, car il n'est pas de même nature. La distance et l'année sont des **caractéristiques** : elles décrivent la maison et forment l'espace où elle se situe. La cible, elle, est ce qu'on **cherche**. Dans la suite, quand nous parlerons des *dimensions* d'un objet, il s'agira toujours de ses caractéristiques.

Nous pouvons maintenant nommer ces deux familles de problèmes, dont vous rencontrerez les noms partout :

- prédire un **nombre** (un prix, une température, une durée) s'appelle une **régression** ;
- prédire une **catégorie** (vendue vite ou non, pourriel ou courriel, chat ou chien, ou encore lequel des dix chiffres est écrit sur une enveloppe) s'appelle une **classification**.

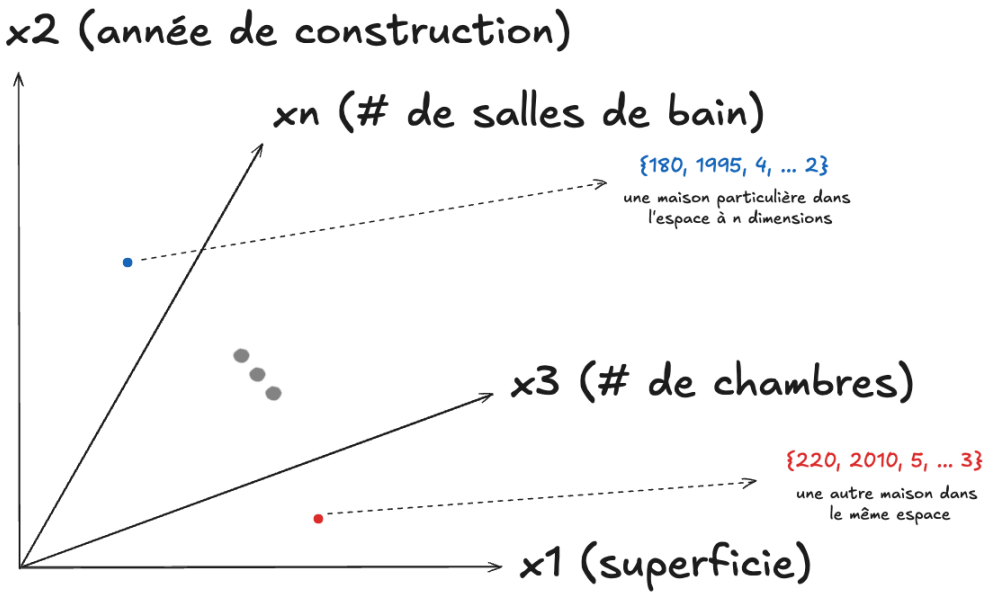
La seule chose qui distingue ces deux familles est la **nature de la cible**. Cette distinction est pourtant l'une des plus utiles du domaine, parce que presque tout problème d'apprentissage à partir d'exemples étiquetés (*labeled examples*) appartient à l'une ou à l'autre. Nous les retrouverons souvent, et nous verrons que certains algorithmes savent faire les deux, alors que d'autres se spécialisent.

Un vecteur, c'est un point dans un espace

Représenter une maison par un vecteur de nombres ne fait pas que ranger ses caractéristiques. Cela lui donne aussi une **place dans l'espace**.

Prenons deux caractéristiques, la superficie et le nombre de chambres. On peut placer chaque maison comme un **point** sur un graphe, avec la superficie à l'horizontale et le nombre de chambres à la verticale. Une maison correspond alors à un endroit du plan, de la même façon que, dans le nuage des pages précédentes, chaque maison était déjà un point. Un vecteur à deux composantes est donc une position dans un plan.

Rien n'oblige à s'arrêter à deux caractéristiques. Si on ajoute l'année de construction, la maison devient un point dans un espace à **trois** dimensions, comme une mouche immobile quelque part dans une pièce. Avec une quatrième caractéristique, elle devient un point dans un espace à quatre dimensions. La règle est toujours la même : **un objet décrit par n nombres est un point dans un espace à n dimensions.**



Chaque caractéristique devient un axe : une maison est un point dans un espace à autant de dimensions qu'elle a de caractéristiques.

Cette représentation géométrique est très utile. Deux maisons aux caractéristiques semblables sont deux points proches, et deux maisons très différentes sont deux points éloignés. La ressemblance entre objets devient donc une **distance** entre points. Le prochain chapitre, « Prédire par ressemblance », repose sur cette idée.

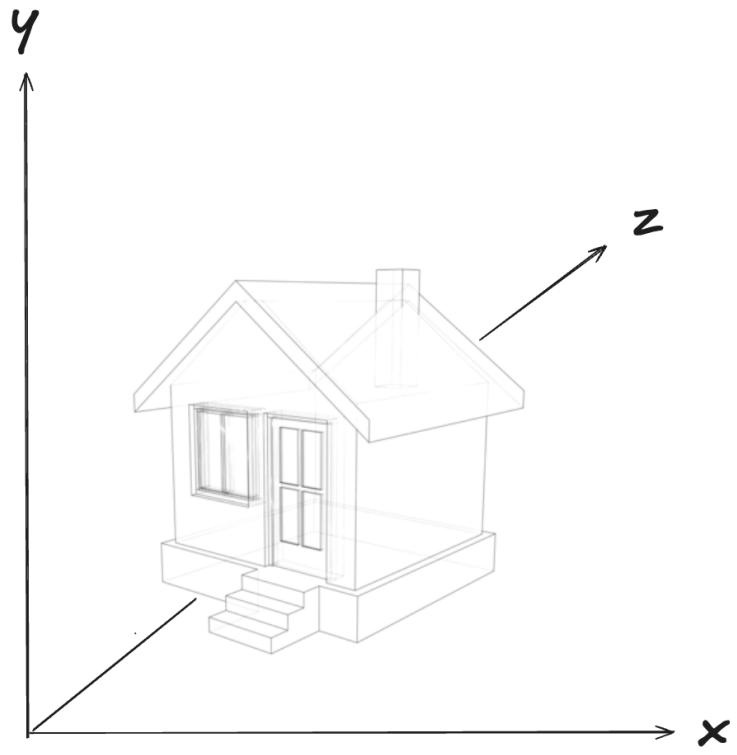
Quand il y a trop de dimensions pour les voir

Nos maisons n'avaient que trois ou quatre caractéristiques, et on pouvait presque les imaginer comme des points dans une pièce. Beaucoup d'objets du monde réel se décrivent cependant par beaucoup plus de nombres.

Prenons une image. Pour une machine, une photo est une **grille de pixels**, et chaque pixel est un nombre (ou trois nombres, pour ses quantités de rouge, de vert et de bleu).

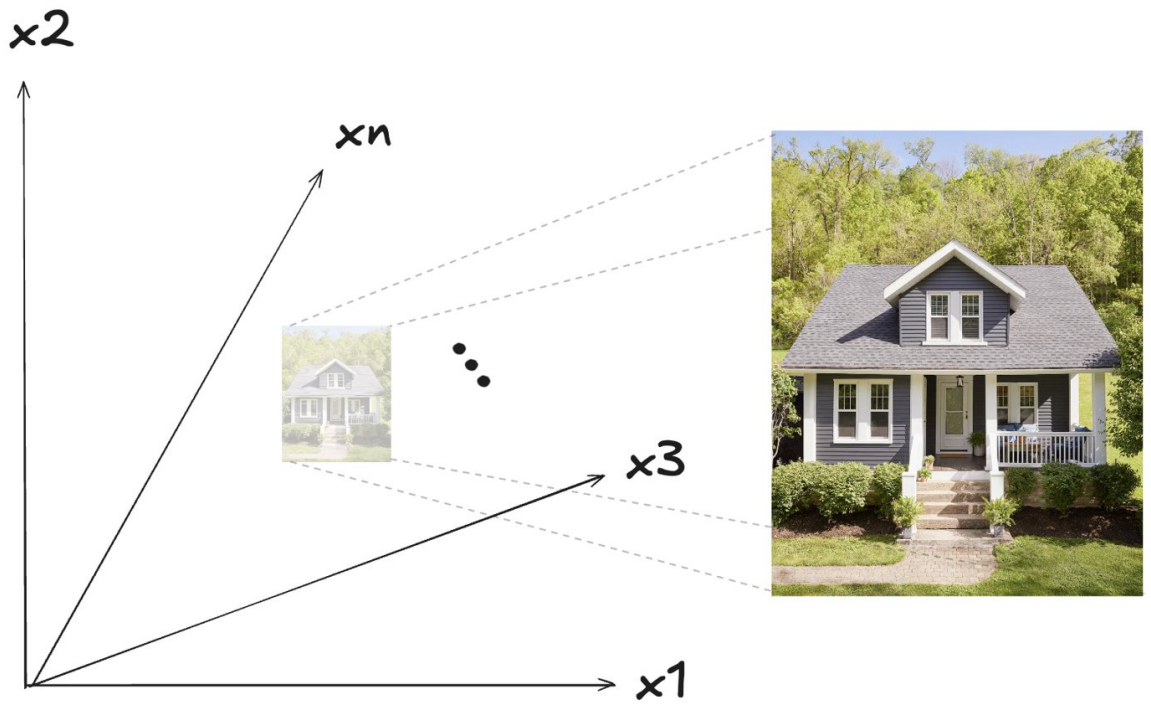
Une image, pour une machine : une grille de pixels, chacun un nombre.

On pourrait aussi la représenter autrement. Par exemple, s'il s'agissait d'une maison de jeu vidéo, on pourrait utiliser un modèle en trois dimensions, avec des axes x, y et z :



Autre représentation : un modèle 3D, repéré par des axes x, y, z.

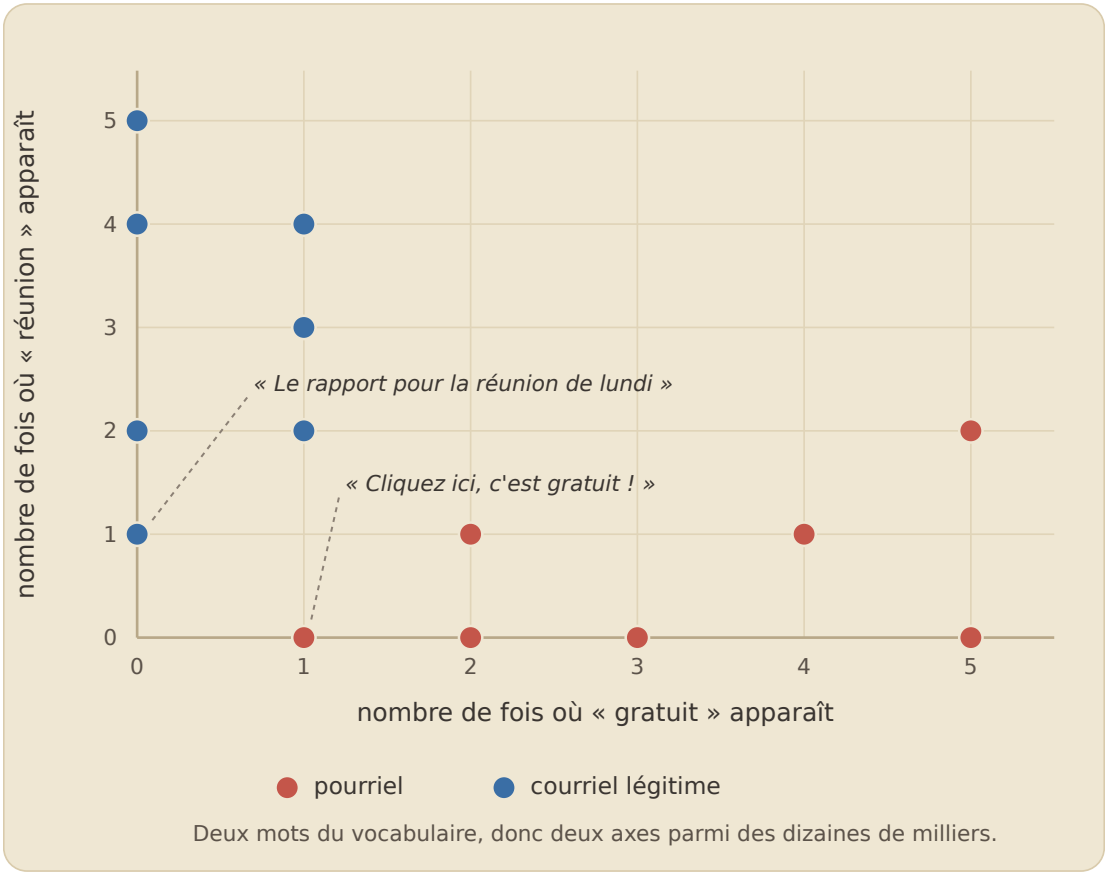
En apprentissage automatique, on procède autrement : on traite l'image **entière** comme un seul point, dans un espace où chaque pixel est une dimension. Une vignette de 100 × 100 pixels est ainsi un point dans un espace à 10 000 dimensions. Une photo ordinaire en compte des **millions**.



En haute dimension : l'image entière devient un seul point ; deux images semblables, deux points voisins.

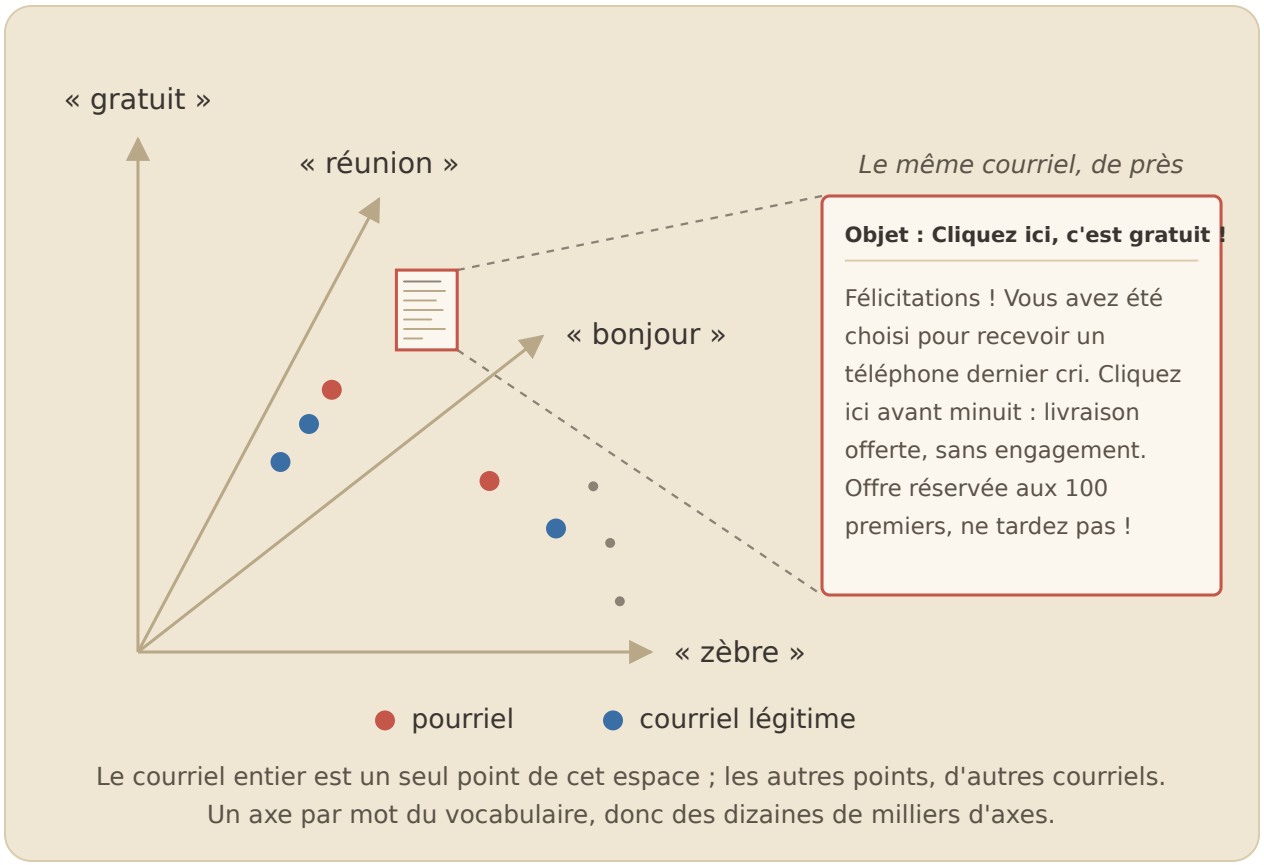
Il est impossible de se représenter un tel espace, puisque nous vivons dans un espace à trois dimensions. L'essentiel reste pourtant valable : **la distance continue d'avoir un sens.** Deux photos presque identiques sont deux points voisins, et deux images sans rapport sont deux points éloignés, comme pour nos maisons.

Le cas du texte est différent. Un courriel n'a ni superficie ni pixels, et rien ne semble s'y mesurer directement. On le décrit pourtant par des nombres, d'une façon qui paraît d'abord assez grossière. On dresse la liste de tous les mots qui peuvent y figurer (le *vocabulaire*) et, pour chacun, on compte combien de fois il apparaît dans le courriel. Chaque mot devient alors un axe. Prenons-en deux, *gratuit* et *réunion*, et plaçons quelques courriels dans leur plan :



Deux mots du vocabulaire, deux axes : chaque courriel devient un point, et les pourriels se rangent d'un côté, les courriels légitimes de l'autre.

Un courriel est donc un point dans un espace qui compte autant de dimensions que le vocabulaire compte de mots, soit des dizaines de milliers. Ce dessin n'en montre que deux. L'opération est la même que pour l'image et ses pixels :



Un axe par mot du vocabulaire, des dizaines de milliers d'axes : le courriel entier devient un seul point de cet espace, comme l'image en devenait un dans celui de ses pixels.

Ces axes ne mesurent rien de physique, ni une taille ni une couleur. Ils contiennent seulement un compte, et l'ordre des mots est perdu. L'idée reste cependant valable : deux courriels qui emploient les mêmes mots sont deux points voisins, et un pourriel ressemble à d'autres pourriels. C'est cette représentation que nous utiliserons, au chapitre « Classer », pour construire un filtre.

On aurait pu décrire les courriels de façon encore plus grossière, en notant seulement si chaque mot est présent ou non, par 1 ou 0, au lieu de compter ses occurrences. Un tel axe n'a que deux barreaux, comme le relief *non/oui* du premier chapitre, et dans notre plan tous les courriels se retrouveraient aux quatre coins d'un carré. Compter les mots ou noter leur simple présence, avec un nombre entier ou avec 0/1, est un choix de représentation, et il en existe beaucoup d'autres. Nous garderons les comptes, qui sont plus précis.

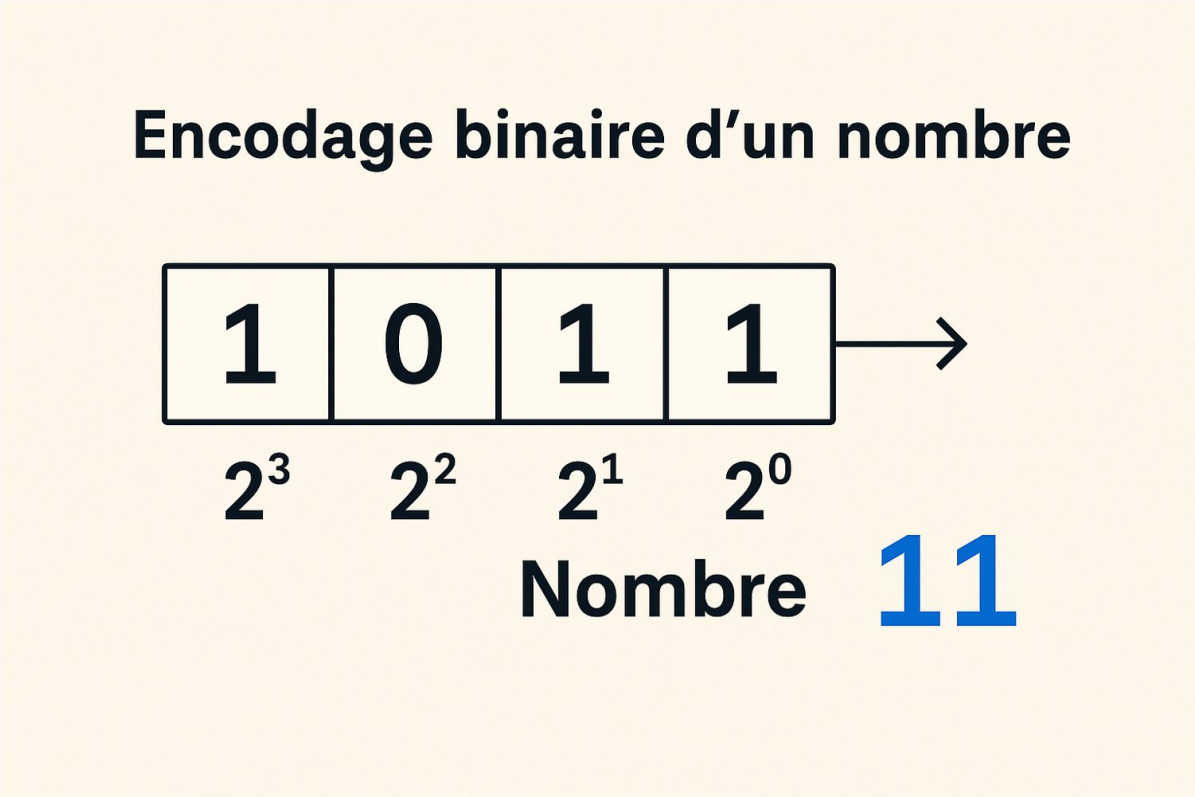
L'idée s'applique donc à des objets très différents. Qu'il s'agisse d'une maison, d'une image ou d'un courriel, dès qu'on sait décrire un objet par des nombres, il devient un point dans un espace, et la ressemblance entre objets se mesure par leur proximité. Le prochain chapitre utilise cette idée pour **prédire par ressemblance**.

Sous le capot : des vecteurs aux bits (optionnel)
Nous avons dit qu'une machine « ne manipule que des nombres ». Il reste à préciser ce qu'est physiquement un nombre pour un ordinateur. Les paragraphes qui suivent descendent du

vecteur jusqu'au signal électrique. Ils ne sont pas indispensables pour la suite du module, mais ils expliquent ce qui se passe à l'intérieur de la machine.

Niveau des bits

Au niveau le plus fondamental, l'ordinateur ne traite qu'un seul type de donnée, le **bit**. Le bit est à la fois un concept mathématique (un symbole dont la valeur ne peut être que `0` ou `1`, ou `vrai/faux` en logique) et une réalité physique, au niveau de l'implémentation : électrique (RAM, CPU, SSD), magnétique (disque dur) ou optique (CD). Les bits *représentent* les nombres au moyen de la convention de l'encodage binaire.



Niveau du processeur (le CPU)

Au niveau suivant se trouve l'ordinateur lui-même, dont le mécanisme central est le microprocesseur (CPU). Un CPU traite les bits sous leur forme physique. Il interprète des « paquets » (ou *mots*) de bits de taille fixe (souvent 32, 64 ou 128 bits) de deux manières très différentes :

1. en tant que *nombre* (ou plus généralement *valeur*) ;
2. en tant qu'*instruction*.

Le flot de bits que reçoit le CPU constitue un *programme*, que le CPU *exécute* dans l'ordre. Un programme dans un « langage machine » fictif pourrait être le suivant :

```
MOV 1000
ADD 0001
STR 2000
```

Les symboles `MOV`, `ADD` et `STR` sont des instructions, qui correspondent elles-mêmes à des nombres. Le CPU pourrait voir la séquence suivante :

```
1000 1000
1001 0001
```

```
1002 2000
```

si `MOV`, `ADD` et `STR` correspondaient par convention aux valeurs 1000, 1001 et 1002. Le programme pourrait signifier ceci :

```
- Prendre la valeur à l'adresse mémoire 1000 et la mettre dans un registre
- Ajouter 1 à cette valeur dans le registre
- Enregistrer le contenu du registre à l'adresse mémoire 2000
```

Le CPU distingue `1000` *instruction* de `1000` *valeur* grâce à des conventions établies à l'avance (par exemple, positions paires pour les instructions et impaires pour les valeurs ; la réalité est un peu plus complexe, mais le principe est celui-là). Pour réaliser une instruction, le CPU exécute un petit programme câblé directement dans ses circuits. C'est à cet endroit que la logique rejoint la matière.

Vous pouvez exécuter vous-même, pas à pas, une version interactive de ce petit programme :
Applet interactive : à utiliser en ligne, sur la page « Regarder les données » du site du cours.

Niveau des langages de programmation

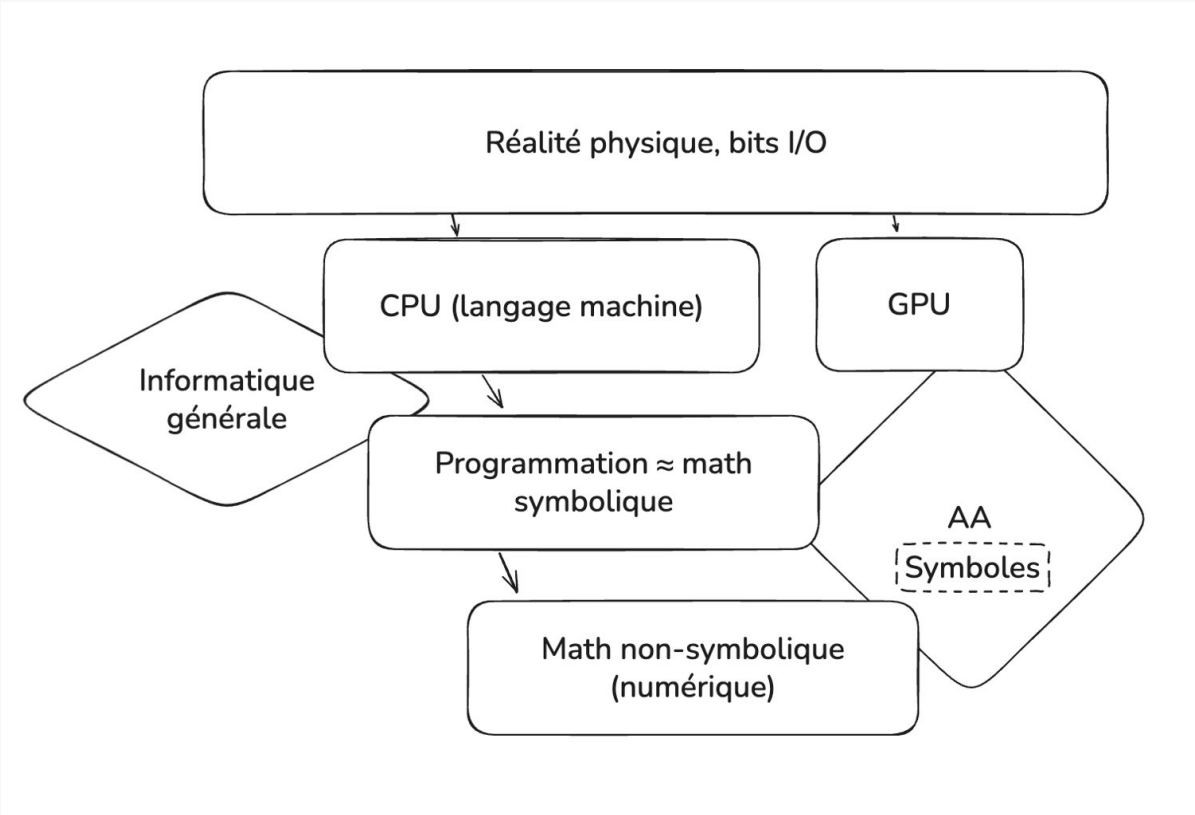
Le niveau suivant est implémenté dans le langage du niveau précédent. De la même façon qu'on peut écrire un jeu ou un système d'exploitation (*operating system*) en langage machine, on peut y écrire un autre langage. Ce langage est plus *abstrait*, plus éloigné de la réalité physique, et il permet d'exprimer des idées plus complexes de façon plus naturelle (C++, Python, JavaScript). On peut le voir comme un « ordinateur virtuel » implémenté au moyen d'un langage moins abstrait. À ce niveau apparaissent des représentations beaucoup plus riches :

- des nombres entiers ;
- des nombres réels (beaucoup plus complexes à représenter) ;
- des chaînes de caractères (*strings*) ;
- des listes de nombres, de mots, de listes… ;
- des images, des sons ;
- etc.

C'est à ce niveau que sont écrits les algorithmes d'apprentissage automatique, et que se trouvent les *vecteurs* dont parle cette page.

Retour vers les symboles

Ces niveaux permettent de mieux comprendre la distinction souvent faite entre l'IA classique, qui manipule des **symboles**, et l'apprentissage automatique, qui manipule des **valeurs numériques** et qu'on qualifie parfois de *sub-symbolique*. Dans les deux cas, les données sont en fin de compte des valeurs numériques (et même des bits physiques). La distinction garde cependant un sens clair, parce que les deux approches reposent sur deux types de mathématiques différents.


Réalité physique, bits I/O
CPU (langage machine)
GPU
Informatique générale
Programmation ≈ math symbolique
AA
Symboles
Math non-symbolique (numérique)

Prédire par ressemblance

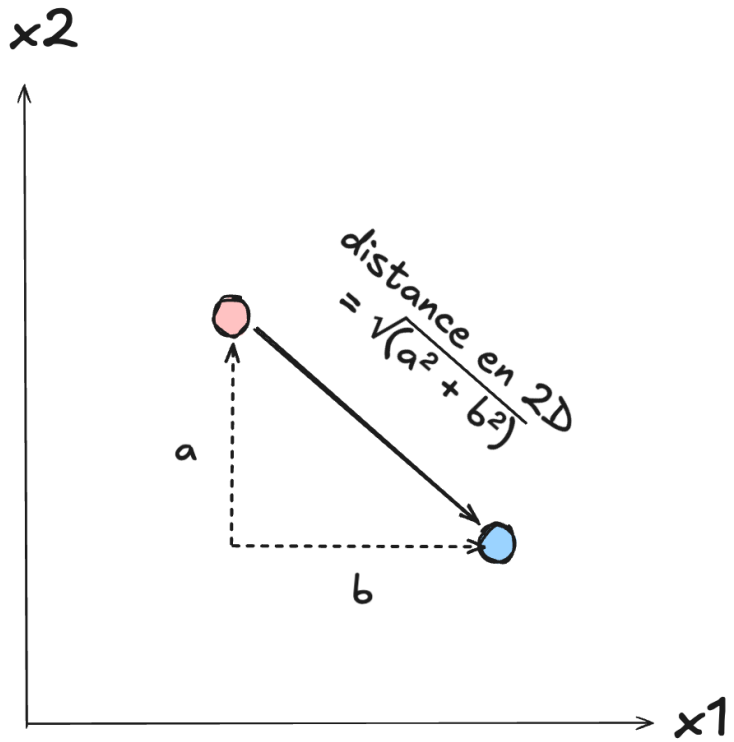
La page « Regarder les données » s'est terminée sur l'idée suivante : si décrire des objets par des nombres transforme leur ressemblance en une distance, il doit exister une façon très simple de prédire. Cette façon est sans doute l'idée la plus intuitive de l'apprentissage automatique : **pour deviner la réponse sur un nouveau cas, on regarde les cas connus qui lui ressemblent le plus, et on reprend leur réponse.**

Pour estimer le prix d'une maison qu'on n'a jamais vue, on cherche les maisons déjà vendues qui lui ressemblent le plus (même quartier, même taille, même âge), et on s'attend à un prix comparable. C'est ce que fait un agent immobilier quand il utilise des « comparables ». Il faut maintenant transformer cette intuition en une procédure précise. Tout repose sur la notion de ressemblance, qu'il faut d'abord rendre mesurable.

Mesurer la ressemblance : la distance

Pour deux objets décrits par des nombres, « se ressembler » a une traduction géométrique directe : être **proches** dans leur espace. La proximité entre deux points se mesure par la **distance**.

Pour deux points dans un plan, il s'agit de la distance ordinaire, c'est-à-dire la longueur du segment droit qui les relie, celle qu'on mesurerait avec une règle. (Les mathématiciens l'appellent la *distance euclidienne*, mais c'est la même notion que dans la vie courante.)



La distance entre deux points : la longueur du trait droit qui les relie.

Cette longueur se calcule avec le théorème de Pythagore. Pour deux points d'un plan, on prend l'écart entre eux sur le premier axe et l'écart sur le second, on élève chacun au carré, on additionne les deux carrés et on prend la racine carrée du résultat :

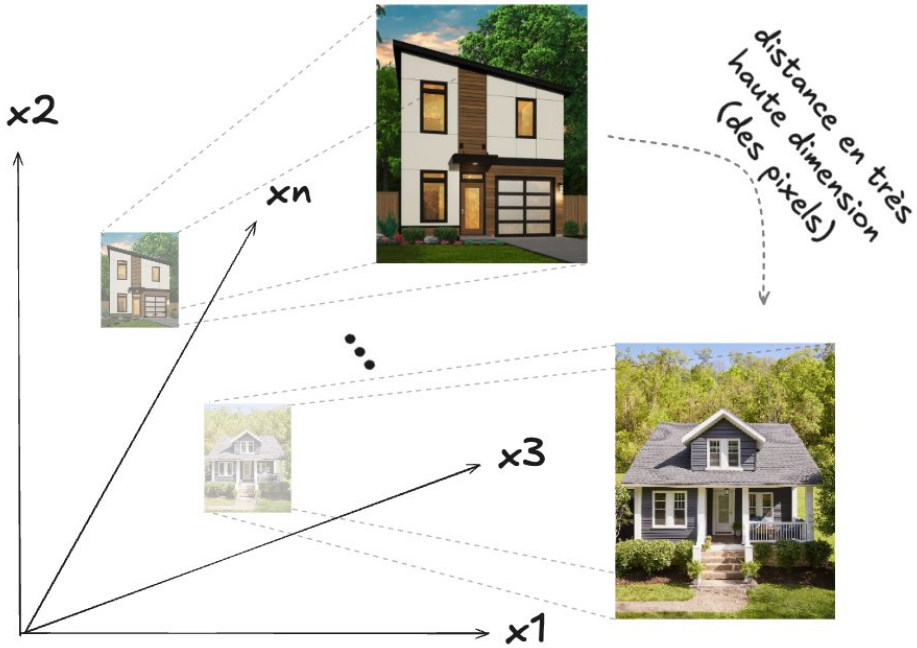
$$\text{distance} = \sqrt{(\text{écart sur l'axe 1})^2 + (\text{écart sur l'axe 2})^2}$$

Rien dans cette méthode ne dépend du nombre d'axes. Si nos maisons ont six caractéristiques, on calcule six écarts au lieu de deux, on les élève au carré, on les additionne et on prend la racine carrée du total. On compare les objets coordonnée par coordonnée et on obtient un seul nombre, petit s'ils se ressemblent, grand s'ils diffèrent. Comme on l'a vu au chapitre « Regarder les données », la même formule s'applique à une image, dont les millions de pixels forment autant de coordonnées. Pour deux photos, l'écart sur un axe est simplement la différence entre la valeur d'un même pixel dans l'une et dans l'autre. On note A_1 la valeur du premier pixel de la photo A, B_1 celle du premier pixel de la photo B, et ainsi de suite jusqu'au n-ième :

$$\text{distance}(A, B) = \sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2 + \cdots + (A_n - B_n)^2}$$

Deux photos identiques donnent une distance nulle, puisque chaque écart est nul. Deux photos qui ne diffèrent que par un pixel donnent un très petit nombre. Plus les pixels diffèrent, en nombre ou

en intensité, plus la distance augmente. La même formule vaut donc du plan à la photo, et n peut valoir deux ou plusieurs millions.



La même distance se calcule, quel que soit le nombre de dimensions.

Une nuance importante. Le fait que la distance se calcule sur des pixels ne signifie pas qu'elle mesure bien la ressemblance entre images. Sur des caractéristiques choisies par un humain (superficie, nombre de pièces…), la proximité a un sens clair. Sur des pixels bruts, elle en a beaucoup moins. Deux photos du même chat, dans deux poses différentes, peuvent être très éloignées pixel à pixel. Une photo et sa version simplement assombrie, presque identiques pour notre œil, peuvent l'être tout autant. Obtenir une distance qui reflète la ressemblance réelle d'objets complexes est un problème à part entière, celui des **bonnes représentations**, que nous retrouverons avec les réseaux de neurones (Module 3) et les plongements (*embeddings*, Module 4). Pour des données tabulaires comme nos maisons, en revanche, la distance brute convient déjà très bien.

La malédiction de la dimension. Une autre raison, plus mathématique, explique pourquoi la distance perd de son sens quand le nombre de dimensions augmente. Dans un plan, vingt maisons suffisent à occuper l'espace, et chacune a des voisines proches. Dans un espace à un million de dimensions, comme celui des pixels, vingt exemples, ou même vingt millions, sont dispersés dans un espace presque vide, et tous les points sont loin les uns des autres.

Les distances entre les points deviennent presque toutes égales, et le « plus proche voisin » n'est plus vraiment proche. C'est la **malédiction de la dimension** (*curse of dimensionality*). Chaque caractéristique ajoutée multiplie la taille de l'espace, et le nombre d'exemples nécessaire pour le remplir croît très rapidement. Ce problème touche directement kNN. C'est l'une des raisons pour lesquelles, sur des images ou du texte, on cherche d'abord à réduire le nombre de dimensions à quelques-unes qui comptent, une idée que les Modules 3 (avec l'autoencodeur) et 4 développeront. Vous avez déjà rencontré ce phénomène sous un autre nom, l'explosion combinatoire du Module 1. Dans ce cas, chaque coup supplémentaire multipliait le nombre de parties à explorer. Ici, chaque caractéristique supplémentaire multiplie l'espace à remplir. Il s'agit de la même croissance exponentielle, que la force brute ne permet pas de traiter et qui demande une méthode plus astucieuse.

Nous disposons donc d'une mesure de ressemblance fiable pour des données tabulaires comme nos maisons. Il reste à l'utiliser pour prédire.

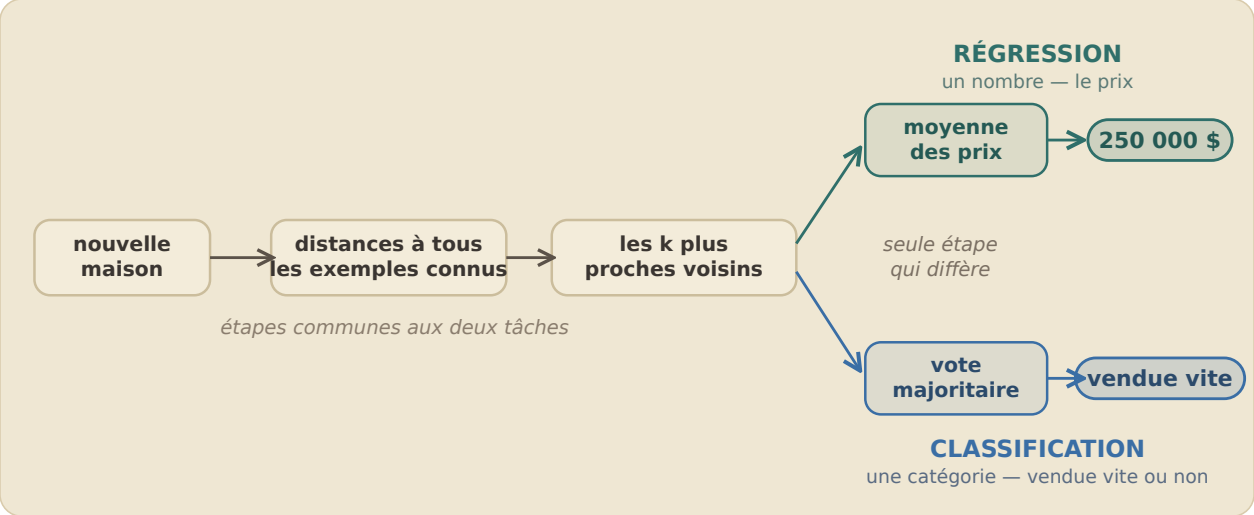
Les k plus proches voisins

Pour prédire le prix d'une maison inconnue, la méthode est très simple. On calcule sa distance à toutes les maisons connues, on retient celles qui lui ressemblent le plus et on prédit la moyenne de leurs prix. On utilise plusieurs voisins plutôt qu'un seul parce qu'un unique voisin serait une base fragile : il pourrait s'agir d'un cas exceptionnel, une aubaine ou une arnaque. En faisant la moyenne de plusieurs voisins, on atténue l'effet de ces cas particuliers.

Le nombre de voisins consultés se note **k**, d'où le nom de l'algorithme : les **k plus proches voisins** (*k-nearest neighbors*, ou kNN). L'idée a été formalisée en 1951 par deux statisticiens, Evelyn Fix et Joseph Hodges, dans un rapport rédigé pour l'armée de l'air américaine. Elle relève donc de la statistique avant de relever de l'informatique.

Nous venons de faire une **régression**, puisque la cible était un prix, donc un nombre, et que nous l'avons obtenu en faisant la moyenne des voisins. Pour une **classification**, il suffit de changer la dernière étape. Au lieu de faire la moyenne des réponses des voisins, on retient la réponse la plus fréquente, par un **vote majoritaire**. Pour prédire si une maison partira vite, on regarde ce qui s'est passé pour ses plus proches voisines et on suit la majorité.

kNN a donc la particularité de faire les deux sans rien changer d'essentiel. La distance et les voisins sont les mêmes, et seule la façon de combiner leurs réponses diffère. La plupart des algorithmes que nous verrons ensuite se spécialiseront dans l'une ou l'autre tâche.



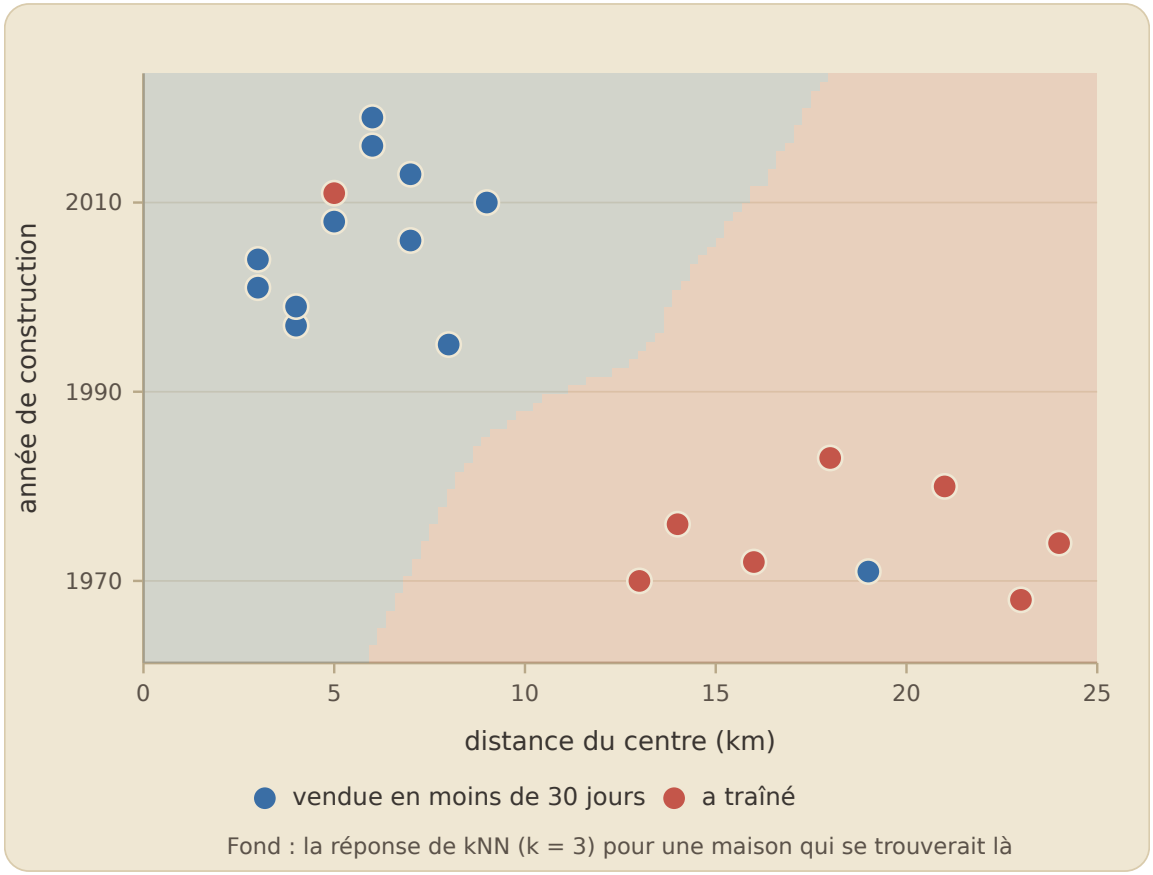
kNN suit les mêmes étapes pour la régression et la classification ; seule la dernière (moyenne ou majorité) les distingue.

La recette des *k* plus proches voisins, pour prédire à propos d'une nouvelle maison :

1. Calculer la **distance** entre cette maison et *chacune* des maisons déjà connues (à partir de leurs caractéristiques).
2. Garder les **k** maisons les plus proches, ses « voisins ».
3. Combiner les réponses de ces voisins :
   - pour un **nombre** (régression) → prendre leur **moyenne** ;
   - pour une **catégorie** (classification) → prendre leur **majorité**.

Seule cette dernière étape distingue les deux tâches ; tout le reste est identique.

Revenons à nos maisons, dans le plan de la seconde question (distance du centre, année de construction). On peut demander à kNN sa réponse pour une nouvelle maison placée à n'importe quel endroit de ce plan. En posant la question pour chaque point du plan et en colorant ce point selon la réponse obtenue, on obtient une carte : bleu pâle si les k voisins votent « vendue vite », rouge pâle s'ils votent « a traîné ». Voici le résultat pour $k = 3$:



La frontière de décision de kNN ($k = 3$) sur nos maisons : le fond donne la réponse du modèle en chaque point du plan, et la ligne où la couleur change est la frontière.

Le plan est ainsi divisé en deux **régions**. La ligne où la couleur change, qui passe dans la bande vide entre les deux amas, s'appelle la **frontière de décision** (*decision boundary*). Personne ne l'a tracée : elle résulte de l'application de la règle en chaque point. C'est elle qui détermine la prédiction. Une maison située d'un côté sera classée « vendue vite », et une maison située de l'autre côté sera classée « a traîné », sans autre nuance. On peut aussi observer ce qui arrive aux deux exceptions du premier chapitre. Avec $k = 3$, chacune se trouve dans la région de la couleur opposée, puisque ses trois voisins les plus proches votent pour l'autre catégorie.

Cette observation vaut pour tout classificateur, et pas seulement pour kNN : **classer revient à diviser l'espace en régions, et un modèle de classification est entièrement décrit par la frontière qu'il trace.** C'est pour cette raison que la classification est plus facile à visualiser que la régression : en deux dimensions, on voit la frontière d'un seul coup d'œil. Au chapitre « Classer », nous verrons des modèles dont la frontière est une simple droite. Celle de kNN peut prendre n'importe quelle forme.

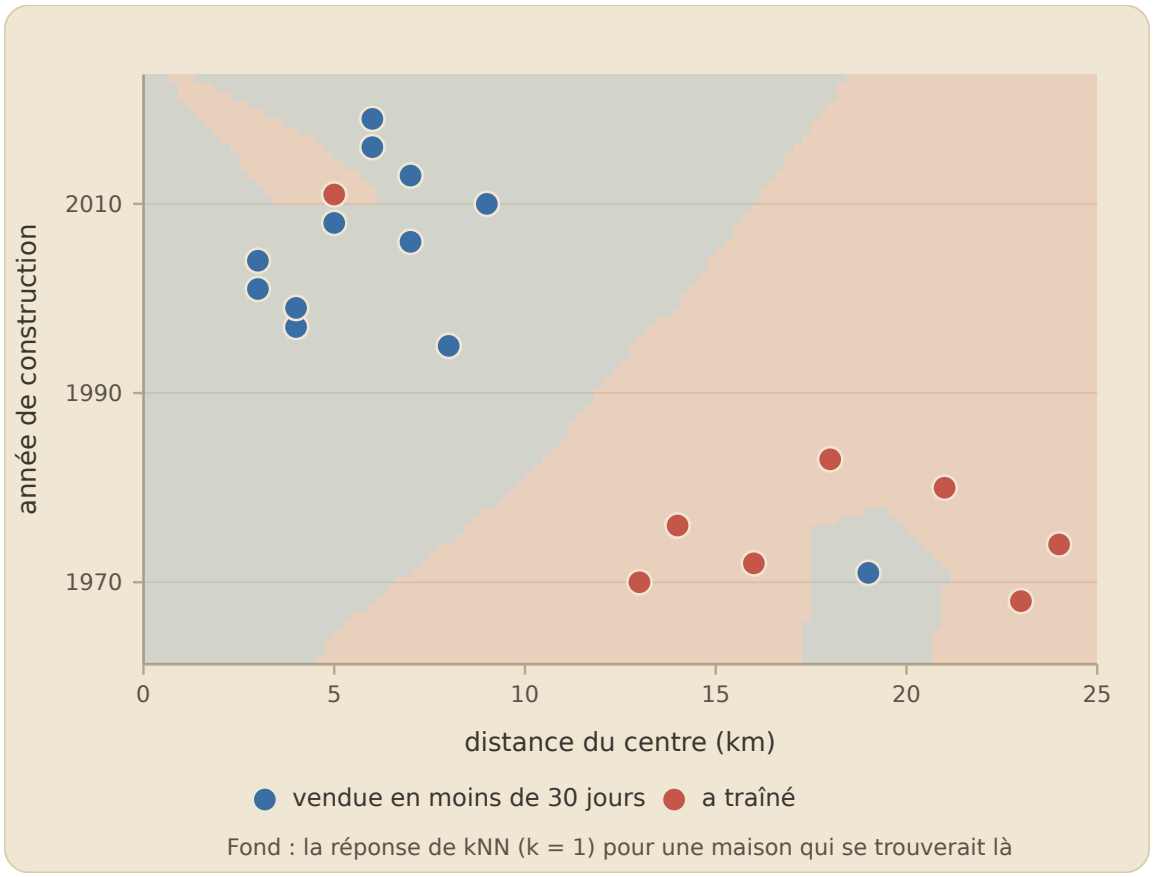
L'applet ci-dessous permet de manipuler cette frontière. Il y a deux catégories, des points rouges et des points bleus. Chaque point coloré est un exemple connu, et le fond montre, comme plus haut, la prédiction de kNN pour tout nouveau point. Ajoutez des points, déplacez-les, faites varier k et observez comment la frontière de décision change.

Applet interactive : à utiliser en ligne, sur la page « Prédire par ressemblance » du site du cours.

Le choix de k

L'applet soulève rapidement une question : quelle valeur faut-il donner à **k** ?

Les deux extrêmes sont instructifs. Avec **k = 1**, chaque prédiction ne s'appuie que sur le voisin le plus proche. La frontière suit alors le moindre détail, fait le tour de chaque point individuel et devient très irrégulière. Le modèle suit les exemples connus de si près qu'il réagit au moindre point aberrant (*outlier*). Ce défaut porte un nom, et il est central dans tout ce qui suit : le **sur-apprentissage** (*overfitting*), qui consiste à apprendre les exemples eux-mêmes au lieu d'apprendre à partir d'eux. Nous y reviendrons en détail dans *Généraliser*. Nos maisons le montrent : avec $k = 1$, chacune des deux exceptions crée autour d'elle une petite zone de sa couleur dans la région opposée, et la frontière se découpe en cellules anguleuses.



La même frontière avec $k = 1$: chaque exception crée sa propre petite zone, et la frontière suit chaque point.

À l'autre extrême, avec un **k très grand**, chaque prédiction fait la moyenne de tant de voisins que les particularités locales disparaissent. La frontière devient lisse, parfois au point d'ignorer des structures qui existent réellement dans les données.

Une « bonne » valeur se trouve entre les deux, ni trop petite ni trop grande. La façon de la trouver est une question qui paraît simple, mais qui est l'une des plus importantes de l'apprentissage automatique. Elle ne concerne pas seulement kNN : tout modèle doit être assez souple pour saisir les vraies régularités des données, sans l'être au point de reproduire leurs variations dues au hasard.

Cette question est assez centrale pour que nous lui consacrions une page entière, *Généraliser*, une fois

que nous connaîtrons quelques modèles de plus. Pour l'instant, retenez seulement l'idée suivante : **k règle un compromis entre « coller aux exemples » et « lisser à l'excès ».**

L'angle mort de kNN

kNN a une particularité : il n'a, à proprement parler, **rien à apprendre**. Il n'y a ni entraînement ni paramètres à régler. Il suffit de garder en mémoire tous les exemples connus et de les consulter au moment de prédire. Les données constituent le modèle.

Cette simplicité a cependant deux inconvénients, qui motivent la suite du module.

Le premier est que kNN est **coûteux**. Pour chaque nouvelle prédiction, il doit calculer la distance à tous les exemples connus, sans exception. Avec quelques dizaines de maisons, ce n'est pas un problème. Pour un système qui doit reconnaître un visage parmi des millions d'images, ou répondre en une fraction de seconde à des millions d'utilisateurs, tout recalculer à chaque fois devient très coûteux. kNN reporte tout le travail au moment de la prédiction, qui est le moment où il coûte le plus cher.

Le second inconvénient est plus fondamental : kNN **ne dégage aucune compréhension** des données. Il n'en tire aucune règle, aucune tendance, aucune forme générale. Il ne « sait » pas que les grandes maisons coûtent plus cher : il se contente de retrouver des voisins. Le modèle le plus bête avait, lui, résumé toute sa connaissance en **un seul nombre**, le prix moyen. kNN fait l'inverse : il ne résume rien et garde tout.

Or c'est justement ce résumé qui nous intéresse. On voudrait qu'un modèle **apprenne** quelque chose des données, c'est-à-dire qu'il en extraie quelques paramètres qui représentent la tendance générale, quitte à ne plus avoir besoin des exemples ensuite. Un tel modèle serait léger à utiliser et contiendrait une forme de compréhension des données.

Le prochain chapitre, « Un modèle qui s'entraîne », montre comment construire un tel modèle.

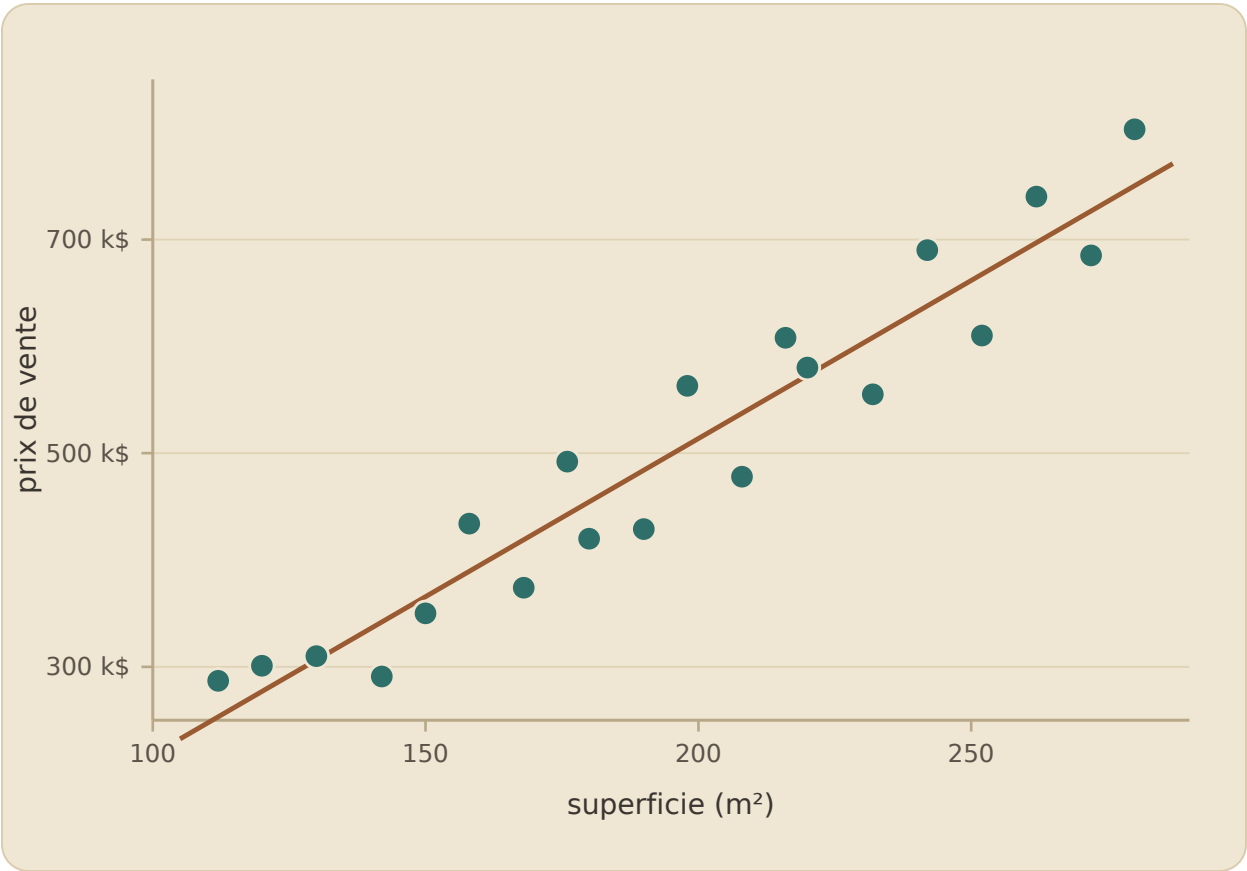
Un modèle qui s'entraîne

Le chapitre « Prédire par ressemblance » s'est terminé sur un objectif : obtenir un modèle qui ne se contente pas de mémoriser les exemples, mais qui en extrait une tendance générale, résumée en quelques paramètres, qu'on pourra ensuite appliquer sans conserver toutes les données.

Nous allons construire ce modèle. Comme nous cherchons le plus simple, nous allons faire l'opération la plus élémentaire qu'on puisse faire sur un nuage de points : **y faire passer une droite.**

Un modèle qui tient en deux nombres

Reprenons notre nuage de maisons (la superficie à l'horizontale, le prix à la verticale). Faire passer une droite à travers ce nuage revient à supposer qu'il existe une relation simple et régulière entre la taille d'une maison et son prix, du type « chaque mètre carré supplémentaire ajoute environ tant de dollars ». Cette droite est notre modèle. C'est aussi le plus ancien modèle de ce module. Legendre et Gauss ajustaient déjà des droites par la méthode des moindres carrés (*least squares*) vers 1805 pour prédire la trajectoire des comètes, et c'est Francis Galton qui, en 1886, lui a donné le nom de *régression*, en étudiant la taille des enfants par rapport à celle de leurs parents.



La droite qui suit la tendance du nuage : c'est notre modèle.

En mathématiques, une droite se résume à deux nombres :

$$\text{prix} = m \times \text{superficie} + b$$

- **m**, la *pente* (*slope*) : de combien le prix monte quand la superficie augmente d'une unité ;
- **b**, l'*ordonnée à l'origine* (*intercept*) : le point de départ, là où la droite croise l'axe vertical.

Ces deux nombres, m et b, sont les **paramètres** du modèle, et ils ont ici une signification. Le modèle le plus bête résumait tout en un seul nombre (la moyenne), et kNN n'avait aucun paramètre. Notre droite en a deux, ce qui suffit pour représenter une *tendance*, avec une direction et une hauteur. Si on change m et b, on obtient une autre droite, donc un autre modèle. Le problème consiste à trouver le couple (m, b) qui donne la droite la mieux ajustée au nuage.

Essayez vous-même. Dans l'applet ci-dessous, déplacez la droite (vous ajustez ainsi m et b à la main) et cherchez la position qui colle le mieux aux points.

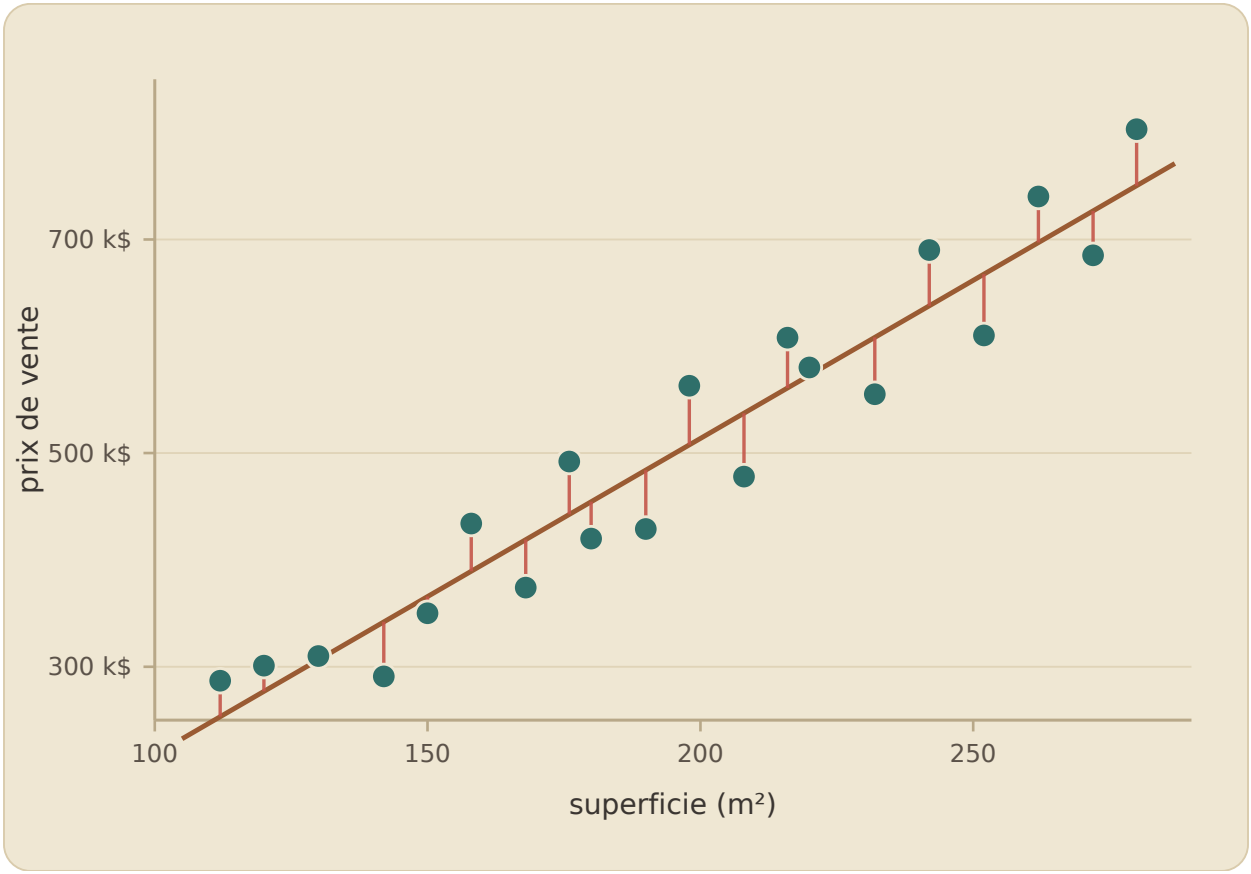
Applet interactive : à utiliser en ligne, sur la page « Un modèle qui s'entraîne » du site du cours.

Mesurer l'erreur

En déplaçant la droite, vous avez cherché à la placer « bien ». Il faut cependant préciser ce que « bien » veut dire. Il nous faut une mesure, que nous avons déjà rencontrée avec le modèle le plus bête sans

encore la définir précisément : l'**erreur**.

Pour une droite donnée, l'erreur sur une maison est l'écart entre le prix que la droite *prédit* (le point de la droite à la verticale de cette maison) et son prix *réel*. Comme pour le modèle le plus bête, cet écart est un segment vertical. La différence est qu'ici la droite est inclinée, et qu'elle peut donc passer beaucoup plus près des points qu'une droite horizontale.



Les mêmes erreurs qu'avec le modèle le plus bête, mais par rapport à une droite inclinée : elles sont beaucoup plus courtes.

L'erreur totale du modèle combine tous ces écarts. On les met au carré, pour que les écarts au-dessus et en dessous de la droite ne s'annulent pas et pour pénaliser davantage les grands écarts, puis on en fait la moyenne. Ce nombre, la moyenne des carrés des écarts, porte un nom technique, l'*erreur quadratique moyenne* (*mean squared error*), mais l'idée est simple : **plus il est petit, mieux la droite s'ajuste au nuage.** (Considérez le cas limite. Si la droite passait exactement par tous les points, chaque écart serait nul, chaque carré aussi, et l'erreur vaudrait zéro. C'est la valeur minimale possible. Avec un nuage comme le nôtre, aucune droite ne l'atteint, et l'objectif est de s'en approcher le plus possible.)

On peut se représenter cette erreur de la façon suivante. Imaginez que chaque point est relié à la droite par un petit ressort vertical. Un point éloigné tire fort, un point proche tire à peine. La meilleure droite est celle où les forces de tous ces ressorts s'équilibrent, c'est-à-dire la position de moindre tension. Cette position correspond à la droite de plus petite erreur. Vous pouvez l'observer dans l'applet suivante :

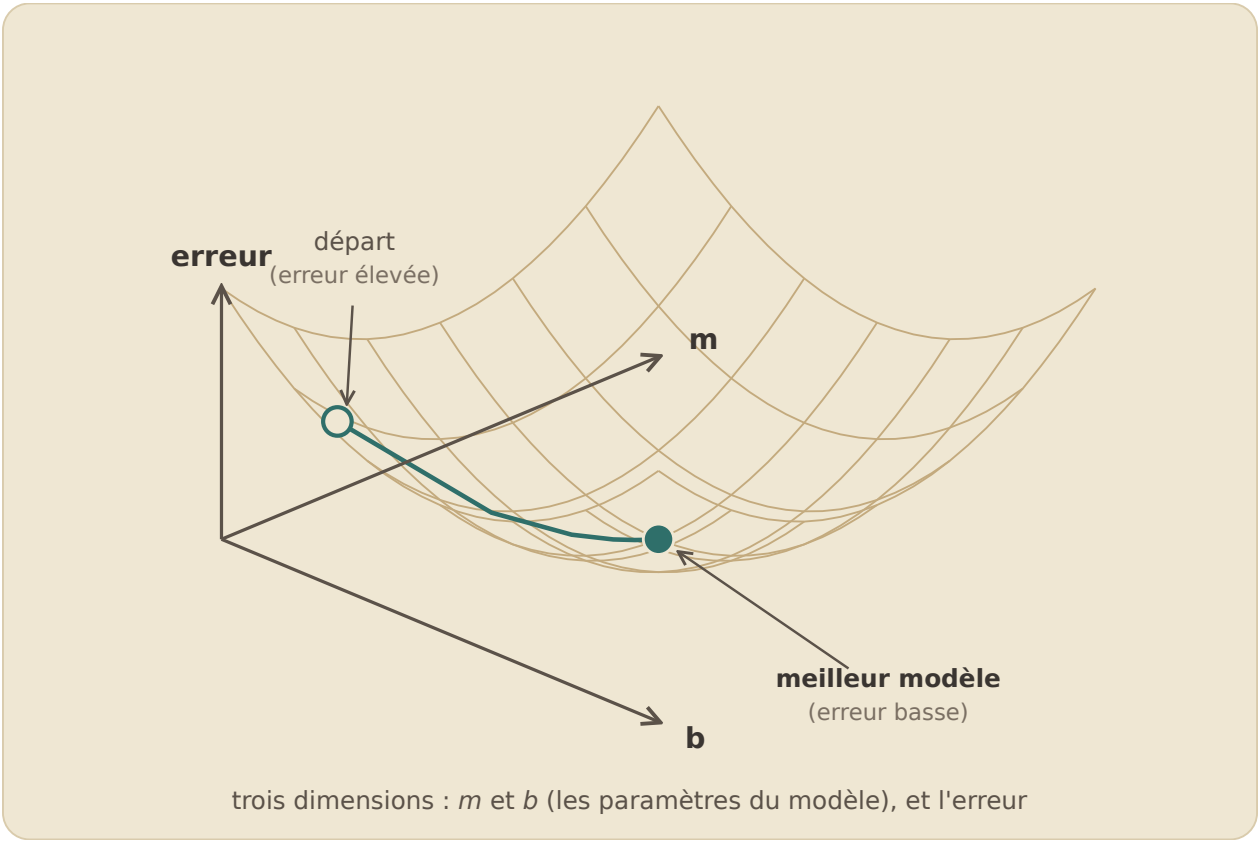
Applet interactive : à utiliser en ligne, sur la page « Un modèle qui s'entraîne » du site du cours.

Trouver cette droite à la main, comme dans l'applet, reste possible en deux dimensions. Il reste à voir comment une machine peut la trouver seule, y compris quand le modèle n'a plus deux paramètres, mais des milliers. C'est l'objet de la section suivante.

Apprendre, c'est descendre la pente

L'idée présentée dans cette section est au cœur de tout l'apprentissage automatique moderne, sous une forme ou une autre, y compris dans les plus grands modèles actuels.

Repartons de l'erreur. Pour chaque choix de paramètres (m, b), le modèle commet une certaine erreur totale. L'erreur est donc elle-même une *fonction* des paramètres. On peut la représenter comme un **paysage**. Les deux paramètres sont les coordonnées sur une carte (est-ouest pour m, nord-sud pour b), et l'erreur est l'**altitude** en chaque point. Le paysage a donc **trois dimensions** : deux pour les paramètres, m et b, et une troisième, verticale, pour l'erreur. Les sommets correspondent aux mauvais modèles (grande erreur), et les vallées aux bons. Trouver le meilleur modèle revient à trouver le **point le plus bas** de ce paysage.



La descente de gradient : l'erreur forme une cuvette au-dessus des réglages (m, b) ; la bille descend jusqu'au creux, qui correspond au meilleur modèle.

La machine ne connaît pas la forme complète de ce paysage. Elle procède comme un randonneur dans le brouillard : elle mesure la **pente** à l'endroit où elle se trouve et fait un pas dans la direction qui descend le plus. Puis elle recommence. Pas après pas, elle descend vers le creux, comme une bille posée sur le flanc d'une vallée roule vers le fond. Quand la pente devient nulle, le fond est

atteint, et les meilleurs paramètres sont trouvés. Cette méthode s'appelle la **descente de gradient** (*gradient descent* ; le « gradient » désigne la direction de plus forte pente). C'est le mécanisme central de l'apprentissage.

La taille des pas a de l'importance. Si les pas sont trop grands, on risque de dépasser le creux et d'osciller autour sans l'atteindre. S'ils sont trop petits, la descente est très longue. Ce réglage, le *taux d'apprentissage* (*learning rate*), n'est pas un paramètre du modèle, mais un réglage de la *procédure*. On l'appelle un **hyper-paramètre** (*hyperparameter*).

Cette idée rejoint la fin du Module 1, où nous annoncions qu'« *apprendre, c'est encore chercher, mais dans un autre espace* ». Il ne s'agit plus d'explorer les coups d'une partie d'échecs, mais l'ensemble très vaste des réglages possibles d'un modèle. La descente de gradient est cette recherche. Le GOFAI cherchait la solution elle-même, alors que l'apprentissage cherche les paramètres qui permettent de la produire. De plus, la méthode reste la même quelle que soit l'échelle : que le paysage ait deux dimensions (m et b) ou plusieurs milliards (les poids d'un grand réseau de neurones), il s'agit toujours de descendre la pente.

Les mathématiques de la régression linéaire (optionnel)
L'erreur quadratique moyenne, pour n maisons, s'écrit :

$$J(m,b) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2$$

où x_i est la superficie de la maison i, y_i son vrai prix, et $mx_i + b$ le prix prédit. Cette fonction correspond à la « carte d'altitude » du paysage : à chaque couple (m, b), elle associe une hauteur d'erreur.
La descente de gradient mesure, en un point, la pente de cette surface dans chaque direction (les *dérivées partielles*) :

$$\frac{\partial J}{\partial m} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (mx_i + b)), \qquad \frac{\partial J}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (mx_i + b))$$

puis fait un pas dans le sens inverse de la pente, d'une taille réglée par le taux d'apprentissage α :

$$m \leftarrow m - \alpha\,\frac{\partial J}{\partial m}, \qquad b \leftarrow b - \alpha\,\frac{\partial J}{\partial b}$$

On répète ces étapes jusqu'à ce que l'erreur ne diminue plus. Pour la régression linéaire, il existe même une formule directe (les moindres carrés de Gauss) qui donne la solution en une seule étape. La descente de gradient a cependant l'avantage de fonctionner pour des modèles beaucoup plus complexes, jusqu'aux réseaux de neurones.

Et pour prédire une catégorie ?

Nous avons maintenant un modèle qui apprend : une droite, deux paramètres, une erreur à minimiser et une descente vers le creux. Ce modèle prédit un **nombre**, ici un prix.

Beaucoup de questions demandent cependant comme réponse une **catégorie** plutôt qu'un nombre. Par exemple, ce courriel est-il un pourriel ou non ? Cette photo montre-t-elle un chat ou un chien ? Nous avons rencontré ce type de tâche au chapitre sur kNN : la **classification**. Cependant, kNN

n'apprenait rien. Nous voulons maintenant un modèle qui s'entraîne comme notre droite, mais dont la sortie est une catégorie plutôt qu'une valeur.

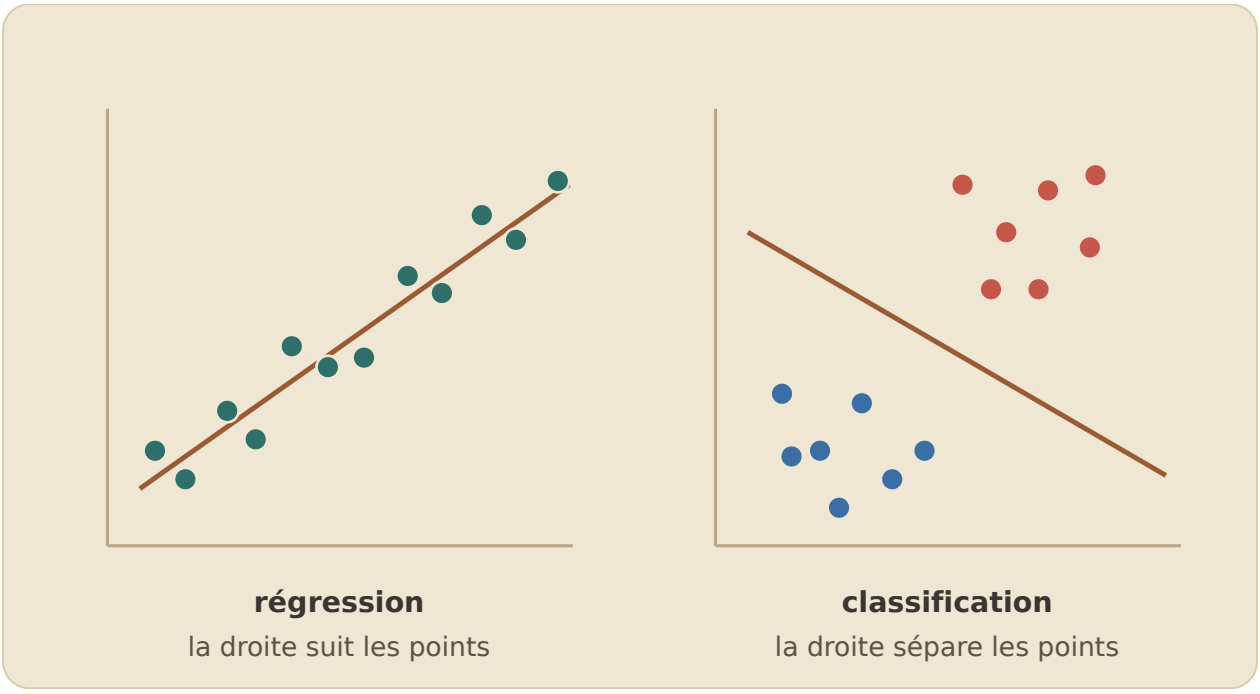
Presque tous les éléments que nous venons de construire vont servir de nouveau. Un modèle réglé par des paramètres, une fonction d'erreur et une descente de gradient pour la minimiser forment un ensemble assez général pour s'appliquer à la classification comme à la régression. Il suffira de modifier la *forme* du modèle (une droite qui *sépare* les points, plutôt qu'une droite qui *suit* leur tendance) et la *façon de calculer l'erreur*. C'est l'objet du prochain chapitre, « Classer ».

Classer

Le chapitre « Un modèle qui s'entraîne » a présenté un modèle qui apprend vraiment : une droite, deux paramètres, une erreur à minimiser et une descente vers le creux. Ce modèle répond cependant toujours par un **nombre**, ici un prix. Or beaucoup de questions n'appellent pas un nombre, mais une **catégorie**. Par exemple, on peut se demander si un courriel est un pourriel, si une tumeur est bénigne ou maligne, ou si une photo montre un chat ou un chien. C'est la tâche de **classification**, déjà rencontrée au chapitre kNN. Cette fois, cependant, nous voulons un modèle qui s'entraîne.

Comme l'annonçait la fin du chapitre précédent, presque toute la machinerie va servir de nouveau. Des paramètres réglables, une fonction d'erreur et une descente de gradient pour la minimiser forment un ensemble assez général pour s'appliquer à la classification comme à la régression. Seules deux choses changent : la **forme** du modèle et la **façon de compter l'erreur**.

Pour la forme, le changement est simple. En régression, la droite suivait le nuage : elle passait à travers les points pour en représenter la tendance. En classification, la droite sépare le nuage : elle passe entre deux groupes pour les départager. L'objet est le même (une droite, deux paramètres), mais son rôle est inversé.



Deux usages de la même droite : à gauche elle suit le nuage (régression), à droite elle le sépare (classification).

Pour simplifier, laissons de côté les maisons et prenons le cas le plus simple : des points de deux couleurs, `bleus` et `rouges`, dispersés dans le plan (deux caractéristiques, x_1 et x_2). Apprendre à classer consiste alors à trouver la droite qui place le mieux les bleus d'un côté et les rouges de l'autre. Il existe deux grandes façons d'y arriver, deux approches que nous allons examiner l'une après l'autre.

Tracer une frontière : la régression logistique

La première approche est la plus directe. Pour séparer les bleus des rouges, il suffit de **tracer la frontière** entre eux. Il n'est pas nécessaire de comprendre ce qui distingue un bleu d'un rouge, seulement de trouver où passe la ligne. C'est l'approche dite **discriminative** : le modèle apprend à discriminer les classes, sans chercher à les décrire.

Dans notre plan, cette frontière est une droite, et nous savons déjà qu'une droite se définit par deux paramètres, une pente et une hauteur. Son rôle a toutefois changé. En régression, on lisait la droite verticalement : à telle superficie correspond tel prix. Ici, on regarde de quel **côté** de la ligne tombe le point. D'un côté, on répond `bleu`, de l'autre, `rouge`. La même équation, $mx_1 + b$, ne sert plus à calculer une valeur mais à partager le plan en deux.

Cet algorithme s'appelle la **régression logistique** (*logistic regression*). Il a été mis au point par des statisticiens dans les années 1950 et, malgré son nom (il contient « régression » alors qu'il classe), c'est l'un des classificateurs les plus utilisés. Dans l'applet ci-dessous, déplacez la ligne de décision pour séparer au mieux les deux groupes. Vous ajustez ainsi ses deux paramètres à la main, comme vous déplaciez la droite de régression au chapitre précédent. Vous pouvez aussi ajouter, retirer ou déplacer des points.

Applet interactive : à utiliser en ligne, sur la page « Classer » du site du cours.

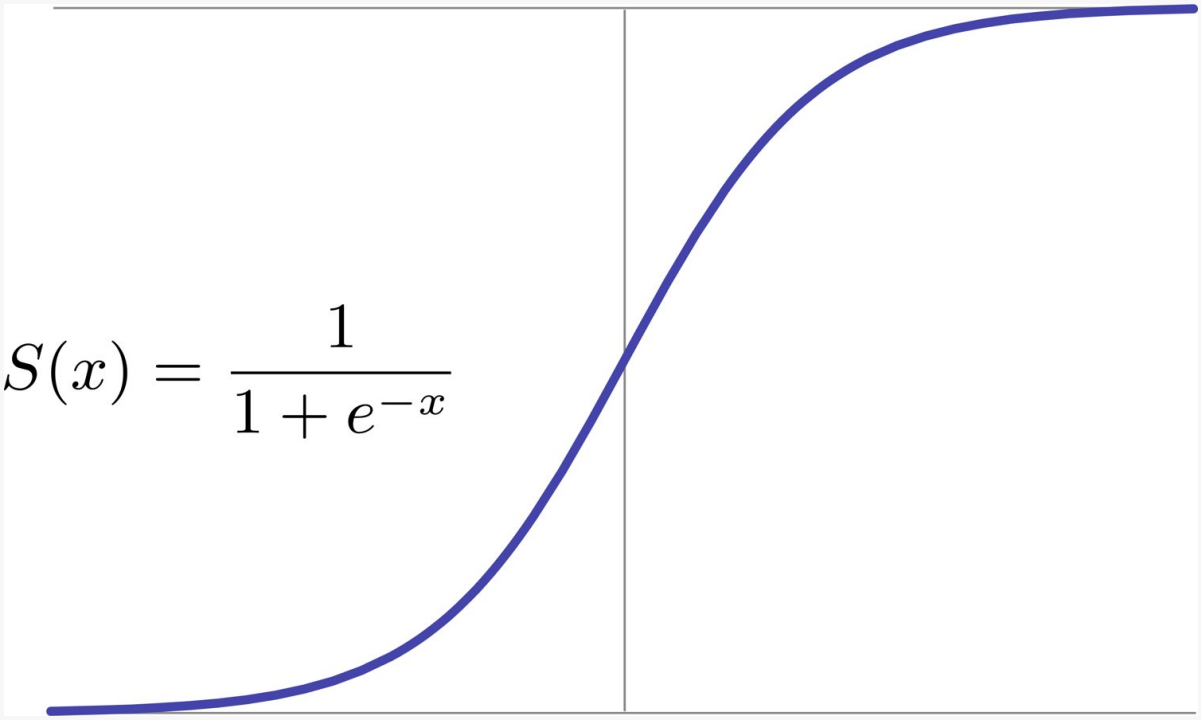
Le second changement concerne l'erreur. On ne peut plus mesurer une distance verticale au point, puisqu'on ne prédit plus une valeur. On compte plutôt les cas où le modèle se **trompe de côté** : un point bien placé ne coûte rien, un point du mauvais côté coûte cher. La barre à droite de l'applet affiche cette erreur, soit la part des points qui sont du mauvais côté. Cherchez à la rendre la plus petite possible. Elle vaut zéro quand la ligne sépare parfaitement les deux couleurs.

Vous remarquerez deux choses. D'abord, il n'est **pas toujours possible** d'atteindre zéro : si les couleurs se chevauchent, aucune droite ne les sépare complètement. Ensuite, l'erreur ne dépasse jamais **50 %**. Même la pire ligne classe correctement la moitié des points par hasard, et si elle fait moins bien, le modèle n'a qu'à inverser sa convention (« ce côté-ci est rouge, pas bleu ») pour repasser sous ce seuil.

> Matière à réflexion : pourquoi n'est-il pas toujours possible de séparer parfaitement les deux groupes par une droite ? Dans quelles conditions y arrive-t-on ? Et qu'est-ce qui pourrait rendre la chose possible quand elle ne l'est pas ? (Nous y reviendrons : c'est l'une des questions centrales du Module 3, dans « Une couche cachée ».)

Il reste à savoir comment la machine trouve seule la bonne ligne, sans qu'on la déplace à la souris. La réponse est la même qu'au chapitre précédent. L'erreur est une fonction des deux paramètres, ce qui définit un paysage, et la **descente de gradient** parcourt ce paysage jusqu'à son creux. Le mécanisme est réutilisé tel quel. Seule la forme de la fonction d'erreur diffère. Les détails sont donnés ci-dessous.

Les mathématiques de la régression logistique (optionnel)
Nous l'avons présentée en termes géométriques, mais la régression logistique est en réalité une méthode *probabiliste*. Plutôt que de répondre simplement `bleu` ou `rouge`, elle estime la **probabilité** qu'un point soit bleu. Un point loin de la frontière, du côté bleu, sera bleu « à 99 % ». Un point situé sur la ligne sera bleu « à 50 % », ce qui correspond à l'incertitude maximale. Pour transformer la position d'un point (une valeur quelconque) en une probabilité (un nombre compris entre 0 et 1), on utilise la **fonction sigmoïde**, ou logistique, qui donne son nom à l'algorithme. Sa courbe en S ramène n'importe quelle valeur dans l'intervalle $[0, 1]$:



Adoptons la notation habituelle de l'apprentissage automatique. Un point est un vecteur $\mathbf{x} = [x_1, x_2]$. Sa vraie classe est $y \in 0, 1$ (0 pour rouge, 1 pour bleu, de façon arbitraire). Les paramètres forment un vecteur $\mathbf{w} = [w_1, w_2]$ accompagné de b. On calcule d'abord un **score**, qui indique de combien et de quel côté le point s'écarte de la frontière :

$$z = w_1 x_1 + w_2 x_2 + b$$

On le passe ensuite dans la sigmoïde pour obtenir la probabilité estimée :

$$\hat{y} = \frac{1}{1 + e^{-z}}$$

Ici, $\hat{y} \in [0, 1]$ est une probabilité, à distinguer de $y \in 0, 1$, la vraie classe. La décision finale est la suivante : bleu si $\hat{y} \geq 0{,}5$, rouge sinon.
L'erreur sur un point compare la probabilité prédite $\hat{y}$ à la vraie classe y. On utilise l'**entropie croisée** (*cross-entropy*) :

$$E(y, \hat{y}) = -[\, y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \,]$$

Cette fonction se comporte comme on le souhaite. Si le point est bleu ($y = 1$) et que le modèle

en est sûr ($\hat{y} = 0{,}9$), l'erreur est faible ($-\log 0{,}9 \approx 0{,}1$). S'il se trompe avec assurance ($\hat{y} = 0{,}1$ pour un vrai bleu), l'erreur devient grande ($-\log 0{,}1 \approx 2{,}3$). Une confiance mal placée est donc fortement pénalisée.
L'erreur totale sur les n points est la moyenne de ces erreurs :

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} E(y^{(i)}, \hat{y}^{(i)})$$

C'est cette fonction $J(\mathbf{w}, b)$ qui joue le rôle de paysage : à chaque choix de paramètres correspond une hauteur d'erreur. La descente de gradient en mesure la pente,

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)})\, \mathbf{x}^{(i)}, \qquad \frac{\partial J}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}^{(i)} - y^{(i)})$$

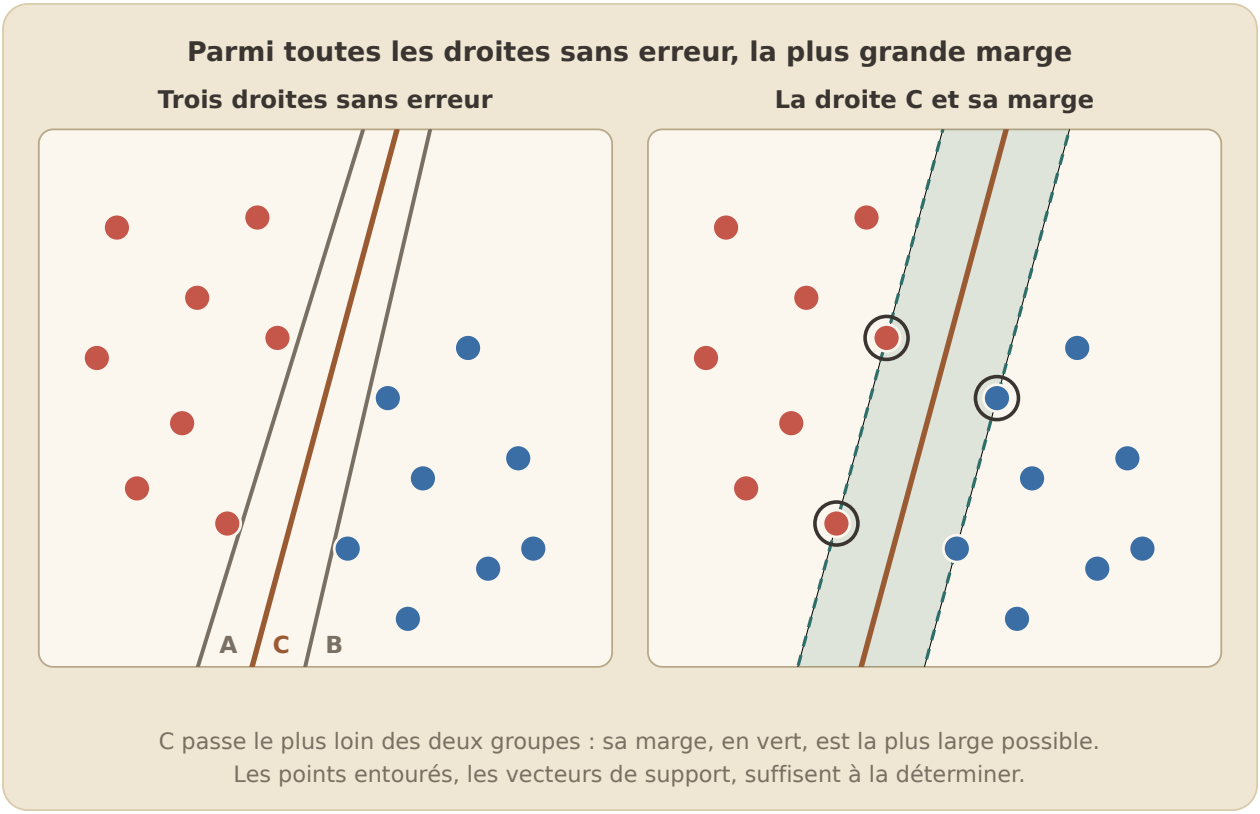
puis fait un pas en sens inverse, d'une taille fixée par le taux d'apprentissage α :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}, \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}$$

On répète jusqu'à ce que l'erreur ne diminue plus. C'est le même mécanisme qu'au chapitre précédent. Seule la fonction d'erreur a changé.

La plus grande marge : les machines à vecteurs de support

Dans l'applet de la section précédente, quand les deux groupes ne se chevauchent pas, on trouve facilement une droite qui ne fait aucune erreur. On en trouve même une infinité : on peut la déplacer ou la faire pivoter un peu, et elle sépare toujours parfaitement les deux couleurs. L'erreur ne permet donc pas de choisir entre elles. Laquelle faut-il préférer ?



Plusieurs droites séparent les deux groupes sans erreur ; celle de plus grande marge passe le plus loin des deux.

L'intuition répond : celle qui passe au milieu, le plus loin possible des deux groupes. Une droite qui frôle les points rouges a toutes les chances de mal classer le prochain point rouge, s'il tombe un peu à côté des autres. Une droite éloignée des deux groupes laisse de la place à ces variations.

La marge. Autour d'une droite qui sépare les deux groupes, on peut tracer une bande vide, aussi large que possible, qui ne contient aucun point. Sa largeur s'appelle la **marge**. La **machine à vecteurs de support** (*support vector machine*, ou SVM) choisit, parmi toutes les droites qui séparent les deux groupes, celle dont la marge est la plus large. En français, on l'appelle aussi **séparateur à vaste marge**, un nom qui garde le sigle SVM et décrit bien la méthode.

Les vecteurs de support. Quand la bande est la plus large possible, quelques points en touchent les bords, souvent deux ou trois. Ce sont eux qui la bloquent : on ne peut pas l'élargir sans les englober. On les appelle les **vecteurs de support**, parce que la frontière s'appuie sur eux (un point du plan est un vecteur, au sens du chapitre « Regarder les données »). Tous les autres points n'ont aucune influence sur la frontière : on peut les déplacer, les retirer ou en ajouter loin de la bande, sans que rien change. C'est une différence importante avec la régression logistique, où chaque point tire un peu sur la frontière.

Dans l'applet ci-dessous, la frontière est recalculée à chaque modification. La bande colorée montre la marge, et les vecteurs de support sont entourés.

Applet interactive : à utiliser en ligne, sur la page « Classer » du site du cours.

1. Déplacez un point éloigné de la frontière. Rien ne change.
2. Déplacez un des points entourés. La frontière et la bande bougent aussitôt.

3. Ajoutez un point au milieu de la bande. La marge se rétrécit, et la frontière se réorganise autour de ce nouveau point.
4. Glissez un point rouge au milieu des bleus, puis réglez le curseur de tolérance, décrit ci-dessous.

La marge souple. Quand les groupes se chevauchent, aucune droite ne les sépare parfaitement, et aucune bande vide n'existe. La méthode accepte alors que certains points entrent dans la bande, ou même passent du mauvais côté, contre une pénalité. Un réglage décide de l'équilibre. Avec une faible tolérance, la méthode cherche à classer chaque point correctement, quitte à obtenir une marge étroite et une frontière qui suit de près les cas difficiles. Avec une forte tolérance, elle préfère une marge large et ignore les points isolés. C'est un **hyperparamètre**, et ce choix entre coller aux exemples et rester simple est un premier aperçu du compromis présenté dans « Généraliser ».

Un peu d'histoire. L'idée de la frontière de plus grande marge est proposée dès 1963 par le mathématicien soviétique Vladimir Vapnik, dans le cadre d'une théorie de l'apprentissage qu'il développe avec Alexeï Tchervonenkis. Émigré aux États-Unis, Vapnik la reprend aux Bell Labs. En 1992, avec Bernhard Boser et Isabelle Guyon, il lui permet de tracer des frontières courbes. En 1995, avec Corinna Cortes, il publie la marge souple (*soft margin*). C'est le laboratoire où Yann Le Cun développe au même moment ses réseaux convolutifs. Pendant une quinzaine d'années, les SVM sont l'une des méthodes les plus utilisées en apprentissage automatique, par exemple pour classer des textes ou analyser des données biologiques. Leur entraînement a un avantage précieux : il n'existe qu'une seule meilleure frontière, et on est sûr de la trouver.

Une frontière droite reste une frontière droite, quelle que soit sa marge. Le chapitre « Généraliser » montrera comment les SVM tracent aussi des frontières courbes, grâce à une idée appelée l'**astuce du noyau** (*kernel trick*).

Les mathématiques des SVM (optionnel)
Reprenons la notation de la régression logistique, avec cette fois des classes notées $y \in -1, +1$. La frontière est la droite $w_1 x_1 + w_2 x_2 + b = 0$. On impose que les points de chaque classe soient du bon côté avec une certaine avance :

$$y^{(i)} \left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{pour tous les points } i$$

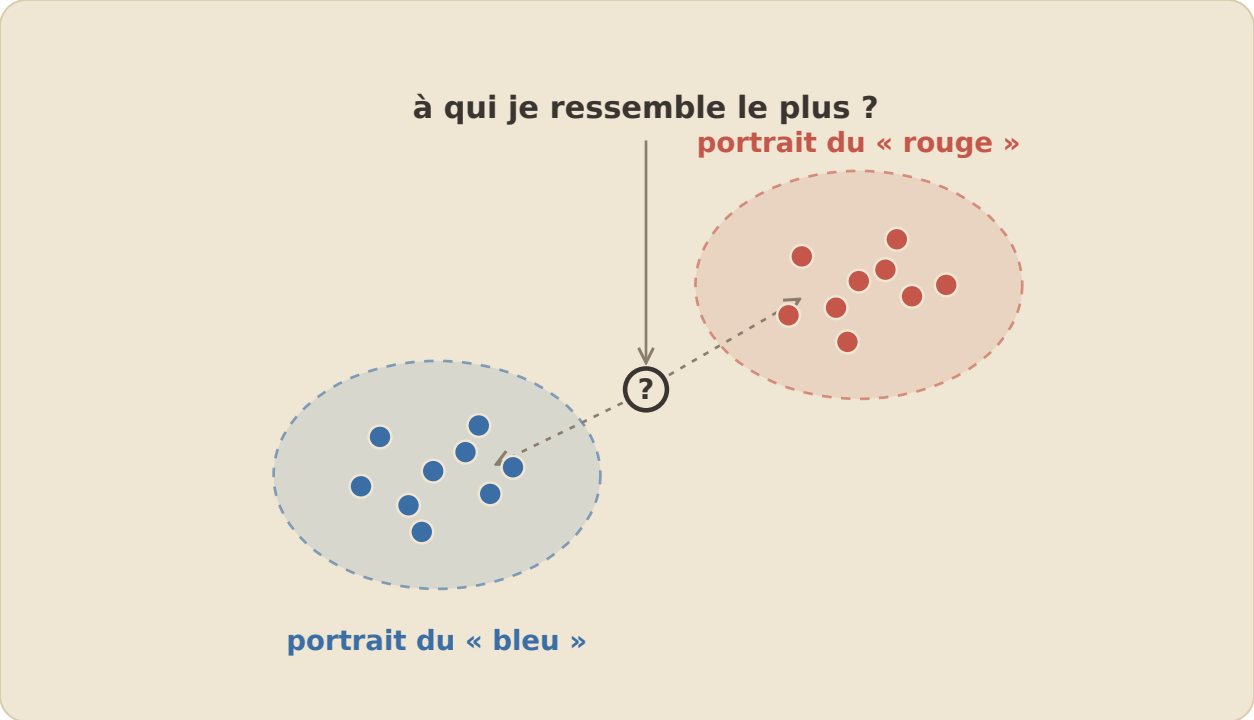
Les bords de la bande sont alors les droites $\mathbf{w} \cdot \mathbf{x} + b = 1$ et $\mathbf{w} \cdot \mathbf{x} + b = -1$. La largeur de la bande vaut $2/\|\mathbf{w}\|$. Maximiser la marge revient donc à minimiser $\|\mathbf{w}\|^2$ en respectant ces contraintes. Les vecteurs de support sont les points pour lesquels l'inégalité devient une égalité.
Pour la marge souple, on autorise chaque point à enfreindre la contrainte, et on pénalise l'écart. On minimise alors

$$\frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{i=1}^{n} \max\left(0,\ 1 - y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)\right)$$

Le second terme, appelé *hinge loss*, ne coûte rien pour un point bien classé hors de la bande, et augmente à mesure que le point s'enfonce dans la bande ou passe du mauvais côté. Le réglage C est l'inverse de la tolérance : un C grand pénalise fortement les erreurs. Ce problème n'a qu'un seul minimum, ce qui garantit qu'on trouve la meilleure frontière.

Renverser le problème : la classification bayésienne

La seconde approche aborde le problème dans l'autre sens. Plutôt que de tracer directement la frontière, elle commence par **décrire chaque classe**. On se demande à quoi ressemble un point bleu typique, puis un point rouge typique. Avec une bonne description de chacun, on peut classer un nouveau point en se demandant s'il ressemble davantage à un bleu ou à un rouge.



L'approche générative : on décrit le « portrait » de chaque classe, puis on demande auquel le nouveau point ressemble le plus.

C'est l'approche dite **générative**, et ce terme demande une explication. Décrire une classe assez précisément pour en reconnaître les membres permet aussi, en principe, d'en produire de nouveaux. Un modèle qui connaît le portrait du « bleu typique » pourrait créer des bleus plausibles qu'il n'a jamais vus. Il est dit *génératif* parce qu'il pourrait générer des données, et pas seulement les classer. Cette idée paraît modeste ici, mais c'est elle qui, développée à grande échelle, a donné l'IA *générative* actuelle. Un **grand modèle de langage** (un LLM, comme celui de ChatGPT) est essentiellement un portrait très détaillé de la classe « texte écrit par des humains », assez précis pour produire du texte nouveau, mot après mot. Les générateurs d'images font la même chose avec les photos. Le Module 4 leur est consacré.

Pour établir le portrait d'une classe, on décrit **comment ses points se répartissent** le long de chaque caractéristique. Par exemple, les points bleus se concentrent peut-être autour d'une certaine valeur de x_1, et les rouges autour d'une valeur plus élevée. Cette répartition se résume par une courbe connue, la **courbe en cloche** (ou *gaussienne*), qui présente un sommet là où les points sont nombreux et des bords qui s'abaissent là où ils sont rares. Le portrait d'une classe est donc formé de quelques-unes de ces cloches, une par caractéristique.

Pour garder le calcul simple, on fait une hypothèse volontairement approximative : on traite **chaque**

caractéristique séparément, comme si elles étaient indépendantes. C'est rarement tout à fait vrai (la superficie et le nombre de pièces, par exemple, varient ensemble), et c'est ce que signifie le mot **naïve** dans le nom de la méthode. Cette méthode naïve est néanmoins très efficace en pratique.

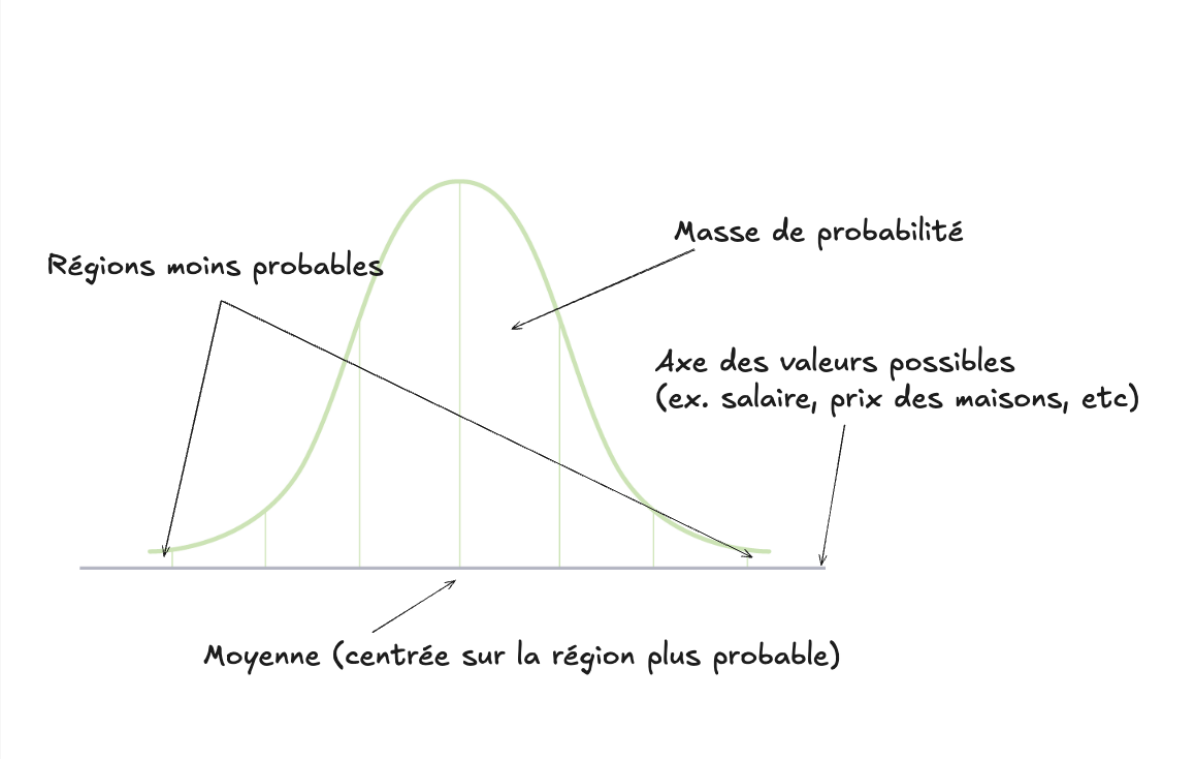
Il reste une dernière étape. Les portraits répondent à la question « si ce point est bleu, à quel point est-il typique ? », c'est-à-dire à la probabilité du point sachant la classe. Ce qu'on veut, c'est l'inverse : la probabilité que le point soit bleu, étant donné ce point. Passer de la *probabilité du point sachant la classe* à la *probabilité de la classe sachant le point* est précisément ce que permet un résultat fondamental des probabilités, le **théorème de Bayes**, publié en 1763. C'est lui qui donne son nom à la méthode, la **classification bayésienne naïve** (*naive Bayes*).

On dispose donc de deux approches pour le même objectif :

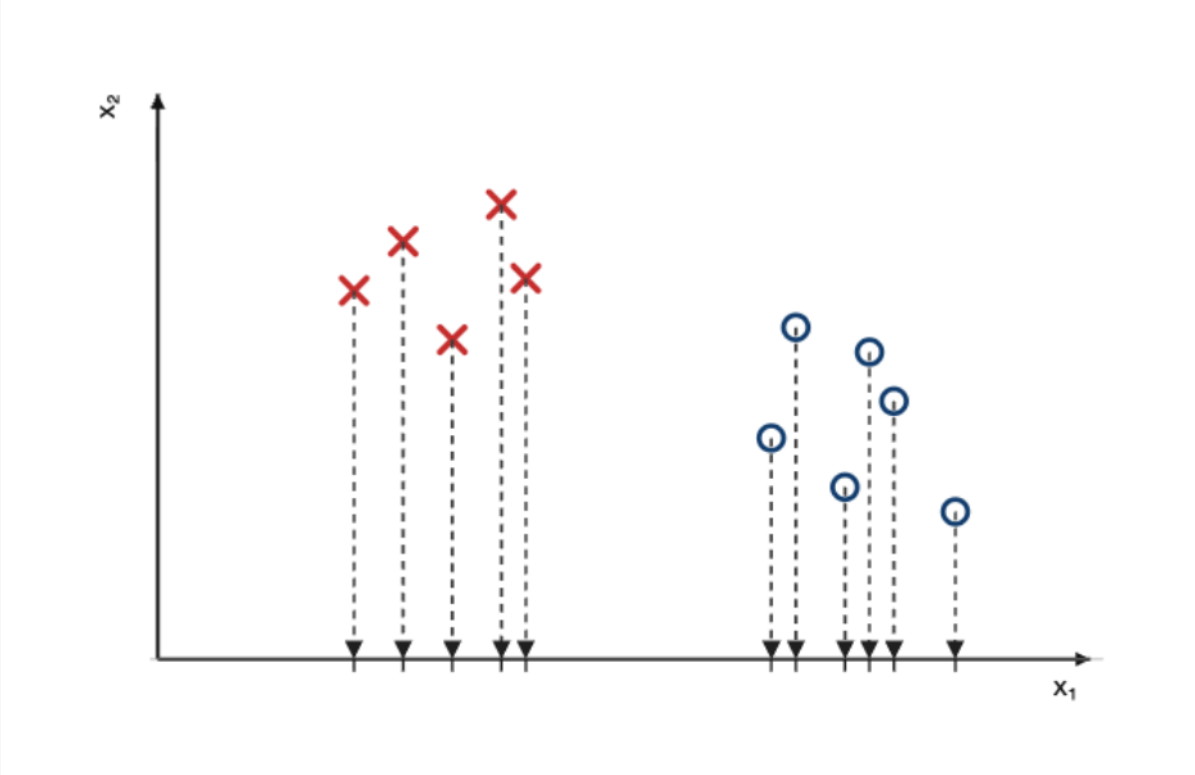
	Régression logistique	**Bayes naïf**
Philosophie	**discriminative**	**générative**
Stratégie	tracer la frontière	décrire chaque classe
Question posée	de quel côté ?	à quel portrait ressemble-t-il le plus ?
Bonus	—	pourrait générer de nouveaux exemples

Sur nos données en deux dimensions, ces deux approches très différentes aboutissent à la **même forme de frontière**, une droite. La distinction entre apprendre à séparer et apprendre à décrire reste cependant l'une des plus importantes du domaine. Nous la retrouverons au Module 4, qui distingue les modèles qui classent et ceux qui produisent du contenu.

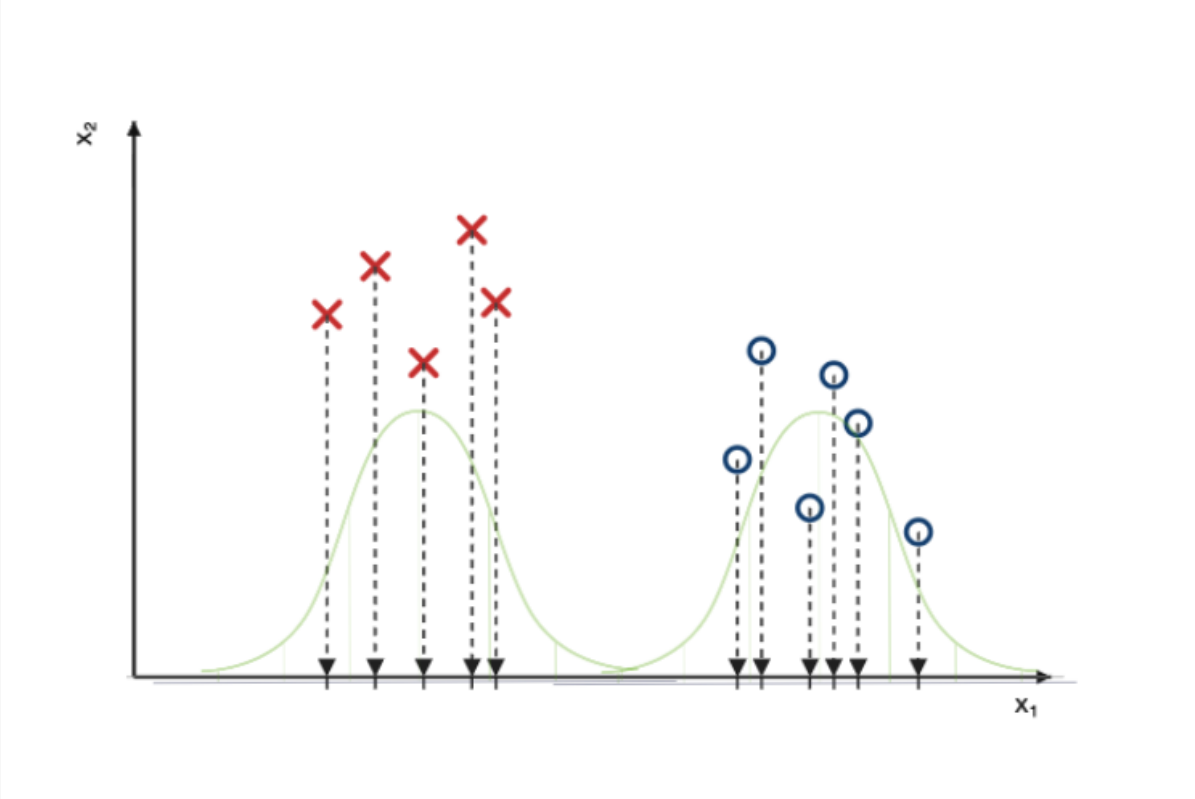
Les mathématiques de la classification bayésienne naïve (optionnel)
Chaque couple **caractéristique + classe** est modélisé par une gaussienne à une dimension. Avec nos deux caractéristiques et nos deux classes, cela fait quatre cloches. La gaussienne (ou loi normale) décrit comment la « masse de probabilité » se répartit autour d'une valeur centrale, la moyenne :



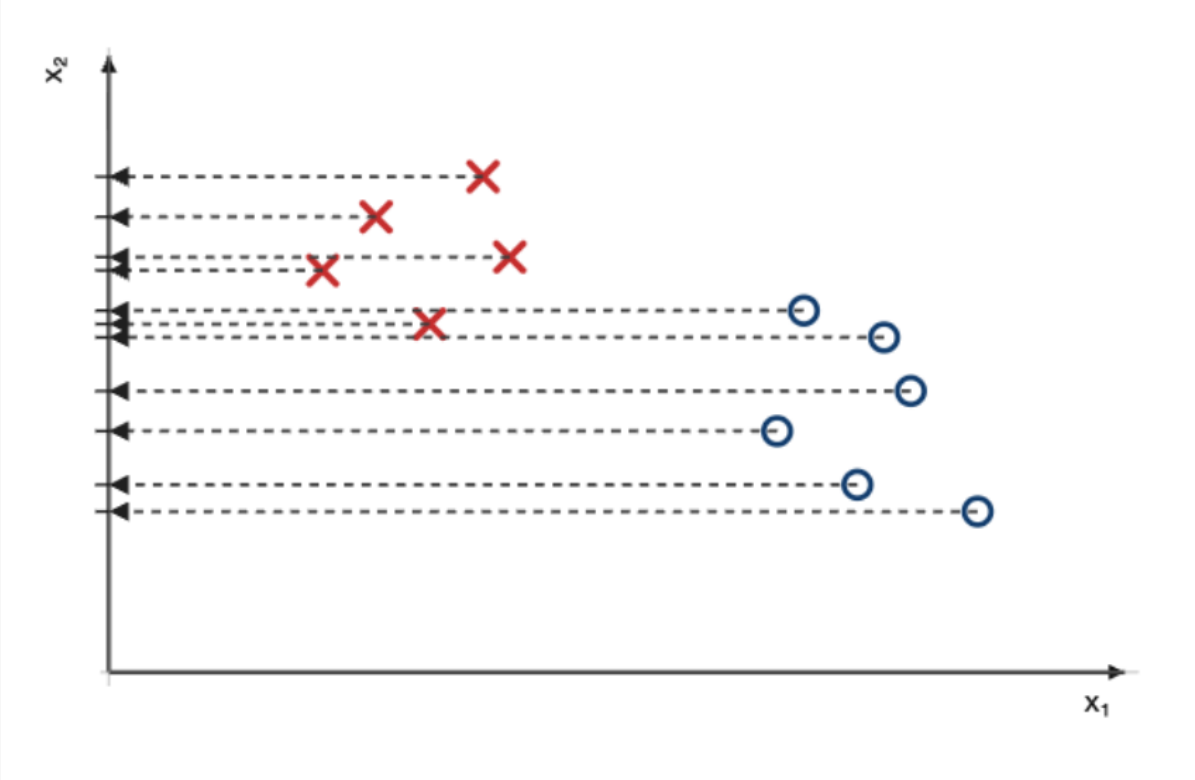
La hauteur de la courbe en un endroit n'est pas la probabilité de ce point. Comme la courbe est continue, une probabilité correspond à une **aire** sous la courbe (entre deux bornes). L'aire totale vaut 1, et l'aire à gauche de la moyenne vaut donc 0,5.
Concrètement, on projette d'abord les points sur l'axe x_1, ce qui les rend unidimensionnels...

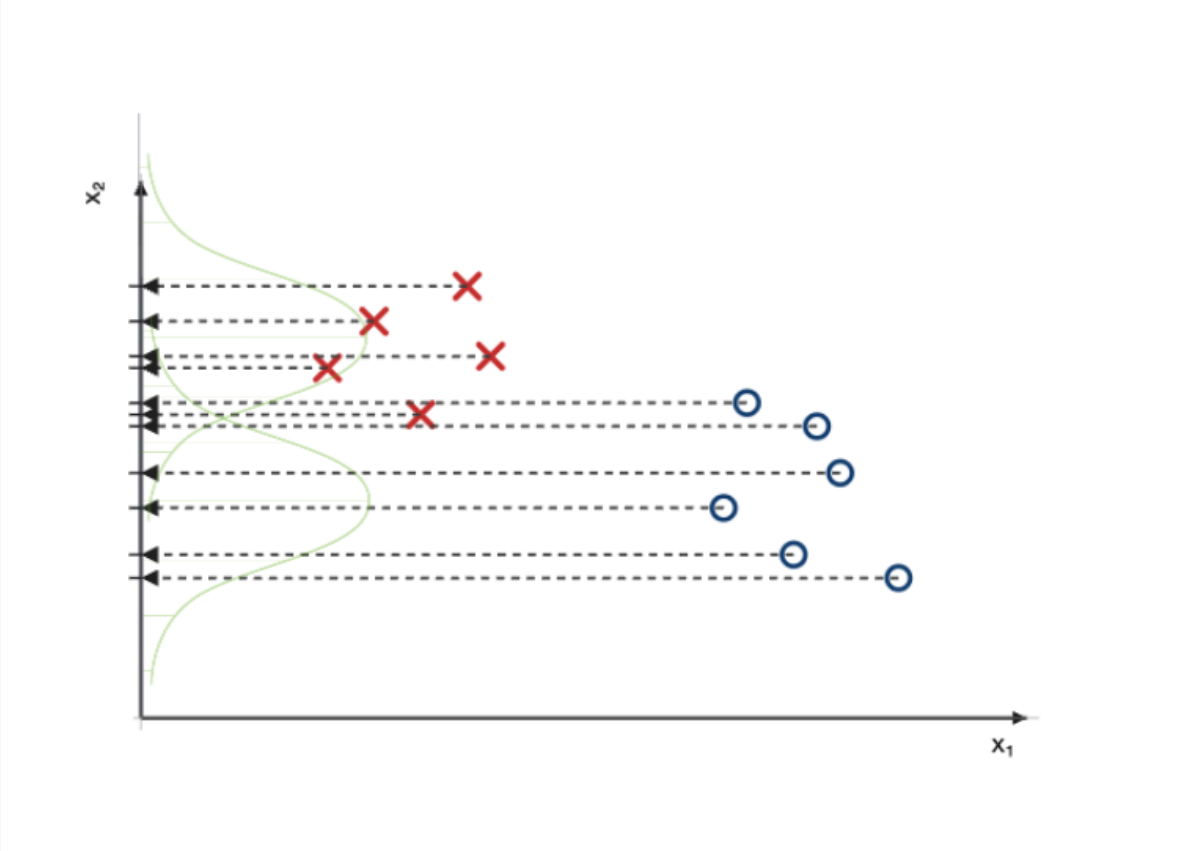


…puis on ajuste une cloche par classe, dont la largeur correspond à la dispersion des points projetés :



On fait ensuite la même chose sur l'axe x_2 :





On dispose alors de quatre modèles $p(x_j \mid \text{classe})$. La moyenne μ et l'écart-type (*standard deviation*) σ de chaque cloche s'obtiennent **directement** par un calcul de moyenne et de dispersion sur les points concernés. Aucune descente de gradient itérative n'est nécessaire ici :

$$\hat{\mu}_{j,c} = \frac{1}{N_c} \sum_{i \in c} x_{ij}, \qquad \hat{\sigma}^2_{j,c} = \frac{1}{N_c} \sum_{i \in c} (x_{ij} - \hat{\mu}_{j,c})^2$$

L'hypothèse naïve d'indépendance permet de combiner les caractéristiques par simple multiplication :

$$p(\mathbf{x} \mid c) = p(x_1 \mid c) \cdot p(x_2 \mid c)$$

Ce modèle est génératif : il décrit la probabilité d'un point $\mathbf{x}$ sachant sa classe, $P(\mathbf{x} \mid y)$. La classification demande l'inverse, $P(y \mid \mathbf{x})$. Le **théorème de Bayes** permet ce passage :

$$P(y \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y)\, P(y)}{P(\mathbf{x})}$$

Ici, $P(y)$ est la proportion de chaque classe (souvent 50/50 si les données sont équilibrées). Comme le dénominateur $P(\mathbf{x})$ ne dépend pas de la classe, on peut l'ignorer pour décider :

$$\text{classe}(\mathbf{x}) = \begin{cases} \texttt{rouge} & \text{si } P(\mathbf{x} \mid \text{rouge})\, P(\text{rouge}) \geq P(\mathbf{x} \mid \text{bleu})\, P(\text{bleu}) \\ \texttt{bleu} & \text{sinon} \end{cases}$$

On détermine donc lequel des deux portraits rend le point observé le plus **vraisemblable**, et c'est ce portrait qui donne la classe.

Le cas des pourriels

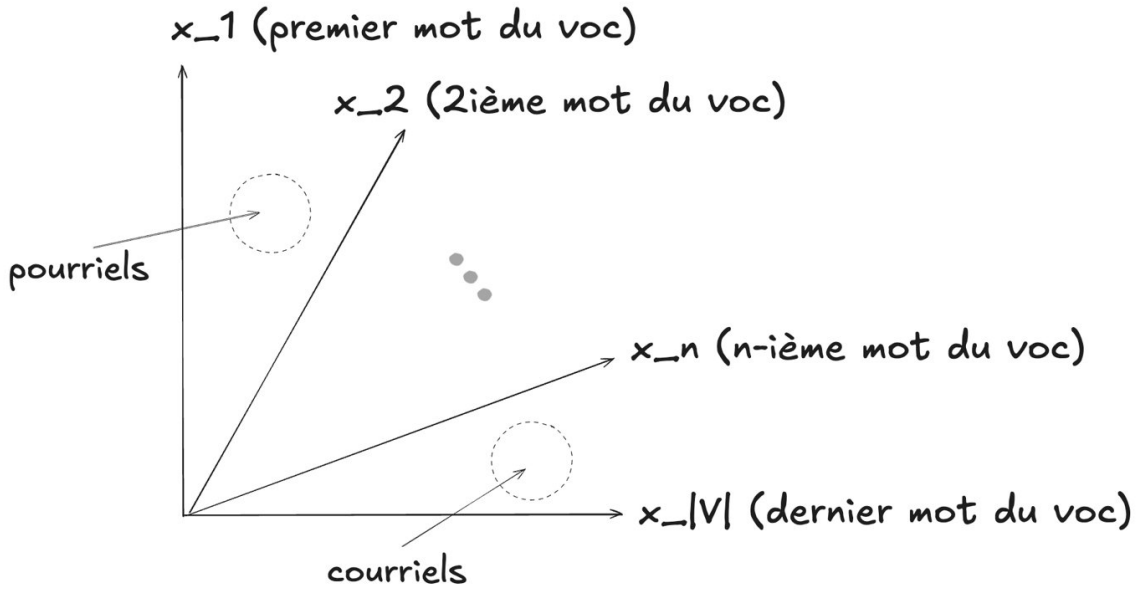
Passons maintenant des points colorés à un problème réel, celui du travail noté 2 : reconnaître automatiquement les **pourriels** (les courriels indésirables, le *spam*). C'est un exemple classique de classification. Il y a deux classes, `pourriel` ou `courriel` légitime, et une décision à prendre pour chaque message qui arrive.

Un premier obstacle se présente. Nos deux classificateurs attendent un **point**, c'est-à-dire une petite liste de nombres. Un courriel, lui, est un texte. Il faut donc trouver comment transformer un texte comme « *Félicitations ! Vous avez gagné un prix…* » en coordonnées.

La solution reprend la démarche de *Regarder les données* : une chose se décrit par une **liste de nombres** et devient ainsi un point dans un espace. Pour un texte, le procédé le plus simple s'appelle le **sac de mots** (*bag of words*). On établit la liste de tous les mots possibles (le *vocabulaire*), puis on décrit un courriel en comptant combien de fois chaque mot y apparaît. Chaque mot du vocabulaire correspond à une dimension, et la valeur est le nombre d'occurrences. Par exemple, le mot « gratuit » apparaît deux fois, « réunion » zéro fois, et ainsi de suite.

$$\mathbf{x} = (n_1, n_2, \dots, n_V)$$

La seule différence avec le vecteur d'une maison est l'échelle. Une maison se décrivait par quelques caractéristiques, alors que le vocabulaire compte des dizaines de milliers de mots. Un courriel est donc bien un point, mais dans un espace de très grande dimension. Comme nos points bleus et rouges, les pourriels et les courriels légitimes y forment deux nuages distincts :



Chaque mot du vocabulaire est un axe ; un courriel devient un point dans cet espace. Pourriels et courriels légitimes y forment deux nuages.

Nous savons déjà traiter deux nuages dans un espace, et la méthode de la section précédente s'applique telle quelle. On établit le **portrait** de chaque classe (quels mots trouve-t-on dans un pourriel typique, et dans un courriel légitime ?), puis on classe un nouveau message en déterminant lequel des deux portraits rend ses mots les plus vraisemblables. Il s'agit encore de l'approche **générative** de Bayes naïf. Ici, l'hypothèse naïve consiste à supposer que les mots sont tirés **indépendamment** les uns des autres. C'est faux (« carte » appelle souvent « bancaire »), mais c'est commode et étonnamment efficace.

Un seul détail technique change par rapport à la section précédente. Les caractéristiques y étaient des valeurs continues, décrites par une courbe en cloche. Ici, ce sont des comptes, des nombres entiers : zéro, une ou deux occurrences, par exemple. On remplace donc la cloche par une loi adaptée aux comptes, la **loi multinomiale** (*multinomial distribution*), mais le principe est le même. Le portrait d'une classe n'est plus une moyenne et une dispersion, mais la liste des mots qu'elle emploie souvent. Des mots comme « gratuit », « urgent » ou « félicitations » sont fréquents dans les pourriels et beaucoup moins dans les courriels ordinaires.

C'est ce classificateur (Bayes naïf multinomial sur un sac de mots) que vous mettrez en œuvre au **travail noté 2** pour construire votre propre filtre anti-pourriel.

Les mathématiques du Bayes naïf multinomial (optionnel)
Un courriel est le vecteur de comptes $\mathbf{x} = (n_1, \dots, n_V)$, où n_i est le nombre d'occurrences du mot i et V la taille du vocabulaire. Si ce courriel appartient à la classe `pourriel`, la probabilité d'observer ce vecteur selon la loi multinomiale est :

$$P(\mathbf{x} \mid \text{pourriel}) = \frac{N!}{n_1!\, n_2! \cdots n_V!} \prod_{i=1}^{V} p_i^{n_i}$$

Ici, $N = \sum_i n_i$ est le nombre total de mots du courriel, et p_i est la probabilité, dans un pourriel, que le mot tiré soit le mot i (par exemple plus élevée pour « prix » que pour « parent »). On définit de la même façon un modèle pour la classe `courriel`. Ces p_i s'estiment directement, en comptant la fréquence de chaque mot dans les courriels d'entraînement de la classe, comme on estimait la moyenne d'une gaussienne.
La décision se prend ensuite avec le théorème de Bayes, comme à la section précédente. En ignorant le dénominateur $P(\mathbf{x})$, qui est le même pour les deux classes, on obtient :

$$\text{classe}(\mathbf{x}) = \begin{cases} \texttt{pourriel} & \text{si } P(\mathbf{x} \mid \text{pourriel})\, P(\text{pourriel}) \geq P(\mathbf{x} \mid \text{courriel})\, P(\text{courriel}) \\ \texttt{courriel} & \text{sinon} \end{cases}$$

Ce calcul correspond à une décision **linéaire** dans l'espace des mots, soit le même genre de frontière que celle de la régression logistique. Les deux familles, discriminative et générative, aboutissent donc de nouveau au même résultat.

Raisonner sous incertitude : les réseaux bayésiens

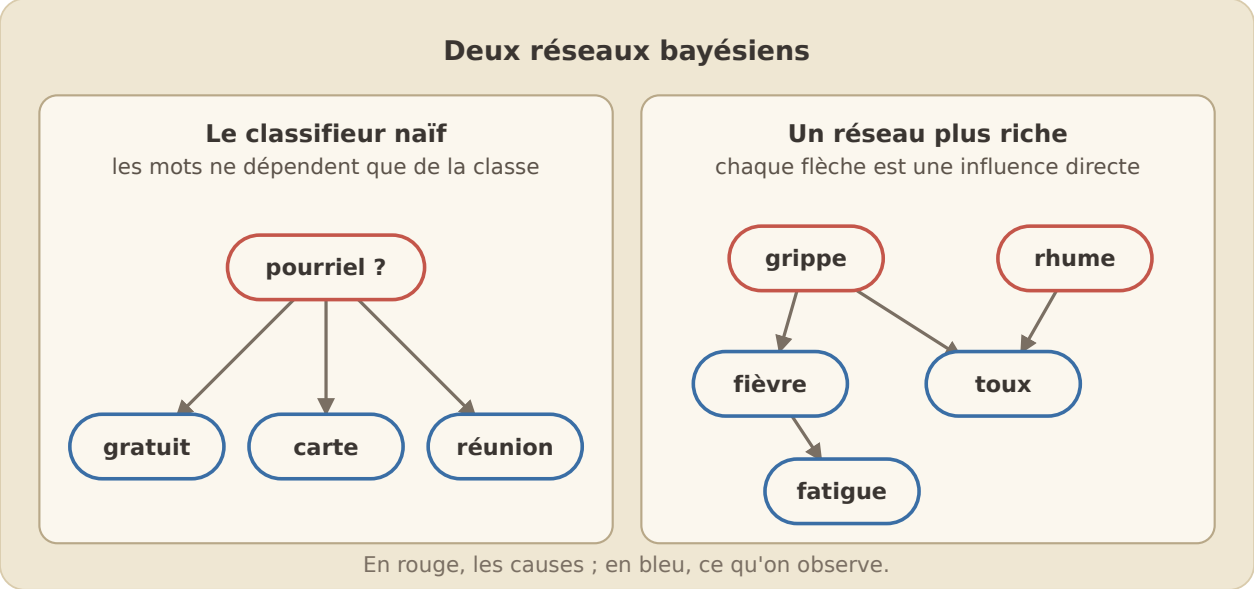
Le classifieur de la section précédente appartient à un courant plus large, qui a transformé l'IA à la fin des années 1980. Les systèmes experts du Module 1 raisonnaient avec des règles. Pour tenir compte de l'incertitude, MYCIN leur ajoutait des facteurs de certitude, une méthode commode mais

approximative. Au cours des années 1980, une autre idée s'impose : raisonner directement avec des **probabilités**, selon les règles du calcul des probabilités et le théorème de Bayes. On a appelé ce tournant la **révolution probabiliste**.

Il touche plusieurs domaines en même temps.

- **La reconnaissance de la parole.** Chez IBM, l'équipe de Frederick Jelinek remplace les règles de linguistes par des modèles statistiques appris sur des enregistrements. On lui attribue une phrase restée célèbre : chaque fois qu'il renvoyait un linguiste, son système s'améliorait.
- **La traduction automatique.** Vers 1990, une autre équipe d'IBM traduit par des probabilités apprises sur des textes déjà traduits, au lieu d'écrire des règles de grammaire.
- **Les filtres anti-pourriel.** En 2002, l'essai de Paul Graham, « A Plan for Spam », popularise le classifieur bayésien de la section précédente, que la plupart des messageries adoptent ensuite.
- **Le raisonnement en général**, avec les **réseaux bayésiens** (*Bayesian networks*) de Judea Pearl, présentés ci-dessous.

Du classifieur naïf au réseau. On peut dessiner le classifieur bayésien naïf comme un graphe. Un nœud représente la classe, « pourriel ou non », et une flèche part de ce nœud vers chaque mot, parce que la classe influence les mots employés. Il n'y a aucune flèche entre les mots : c'est l'hypothèse naïve.

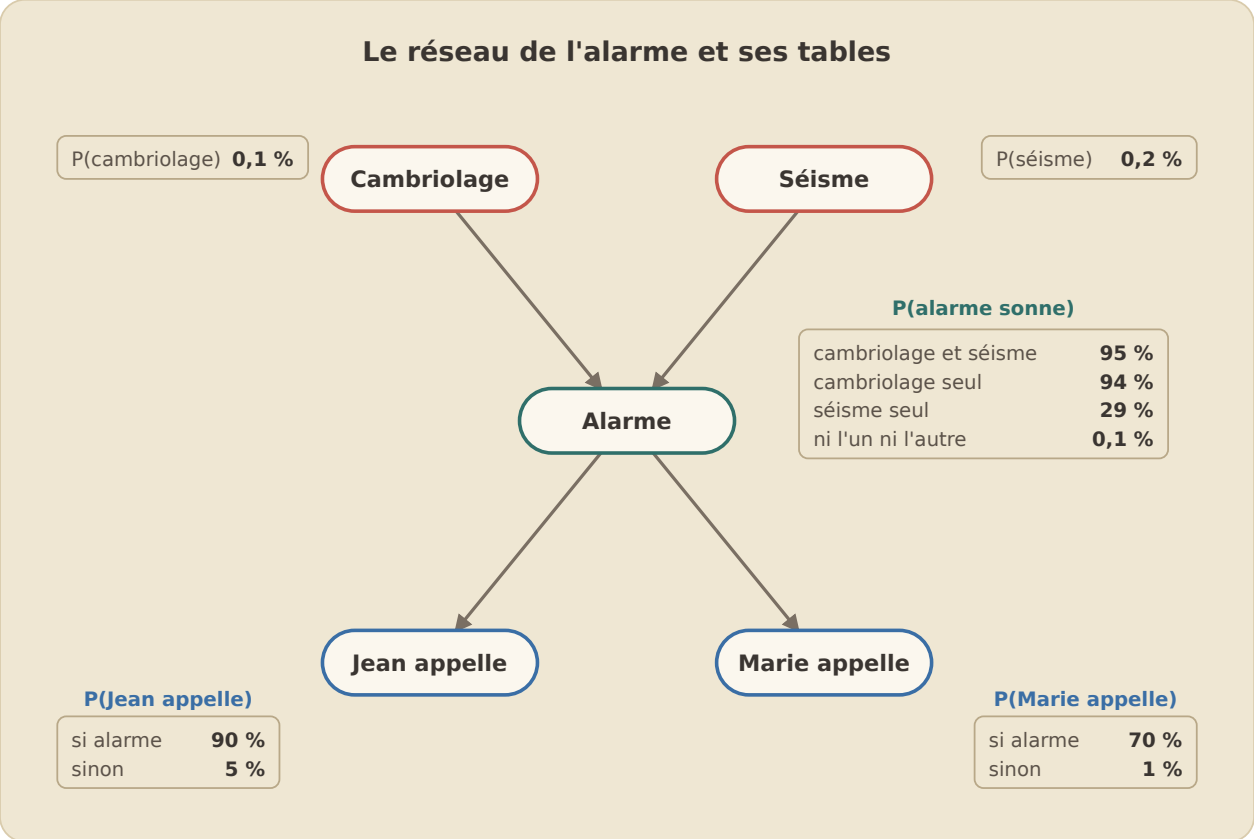


Le classifieur naïf est le plus simple des réseaux bayésiens ; un réseau plus riche dessine les dépendances réelles entre les variables.

Un **réseau bayésien**, proposé par Judea Pearl en 1985, généralise ce dessin. Chaque nœud est une variable incertaine. Chaque flèche indique une influence directe, en général d'une cause vers un effet. Chaque nœud porte une petite table, qui donne sa probabilité selon les valeurs de ses parents. Ces tables suffisent à calculer la probabilité de n'importe quel nœud, à partir de n'importe quelle observation.

Un exemple : l'alarme. Une maison a une alarme contre les cambriolages. Elle se déclenche aussi, parfois, lors d'un tremblement de terre. Si elle sonne, vos voisins Jean et Marie vous appellent peut-être. Jean appelle presque toujours, mais il appelle aussi parfois quand il confond l'alarme avec

la sonnerie de son téléphone. Marie entend moins bien, mais se trompe rarement.



Le réseau de l'alarme : cinq variables, et une petite table de probabilités par variable.

Dans l'applet ci-dessous, cliquez sur un nœud pour indiquer ce que vous savez : vrai, faux ou inconnu. Les probabilités des autres nœuds sont recalculées aussitôt.

Applet interactive : à utiliser en ligne, sur la page « Classer » du site du cours.

1. Sans aucune observation, la probabilité d'un cambriolage est de 0,1 %. Indiquez que l'alarme sonne. Elle passe à environ 37 %.
2. Indiquez ensuite qu'il y a eu un tremblement de terre. La probabilité du cambriolage retombe à 0,3 %. Le séisme explique l'alarme, et le cambriolage n'est plus nécessaire pour l'expliquer. Ce raisonnement s'appelle l'**explication concurrente** (*explaining away*). Pourtant, dans le réseau, cambriolage et séisme sont indépendants l'un de l'autre.
3. Effacez ces observations, puis indiquez seulement que Jean appelle : le cambriolage passe à 1,6 %. Avec Marie seule : 5,6 %. Avec les deux : 28 %. Deux indices faibles et indépendants, réunis, deviennent un indice fort.

Les flèches vont des causes vers les effets, mais le calcul va dans les deux sens. On part des effets observés, les appels, pour remonter vers les causes probables, le cambriolage ou le séisme. C'est le raisonnement d'un diagnostic, celui que MYCIN cherchait à mener avec ses règles. Le réseau le fait selon les règles exactes du calcul des probabilités.

Les réseaux bayésiens sont utilisés en diagnostic médical, en détection de pannes, en analyse de risques ou dans des filtres comme celui des pourriels. Pearl a ensuite cherché à distinguer, dans ces

graphes, les influences causales des simples corrélations. Ces travaux sur la **causalité** lui ont valu le prix Turing en 2011. La distinction entre corrélation et causalité sera reprise au Module 5.

Les mathématiques des réseaux bayésiens (optionnel)
Un réseau bayésien décrit la probabilité conjointe (*joint probability*) de toutes ses variables comme un produit de petites tables. Pour l'alarme, avec C (cambriolage), S (séisme), A (alarme), J et M (les appels) :

$$P(C, S, A, J, M) = P(C)\, P(S)\, P(A \mid C, S)\, P(J \mid A)\, P(M \mid A)$$

Avec cinq variables vrai ou faux, la table complète compterait 32 cases. Le réseau n'a besoin que de 10 nombres. Pour obtenir une probabilité comme $P(C \mid A)$, on additionne les cases compatibles avec l'observation, puis on divise :

$$P(C \mid A) = \frac{P(C, A)}{P(A)}$$

Ce calcul par énumération est exact, mais son coût double avec chaque variable ajoutée. Pour les grands réseaux, on utilise des algorithmes qui exploitent la forme du graphe, ou des méthodes d'approximation.

Sous les modèles, des probabilités

Avant de quitter la classification, il faut revenir sur une notion présente dans toute cette page, celle de **probabilité**. La régression logistique ne répond pas « bleu » ou « rouge », mais « bleu à 80 % ». Le classificateur bayésien compare lui aussi deux probabilités. Ce n'est pas un simple détail d'implantation. Un modèle qui donne une probabilité exprime une **croyance** avec son degré de certitude, et cette information est souvent plus utile que la seule décision. Un filtre qui rejette un courriel à 51 % n'indique pas la même chose qu'un filtre qui le rejette à 99,9 %. De plus, le seuil que nous déplacerons dans *Bien évaluer un modèle* n'existe que parce que le modèle exprime une incertitude. Il faut cependant que ces probabilités soient fiables : parmi les courriels qu'un bon modèle rejette « à 90 % », neuf sur dix doivent réellement être des pourriels. Cette propriété s'appelle la **calibration**, et on peut la vérifier.

Cette notion va plus loin. Rappelez-vous la fonction d'erreur de la droite, la moyenne des carrés des écarts, et celle de la régression logistique, qui pénalise d'autant plus le modèle qu'il se trompe avec assurance. Ces deux fonctions semblent sans rapport, mais elles découlent d'un même principe, l'un des plus importants de la statistique : le **maximum de vraisemblance** (*maximum likelihood*). Au lieu de chercher les paramètres qui réduisent l'erreur, on cherche les paramètres sous lesquels les données observées étaient les plus probables. Si l'on suppose que les prix des maisons s'écartent de la droite selon une courbe en cloche, chercher la droite qui rend les vingt prix les plus probables revient exactement à minimiser les carrés des écarts. Si l'on suppose que chaque étiquette est tirée avec la probabilité annoncée par le modèle, chercher les paramètres qui rendent les étiquettes observées les plus probables donne exactement la fonction d'erreur de la régression logistique. Minimiser une erreur et maximiser une vraisemblance sont donc deux descriptions de la même opération.

Nous avons employé le théorème de Bayes comme une formule, mais il décrit aussi une manière d'**apprendre**. On part d'une croyance *a priori*. Par exemple, avant de lire un courriel, on estime qu'il a une chance sur dix d'être un pourriel, parce que c'est la proportion habituelle. On observe

ensuite des indices, ici les mots du courriel, et le théorème indique comment **réviser** la croyance. La probabilité *a posteriori* est la croyance de départ, corrigée selon ce que les indices rendent plus ou moins probable. Chaque mot lu modifie un peu l'estimation : « gratuit » la rapproche du pourriel, « réunion » du courriel légitime. La probabilité a posteriori d'aujourd'hui devient ensuite l'a priori de demain. Dans cette perspective, apprendre consiste à mettre à jour ses croyances à mesure que les indices arrivent, et le théorème de Bayes est la règle exacte de cette mise à jour. L'approche bayésienne est présente dans tout le domaine, y compris dans les méthodes qui cherchent aujourd'hui à faire indiquer aux grands modèles leur degré de certitude.

Tous les modèles de cette page apprennent leurs paramètres à partir des exemples, en optimisant un critère : une fonction d'erreur, une marge ou une vraisemblance. Avant d'examiner la qualité de leurs prédictions sur des données nouvelles, la page suivante, « Poser des questions : les arbres de décision », présente un dernier modèle, d'un autre type, qui ne calcule pas mais pose des questions.

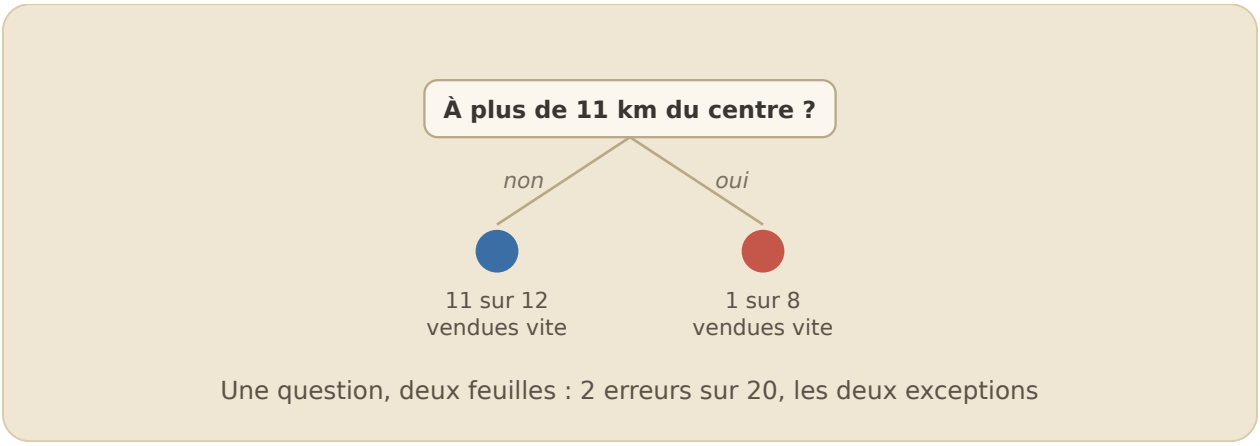
Poser des questions : les arbres de décision

Tous les modèles vus jusqu'ici raisonnent avec des nombres. kNN mesure des distances, la droite multiplie et additionne, la régression logistique et Bayes calculent des probabilités. Ces méthodes sont puissantes, mais ce n'est pas ainsi que nous prenons la plupart de nos décisions. Un médecin devant un patient ne fait pas de calcul : il pose des questions l'une après l'autre, et chaque réponse oriente la suivante. Le jeu des vingt questions fonctionne de la même façon. Un joueur pense à un objet, et l'autre doit le deviner en posant au plus vingt questions auxquelles on ne répond que par oui ou par non, comme « est-ce un animal ? » ou « est-il plus gros qu'un chat ? ». Chaque réponse réduit l'ensemble des possibilités. Le guide de dépannage à la fin d'un manuel suit le même principe : « l'appareil s'allume-t-il ? si non, est-il branché ? ». Il s'agit chaque fois d'une suite de questions à réponse oui ou non, qui se ramifie et aboutit à une conclusion.

Cette façon de décider correspond à l'un des modèles les plus anciens et les plus intuitifs de l'apprentissage automatique, l'**arbre de décision** (*decision tree*). Comme kNN, il traite aussi bien la régression que la classification, et son principe peut se dessiner. Il apporte cependant deux choses qu'aucun modèle du module n'offrait encore : une frontière qui n'est pas une droite, et une décision qu'on peut lire.

Une question suffit presque

Reprenons la seconde question du module dans le plan où elle se représente : la distance du centre en abscisse, l'année de construction en ordonnée, et la couleur qui indique si la maison s'est vendue vite. Laissons un arbre apprendre sur ces vingt maisons, avec une seule question permise. Voici le résultat :



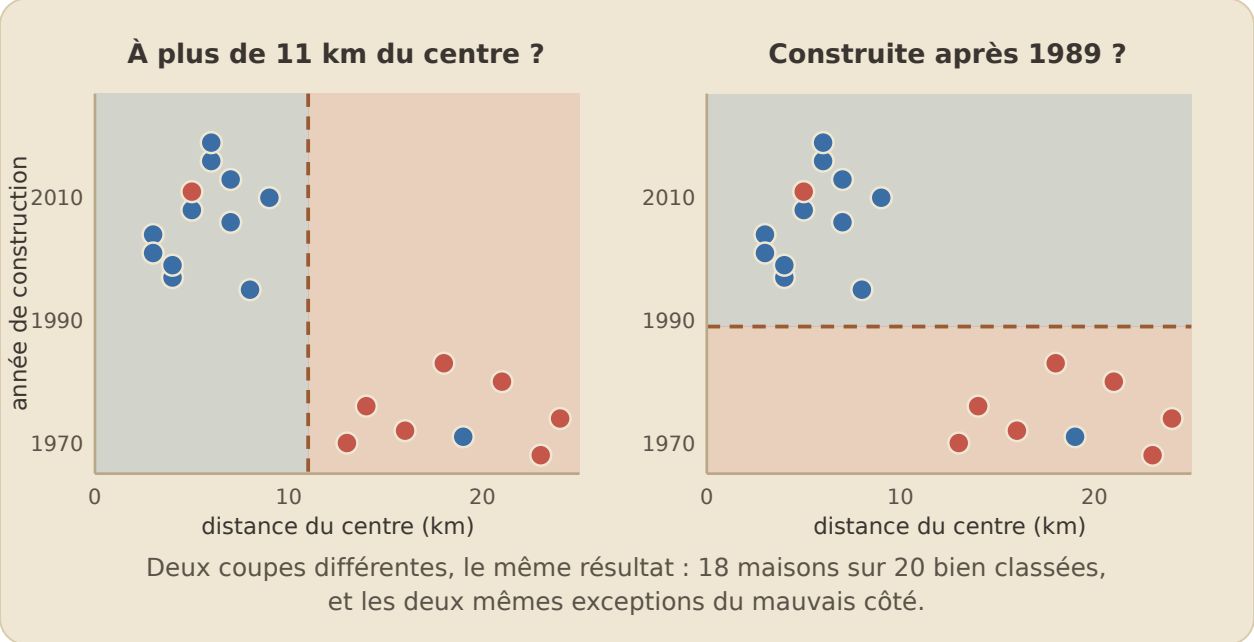
Un arbre à une question : une racine, deux branches, deux feuilles. Dix-huit maisons sur vingt bien classées.

La question est « à plus de 11 km du centre ? ». Si la réponse est non, la maison va dans la feuille de gauche, où 11 maisons sur 12 se sont vendues vite, et l'arbre répond *oui, vendue vite*. Si la réponse est oui, elle va dans la feuille de droite, où 1 maison sur 8 seulement s'est vendue vite, et l'arbre répond *non*. Avec une question et deux feuilles, 18 maisons sur 20 sont correctement classées. Les deux maisons restantes sont les deux exceptions déjà repérées dans le nuage coloré, chacune du mauvais côté de la question.

Le vocabulaire est celui d'un arbre dessiné à l'envers. La première question est la **racine**, chaque réponse est une **branche**, et les cases du bas, où l'on ne pose plus de question mais où l'on répond, sont les **feuilles**. Pour prédire, on part de la racine avec une nouvelle maison, on répond aux questions et on descend jusqu'à une feuille. La couleur majoritaire de cette feuille est la prédiction.

Dans le plan, une question sur la distance correspond à une **coupe verticale** à 11 km : tout ce qui est à gauche est bleu, tout ce qui est à droite est rouge. C'est le panneau de gauche de la figure ci-dessous, avec le même fond teinté que pour kNN. Comparez cette frontière à celle de kNN sur les mêmes maisons. La frontière de kNN était une ligne sinueuse entre les amas. Celle de l'arbre est un trait droit et vertical, parce qu'une question ne porte que sur une seule caractéristique à la fois.

On peut remarquer que l'arbre n'a rien demandé sur l'année de construction, alors que le nuage coloré semblait indiquer que les maisons anciennes se vendent lentement. Une question sur l'année aurait donné un résultat tout aussi bon, comme le montre le panneau de droite. La question « construite après 1989 ? » correspond à une **coupe horizontale**, et elle place les mêmes 11 maisons sur 12 d'un côté et 1 sur 8 de l'autre, avec les mêmes deux exceptions.



Deux questions à égalité : la coupe sur la distance et la coupe sur l'année classent les mêmes dix-huit maisons, et ratent les deux mêmes.

L'arbre disposait donc de deux questions équivalentes. Il n'en a pris qu'une, parce qu'une fois la première posée, la seconde n'apporte plus rien, sauf pour corriger les deux exceptions. De plus, l'arbre ne conserve pas les vingt maisons : une fois la question trouvée, il n'en a plus besoin. Il fait donc partie des modèles qui résument les données, comme la droite, et ses seuls paramètres sont un

seuil et deux réponses. La section suivante explique comment il a choisi entre les deux questions et, plus généralement, comment il choisit ses questions.

Comment l'arbre choisit ses questions

Le choix de la question repose sur un simple **comptage**. Pour chaque caractéristique et chaque seuil possible, l'arbre fait un essai : il sépare les vingt maisons en deux groupes et mesure, de chaque côté, à quel point les couleurs sont **mélangées**. Une feuille où toutes les maisons sont de la même couleur est dite *pure*. Une feuille où elles sont moitié-moitié est le pire cas, puisqu'elle ne donne aucune information. La meilleure question est celle dont les deux côtés sont, ensemble, les plus purs possible.

Le nombre de seuils candidats est limité. Entre deux maisons voisines sur une caractéristique, toutes les coupes donnent le même résultat, et il suffit d'essayer celle du milieu. Avec vingt maisons et deux caractéristiques, cela fait une quarantaine de questions à essayer. Un ordinateur les évalue toutes très rapidement et garde la meilleure. Sur nos maisons, la question « à plus de 11 km du centre ? » donne d'un côté 11 bleues sur 12 et de l'autre 7 rouges sur 8, soit deux feuilles presque pures. La question « construite après 1995 ? » aurait laissé 2 rouges parmi 11 d'un côté et 2 bleues parmi 9 de l'autre. Le mélange étant plus important, cette question est écartée.

Cela explique l'observation précédente. Les questions « construite après 1989 ? » et « à plus de 11 km du centre ? » obtenaient exactement le même score, 11 sur 12 et 1 sur 8. En cas d'égalité parfaite, l'arbre prend la première question trouvée. Il faut donc retenir que, lorsque deux caractéristiques apportent la même information, l'arbre en choisit une et ignore l'autre, sans que cela signifie que l'autre soit sans importance. Un arbre ne donne pas une explication du monde, mais un chemin de décision qui fonctionne.

Une fois la première question posée, on recommence séparément dans chacune des deux feuilles. On cherche la meilleure question pour les maisons proches du centre, puis la meilleure pour les maisons éloignées, et ainsi de suite, jusqu'à ce que les feuilles soient pures ou qu'on décide d'arrêter. Chaque question est choisie sans tenir compte des suivantes. Les informaticiens appellent cela une stratégie *gloutonne* (*greedy*, en anglais, le terme le plus souvent employé) : on choisit la meilleure étape immédiate, sans regarder plus loin. Cette stratégie ne garantit pas le meilleur arbre possible, mais elle est rapide et donne de très bons résultats en pratique.

Cet apprentissage diffère de ceux vus précédemment. Le modèle le plus bête calculait une moyenne. La droite et les modèles apparentés descendaient progressivement la pente d'une fonction d'erreur. L'arbre, lui, **cherche** : il énumère des questions, les essaie et garde la meilleure. Il n'y a ni paramètres ajustés de façon continue, ni gradient, mais une exploration parmi des choix discrets, semblable à la recherche dans un arbre de coups du Module 1, appliquée ici à l'apprentissage. C'est la troisième façon d'apprendre présentée dans ce module, et elle a la même structure que les deux autres : des données, une mesure de ce qui est bon (la pureté) et une procédure qui maximise cette mesure.

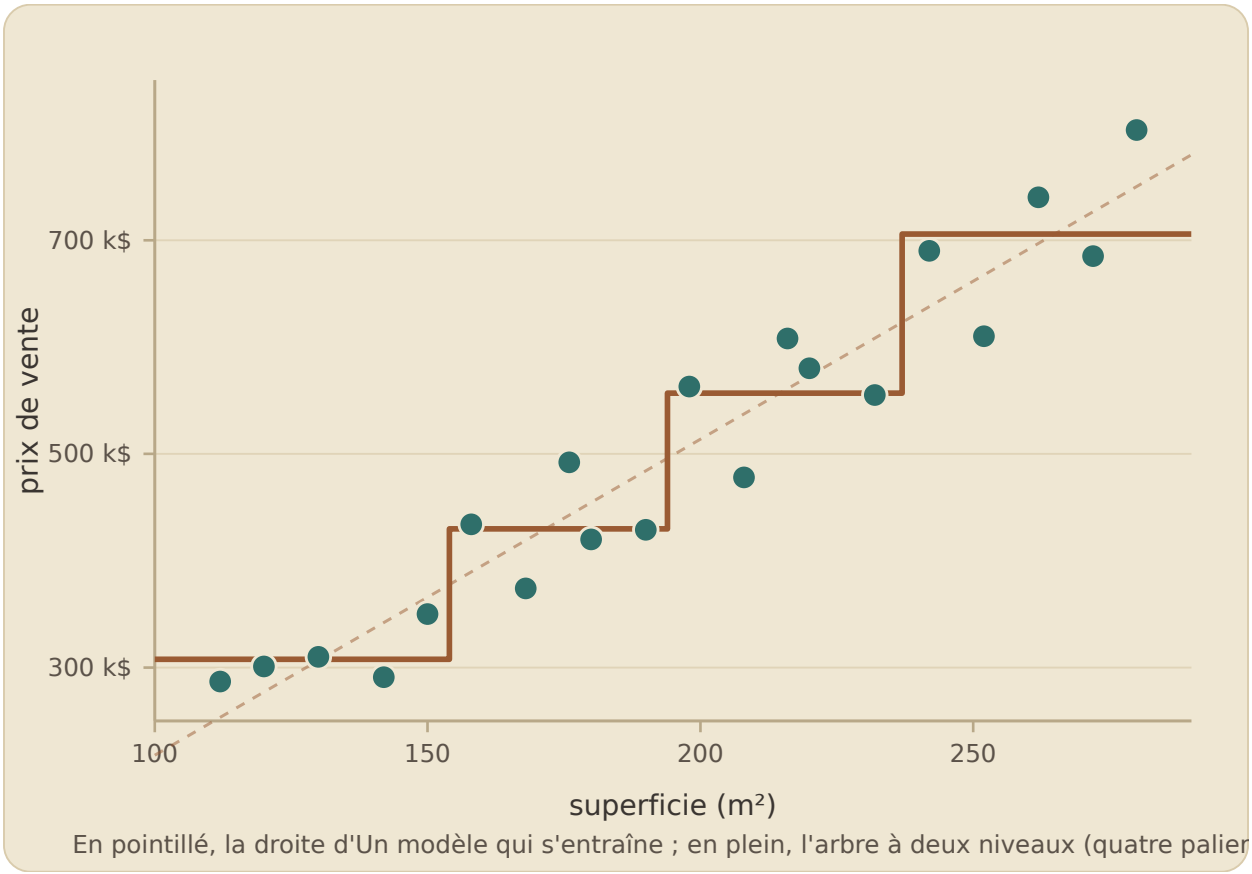
Pour aller plus loin : mesurer le mélange. La mesure la plus courante est l'*indice de Gini* (*Gini impurity*). Dans une feuille où une proportion p des maisons est bleue, il vaut $2p(1-p)$, soit 0 pour une feuille pure et 0,5 pour une feuille moitié-moitié. Le score d'une question est la moyenne des indices de ses deux côtés, pondérée par leur taille. Sur nos maisons, l'indice de départ vaut 0,48, et il descend à 0,18 après la question sur la distance. Une autre mesure,

l'*entropie*, issue de la théorie de l'information, donne presque toujours le même arbre.

Un arbre qui prédit un nombre

Rien de ce qui précède ne dépend du fait que la réponse soit une catégorie. Reprenons la première question du module, le prix, avec une seule caractéristique, la superficie. Un arbre peut la traiter de la même façon. Il pose des questions sur la superficie et, dans chaque feuille, il donne le **prix moyen** des maisons qui s'y trouvent au lieu d'une couleur majoritaire. Seule la mesure du mélange change. On regarde à quel point les prix d'une feuille sont dispersés autour de leur moyenne, et la meilleure question est celle qui réduit le plus cette dispersion de chaque côté.

Voici l'arbre à deux niveaux appris sur nos vingt maisons. La première question est « plus de 194 m² ? ». Pour les petites maisons, la question suivante est « plus de 154 m² ? » et, pour les grandes, « plus de 237 m² ? ». On obtient quatre feuilles et quatre prix : 308 000, 430 000, 557 000 et 706 000 $.



L'arbre de régression, dessiné sur le nuage : un escalier à quatre paliers, un par feuille. En pointillé, la droite, pour comparer.

Dessiné sur le nuage, l'arbre forme un **escalier** : quatre paliers horizontaux, un par feuille, avec une marche à chaque seuil. Cette courbe ne monte pas de façon continue, elle avance par sauts. Par exemple, entre 154 et 194 m², toutes les maisons valent 430 000 $, qu'elles fassent 155 ou 193 m². Pourtant, avec ses quatre paliers, l'escalier suit déjà mieux le nuage que la droite d'*Un modèle qui*

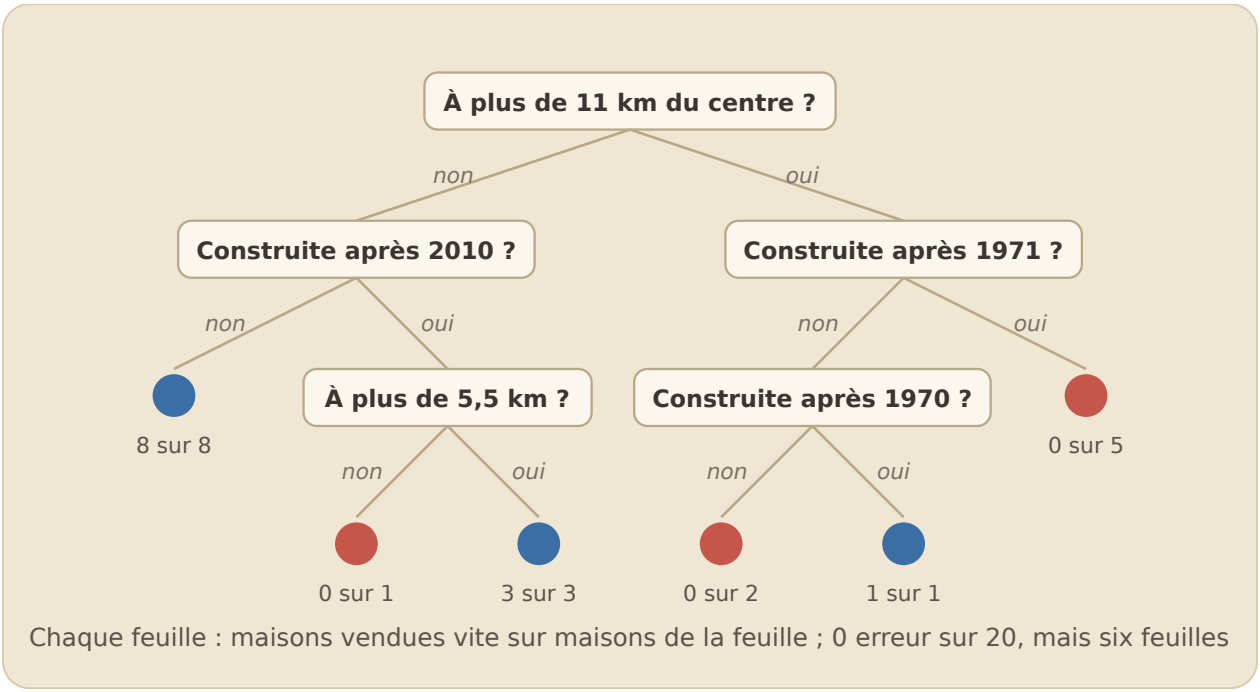
s'entraîne : son erreur moyenne est de 32 000 $, contre 41 000 pour la droite. Avec un niveau de plus, soit huit paliers, l'erreur descend à 23 000 $.

Cette comparaison met en évidence les différences entre les deux modèles. La droite suppose une forme précise, la ligne droite, et ne peut rien représenter d'autre. Si les prix suivaient une courbe, elle ne pourrait pas la suivre. L'escalier ne suppose aucune forme : avec assez de marches, il peut s'adapter à n'importe quelle forme. C'est sa force. Cependant, chaque palier est calculé sur un petit nombre de maisons, cinq ici, et c'est aussi sa faiblesse. La droite résume ses vingt maisons en deux nombres, alors que l'escalier les découpe en petits groupes traités séparément. De plus, l'escalier ne peut pas extrapoler : au-delà de 280 m², il répond toujours 706 000 $, alors que la droite continue de monter. Nous verrons dans *Bien évaluer un modèle* qu'on ne peut pas dire lequel des deux a raison hors du nuage, car aucun des deux n'offre de garantie dans cette zone.

Un arbre de régression utilise donc les mêmes questions et les mêmes feuilles qu'un arbre de classification, avec une moyenne à la place d'un vote. C'est la même distinction que celle que kNN avait présentée à sa dernière étape.

Jusqu'où laisser pousser l'arbre ?

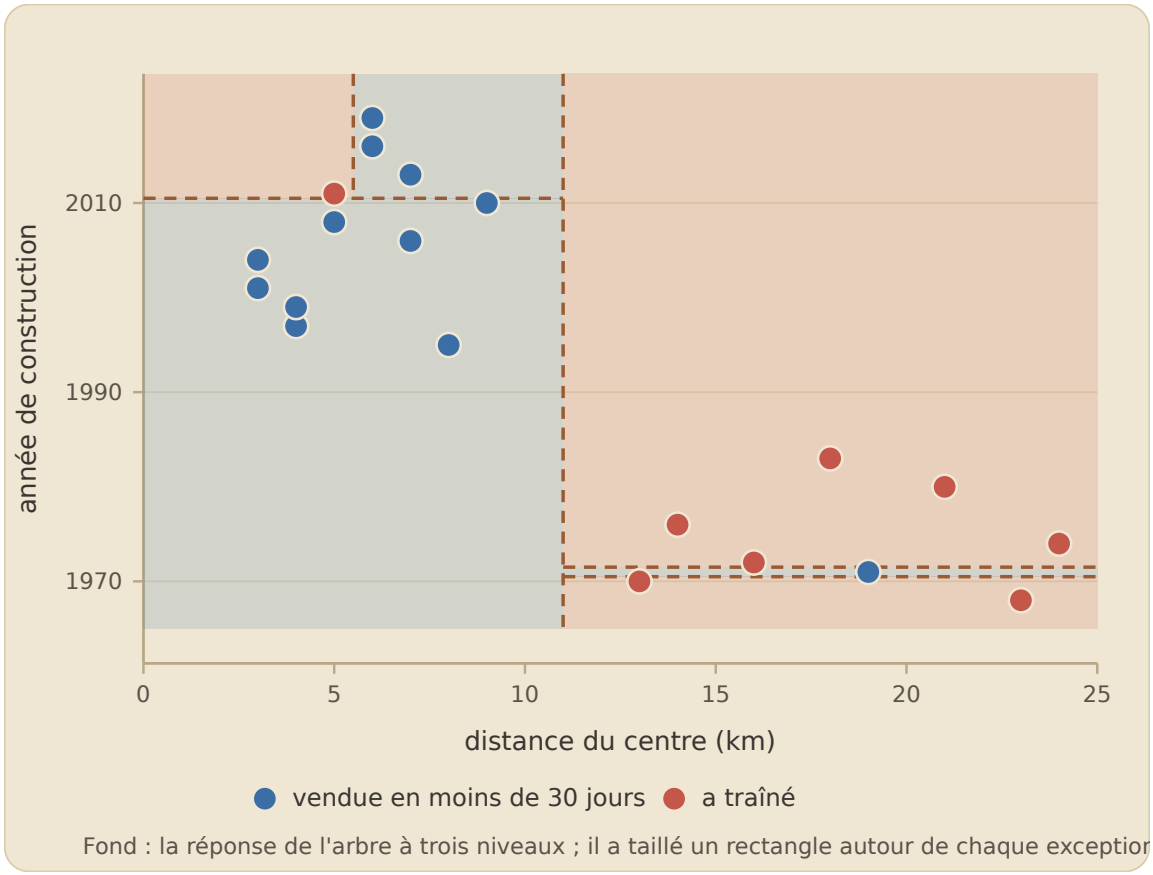
Revenons à la classification et laissons l'arbre poser d'autres questions. Après « à plus de 11 km du centre ? », il cherche une question dans chaque feuille, puis dans chaque nouvelle feuille, jusqu'à ce qu'aucune feuille ne mélange plus les couleurs. Voici l'arbre obtenu sur trois niveaux :



Le même arbre, laissé pousser sur trois niveaux : six feuilles, zéro erreur, et une feuille sur mesure pour chaque exception.

L'arbre compte maintenant six feuilles et ne fait plus aucune erreur. Il a trouvé le moyen de traiter les deux exceptions. Une question sur l'année, puis une question sur la distance, isolent dans sa propre feuille la maison ancienne qui s'est vendue vite. Deux questions de plus isolent de même la

maison récente qui s'est vendue lentement. Dans le plan, chaque question ajoute une coupe, et la frontière devient un assemblage de rectangles :



La frontière de l'arbre à trois niveaux : des rectangles, dont deux taillés sur mesure autour des exceptions.

Observez les deux petits rectangles découpés autour des exceptions. Ils correspondent aux îlots que kNN dessinait avec $k = 1$, dans *Prédire par ressemblance*. Il s'agit du même phénomène, dans un autre modèle. Un arbre sans limite continue de pousser jusqu'à ce que chaque feuille soit pure, au besoin en donnant une feuille à chaque maison. Il s'ajuste alors aux vingt maisons dans leurs moindres détails et traite comme des règles les deux cas qui font exception. Une nouvelle maison, ancienne et éloignée du centre, qui tomberait dans le rectangle de l'exception serait déclarée « vendue vite » sur la base d'un seul exemple.

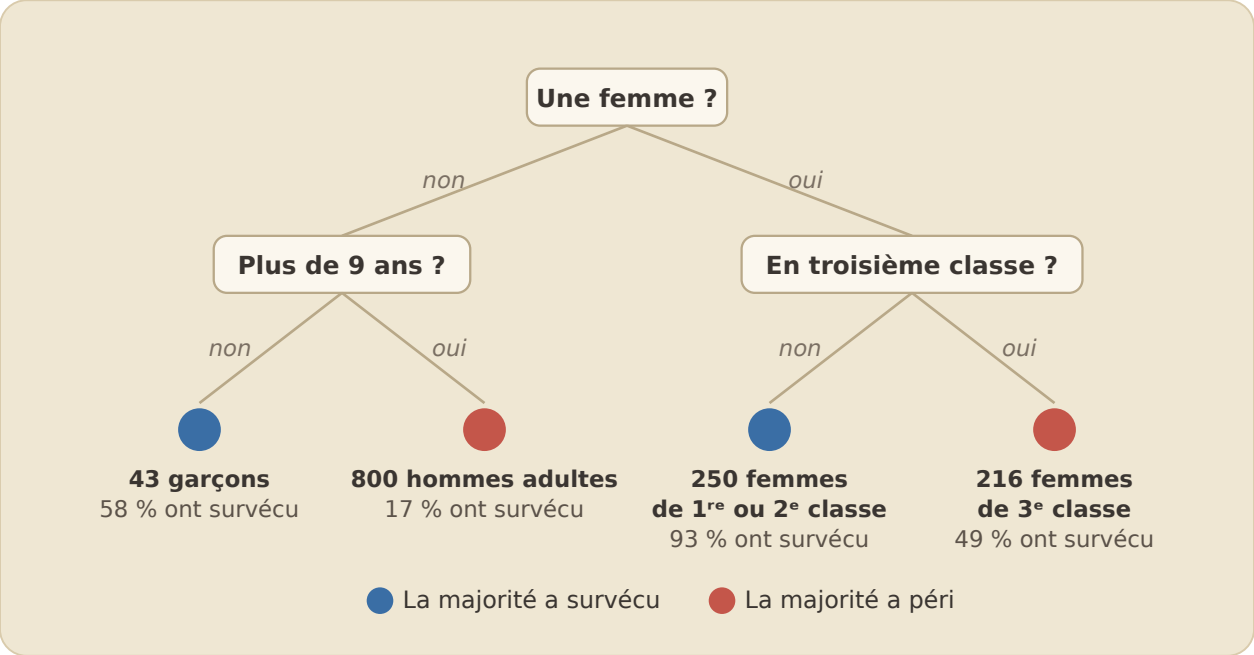
Il faut donc, pour l'arbre comme pour kNN, régler un paramètre : la **profondeur**, c'est-à-dire le nombre de questions que l'arbre peut poser à la suite. Avec une profondeur trop faible, l'arbre est grossier : sur des données où les deux amas seraient moins nets, une seule coupe verticale ne suffirait pas. Avec une profondeur trop grande, il apprend les exemples par cœur. C'est le sur-apprentissage (*overfitting*) déjà observé avec kNN, sous une autre forme, et *Généraliser* en fera son sujet principal. La bonne profondeur se situe entre les deux : elle permet de saisir les vraies régularités sans reproduire les cas accidentels. Rien, dans les vingt maisons, n'indique quelle est cette profondeur. Nous avons rencontré ce problème avec k, et nous le retrouverons pour tous les modèles dans *Généraliser*, où l'arbre servira d'exemple principal. Pour l'instant, retenez les deux façons de limiter un arbre : fixer une profondeur maximale, ou le laisser pousser puis **l'élaguer** (*pruning*), c'est-à-dire supprimer après coup les branches qui n'apportent que du détail.

L'applet ci-dessous vous permet d'en faire l'expérience. Elle présente des points rouges et bleus, un arbre appris en direct et le fond correspondant dans le plan. À droite, l'arbre lui-même se redessine à chaque changement. Faites varier la profondeur maximale de 1 à 8 et observez les rectangles se multiplier autour des points isolés. Ajoutez un point d'une couleur au milieu de l'autre couleur et observez comment l'arbre s'adapte. Survolez une zone du plan pour voir son chemin s'allumer dans l'arbre.

Applet interactive : à utiliser en ligne, sur la page « Poser des questions : les arbres de décision » du site du cours.

L'arbre du *Titanic*

Passons maintenant de nos maisons à des données réelles, celles que nous utiliserons à propos des fuites dans *Bien évaluer un modèle* : la liste des 1309 passagers du *Titanic*, avec pour chacun la classe, le sexe, l'âge et l'indication de sa survie. Trente-huit pour cent des passagers ont survécu. Le modèle le plus bête, qui répond « tout le monde meurt », obtient donc 62 % de bonnes réponses. Il sert de point de comparaison. Laissons un arbre apprendre sur ces passagers, avec deux niveaux de questions :



L'arbre du Titanic, appris sur les vraies données : trois questions, 79 % de bonnes réponses, et une histoire qu'on peut lire.

La première question que l'arbre trouve, sans aucune indication, est « une femme ? ». À elle seule, elle donne 78 % de bonnes réponses, soit seize points de plus que le modèle de comparaison. Les deux branches posent ensuite des questions différentes. Pour les hommes, la question est « plus de 9 ans ? » : les 43 garçons ont survécu à 58 %, les 800 hommes adultes à 17 %. Pour les femmes, la question est « en troisième classe ? » : celles de première et de deuxième classe ont survécu à 93 %, celles de troisième classe à 49 %. Avec trois questions, l'arbre obtient 79 % de bonnes réponses.

Ce que cet arbre a appris peut être lu par tous : « les femmes et les enfants d'abord », la consigne donnée cette nuit-là, avec une nuance que les historiens confirment. Les femmes de troisième classe,

logées dans les entreponts, loin des canots, ont eu une chance sur deux de survivre. L'arbre n'a utilisé aucune source historique. Il a seulement compté, et il retrouve en trois questions ce que les historiens décrivent. Cet exemple montre aussi le comptage appliqué à un cas ambigu. La feuille des femmes de troisième classe, à 49 %, est presque moitié-moitié, et l'arbre y répond « a péri » à une voix près. Un niveau de plus la découperait selon l'âge, sans gain important : au-delà de trois questions, le taux de bonnes réponses ne change presque plus. Sur ces données, l'essentiel tient en trois questions, et le reste est du détail.

Un dernier point sera repris dans *Bien évaluer un modèle* : ce taux de 79 % a été mesuré sur les passagers qui ont servi à l'apprentissage. Sur un jeu de test, il serait un peu plus bas, et avec un arbre plus profond, l'écart serait plus grand.

Ce qu'un arbre dit, et ce qu'il tait

Comme l'a montré l'exemple du *Titanic*, la grande force de l'arbre est qu'il peut **expliquer** ses décisions. Pour toute prédiction, le chemin de la racine à la feuille constitue la justification, et elle se lit directement : « une femme, en troisième classe, donc une chance sur deux ». Cette propriété s'appelle l'**explicabilité** (ou *interprétabilité* ; en anglais *explainability*, *interpretability*). C'est la capacité d'un modèle à rendre compte de ses décisions dans des termes qu'un humain peut comprendre, vérifier et contester. L'arbre est l'un des modèles les plus explicables, parce que le modèle est lui-même son explication : en le lisant, on sait pourquoi il donne telle réponse.

Aucun autre modèle de ce module n'offre cette propriété au même degré. La droite donne une pente, qui se comprend facilement. La régression logistique donne des poids associés à chaque caractéristique, qu'il faut interpréter. kNN donne une liste de voisins, qui décrit la décision sans la justifier. Les grands réseaux de neurones du Module 3 n'offriront rien de comparable : ils comptent des milliards de paramètres, dont aucun n'a de sens pris isolément. On parle alors de **boîte noire** (*black box*) : le modèle répond, souvent très bien, mais personne ne peut dire pourquoi. Tout un domaine de recherche, l'*IA explicable* (*XAI*, pour *explainable AI*), cherche aujourd'hui à analyser ces modèles après coup, en déterminant quelles caractéristiques ont compté dans une décision. L'arbre, lui, n'a pas besoin de cette analyse.

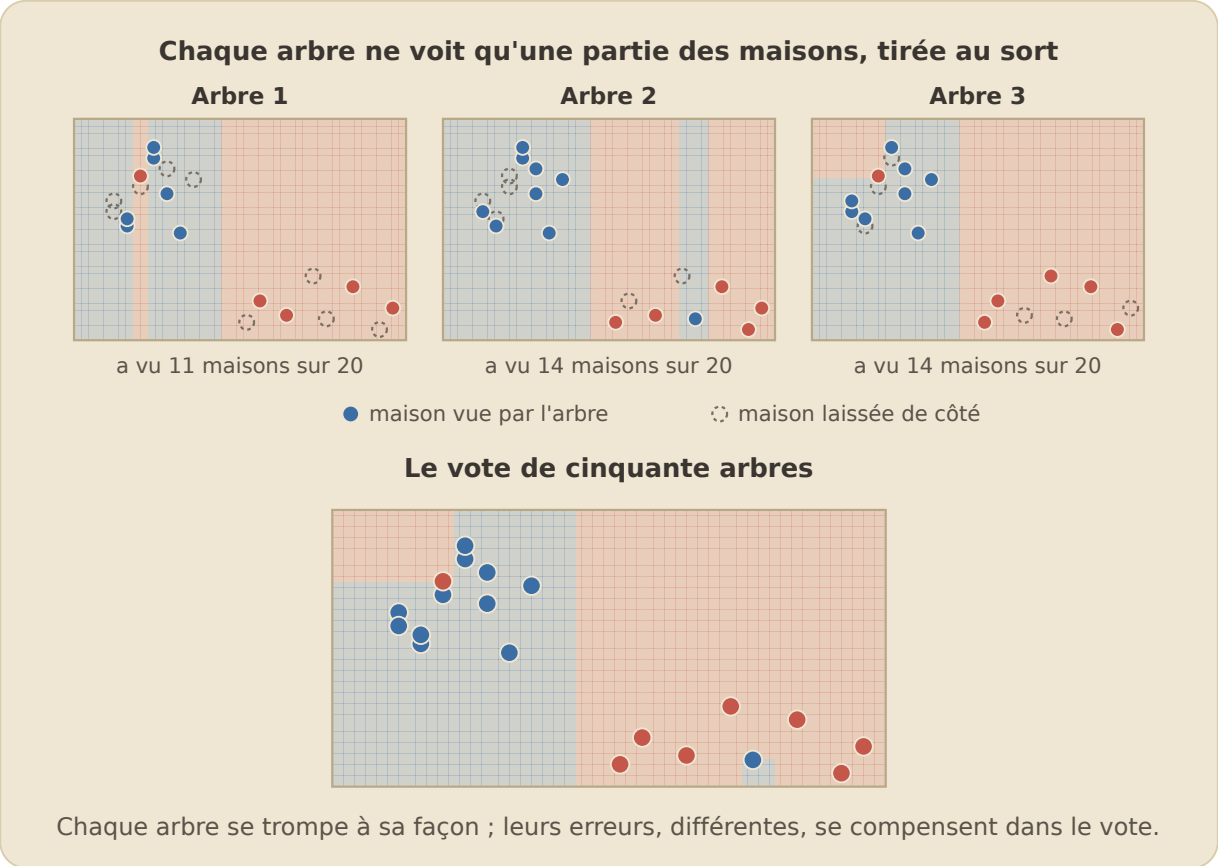
L'enjeu n'est pas seulement théorique. Dans les domaines où une décision doit pouvoir être contestée (un prêt refusé, un diagnostic, un dossier trié, une peine évaluée), un modèle qui ne peut pas expliquer ses décisions est difficile à accepter. La lisibilité compte alors davantage que quelques points de performance. C'est l'une des raisons pour lesquelles les arbres, apparus dans les années 1960 et mis au point en 1984 par quatre statisticiens (c'est la méthode CART, celle que nous avons suivie ici), sont toujours utilisés. Nous retrouverons cette opposition entre performance et explicabilité avec les grands modèles de langage du Module 4.

Les faiblesses de l'arbre découlent de sa méthode. D'abord, un arbre est **instable**. Rappelez-vous l'égalité entre la distance et l'année : si l'on retire deux maisons, la première question peut changer, et tout l'arbre avec elle. Deux jeux de données presque identiques peuvent produire deux arbres très différents, qui font pourtant à peu près les mêmes prédictions. Ensuite, ses coupes sont toujours parallèles aux axes, puisqu'une question ne porte que sur une caractéristique. Pour séparer deux amas le long d'une **diagonale**, l'arbre doit construire un escalier de nombreuses coupes, alors qu'une simple droite, comme celle de la régression logistique, y parviendrait en une fois. L'arbre est donc un modèle non linéaire, mais d'un type particulier, fait de marches.

La solution à l'instabilité est l'une des idées les plus productives du domaine, et elle se résume

simplement : si un arbre est fragile, on en utilise **cent**. On entraîne des centaines d'arbres, chacun sur une variante des données (un tirage au sort des maisons, un sous-ensemble des caractéristiques), puis on les fait voter ou on fait la moyenne de leurs réponses. Comme les erreurs des arbres sont différentes, elles se compensent, et la réponse collective est plus stable et plus juste que celle d'un arbre seul. C'est la **forêt aléatoire** (*random forest*), proposée par Leo Breiman en 2001. Une méthode apparentée et plus élaborée, le *gradient boosting*, construit chaque arbre de façon à corriger les erreurs des précédents. Sur des données en tableau (*tabular data*), comme nos maisons ou les passagers du *Titanic*, ces méthodes restent aujourd'hui parmi les plus performantes, et elles remportent la plupart des compétitions Kaggle dont parle *Bien évaluer un modèle*. Elles perdent cependant la lisibilité de l'arbre seul, car on ne peut pas lire cent arbres comme on en lit un.

La figure suivante illustre cette idée sur nos maisons. Trois arbres, chacun appris sur une partie des vingt maisons tirée au sort et laissé pousser sans limite, produisent trois frontières très différentes, avec des bandes et des îlots à des endroits différents. Si l'on fait voter cinquante arbres de ce type, ces particularités s'annulent, et il ne reste que ce sur quoi les arbres s'accordent.



La forêt aléatoire : chaque arbre ne voit qu'une partie des maisons et se trompe différemment ; le vote de cinquante arbres garde ce sur quoi ils s'accordent.

Deux cultures. Leo Breiman, l'inventeur de la forêt aléatoire, était un statisticien venu à l'apprentissage automatique. Il a tiré de ce parcours un texte devenu classique, « Statistical Modeling: The Two Cultures » (2001), annoncé dans la page d'accueil de ce module. Selon sa thèse, deux cultures se partagent l'analyse des données. La première, celle de la statistique classique, sup-

pose que les données ont été produites par un mécanisme simple qu'un modèle lisible peut décrire, et elle évalue le modèle selon sa fidélité supposée à ce mécanisme. La seconde, celle de l'apprentissage automatique, considère le mécanisme comme inconnu et n'évalue un modèle que sur la qualité de ses prédictions sur des données nouvelles. L'arbre seul appartient à la première culture, la forêt à la seconde. Breiman parle d'un **dilemme d'Occam**. Le rasoir d'Occam recommande de choisir le modèle le plus simple, mais sur des données réelles le modèle le plus précis est rarement le plus simple, et il faut alors choisir entre comprendre et prédire, alors qu'on voudrait les deux. Breiman relevait aussi une conséquence qu'il appelait l'**effet Rashomon**, du nom du film où quatre témoins racontent quatre versions du même crime : sur les mêmes données, des modèles très différents sont souvent également bons. Vous l'avez constaté avec nos maisons, que la distance et l'année séparaient aussi bien l'une que l'autre. Il n'existe donc pas un seul bon modèle, et l'explication fournie par un arbre n'est qu'une explication possible parmi d'autres. Il faut donc rester prudent envers un modèle qui s'explique : il décrit un chemin de décision qui fonctionne, et non le mécanisme réel du monde. Nous reviendrons sur cette question au Module 5, à propos de ce que signifie expliquer.

Et sur des données neuves ?

Récapitulons. Nous disposons maintenant de plusieurs modèles : une droite qui prédit un nombre, deux classificateurs qui prédisent une catégorie, et un arbre qui fait les deux en posant des questions. Nous avons aussi vu deux façons de les entraîner : minimiser une fonction d'erreur, ou chercher la question la plus pure. Chacun de ces modèles apprend en s'ajustant le mieux possible **aux exemples qu'on lui a montrés**.

Or ce n'est pas ce qui nous intéresse vraiment. Un filtre anti-pourriel qui classe parfaitement les courriels d'hier (ceux qui ont servi à l'entraîner) est inutile s'il se trompe sur ceux qui arriveront demain. Depuis le début de ce module, le but n'est pas de mémoriser des exemples, mais d'en tirer ce qu'il faut pour traiter des cas **encore jamais vus**.

Il y a là une difficulté. Un modèle peut ne faire aucune erreur sur ses données d'entraînement et faire beaucoup d'erreurs sur des données nouvelles, comme un étudiant qui aurait appris par cœur les réponses du corrigé sans comprendre la matière. C'est le sur-apprentissage (*overfitting*), déjà rencontré deux fois, que la page suivante, « Généraliser », traite en détail. Nous venons d'en voir l'exemple le plus net : l'arbre laissé pousser jusqu'à isoler chaque exception, qui ne se trompe plus sur les vingt maisons mais n'en a retenu que ce qu'il peut reproduire. Nous avions déjà rencontré ce problème avec kNN et le réglage de k : trop s'ajuster aux exemples peut être une faiblesse plutôt qu'une force.

Il faut donc savoir mesurer si un modèle a vraiment appris ou s'il a seulement retenu les exemples, et savoir éviter ce problème. C'est la question de la **généralisation**, qui fait l'objet de la page suivante.

Généraliser

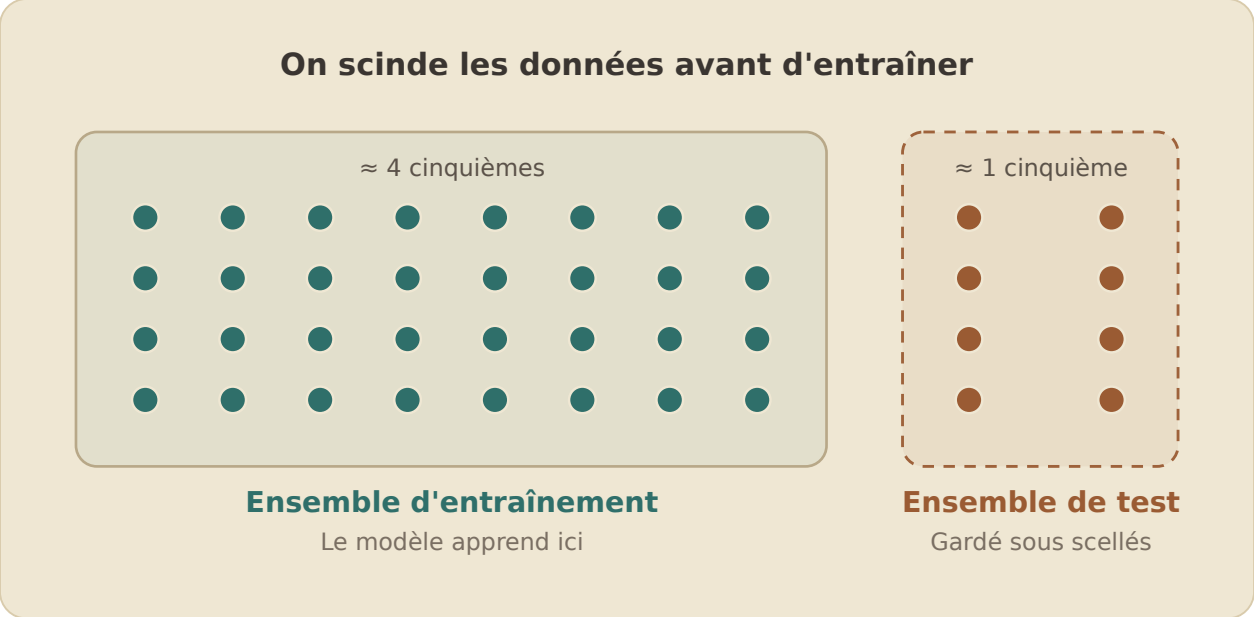
Le chapitre « Poser des questions : les arbres de décision » s'est terminé sur un doute. Nous savons maintenant entraîner plusieurs sortes de modèles (une droite qui prédit un prix, des classificateurs qui rangent en catégories), et tous apprennent de la même façon, en rendant leur erreur la plus petite possible sur les exemples qu'on leur montre. Ce résultat ne prouve cependant rien. Ce qui compte n'est pas qu'un modèle réussisse bien sur les données passées, mais qu'il se comporte bien sur des données nouvelles.

Le problème tient à une distinction. Un modèle peut avoir vraiment **appris** quelque chose des données, c'est-à-dire une régularité qu'il pourra appliquer ailleurs, ou s'être contenté de les **retenir**. Les deux cas se ressemblent beaucoup tant qu'on regarde les exemples d'entraînement. Ils ne se distinguent que devant des données nouvelles, et on découvre parfois alors qu'un modèle qui semblait bon ne vaut rien.

La capacité à bien se comporter au-delà des exemples appris s'appelle la **généralisation**. C'est elle, et non l'erreur d'entraînement, qui mesure la valeur réelle d'un modèle. Ce chapitre lui est consacré. Il explique comment la mesurer, pourquoi elle est difficile à obtenir et ce qu'elle révèle sur la nature des modèles.

Un modèle se juge sur ce qu'il n'a jamais vu

La solution est simple : **on cache des exemples au modèle.** Avant l'entraînement, on met de côté une partie des données, par exemple un cinquième. Le modèle apprend sur le reste, sans jamais voir cette réserve. Une fois le modèle entraîné, on l'évalue sur la réserve. Ces exemples sont nouveaux pour lui, mais nous connaissons les bonnes réponses. Sa performance sur cette réserve donne une estimation honnête de ce qu'il fera face à des données réellement nouvelles.



On sépare les données : le modèle apprend sur l'ensemble d'entraînement, et l'ensemble de test, mis de côté, sert à mesurer sa généralisation.

L'analogie de l'examen s'applique bien. Un enseignant qui noterait ses étudiants uniquement sur les questions distribuées d'avance pour la révision ne mesurerait pas grand-chose, parce qu'on peut apprendre ces réponses par cœur sans comprendre. C'est le sur-apprentissage (*overfitting*) de l'étudiant, et le jeu de test permet de le détecter. Pour évaluer la compréhension, il faut des questions nouvelles. On procède de la même façon avec un modèle.

Ces deux parties des données ont un nom. L'**ensemble d'entraînement** (*training set*) sert à l'apprentissage du modèle, et on y mesure l'**erreur d'entraînement**. L'**ensemble de test** (*test set*) reste à l'écart jusqu'à la fin et sert à mesurer l'**erreur de test**, la seule qui estime la généralisation. La règle d'or est qu'*on ne touche jamais au jeu de test pendant l'entraînement*. Si le modèle apprend, même indirectement, sur ses propres données d'examen, le test ne veut plus rien dire.

Lorsqu'il faut *régler* quelque chose (la valeur de k pour kNN, ou le taux d'apprentissage vu dans *Un modèle qui s'entraîne*), on ne peut pas non plus utiliser le jeu de test pour choisir, car il ne serait alors plus neuf. On réserve donc un troisième ensemble, l'**ensemble de validation** (*validation set*), consacré à ces réglages. Le jeu de test reste intact pour l'évaluation finale.

La règle d'or, à l'échelle d'aujourd'hui : les grands modèles de langage. Toute cette section suppose qu'on sait ce que le modèle a vu pendant l'entraînement. Pour un LLM, ce n'est plus vraiment possible. Ses données d'entraînement, des milliers de milliards de mots recueillis sur le Web, sont si vastes que personne n'en connaît le contenu précis, pas même ceux qui l'ont entraîné. Or les examens qu'on lui fait passer (un problème de mathématiques, une question d'un test standardisé, un exercice de programmation) ont très probablement circulé sur le Web, avec leur corrigé. On ne peut donc plus savoir avec certitude si le modèle a raisonné ou s'il a retrouvé la réponse. C'est le cas interdit par la règle d'or, un modèle qui a vu, même

indirectement, ses données d'examen, mais à une échelle où personne ne peut plus le contrôler. Il faut donc se méfier des scores impressionnants qu'on annonce.
Le problème ne concerne pas seulement la mesure. Au début du module, la question de la 1001e image reposait sur le fait de savoir si elle faisait partie des 1000. Pour un LLM, on ne peut plus répondre à cette question. Distinguer ce qu'un modèle sait déjà (parce que c'était dans ses données) de ce qu'il apporte de nouveau (parce qu'il généralise) est devenu, pour ces systèmes, une question ouverte, et en partie philosophique : où finit la mémoire, et où commence la compréhension ? Ce chapitre a posé cette question sur vingt maisons. Le Module 4 la reprendra à propos d'un modèle qui a lu une grande partie de ce que l'humanité a écrit.

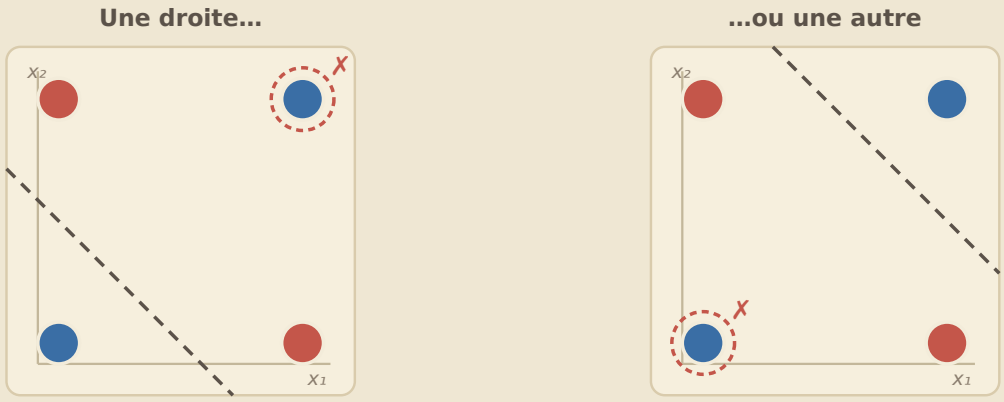
Linéaire ou non-linéaire : ce qu'un modèle peut dessiner

Revenons sur les modèles que nous avons construits. La régression linéaire d'*Un modèle qui s'entraîne* produit une droite. La frontière de la régression logistique est une droite. Celle de Bayes naïf est aussi une droite, comme nous l'avions remarqué. Même notre filtre anti-pourriel, malgré son grand nombre de mots, prenait une décision linéaire. Ces modèles ont donc un point commun.

Ils tirent tous leur décision d'une **somme pondérée** (*weighted sum*) des caractéristiques. Chaque attribut pousse d'un côté ou de l'autre, proportionnellement à son poids, et la décision dépend du total. Sur le plan géométrique, le résultat est toujours une droite (un plan en trois dimensions, un *hyperplan* au-delà). On appelle ces modèles des **modèles linéaires**.

La question est alors de savoir s'il existe des choses qu'**aucune droite** ne peut apprendre.

La réponse est oui, et vous connaissez déjà l'exemple le plus célèbre de cette limite. Le **perceptron**, présenté au Module 1, a été mis en échec sur ce point en 1969. L'exemple en cause est le **XOR**, *l'un ou l'autre, mais pas les deux*, la règle de l'interrupteur va-et-vient qui commande une lampe au bout d'un couloir. Si on place ses quatre cas dans le plan, les deux cas « éteint » se trouvent sur une diagonale et les deux cas « allumé » sur l'autre.



Le XOR : les bleus sur une diagonale, les rouges sur l'autre. Quelle que soit la droite, un point reste toujours du mauvais côté (cerclé de rouge).

Le XOR : quelle que soit la droite tracée, un point reste du mauvais côté. C'est la limite d'un modèle linéaire.

Quelle que soit la droite choisie et son inclinaison, il reste toujours un point du mauvais côté. Il ne s'agit pas d'un manque d'astuce, mais d'une impossibilité.

Il faut bien distinguer ce point de tout ce qui précède : **cet échec n'a rien à voir avec le bruit ni avec le manque de données**. Avec un million d'exemples parfaits, sans aucune erreur de mesure, un modèle linéaire échouera de la même façon. Le problème ne vient pas d'un mauvais réglage ni d'un manque d'entraînement. Il vient du **répertoire de formes** du modèle, qui ne contient pas la réponse. On appelle ce répertoire la **capacité** du modèle (ou son *expressivité*), c'est-à-dire l'ensemble des frontières qu'il peut dessiner. Un modèle linéaire ne peut dessiner qu'une seule famille de frontières, les droites.

kNN, au contraire, a une frontière qui peut prendre n'importe quelle forme. Elle peut entourer des îlots et contourner des groupes de points, car aucune contrainte de forme ne s'applique à elle. kNN résout le XOR sans difficulté, parce que chaque point regarde ses voisins et que les voisins d'un coin bleu sont bleus. kNN est donc un modèle **non-linéaire**, et c'était le premier que nous avons vu.

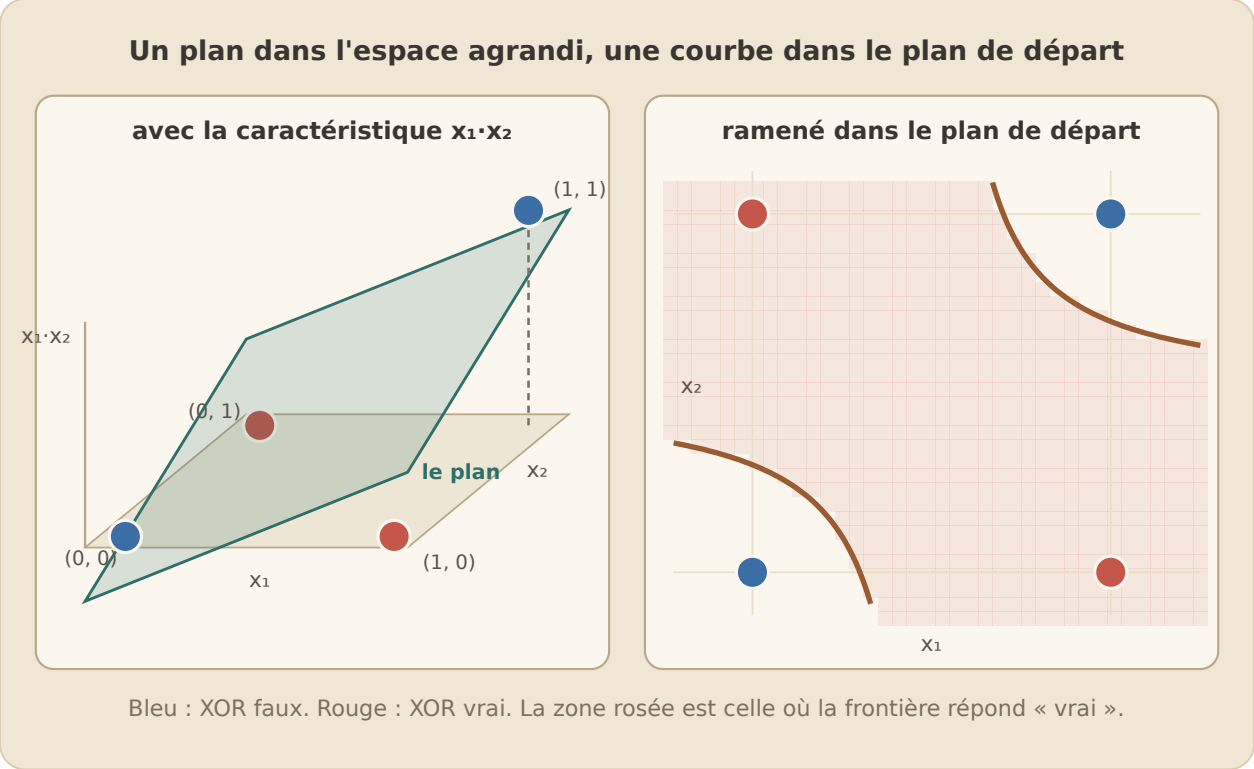
L'arbre de décision de la page précédente résout lui aussi le XOR, en deux questions, « à droite ? » puis « en haut ? ». Il obtient quatre rectangles, un par coin, soit la frontière que le perceptron ne pouvait pas tracer. Ses coupes parallèles aux axes sont une autre façon d'être non-linéaire, très différente de celle de kNN.

Il y a donc deux familles de modèles, et une question à se poser avant toute autre devant un problème : *la vérité que je cherche peut-elle tenir dans le répertoire de mon modèle ?* Si elle n'y est pas, aucun réglage et aucune donnée supplémentaire ne l'y mettront.

L'apprentissage automatique classique offre deux moyens de contourner cette limite. Le premier vient d'être présenté, il consiste à prendre un modèle non-linéaire comme kNN. Le second consiste à **construire soi-même la bonne caractéristique** (*feature engineering*). Si on ajoute aux deux entrées leur produit $x_1 \cdot x_2$, le XOR devient séparable par une droite dans ce nouvel espace. Cependant, c'est l'humain qui a dû trouver cette astuce, et il arrive que personne ne sache quelle caractéristique construire.

Ajouter des dimensions sans les calculer : l'astuce du noyau

La caractéristique $x_1 \cdot x_2$ résout le XOR, mais il a fallu la trouver. On peut contourner cette difficulté par la quantité : au lieu de chercher la bonne caractéristique, on en ajoute beaucoup, par exemple tous les produits et tous les carrés des entrées, $x_1 \cdot x_2$, x_1^2, x_2^2, et ainsi de suite. Dans cet espace agrandi, un modèle linéaire a beaucoup plus de chances de trouver une séparation par un plan. Et une séparation par un plan dans l'espace agrandi correspond à une frontière **courbe** dans le plan de départ.



Le XOR avec la caractéristique $x_1 \cdot x_2$: un plan le sépare dans l'espace agrandi, ce qui donne une frontière courbe dans le plan de départ.

Cette méthode a un coût. Le nombre de caractéristiques ajoutées explose avec le nombre d'entrées. Pour une image de 784 pixels, les seuls produits de deux pixels donnent déjà plus de 300 000 caractéristiques. Les produits de trois pixels en donnent environ 80 millions.

L'**astuce du noyau** évite ce calcul. Elle repose sur une propriété des SVM : pour trouver la frontière de plus grande marge, ils n'ont jamais besoin des coordonnées des points. Ils n'utilisent qu'une mesure de **ressemblance** entre deux points, calculée pour chaque paire. Or, pour de nombreux espaces agrandis, cette ressemblance peut se calculer directement à partir des coordonnées de départ, avec une formule simple appelée **noyau**, sans jamais construire les nouvelles caractéristiques. Le SVM travaille donc comme s'il disposait de millions de caractéristiques, sans en calculer aucune.

Le noyau le plus utilisé, le **noyau gaussien**, mesure la ressemblance de deux points par leur proximité : elle vaut 1 pour deux points confondus, et diminue rapidement avec la distance. Il correspond à un espace agrandi d'une infinité de dimensions. Un SVM équipé de ce noyau rappelle les plus proches voisins : un nouveau point est classé selon sa ressemblance avec les vecteurs de support, comme kNN le classait selon ses voisins. La différence est que le SVM ne garde que les vecteurs de support, et qu'il pondère leur influence pour obtenir la plus grande marge.

L'applet ci-dessous est celle de la section sur la marge, avec le jeu de points « XOR » et le noyau gaussien. Le curseur de portée règle la distance à laquelle deux points se ressemblent encore.

Applet interactive : à utiliser en ligne, sur la page « Généraliser » du site du cours.

1. Choisissez le noyau linéaire. Aucune droite ne convient, et des points restent mal classés.
2. Revenez au noyau gaussien. La frontière se courbe et sépare les quatre groupes.

3. Réduisez la portée. La frontière se met à entourer chaque groupe de près, et presque tous les points deviennent des vecteurs de support. C'est le sur-apprentissage (*overfitting*) de la section suivante.

L'idée du noyau date de 1964, avec les travaux de Mark Aizerman et de ses collègues à Moscou. Son union avec la marge maximale, en 1992 (Boser, Guyon et Vapnik), a fait le succès des SVM pendant les années 1990 et 2000.

L'astuce du noyau a pourtant deux limites. D'abord, c'est encore l'humain qui choisit le noyau, c'est-à-dire la façon de mesurer la ressemblance entre deux exemples. Le problème de la bonne caractéristique est déplacé, pas supprimé. Ensuite, le calcul porte sur toutes les paires d'exemples. Avec 10 000 exemples, cela fait 50 millions de paires. Avec un million d'exemples, cela devient impraticable. Ces deux limites expliquent en partie pourquoi les réseaux de neurones, qui apprennent leurs caractéristiques et s'adaptent bien aux très grandes quantités de données, ont fini par prendre le relais.

Le Module 3 reprend cette question, dans « Une couche cachée ». Les réseaux de neurones apprennent à **construire eux-mêmes** les caractéristiques qui rendent le problème séparable, en empilant des couches. La limite constatée en 1969 sera dépassée en 1986. C'est la question que le Module 1 avait laissée ouverte, et le Module 3 y répondra.

Cependant, la capacité de produire des frontières courbes n'est pas un avantage en soi. Un modèle capable de suivre n'importe quelle forme peut aussi suivre des formes qui ne correspondent à rien. C'est le sujet de la section suivante.

Trop coller, ou trop lisser : le compromis biais-variance

Nous venons de voir qu'un modèle doit être assez *expressif* pour que la vérité tienne dans son répertoire. L'excès inverse pose aussi un problème : un modèle trop souple finit par suivre le hasard autant que le signal. Pour observer ce compromis (et le mesurer, puisque nous savons maintenant le faire), revenons à kNN. Son seul réglage, le nombre de voisins k, contrôle directement la souplesse du modèle. Dans l'applet ci-dessous, faites glisser lentement k d'une extrémité à l'autre et observez la frontière.

Applet interactive : à utiliser en ligne, sur la page « Généraliser » du site du cours.

Avec $k = 1$, chaque point ne consulte que son plus proche voisin. La frontière se déforme pour entourer chaque exemple, forme des îlots autour des points isolés et suit tous les détails. L'erreur d'entraînement est nulle, puisque chaque exemple est son propre plus proche voisin. Cependant, cette frontière tient compte de tous les accidents des données. Pour un point un peu aberrant ou pour du bruit, elle se déforme quand même. C'est le **sur-apprentissage** (*overfitting*). Il correspond à l'étudiant qui a appris le corrigé par cœur, avec ses erreurs, sans le comprendre. Sur des données nouvelles, il se trompe.

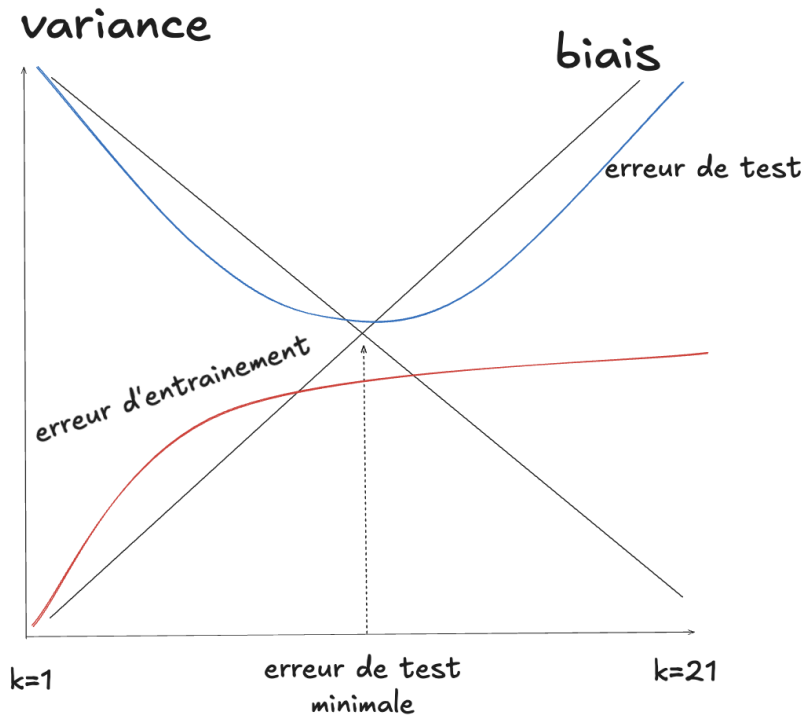
À l'autre extrême, avec k **très grand**, chaque prédiction fait la moyenne de tant de voisins que la frontière devient une courbe lisse, presque droite. Elle peut alors effacer des structures qui existent réellement dans les données. C'est le défaut inverse, le **sous-apprentissage** (*underfitting*) : le modèle est trop rigide pour suivre la forme réelle des données.

Il y a donc deux façons d'échouer, qui ont chacune un nom :

- la **variance** est la sensibilité du modèle au hasard de l'échantillon (du côté des petites valeurs de k). Si on change quelques points d'entraînement, un modèle à haute variance change complètement ;
- le **biais** est la rigidité propre au modèle (du côté des grandes valeurs de k), c'est-à-dire son incapacité *systématique* à représenter la forme réelle des données, quels que soient les points qu'on lui montre.

Qu'est-ce que le bruit ? Nous disons qu'un modèle trop souple « suit le bruit ». Cela signifie que les données ne sont jamais un signal pur. Le prix d'une maison est une tendance (plus la maison est grande, plus elle est chère) à laquelle s'ajoute une part aléatoire : l'humeur du vendeur, la saison, une négociation et beaucoup d'autres facteurs que le registre n'a pas notés. Nos vingt maisons ne sont pas sur la droite, elles sont autour, et l'écart de chacune correspond à ce bruit. Un bon modèle apprend la tendance et ignore le bruit. Un modèle trop souple apprend les deux, et le bruit qu'il a appris ne se reproduira pas pour la maison suivante, puisqu'il est aléatoire. La courbe en U présentée plus bas découle de cette idée : toute donnée contient une part qu'il ne faut pas apprendre.

Voici l'une des idées les plus importantes du domaine. Quand on rend un modèle plus souple (ici, en diminuant k), son erreur d'entraînement ne fait que diminuer, car un modèle flexible s'ajuste toujours mieux aux données qu'il a déjà vues. Son erreur de test suit au contraire une courbe en **U**. Elle diminue d'abord, parce que le modèle représente mieux les régularités réelles, puis elle **augmente** dès que le modèle commence à suivre le bruit. Le meilleur modèle se trouve au bas du U, à l'équilibre entre biais et variance.



L'erreur de test (en bleu) suit une courbe en U : trop de variance à gauche, trop de biais à droite. Le meilleur modèle est au creux.

Ce phénomène **n'est pas propre à kNN.** Chaque modèle a un réglage de souplesse : le nombre de termes d'une courbe plus souple qu'une droite (nous le verrons plus bas), la profondeur d'un arbre de décision ou le nombre de paramètres d'un réseau de neurones. Chacun présente la même courbe en U et le même arbitrage entre s'ajuster aux données et lisser. C'est le **compromis biais-variance** (*bias-variance tradeoff*), et savoir le régler est une compétence essentielle en apprentissage automatique.

Garder un modèle riche, mais le tenir en laisse : la régularisation

Le compromis biais-variance semble ne laisser qu'un seul moyen d'action, diminuer la souplesse du modèle, comme nous l'avons fait avec kNN en augmentant k ou avec l'arbre en limitant sa profondeur. Cette solution a cependant des limites. Elle restreint le modèle avant même qu'il ait vu les données, et le réglage est grossier, puisqu'il se fait par paliers. Or on veut parfois un modèle riche, capable de dessiner des formes compliquées si les données l'exigent, mais qui ne suive pas les moindres accidents des données.

Il existe une troisième approche, plus fine, qui se résume ainsi : **au lieu de limiter la souplesse du modèle, on la pénalise.** La fonction d'erreur vue dans *Un modèle qui s'entraîne* mesure à quel point le modèle se trompe sur les exemples, et l'entraînement consiste à la faire diminuer. On y ajoute un second terme, une **pénalité** qui augmente avec la *complexité* du modèle :

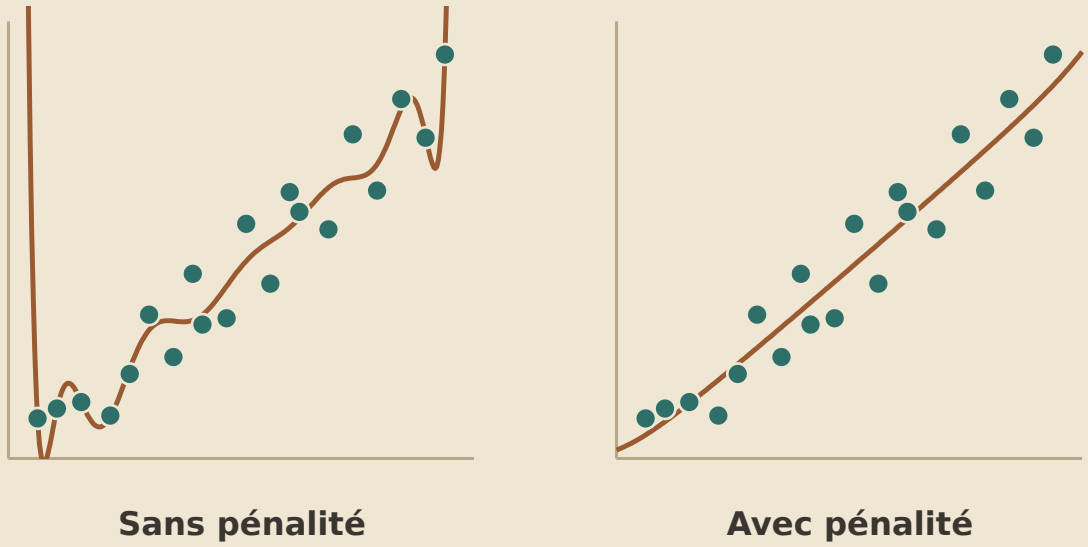
$$\text{erreur totale} = \text{erreur sur les données} + \lambda \times \text{complexité}$$

Le modèle doit alors trouver un équilibre. S'ajuster aux points fait diminuer le premier terme, mais augmente le second. Le modèle ne prendra une forme compliquée que si le gain le justifie. La descente de gradient va toujours vers le point le plus bas, mais dans un paysage modifié où les régions qui correspondent à des modèles trop compliqués ont été relevées.

Ce principe correspond à une idée vieille de sept siècles, le **rasoir d'Occam** : entre deux explications qui rendent compte des mêmes faits, il faut préférer la plus simple. La pénalité en est une traduction chiffrée. Elle n'affirme pas que le monde est simple. Elle impose qu'un modèle reste simple tant que les données n'exigent pas davantage, et que chaque complication apporte un gain visible. La courbe en U de la section précédente illustre expérimentalement ce principe : au-delà d'un certain point, la complexité supplémentaire ne sert qu'à apprendre du bruit.

Pour observer l'effet de la pénalité, il faut un modèle assez souple pour faire du sur-apprentissage (*overfitting*). Reprenons la droite d'*Un modèle qui s'entraîne* et rendons-la plus souple. Au lieu de prix $= m \times$ superficie $+ b$, on autorise aussi des termes en superficie^2, superficie^3, et ainsi de suite jusqu'à la puissance 12. Une telle courbe s'appelle un *polynôme*, et chaque terme ajouté apporte un paramètre de plus, ce qui fait passer le modèle de deux à treize paramètres. Le réglage de souplesse est ici le nombre de termes, appelé le *degré*. Avec treize paramètres pour vingt maisons, la courbe peut facilement passer en zigzag entre les points.

Il reste à définir concrètement la complexité. La réponse la plus courante est simple : **la taille des paramètres.** Un polynôme qui passe en zigzag entre les points doit monter et descendre rapidement. Pour cela, il lui faut des coefficients très grands, qui se compensent presque entre eux et ne laissent apparaître que de petites bosses. Obliger les coefficients à rester petits oblige donc la courbe à rester régulière. La pénalité s'écrit alors comme la somme des carrés des paramètres, et le reste ne change pas : même modèle, même degré, même descente.



Le même polynôme de degré 12, ajusté aux mêmes maisons ; seule la pénalité change.

Même modèle, même degré ; seule la pénalité change. À gauche, le polynôme sans pénalité suit le bruit ; à droite, avec la pénalité, il suit la tendance.

Le nombre λ règle la sévérité de la pénalité. Avec une valeur nulle, on retrouve le modèle sans contrainte. Avec une valeur trop grande, tous les paramètres sont écrasés et on retombe dans le sous-apprentissage (*underfitting*), avec au bout du compte une droite horizontale. C'est un hyper-paramètre de plus, et on le choisit comme les autres, sur l'ensemble de validation et jamais sur le jeu de test.

Cette idée porte des noms différents selon les modèles. Pour la droite et les modèles apparentés, on parle de *régression ridge* (pénalité sur les carrés) ou de *lasso* (pénalité sur les valeurs absolues, qui a la propriété de mettre certains paramètres exactement à zéro, et donc d'*éliminer* des caractéristiques). Pour les réseaux de neurones, on parle de *weight decay*, qui désigne le même terme. D'autres méthodes ne passent pas par la fonction d'erreur, mais visent le même but, empêcher le modèle de suivre le bruit. On peut arrêter l'entraînement avant que le modèle s'ajuste trop aux données (l'*arrêt précoce*, en anglais *early stopping*) ou désactiver au hasard une partie des neurones à chaque étape (le *dropout*). Pour l'arbre, l'**élagage** consiste à le laisser croître, puis à couper les branches qui n'apportent que du détail. Le Module 3 reviendra sur ces méthodes. Elles reposent toutes sur la même idée : la souplesse est une ressource, et un bon modèle est un modèle riche dont on contrôle la souplesse.

Une question reste ouverte pour plus tard. Si trop de souplesse nuit, comment les très grands réseaux de neurones actuels, qui ont des centaines de milliards de paramètres et donc une souplesse considérable, parviennent-ils à généraliser ? La réponse, qui remet en question la courbe en U, sera présentée au Module 3.

Paramétrique ou non-paramétrique

Il existe une seconde façon importante de classer les modèles. Elle ne porte pas sur leur souplesse, mais sur ce qu'il en reste une fois l'entraînement terminé. La question est de savoir si le modèle a **résumé** les données en un petit nombre de réglages ou s'il les **conserve**.

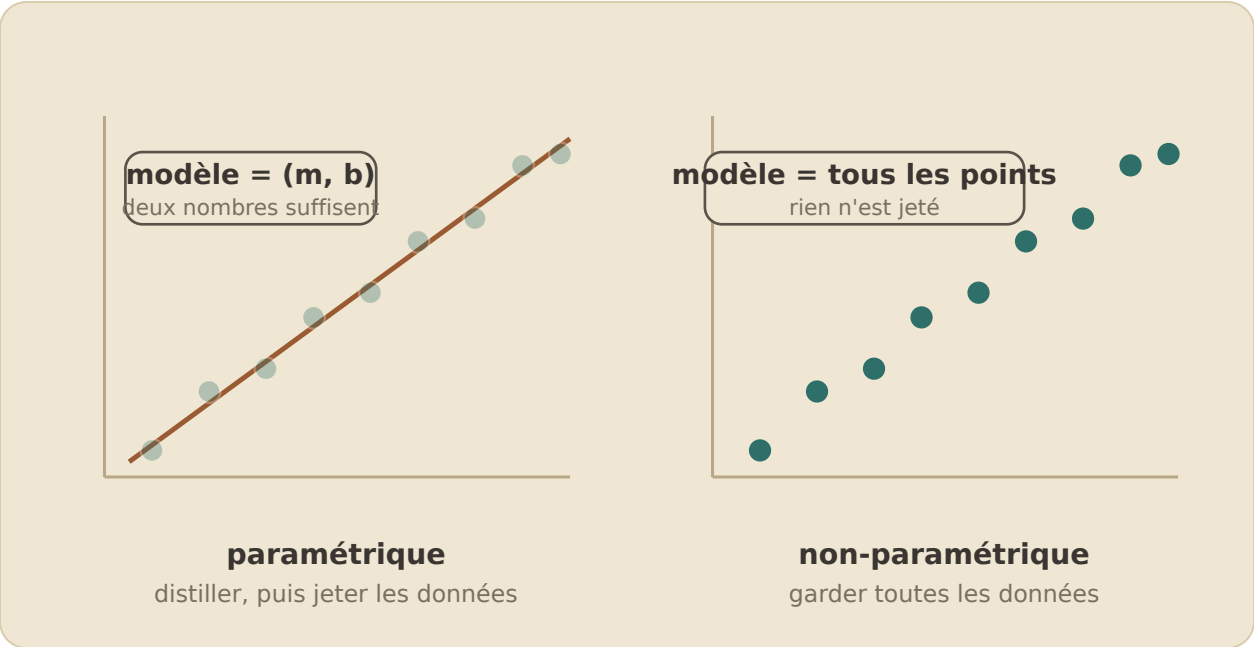
kNN, comme on l'a vu à propos de son angle mort, n'a rien à apprendre à proprement parler. Il n'a pas de paramètres à régler. Pour prédire, il consulte directement les exemples mémorisés, et les données constituent donc le modèle. Par conséquent, sa taille augmente avec le jeu de données. Avec mille exemples, il faut conserver mille exemples, et avec un million, un million. On dit d'un tel modèle qu'il est **non-paramétrique**, parce qu'il ne résume pas les données dans un nombre fixe de réglages et qu'il s'appuie directement sur elles.

À l'opposé, une fois la pente et l'ordonnée à l'origine de notre droite de régression trouvées, on peut **se débarrasser des données**. Il ne reste que deux nombres, m et b, et ils suffisent pour prédire. C'est aussi le cas de la régression logistique (un poids par caractéristique) et de Bayes naïf (une moyenne et une dispersion par classe, ou une probabilité par mot). Ces modèles sont **paramétriques**. Ils résument toute leur connaissance dans un ensemble de paramètres dont la taille est *fixée d'avance*, que l'apprentissage ait porté sur cent exemples ou sur cent millions. Cette distinction était déjà présente dans *Un modèle qui s'entraîne* : le modèle le plus bête résumait tout en *un* nombre, la droite en *deux*, et kNN en *aucun*. Il s'agissait déjà de l'axe paramétrique / non-paramétrique.

Chaque famille a ses avantages et ses inconvénients :

- le modèle **paramétrique** est léger et rapide pour la prédiction, et il généralise grâce à la compression qu'il impose aux données. Cependant, il *suppose une forme* (une droite, par exemple). Si la structure réelle des données n'a pas cette forme, aucun réglage ne corrigera le problème. C'est du **biais** ;
- le modèle **non-paramétrique** ne suppose presque rien sur la forme et peut représenter des structures très complexes. Cependant, il est lourd (il faut tout conserver), lent pour la prédiction et plus exposé au risque de suivre le bruit. C'est de la **variance**.

On retrouve ici le compromis biais-variance. Résumer les données ou tout conserver, supposer une forme ou suivre les données : il n'existe pas de réponse universelle. Il faut faire des choix adaptés au problème, et c'est une part importante du travail dans la discipline.



Paramétrique : résumer les données en quelques réglages, puis s'en débarrasser. Non-paramétrique : conserver toutes les données et s'appuyer dessus.

Nous savons maintenant ce qu'un modèle peut dessiner, jusqu'à quel point il faut le laisser se courber et comment contrôler sa souplesse. Il reste à mesurer tout cela correctement, car un score peut être trompeur de plusieurs façons. C'est l'objet de la page suivante, « Bien évaluer un modèle ».

Bien évaluer un modèle

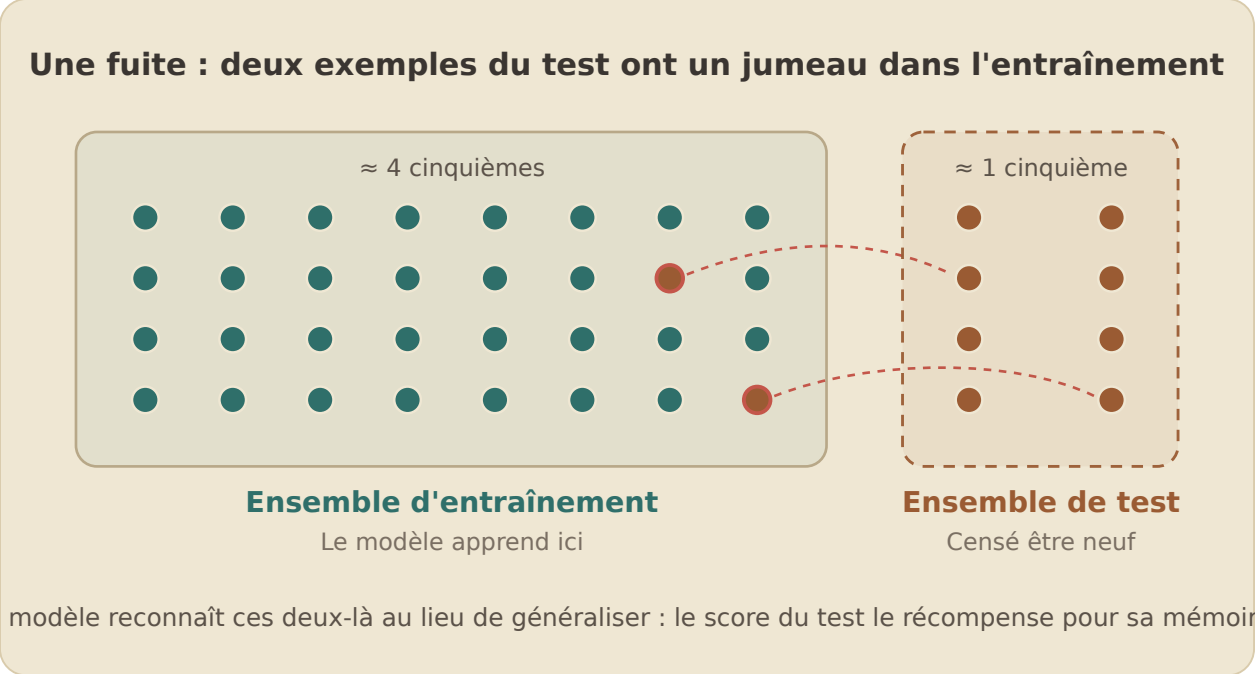
La page « Généraliser » a posé la règle d'or : on juge un modèle sur des exemples qu'il n'a jamais vus. Cette règle est nécessaire, mais elle ne suffit pas. Un score sur le jeu de test n'a de valeur que s'il est **honnête** (le modèle n'a-t-il vraiment rien vu du test ?), **fiable** (le score dépend-il du hasard du partage entre entraînement et test ?), **pertinent** (mesure-t-il ce qui coûte vraiment en cas d'erreur ?) et **valable** (s'applique-t-il aux données qu'on rencontrera réellement ?). Les quatre sections de cette page traitent ces quatre questions. Aucune ne demande de nouveau modèle. Toutes demandent de la méthode, et c'est surtout la méthode, bien plus que le choix d'un algorithme, qui distingue un modèle qui fonctionne d'un modèle qui semble fonctionner.

Le score trop beau pour être vrai : la fuite de données

Le premier piège est le plus courant et le plus difficile à repérer. Un modèle obtient 99 % de bonnes réponses sur le jeu de test, on le met en service, et ses résultats deviennent très mauvais. La cause est le plus souvent une **fuite de données** (*data leakage*) : une information qui n'aurait pas dû être disponible a été utilisée pendant l'entraînement, et le test n'était plus vraiment nouveau.

La fuite prend deux formes. La première est la **contamination** : le jeu de test contient, sous une autre forme, des exemples que le modèle a déjà vus. Par exemple, la même maison vendue deux fois en trois ans, une fois dans chaque ensemble, la même photo en deux résolutions ou le même courriel transféré à trois personnes. Le modèle ne généralise pas, il *reconnaît* les exemples, et le score récompense sa mémoire. À grande échelle, c'est le problème des grands modèles de langage présenté dans l'encart de la page précédente : quand les données d'entraînement couvrent tout le Web, on ne sait plus ce qui a fui.

La seconde forme est moins évidente. Il s'agit d'**une caractéristique qui contient la réponse**, ou dont on ne disposera pas au moment de prédire. L'exemple le plus connu vient du naufrage du *Titanic*, dont la liste des passagers est un classique des cours d'apprentissage automatique. On y prédit qui a survécu à partir de la classe, du sexe, de l'âge et du tarif payé. La version complète de cette liste, compilée à partir de l'*Encyclopedia Titanica* et hébergée par l'Université Vanderbilt, comporte deux colonnes de plus, le numéro du canot de sauvetage et le numéro d'identification du corps repêché. Si on les donne au modèle, il devient parfait, parce qu'avoir un numéro de canot signifie avoir survécu et qu'avoir un numéro de corps signifie être mort. Le modèle n'a rien deviné, il a lu la réponse, à peine cachée, dans les données. Nos maisons présentent le même piège. Pour prédire le prix, la « taxe foncière » donnerait d'excellents résultats, parce qu'elle est calculée à partir de la valeur de la maison.



Une fuite par contamination : deux exemples du test ont un jumeau dans l'entraînement. Le modèle les reconnaît, et le score récompense sa mémoire.

Il n'existe pas de détecteur automatique de fuites, mais il existe une méthode. D'abord, il faut se méfier : un score exceptionnel est un signal d'alerte avant d'être une bonne nouvelle, et la première question à poser est « qu'est-ce qui pourrait avoir fui ? ». Ensuite, il faut appliquer une règle systématique : **tout ce qu'on calcule à partir des données se calcule sur l'ensemble d'entraînement seulement**, qu'il s'agisse de moyennes, d'écarts, du choix des caractéristiques ou des réglages. Le jeu de test ne sert qu'une fois, à la fin, comme un examen qu'on ne corrige pas en cours de route.

Le jeu de test sous clé : Kaggle. Cette méthode existe aussi sous une forme institutionnelle. Sur Kaggle, la plateforme de compétitions d'apprentissage automatique, les participants reçoivent les données d'entraînement avec leurs réponses et les données de test *sans* leurs réponses. Les vraies réponses du test restent **privées**, et ce sont les organisateurs qui calculent le score de chaque soumission. Personne ne peut donc, même par erreur, entraîner son modèle sur l'examen. La plateforme va plus loin. Pendant la compétition, le classement public n'est calculé que sur une *partie* du jeu de test, et le classement final est calculé sur l'autre partie, gardée secrète jusqu'à la fin. Cette mesure empêche une fuite encore moins évidente, qui consiste à ajuster son modèle, soumission après soumission, au score affiché, jusqu'à s'ajuster au jeu de test sans l'avoir jamais vu. Les participants connaissent bien les changements importants de classement qui se produisent à la fin quand les deux parties donnent des résultats différents.

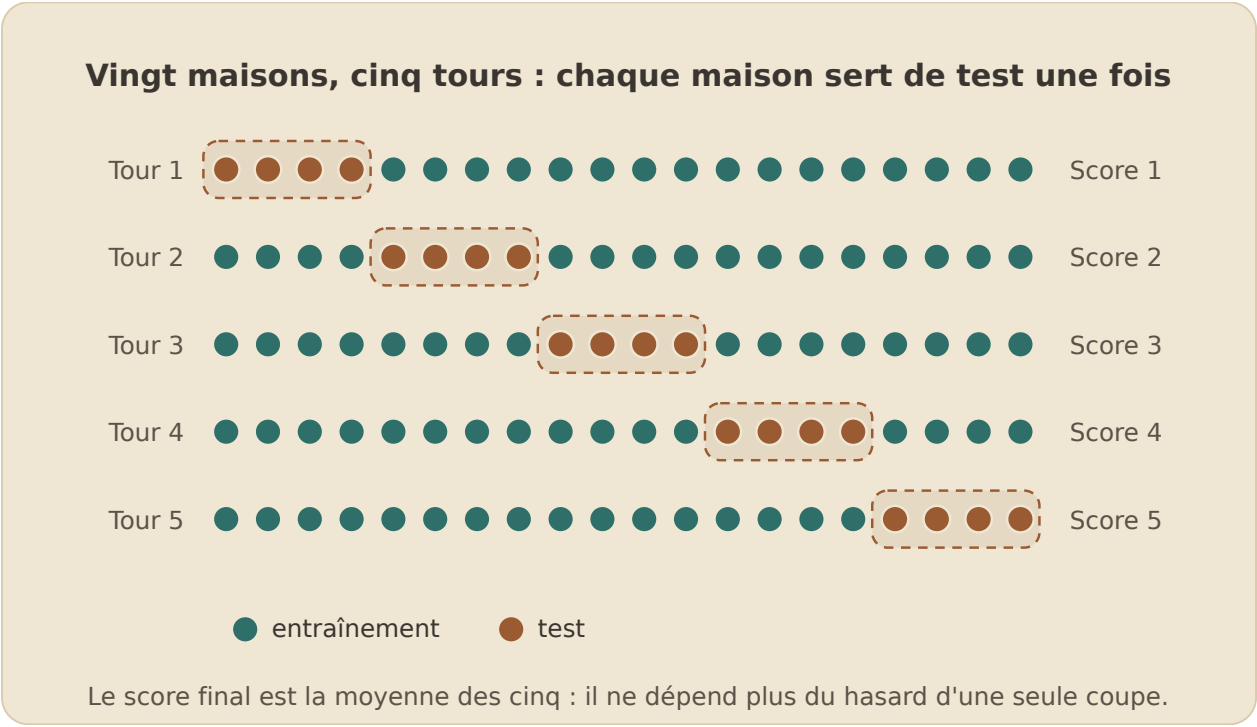
Une unité n'est pas l'autre : mettre à l'échelle. Cette règle a une application très concrète, que nous avons évitée sans la mentionner. Dans le plan des maisons, kNN mesure une distance qui combine une différence de kilomètres et une différence d'années. Or rien n'indique qu'un kilomètre « vaut » une année. Si on avait placé la superficie en mètres carrés à côté du nombre

de chambres, un écart de 50 m² aurait été beaucoup plus important qu'un écart d'une chambre, et la distance n'aurait plus eu beaucoup de sens. On corrige ce problème en **mettant à l'échelle** (*feature scaling*) chaque caractéristique, par exemple en la ramenant entre 0 et 1, ou en soustrayant sa moyenne et en divisant par son écart-type. La descente de gradient en profite aussi, car elle fonctionne mal dans une cuvette très étirée dans une direction. La règle s'applique ici aussi : les moyennes et les écarts utilisés pour la mise à l'échelle se calculent sur l'entraînement, puis s'appliquent tels quels au test.

Quand les données sont rares : la validation croisée

La deuxième question porte sur la fiabilité du score. Reprenons nos vingt maisons. Suivre la règle d'or revient à en mettre quatre de côté et à n'apprendre que sur seize. Cela pose deux problèmes. D'abord, seize maisons, c'est peu pour apprendre, et on voudrait utiliser les vingt. Ensuite, et surtout, le score dépend beaucoup du choix des quatre maisons mises de côté. Si ce sont les deux exceptions du nuage coloré, le modèle paraîtra mauvais. Si ce sont quatre maisons typiques, il paraîtra excellent. Le hasard d'un seul partage a donc trop d'influence.

La solution consiste à faire une **rotation** plutôt qu'un seul partage. On divise les vingt maisons en cinq groupes de quatre. Au premier tour, le premier groupe sert de test et les seize autres maisons servent à l'apprentissage. Au deuxième tour, c'est le deuxième groupe qui sert de test, et ainsi de suite. On obtient cinq tours et cinq scores, dont on calcule la moyenne. Chaque maison a servi de test exactement une fois, et chacune a servi à l'entraînement quatre fois sur cinq. C'est la **validation croisée** (*cross-validation*), et le nombre de groupes se choisit librement. On en utilise le plus souvent cinq ou dix, et jusqu'à n groupes d'un seul exemple quand les données sont très rares.



La validation croisée sur nos vingt maisons : cinq tours, chaque maison testée une fois, et le score final est la moyenne des cinq.

La validation croisée demande cinq entraînements au lieu d'un. C'est peu pour une droite, mais beaucoup pour un très grand réseau de neurones. C'est pourquoi on l'utilise souvent sur de petits jeux de données et presque jamais sur les très grands, pour lesquels un seul jeu de test suffit, parce qu'il est lui-même très grand.

La validation croisée est aussi le cadre habituel d'une opération que nous avons faite plusieurs fois sans vraiment la nommer, le réglage de ce qui ne s'apprend pas. Le nombre de voisins k, la sévérité λ de la pénalité et le taux d'apprentissage de la descente ne sont pas des paramètres du modèle, et aucun n'est ajusté par la descente de gradient avec les autres. Ce sont des **hyperparamètres**, c'est-à-dire des réglages *de la procédure*, fixés avant l'entraînement. On les choisit en essayant plusieurs valeurs et en gardant celle qui donne le meilleur score de validation croisée. Ensuite seulement, on utilise le jeu de test pour l'évaluation finale. C'est le rôle de l'ensemble de validation de la page précédente, appliqué en rotation.

Compter juste : les métriques

La troisième question est de savoir si le score mesure ce qui coûte vraiment. Jusqu'ici, pour une catégorie, nous avons compté le **taux de bonnes réponses** (*accuracy*). Nous connaissons déjà son défaut. Dans *Le modèle le plus bête*, un filtre qui ne signalait *jamais* de pourriel obtenait 99 % de bonnes réponses, parce que les pourriels étaient rares. Un seul nombre ne peut pas indiquer à la fois combien de pourriels on intercepte et combien de vrais courriels on élimine. Il faut compter ces deux quantités séparément.

Prenons 1000 courriels, dont 50 pourriels, et un filtre qui en élimine une partie. Quatre cas sont possibles pour chaque courriel, et on les range dans un tableau appelé la **matrice de confusion** (*confusion matrix*) : un pourriel éliminé (**vrai positif**), un pourriel qui passe (**faux négatif**), un vrai courriel éliminé (**faux positif**) et un vrai courriel conservé (**vrai négatif**). Ici, « positif » signifie « signalé par le filtre » et n'indique pas un résultat favorable.

Ces deux types d'erreur ne sont pas propres à l'intelligence artificielle. La statistique les a nommés bien avant, en 1933, dans les travaux de Jerzy Neyman et Egon Pearson. L'**erreur de première espèce** consiste à voir un effet là où il n'y en a pas (notre faux positif), et l'**erreur de seconde espèce** consiste à manquer un effet réel (notre faux négatif). En anglais, on parle de *type I and type II errors*. Ces notions s'appliquent à tout test, de l'essai clinique au contrôle de qualité en usine. Un tribunal qui condamne un innocent ou acquitte un coupable commet ces deux types d'erreur. L'apprentissage automatique a repris ces notions avec son propre vocabulaire.

1000 courriels, dont 50 pourriels : ce que le filtre en a fait

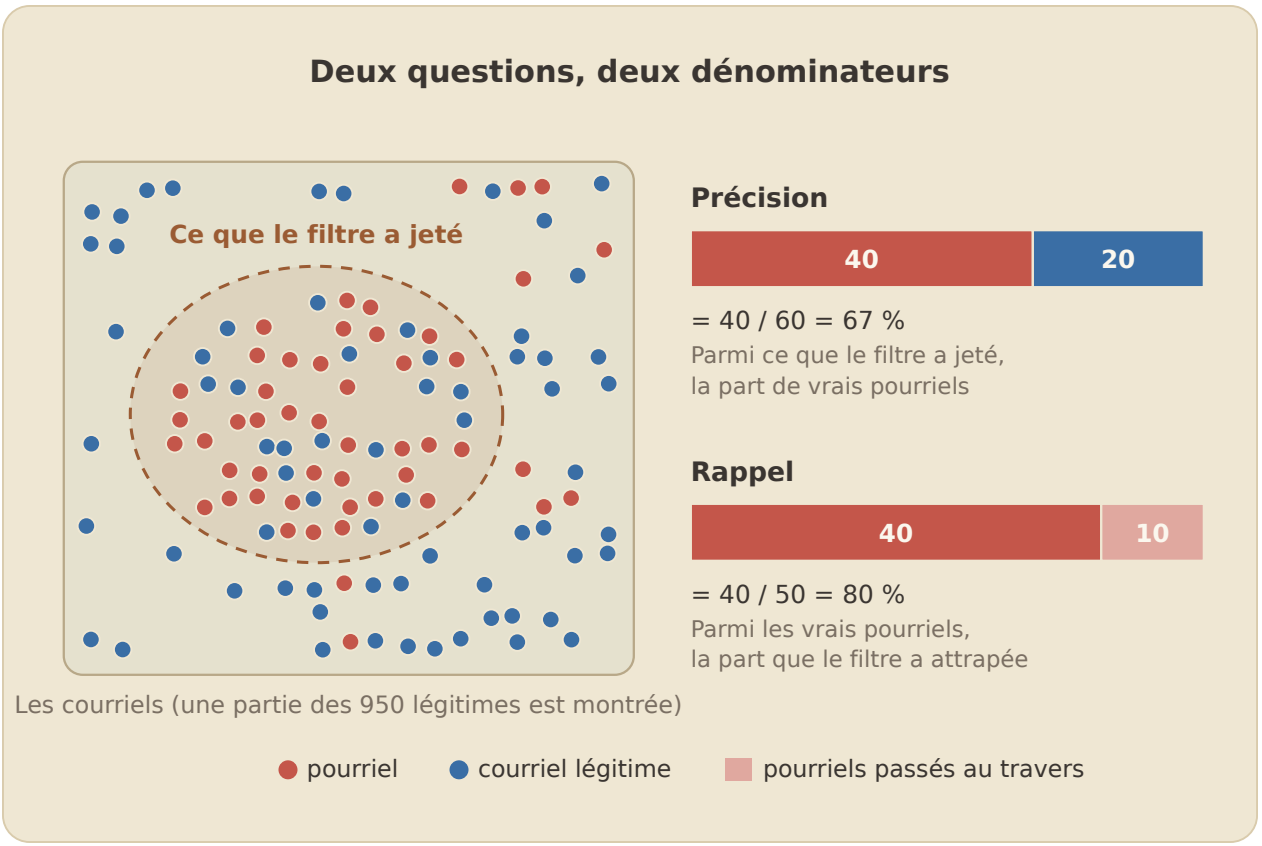
réalité \ *décision du filtre*	**jeté**	**gardé**
pourriel 50 en tout	**40** **vrais positifs** pourriels bien jetés	**10** **faux négatifs** pourriels passés au travers
légitime 950 en tout	**20** **faux positifs** vrais courriels jetés	**930** **vrais négatifs** vrais courriels bien gardés

Taux de bonnes réponses : (40 + 930) / 1000 = 97 %, à peine mieux que « jamais un pourriel » (95 %)

Précision : 40 / (40 + 20) = 67 % · Rappel : 40 / (40 + 10) = 80 %

La matrice de confusion : quatre cases au lieu d'un seul score. Les deux cases rouges sont les deux types d'erreur, qui n'ont pas le même coût.

Ce filtre obtient 97 % de bonnes réponses, alors qu'un filtre qui ne signale jamais de pourriel en obtiendrait 95 %. Le taux global cache donc presque tout. Le tableau permet de répondre à deux questions plus précises, qui ont chacune un nom. La **précision** est la part de vrais pourriels parmi les courriels que le filtre a éliminés. Ici, elle est de 40 sur 60, soit 67 %, et le reste correspond à des courriels légitimes perdus. Le **rappel** (*recall*) est la part des vrais pourriels que le filtre a interceptés. Ici, il est de 40 sur 50, soit 80 %, et le reste est passé. On obtient ainsi deux nombres au lieu d'un, et ils varient en sens contraire.



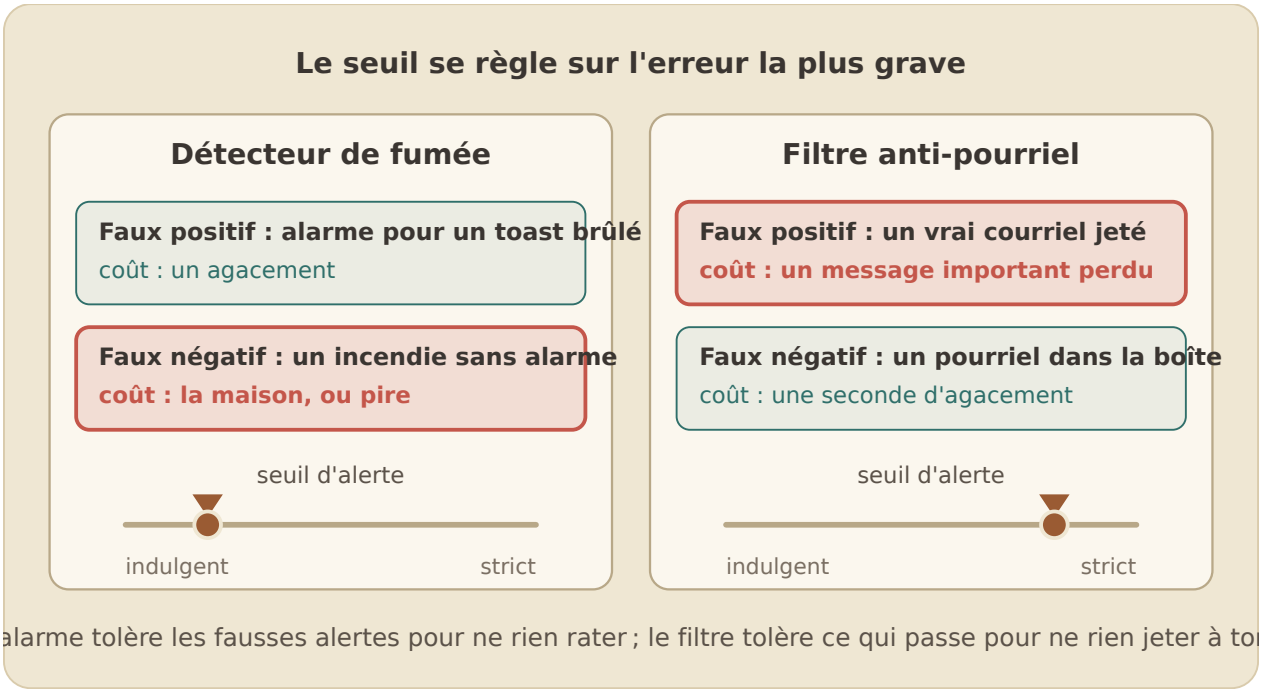
Précision et rappel, avec les nombres de la matrice : deux questions, deux dénominateurs. La précision se lit à l'intérieur du lasso ; le rappel, parmi les points rouges.

Ces deux termes ne viennent pas non plus de l'IA, mais de la **recherche d'information** (*information retrieval*), la discipline des catalogues de bibliothèque puis des moteurs de recherche. Pour une requête, un bon système retourne des documents *pertinents* sans y mêler de documents inutiles (la précision), et n'en oublie pas (le rappel). Dès les années 1960, on a fait de ces deux mesures la référence pour comparer les systèmes de recherche documentaire, et elles le sont restées. Précision et rappel forment aujourd'hui le vocabulaire commun de tous les systèmes qui *trient*. Un filtre anti-pourriel est d'ailleurs une forme de recherche, puisqu'il retrouve les pourriels parmi les courriels.

Ces deux mesures varient en sens contraire. La régression logistique de *Classer* donne une probabilité, et nous avons placé le seuil de décision à 0,5. Rien n'oblige à choisir cette valeur. Avec un seuil de 0,9, le filtre n'élimine que les courriels pour lesquels il est presque sûr, donc la précision augmente et le rappel diminue fortement. Avec un seuil de 0,1, il élimine un courriel au moindre doute, donc le rappel augmente et la précision diminue fortement. Le même modèle, sans nouvel entraînement, peut être strict ou tolérant, et ce choix dépend du problème et non des mathématiques. Pour un filtre anti-pourriel, un vrai courriel éliminé coûte plus cher qu'un pourriel qui passe, et on favorise donc la précision. Pour un test de dépistage, un malade non détecté coûte plus cher qu'une fausse alerte qu'un second examen permettra d'écarter, et on favorise donc le rappel.

Le cas d'un système d'**alerte** illustre bien ce choix. Un détecteur de fumée peut commettre deux erreurs, sonner pour un toast brûlé ou rester silencieux pendant un incendie. La première erreur cause un désagrément. La seconde peut coûter la maison, et parfois des vies. En raison de cette **asymétrie**, on règle l'appareil pour qu'il ne commette *jamais* la seconde erreur, même s'il commet

souvent la première. Un détecteur de fumée se déclenche sans raison plusieurs fois par année, et c'est voulu. Sa tolérance aux faux positifs n'est pas un défaut de conception. C'est le prix, accepté d'avance, d'un rappel proche de 100 %. Le filtre anti-pourriel suit le raisonnement inverse, parce que dans son cas c'est le faux positif qui coûte cher. Avec le même modèle et le même seuil réglable, on choisit des réglages opposés selon l'erreur qu'on ne peut pas se permettre. Ce réglage reste toutefois délicat, parce qu'il a deux extrêmes absurdes. Une alarme qui sonnerait *en permanence* ne manquerait aucun incendie, et un filtre qui n'éliminerait *aucun* courriel ne perdrait aucun message. L'une aurait un rappel parfait et l'autre une précision parfaite, mais les deux seraient inutiles, puisqu'ils ne trieraient plus rien. Tolérer l'erreur la moins grave ne veut donc pas dire l'accepter sans limite. On déplace le seuil du côté le plus sûr, mais seulement jusqu'au point où l'alerte garde un sens.



Le seuil se règle en fonction de l'erreur la plus grave. L'alarme tolère les fausses alertes pour ne rien manquer ; le filtre tolère les pourriels qui passent pour ne rien éliminer à tort.

Le travail noté vous posera cette question, *quelle erreur coûte le plus ?*, à propos d'un vrai filtre.

Pour une prédiction numérique, la même question se pose, de façon plus simple. L'erreur quadratique moyenne d'*Un modèle qui s'entraîne* est conçue pour être *minimisée*. Ses carrés sont pratiques pour la descente de gradient, mais difficiles à interpréter (ils sont exprimés en dollars au carré). Pour présenter les résultats, on préfère l'**erreur absolue moyenne** (*mean absolute error*). Sur nos vingt maisons, la droite se trompe de 41 000 $ en moyenne, et de 59 000 $ au maximum. Ces valeurs sont faciles à comprendre et permettent de juger si le modèle convient *à l'usage prévu*. Il est excellent pour obtenir un ordre de grandeur, mais insuffisant pour fixer un prix de vente.

Une métrique est donc un choix, qui définit ce qu'on considère comme une réussite. Avant de lire un score, il faut toujours savoir de quelle métrique il s'agit.

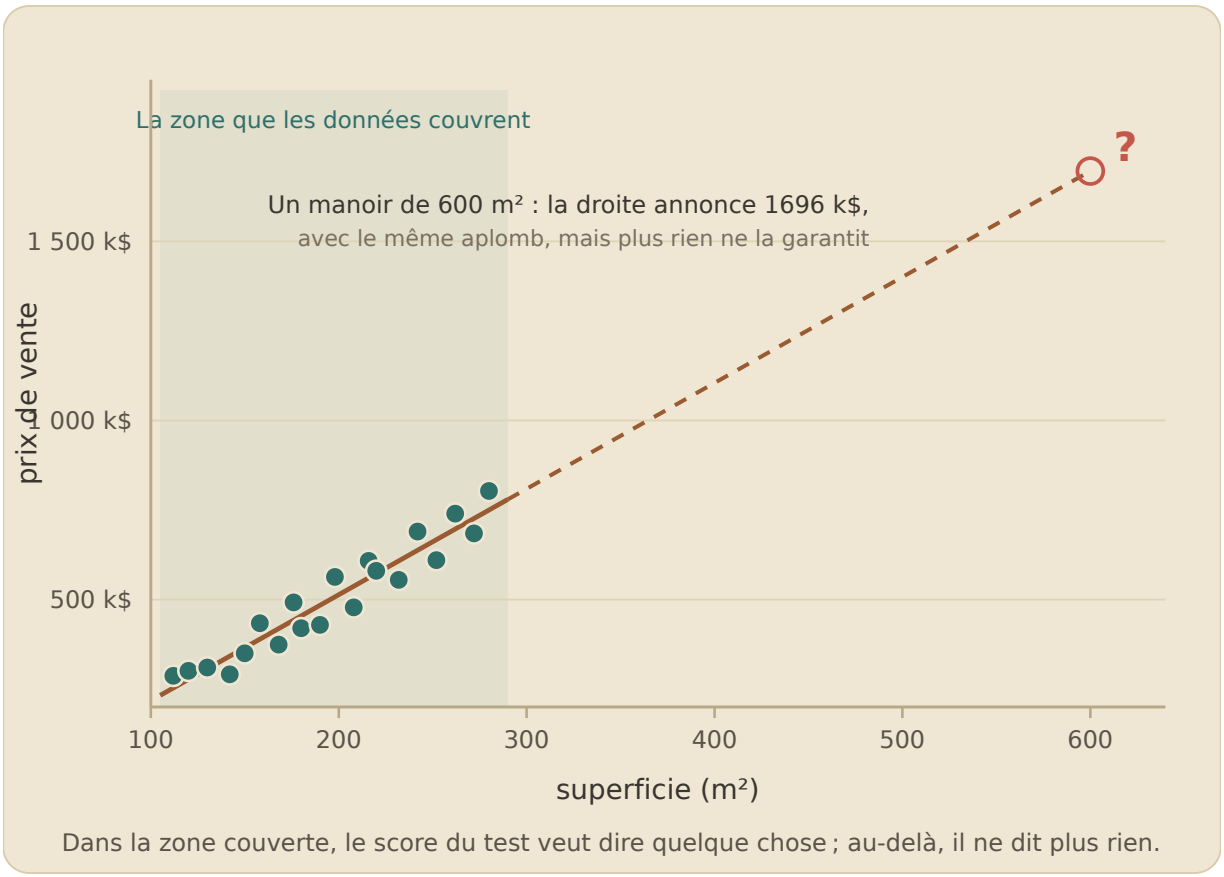
Jamais vu, mais du même monde : la question de la distribution

La dernière question est la plus facile à oublier. Il s'agit de savoir si le score s'applique aux données qu'on rencontrera réellement. Les sondages permettent de bien la comprendre. Interroger mille personnes pour connaître l'opinion de millions d'autres suppose que l'échantillon **ressemble** à la population, c'est-à-dire qu'il en est une version réduite. Quand ce n'est pas le cas, le sondage se trompe, même s'il paraît très sûr. L'exemple le plus connu date de 1936. Un grand magazine américain, le *Literary Digest*, avait recueilli plus de deux millions de réponses et prédisait une nette défaite de Roosevelt. Roosevelt fut réélu avec une large majorité. Les réponses provenaient de listes d'abonnés au téléphone et de propriétaires d'automobiles, en pleine crise économique. L'échantillon était très grand, mais il ne représentait pas l'ensemble des électeurs. Ce n'est pas la quantité qui garantit la qualité d'un sondage, c'est la **représentativité**.

Un jeu de données fonctionne comme un sondage. Nos vingt maisons ne sont pas l'ensemble des maisons. Ce sont vingt tirages dans un ensemble beaucoup plus vaste, celui de toutes les maisons qu'un tel modèle pourrait rencontrer, et les statisticiens appellent cet ensemble une **distribution**. Le jeu d'entraînement en est un échantillon, et le jeu de test en est un autre. Toute la méthode de ce chapitre repose sur une hypothèse implicite : les deux proviennent de la même distribution, et les données futures aussi. On dit alors que le test est **en distribution**. Un modèle généralise toujours à cette distribution, jamais au monde entier.

Quand cette hypothèse n'est plus vraie, c'est-à-dire quand une donnée est **hors distribution** (*out of distribution*), le problème peut prendre trois formes.

- **L'extrapolation.** Nos maisons vont de 112 à 280 m². Si on demande à la droite le prix d'un manoir de 600 m², elle répond 1 696 000 $, avec la même assurance que pour une maison de 180 m². Cependant, aucune donnée ne justifie cette réponse. La droite est prolongée hors de la zone couverte par les données, et rien n'indique que les manoirs suivent la même règle que les bungalows.
- **Le changement de lieu ou de population.** Un modèle entraîné sur les ventes de Montréal est appliqué à Vancouver, ou un test de dépistage mis au point sur des adultes est appliqué à des enfants. Les caractéristiques sont les mêmes, mais la distribution est différente, et le score obtenu dans un contexte ne dit rien de l'autre.
- **La dérive dans le temps.** Un filtre anti-pourriel entraîné sur les pourriels de 2010 est confronté à ceux de 2025. Les mots et les procédés ont changé. La distribution s'est modifiée sans que le modèle le détecte, et son score passé ne dit plus rien de sa performance actuelle.



Hors distribution : dans la zone couverte par les données, le score du test a un sens ; au-delà, la droite donne encore une réponse, mais rien ne la justifie.

Ces trois cas reproduisent la situation du *Literary Digest* : un échantillon, parfois très grand, mais qui ne provient pas de la population sur laquelle on fait les prédictions. On en tire deux règles pratiques. D'abord, le jeu de test doit ressembler aux données du **déploiement**, et pas seulement à celles de l'entraînement. Si le modèle doit servir à Vancouver, il faut des maisons de Vancouver dans le test, et on constatera rapidement qu'il en faut aussi dans l'entraînement. Ensuite, un modèle mis en service doit être **surveillé**, parce que le monde change et que rien, dans le modèle, n'indiquera que la distribution a changé.

Au début du module, la question de la 1001e image était de savoir si elle faisait partie des 1000. Une autre question s'y ajoutait : l'image provient-elle de la même distribution que les 1000 autres ? Une photo prise de nuit, alors que toutes les autres ont été prises de jour, est nouvelle d'une façon que le modèle ne sait pas traiter. Pour les grands modèles de langage présentés dans l'encart de *Généraliser*, la question devient très difficile : quand l'entraînement porte sur tout le Web, où se trouve la limite de ce qui est « en distribution » ?

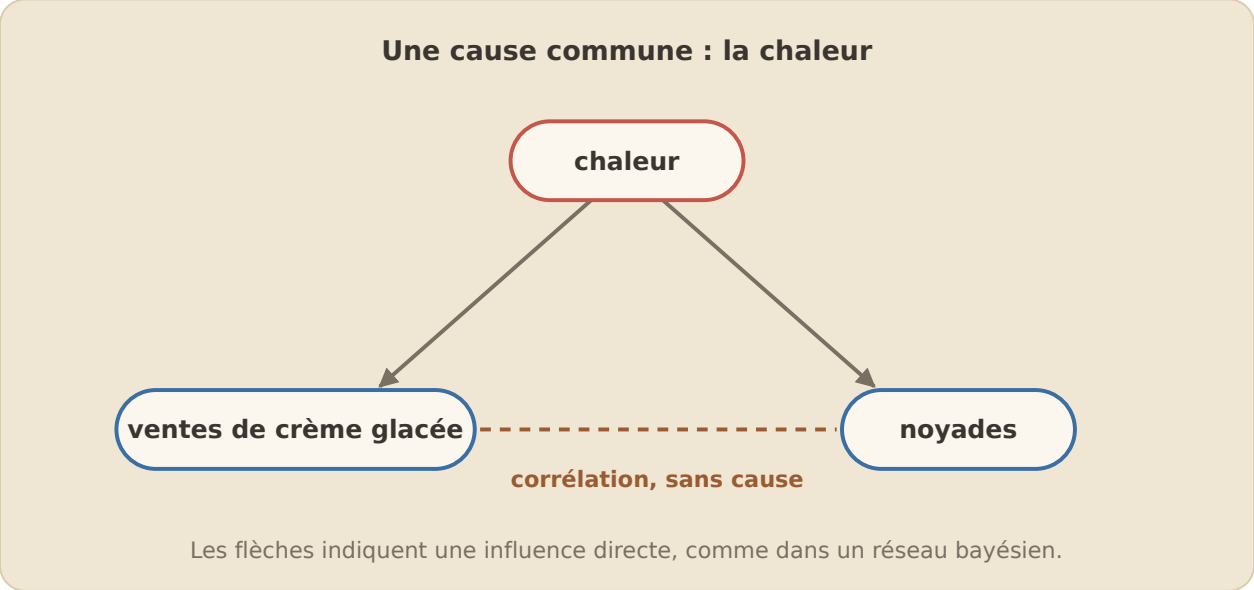
Une dernière conséquence dépasse le plan technique. Un modèle apprend *ce qui se trouve* dans ses données, et non ce qui devrait s'y trouver. Si les vingt maisons du registre viennent toutes des mêmes quartiers, le modèle reproduira leurs particularités. De même, si des décisions humaines passées ont laissé des traces dans les exemples (des prêts refusés plus souvent à certains groupes, des dossiers triés selon des habitudes que personne n'a jamais écrites), le modèle apprendra ces habitudes avec le reste, sans que cela soit visible, et les appliquera de façon systématique. C'est le **biais des données**

(*data bias*). Il ne se mesure sur aucun jeu de test, puisque le test provient des mêmes données. Un score honnête, fiable, pertinent et valable peut donc correspondre à un modèle injuste. C'est une raison de plus d'examiner la provenance des données, et pas seulement leur quantité. Le Module 5 reviendra sur cette question.

Corrélation n'est pas causalité

Les modèles de ce module apprennent des **associations** : quand telle caractéristique est présente, telle réponse est plus probable. Ils ne disent pas **pourquoi**. Cette différence semble théorique. Elle a pourtant des conséquences très concrètes.

L'exemple classique est celui de la crème glacée. Dans une ville, les jours où l'on vend beaucoup de crème glacée sont aussi des jours où il y a plus de noyades. Un modèle entraîné sur ces données prédirait très bien les noyades à partir des ventes de crème glacée. Pourtant, interdire la crème glacée ne sauverait personne. Les deux augmentent ensemble parce qu'elles ont une **cause commune**, la chaleur, qui pousse à la fois à acheter de la crème glacée et à se baigner. Une telle variable, qui influence à la fois ce qu'on observe et ce qu'on veut prédire, s'appelle un **facteur de confusion** (*confounder*).



La chaleur fait monter à la fois les ventes de crème glacée et les noyades : les deux sont corrélées, sans que l'une cause l'autre.

On reconnaît le dessin des réseaux bayésiens de *Classer* : une flèche pour chaque influence directe. Judea Pearl s'est servi de ces graphes pour distinguer une corrélation d'une cause.

Prédire n'est pas décider. Dans les années 1990, des chercheurs entraînent un modèle à prédire le risque de décès des patients atteints de pneumonie, pour décider lesquels hospitaliser. Le modèle découvre une règle surprenante : les patients asthmatiques auraient **moins** de risque de mourir. La corrélation est réelle. Mais elle vient de ce que les médecins envoyaient systématiquement les asthmatiques aux soins intensifs, où ils étaient mieux soignés. Le modèle prédisait bien. Utilisé pour décider qui hospitaliser, il aurait renvoyé chez eux les patients les plus fragiles. Ce cas a été rendu

public en 2015 par Rich Caruana et ses collègues, qui plaidaient pour des modèles dont on peut lire les règles.

Une corrélation peut disparaître. Une relation causale reste vraie quand le contexte change. Une simple corrélation peut disparaître. C'est l'une des raisons pour lesquelles les modèles échouent hors distribution. Le Module 3 en donne un exemple : un classifieur qui distinguait les loups des huskies en regardant la neige à l'arrière-plan. La neige était associée aux loups dans les photos d'entraînement, mais elle n'en est pas la cause. Ailleurs, la règle ne tient plus.

Établir une cause. La méthode de référence est l'**essai contrôlé à répartition aléatoire** (*randomized controlled trial*). On tire au sort les personnes qui reçoivent un traitement et celles qui n'en reçoivent pas. Le tirage au sort répartit les facteurs de confusion de la même façon dans les deux groupes, si bien qu'une différence entre eux ne peut venir que du traitement. La médecine l'emploie depuis un essai sur la streptomycine contre la tuberculose, publié en 1948. Les **tests A/B** (*A/B testing*) des sites Web, qui montrent deux versions d'une page à deux groupes d'internautes tirés au sort, reposent sur la même idée. Quand une expérience est impossible ou contraire à l'éthique, on peut parfois établir une cause à partir de simples observations, avec des précautions. Les économistes David Card, Joshua Angrist et Guido Imbens ont reçu le prix Nobel d'économie en 2021 pour leurs méthodes en ce sens.

L'échelle de la causalité. Dans *The Book of Why* (2018), écrit avec Dana Mackenzie, Pearl distingue trois niveaux de raisonnement, qu'il appelle l'échelle de la causalité (*ladder of causation*).

Niveau	Question	Exemple
1. Voir	Que puis-je prédire en observant ?	Les jours de grande vente de crème glacée, y a-t-il plus de noy
2. Agir	Que se passe-t-il si j'interviens ?	Si j'interdis la crème glacée, y aura-t-il moins de noyades ?
3. Imaginer	Que se serait-il passé autrement ?	Ce patient aurait-il survécu s'il avait été hospitalisé ?

Les modèles de ce module restent en général au premier niveau : ils apprennent des associations dans des données observées. Pour Pearl, une intelligence comparable à la nôtre doit monter les deux autres barreaux.

Cette question a pris de l'importance ces dernières années. Le prix Turing de Pearl en 2011, son livre de 2018 et le Nobel de 2021 l'ont fait connaître bien au-delà de la statistique. Des chercheurs en apprentissage profond, comme Yoshua Bengio et Bernhard Schölkopf, y voient une piste pour obtenir des modèles plus robustes. Et l'on se demande aujourd'hui si les grands modèles de langage raisonnent sur des causes, ou s'ils reproduisent des associations présentes dans leurs textes d'entraînement. Le Module 5 reviendra sur ces questions.

Tout cela portait un nom : l'apprentissage supervisé

Depuis la première page de ce module, un élément n'a jamais changé, et nous l'avons à peine remarqué : **la bonne réponse était toujours fournie.** Chaque maison avait son prix, chaque courriel son étiquette (pourriel ou non) et chaque point sa couleur. Le modèle devait seulement apprendre à passer de la question à une réponse *fournie d'avance*.

Cette situation porte un nom, que nous pouvons maintenant introduire après l'avoir étudiée en détail : l'**apprentissage supervisé** (*supervised learning*). Le terme « supervisé » renvoie à un élève

corrigé par un professeur qui connaît la réponse attendue. Régression ou classification, droite ou Bayes, paramétrique ou non, tout ce que nous avons construit appartient à cette grande famille, celle de l'apprentissage à partir d'exemples **étiquetés**.

Il reste à savoir d'où vient cette réponse fournie d'avance. Quelqu'un a dû étiqueter ces milliers d'exemples un à un, un travail souvent long et coûteux, et parfois impossible. Ce travail est si important qu'il est devenu une **industrie à part entière**, celle de l'*étiquetage de données* (*data labeling*). Des entreprises comme Scale AI (dans laquelle Meta a investi une quinzaine de milliards de dollars en 2025), Appen, Sama ou Labelbox, ou des plateformes comme le Mechanical Turk d'Amazon, emploient ou mobilisent des centaines de milliers de personnes pour tracer des contours sur des images, transcrire des enregistrements, classer des textes et, depuis l'arrivée des grands modèles de langage, comparer et noter des réponses générées. Ce travail, souvent invisible et mal payé, est réalisé en bonne partie au Kenya, aux Philippines ou au Venezuela. Il constitue la face cachée de l'apprentissage supervisé, puisque chaque « bonne réponse » fournie au modèle a été produite par un humain. Par ailleurs, la plupart des données disponibles n'ont *pas* d'étiquette : des millions de photos que personne n'a triées, des historiques d'achats sans catégories, des textes non classés. On peut se demander si un modèle peut apprendre quelque chose de données brutes, sans aucune bonne réponse. À l'inverse, quand un robot apprend à marcher, personne ne lui indique le « bon » mouvement à chaque instant. Il reçoit seulement un encouragement ou constate une chute, souvent bien plus tard. On peut se demander s'il s'agit encore d'apprentissage.

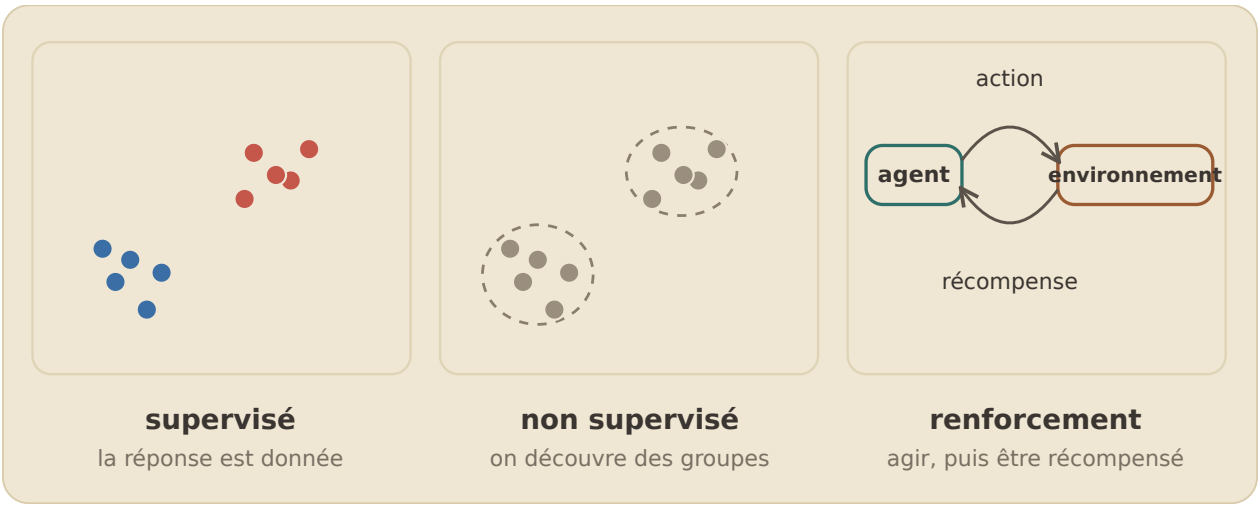
C'est bien de l'apprentissage, mais d'un autre type. Ces situations se distinguent par la nature du **signal** à partir duquel le modèle apprend : une réponse fournie, une structure à découvrir sans guide ou une récompense différée. Le chapitre suivant, « Trois façons d'apprendre », présente cette typologie et les nouvelles possibilités qu'elle ouvre.

Trois façons d'apprendre

Le chapitre « Bien évaluer un modèle » s'est terminé sur un constat. Tout ce que nous avons construit dans ce module (régression, classification, la droite, Bayes) appartient à une seule famille, l'**apprentissage supervisé**, dans laquelle chaque exemple est accompagné de sa bonne réponse. Nous avons aussi vu que ce n'est pas la seule façon d'apprendre.

Les grandes familles de l'apprentissage automatique se distinguent par la nature du **signal** que le modèle utilise, c'est-à-dire ce qui, dans les données, lui sert de guide. Il en existe trois grands types :

- une **réponse fournie** pour chaque exemple, ce qui correspond à l'apprentissage **supervisé**, étudié depuis le début du module ;
- **aucune réponse**, seulement des données brutes dans lesquelles il faut trouver une structure, ce qui correspond à l'apprentissage **non supervisé** (*unsupervised learning*) ;
- ni réponse ni structure donnée, mais une **récompense** qui arrive après coup, au fil des actions, ce qui correspond à l'apprentissage par **renforcement** (*reinforcement learning*).



Les trois grandes familles, par la nature de leur signal : réponse donnée (supervisé), structure à découvrir (non supervisé), récompense au fil de l'action (renforcement).

Nous avons passé tout le module dans la première famille. Ce dernier chapitre présente les deux autres. Le but n'est pas de les maîtriser, mais de situer ce que nous avons appris dans un ensemble plus large et de préparer les modules suivants.

Apprendre avec un professeur : le supervisé

Commençons par ce que nous connaissons déjà. En apprentissage supervisé, le signal est une **réponse fournie** avec chaque exemple, par exemple le prix d'une maison, l'étiquette *pourriel* d'un courriel ou la couleur d'un point. Le modèle apprend à relier l'entrée à cette réponse. Une fois entraîné, il applique ce qu'il a appris à de nouveaux cas. On peut le comparer à un professeur qui corrige : il connaît la bonne réponse, et l'élève progresse en comparant sa réponse à celle-ci.

Nous en avons vu les deux formes : la **régression**, quand la réponse est un nombre (un prix), et la **classification**, quand c'est une catégorie (pourriel ou non). Malgré leurs différences, les deux utilisent le même signal (une cible fournie d'avance) et le même mécanisme, qui consiste à régler des paramètres pour minimiser l'écart à cette cible.

Cette dépendance à des réponses fournies est à la fois la principale force du supervisé et sa principale limite. En effet, il faut que quelqu'un fournisse ces étiquettes, souvent à la main, un exemple à la fois.

Le socle humain du supervisé
Nous avons vu, dans *Bien évaluer un modèle*, que l'étiquetage des données est devenu une industrie à part entière. Cette industrie a aussi des aspects difficiles. Décider des milliers de fois par jour qu'un message est haineux et qu'un autre ne l'est pas, c'est aussi être exposé aux contenus les plus violents du Web, et la modération de contenus est l'un des métiers les plus éprouvants de cette chaîne. Les assistants les plus récents en dépendent aussi : une partie de l'entraînement de ChatGPT repose sur des humains qui notent et corrigent ses réponses (nous y reviendrons au Module 4). L'« intelligence » de ces systèmes repose donc en partie sur un travail humain, ce qui soulève des questions que nous retrouverons au Module 5.

La section suivante traite du cas où personne n'a fourni ces réponses.

Apprendre sans réponses : le non-supervisé

Souvent, les données n'ont pas d'étiquettes. On dispose d'un grand ensemble de données brutes (des clients, des photos, des textes), sans aucune « bonne réponse » à imiter. On peut quand même en apprendre quelque chose, mais le but est différent. Il ne s'agit plus de prédire une réponse donnée, mais de **découvrir une structure** présente dans les données elles-mêmes. C'est l'apprentissage **non supervisé**.

La tâche la plus courante est le **regroupement** (ou *clustering*), qui consiste à rassembler les exemples qui se ressemblent. Nous savons déjà ce que « se ressembler » veut dire : c'est être **proches** dans l'espace des caractéristiques, comme l'a expliqué *Prédire par ressemblance*. Regrouper consiste donc à repérer les amas naturels de points, c'est-à-dire les zones denses séparées par des zones vides.

L'algorithme classique pour cette tâche s'appelle **k-means**. On peut l'expliquer par une comparaison. Pour organiser une fête, on veut répartir les invités autour de k tables, en plaçant chaque table au centre de son groupe, pour que chacun soit le plus près possible de la sienne. k-means procède de cette façon, par ajustements successifs :

1. placer k « centres » au hasard ;
2. rattacher chaque point au centre le plus proche (ce qui donne des groupes provisoires) ;
3. déplacer chaque centre au milieu de son groupe ;

4. répéter les étapes 2 et 3 jusqu'à ce que plus rien ne change.

Dans l'applet ci-dessous, choisissez le nombre de groupes et observez les centres se déplacer, étape par étape, vers le centre des amas.

Applet interactive : à utiliser en ligne, sur la page « Trois façons d'apprendre » du site du cours.

Ce résultat ressemble à de la classification, puisque les points sont répartis en groupes de couleurs. La différence est cependant importante : ici, **les points n'avaient aucune étiquette au départ.** Vous n'avez pas indiqué à l'algorithme ce que représente chaque groupe. Vous lui avez seulement donné le nombre de groupes, et il a déterminé le reste. Il n'y a pas de « bonne réponse » à retrouver, seulement une structure à mettre en évidence.

Ce type de méthode est très répandu. On l'utilise pour segmenter une clientèle en profils types pour le marketing, pour repérer une transaction anormale parmi des millions d'autres (détection de fraude) ou pour compresser des données en les résumant par leurs groupes.

Le regroupement n'est qu'une partie du non-supervisé. Celui-ci comprend aussi la **réduction de dimension** (simplifier des données à mille variables en perdant le moins d'information possible) et l'**apprentissage de représentations** (*representation learning*), qui consiste à découvrir sans étiquettes de bonnes caractéristiques pour décrire les données. Cette idée, qui consiste à laisser la machine construire ses propres descripteurs, joue un rôle central dans l'IA moderne. Nous la retrouverons avec les **autoencodeurs** (Module 3) et les **plongements** (*embeddings*) de mots (Module 4).

Les mathématiques de k-means (optionnel)
k-means cherche à minimiser une fonction d'erreur, l'**inertie**, qui est la somme des carrés des distances de chaque point à son centre :

$$J = \sum_{i=1}^{n} \min_{j=1}^{k} \|\mathbf{x}_i - \mu_j\|^2$$

où $\mathbf{x}_i$ est le i-ème point et μ_j le centre du groupe j. L'algorithme alterne deux étapes :

- **assignation** : chaque point est rattaché au centre le plus proche, $c_i = \arg\min_j \|\mathbf{x}_i - \mu_j\|^2$;
- **mise à jour** : chaque centre est placé à la moyenne de ses points, $\boldsymbol{\mu}j = \frac{1}{n_j} \sum{i : c_i = j} \mathbf{x}_i$.

On répète ces étapes jusqu'à convergence. Deux remarques s'imposent. Le nombre de groupes k est un **hyper-paramètre**, qu'on choisit d'avance comme le k de kNN. De plus, l'algorithme peut s'arrêter dans un minimum local. C'est pourquoi on le relance habituellement plusieurs fois avec des centres initiaux différents, et on garde la meilleure solution.

Fabriquer soi-même ses réponses : l'auto-supervision

Entre le supervisé, qui exige des étiquettes, et le non-supervisé, qui n'en utilise pas, il existe une approche qui a beaucoup transformé le domaine : **fabriquer les étiquettes à partir des données elles-mêmes**. On peut prendre un texte, en cacher un mot et demander au modèle de le deviner. La « bonne réponse » se trouve dans le texte, sans qu'aucun humain ait eu à étiqueter quoi que ce soit. On peut aussi cacher une partie d'une photo et demander au modèle de la reconstituer, ou faire pivoter une image et lui demander de quel angle. Dans chaque cas, le problème a la forme d'un problème supervisé, avec une entrée et une réponse attendue, mais la réponse est tirée de la donnée brute. C'est l'apprentissage **auto-supervisé** (*self-supervised learning*).

Cette approche semble simple, mais elle a une portée considérable, parce qu'elle supprime le besoin d'étiquetage. Le Web contient des milliers de milliards de mots disponibles, qui ne demandent aucune heure de travail humain : chaque phrase fournit un exercice avec son corrigé. C'est de cette façon, en apprenant à prédire le mot suivant, que les grands modèles de langage du Module 4 sont entraînés, et c'est pour cette raison qu'ils ont pu traiter une grande partie de ce que l'humanité a écrit. L'apprentissage supervisé intervient ensuite, sur des quantités beaucoup plus petites, avec les réponses notées par des humains dont parlait l'encadré plus haut. Cependant, l'essentiel de leurs connaissances provient de cette tâche de prédiction sur les textes eux-mêmes.

Apprendre par l'expérience : le renforcement

La troisième situation est très différente des deux premières. Il n'y a pas d'étiquettes, et il n'y a pas non plus seulement un ensemble de données à structurer. Le modèle, qu'on appelle ici un **agent**, doit agir, et il apprend à partir des **conséquences** de ses actions. On peut penser à un enfant qui apprend à faire du vélo, à un robot qui apprend à marcher ou à un joueur qui découvre un jeu. Personne ne leur indique le bon mouvement à chaque instant. Ils essaient, échouent, recommencent et retiennent ce qui fonctionne.

Le signal est ici une **récompense** (*reward*), par exemple un point gagné, une partie remportée ou une chute évitée. Cette récompense arrive souvent **bien après** les actions qui l'ont produite. Quand on gagne une partie d'échecs, il est difficile de savoir quel coup a été décisif. Ce décalage est la principale difficulté du renforcement, parce qu'il faut apprendre à relier une récompense tardive aux actions qui l'ont rendue possible.

La stratégie utilisée est celle de l'**essai-erreur**. L'agent explore, reçoit des récompenses et des punitions, et **renforce** peu à peu les actions qui mènent au succès, d'où le nom de cette famille. Dans l'applet ci-dessous, un agent (le point jaune) cherche la sortie (**+1**) d'une petite grille en évitant un piège (**−1**), sans aucune connaissance au départ. Lancez l'entraînement.

Applet interactive : à utiliser en ligne, sur la page « Trois façons d'apprendre » du site du cours.

Au début, l'agent se déplace au hasard. Puis, d'un épisode à l'autre, une **carte de valeur** se forme (les cases se colorent selon leur valeur estimée) et une **politique** (*policy*) apparaît. Les flèches, qui indiquent pour chaque case le meilleur mouvement, finissent par former un chemin sûr vers le but. Le curseur d'**exploration** (ε) est instructif. À zéro, l'agent ne fait qu'exploiter ce qu'il croit déjà savoir, et il peut rester dans un chemin médiocre. Un peu de hasard l'amène à explorer d'autres chemins, et parfois à en trouver de meilleurs. Ce compromis entre **explorer et exploiter** (*exploration vs. exploitation*) est au centre du renforcement.

Cette approche rappelle une notion déjà vue. Au Module 1, nous avons étudié les machines qui jouent aux échecs en cherchant dans l'arbre des coups possibles. Le renforcement reprend cette recherche, mais la machine apprend elle-même à évaluer les positions au lieu de tout calculer. C'est cette combinaison de la recherche du GOFAI et de l'apprentissage qui a permis à **AlphaGo** de battre les meilleurs joueurs de go humains, alors que la force brute seule n'y parvenait pas.

Le renforcement est aussi la famille qui demande le plus de calcul, et il s'est surtout développé lorsqu'on l'a combiné aux **réseaux de neurones** (le *deep reinforcement learning*, Module 3). On le retrouve aussi dans les assistants modernes. Le comportement de ChatGPT est en partie ajusté par renforcement, à partir des préférences d'évaluateurs humains (le **RLHF**). Le renforcement sert également, avec des récompenses vérifiables (un problème de mathématiques a une bonne réponse,

un programme réussit ou non ses tests, ce qu'on appelle le **RLVR**), à entraîner les modèles les plus récents à « raisonner » longuement avant de répondre. Nous verrons ces méthodes au Module 4.

Un même squelette, d'un bout à l'autre

Les trois familles répondent de trois façons à une même question, celle de savoir à partir de quoi le modèle apprend : une réponse fournie, une structure à découvrir ou une récompense à obtenir. Ces trois familles partagent cependant la même structure de base, celle qui a servi tout au long de ce module.

Cette structure comporte plusieurs étapes. On part de **données**, qu'on décrit par des nombres, donc par des points dans un espace. On choisit un **modèle**, c'est-à-dire une fonction de prédiction réglée par des paramètres. On mesure son **erreur** de prédiction. On règle ensuite les paramètres du modèle pour **minimiser** cette erreur, le plus souvent par descente de gradient. On vérifie enfin que le modèle **généralise** au-delà des exemples appris. Toutes les méthodes vues dans ce module sont des variantes de ce schéma : le modèle simple et son unique nombre, kNN et ses zéro paramètre, la droite et ses deux paramètres, l'arbre et ses questions, la régression logistique et Bayes qui classent, k-means qui regroupe sans étiquettes et l'agent qui apprend à partir d'une récompense. La démarche est la même, mais elle s'applique à des problèmes différents.

C'est ce que ce module voulait montrer. Pour une machine, « apprendre » n'a rien de magique : c'est **ajuster des réglages pour réduire une erreur sur des exemples**. La question posée dès la première page, qui était de savoir en quoi il s'agit d'intelligence, reste ouverte. Vous savez cependant maintenant, concrètement, comment ces systèmes fonctionnent.

Il reste à revenir sur le lien avec la statistique annoncé au début du module. Un statisticien reconnaîtrait tout ce que nous avons fait ici : ajuster une droite, estimer une probabilité, tirer un échantillon, se méfier d'un score. Ce qui a changé, ce n'est pas la base, mais l'accent et l'échelle. L'accent est mis sur la prédiction d'abord, l'explication ensuite, et parfois jamais. L'échelle se compte en millions d'exemples et en milliards de paramètres, avec des modèles que personne ne lit plus. C'est à cette échelle que la statistique a pris le nom d'apprentissage automatique, et qu'elle a produit des résultats que ses fondateurs n'avaient pas prévus.

L'étape suivante est ce changement d'échelle. Richard Sutton, un chercheur qui a consacré sa carrière à l'apprentissage par renforcement, a tiré de soixante-dix ans d'histoire de l'IA une leçon qu'il qualifie d'amère, « The Bitter Lesson » (2019). Selon lui, à long terme, les méthodes générales qui exploitent la puissance de calcul finissent toujours par l'emporter sur celles où l'on avait inscrit à la main le savoir humain, et seules deux méthodes passent vraiment à l'échelle, la **recherche**, celle du Module 1, et l'**apprentissage**, celui de ce module. Le sens et le coût de cette leçon sont examinés au **Module 3**. Nous y combinerons ces fonctions réglables en **réseaux de neurones** profonds, et nous verrons pourquoi ces réseaux ont permis de grands progrès dans le traitement de l'image et du langage. Au **Module 4**, ces mêmes réseaux deviendront **génératifs**, capables de produire des textes et des images : ce sont les grands modèles de langage. La structure de base restera la même, mais à une échelle beaucoup plus grande.

Classification naïve bayésienne pour détecter les pourriels (travail noté 2)

La classification naïve bayésienne est un algorithme d'apprentissage supervisé qui repose sur les probabilités. Nous avons vu deux variantes de cet algorithme :

1. La classification de simples points en 2d avec un modèle gaussien
2. La classification de vecteurs en haute dimension avec un modèle multinomial

La classification de courriels est un problème classique qu'on peut traiter avec cet algorithme. On cherche à estimer la probabilité qu'un courriel soit un pourriel (*spam*) à partir des mots qu'il contient, parce que certains mots sont plus souvent utilisés dans les pourriels et d'autres dans les courriels.

Comme nous l'avons vu au chapitre « Classer », la classification naïve bayésienne est un algorithme d'apprentissage *génératif*. On considère donc d'abord deux modèles (un pour chaque classe, `pourriel` ou `courriel`), dont le rôle est de générer les données observées plutôt que de les classifier directement :

$$P(\text{les mots générés} \mid \text{il s'agit d'un pourriel})$$

$$P(\text{les mots générés} \mid \text{il s'agit d'un courriel})$$

ou, de manière plus compacte :

$$P(\mathbf{x} \mid \texttt{pourriel})$$

$$P(\mathbf{x} \mid \texttt{courriel})$$

Comme ce qui nous intéresse ici est de classifier les courriels, nous utilisons le théorème de Bayes pour « inverser » les modèles et obtenir une règle de classification simple, qui prédit la classe plutôt que les mots :

$$\text{classification}(\mathbf{x}) = \begin{cases} \texttt{pourriel} \text{ si } P(\mathbf{x} \mid \texttt{pourriel})P(\texttt{pourriel}) \geq P(\mathbf{x} \mid \texttt{courriel})P(\texttt{courriel}) \\ \texttt{courriel} \text{ sinon} \end{cases}$$

Consignes

1. Suivez les instructions ci-dessous pour construire d'abord le fichier Google Sheets avec toutes les données nécessaires.
2. Une fois le fichier complété et fonctionnel, partagez votre fichier et copiez le lien vers celui-ci dans un document PDF (**Attention : aucun autre format que PDF ne sera accepté**).
3. Répondez aux questions d'interprétation de la dernière section dans le même fichier PDF, en fournissant des réponses claires et précises.

Entraînement du modèle

Cette section montre comment calculer ces probabilités en entraînant un modèle de classification sur une série de courriels.

Comme nous allons utiliser le tableur en ligne Google Sheets, assurez-vous d'abord qu'il est correctement configuré.

Copiez d'abord ces 10 mini-courriels dans la colonne A d'une nouvelle « feuille » Google Sheets, un courriel par rangée (si vous utilisez le copier-coller, assurez-vous de bien appliquer la fonction « copier-coller ») :

```
voici le colis est arrivé
bonjour voici le lien
offre spéciale colis gratuit
merci pour votre colis
colis livré demain matin
voici votre carte gratuite
réunion demain à midi
voici le code pour carte
livraison spéciale pour vous
merci encore pour votre carte
```

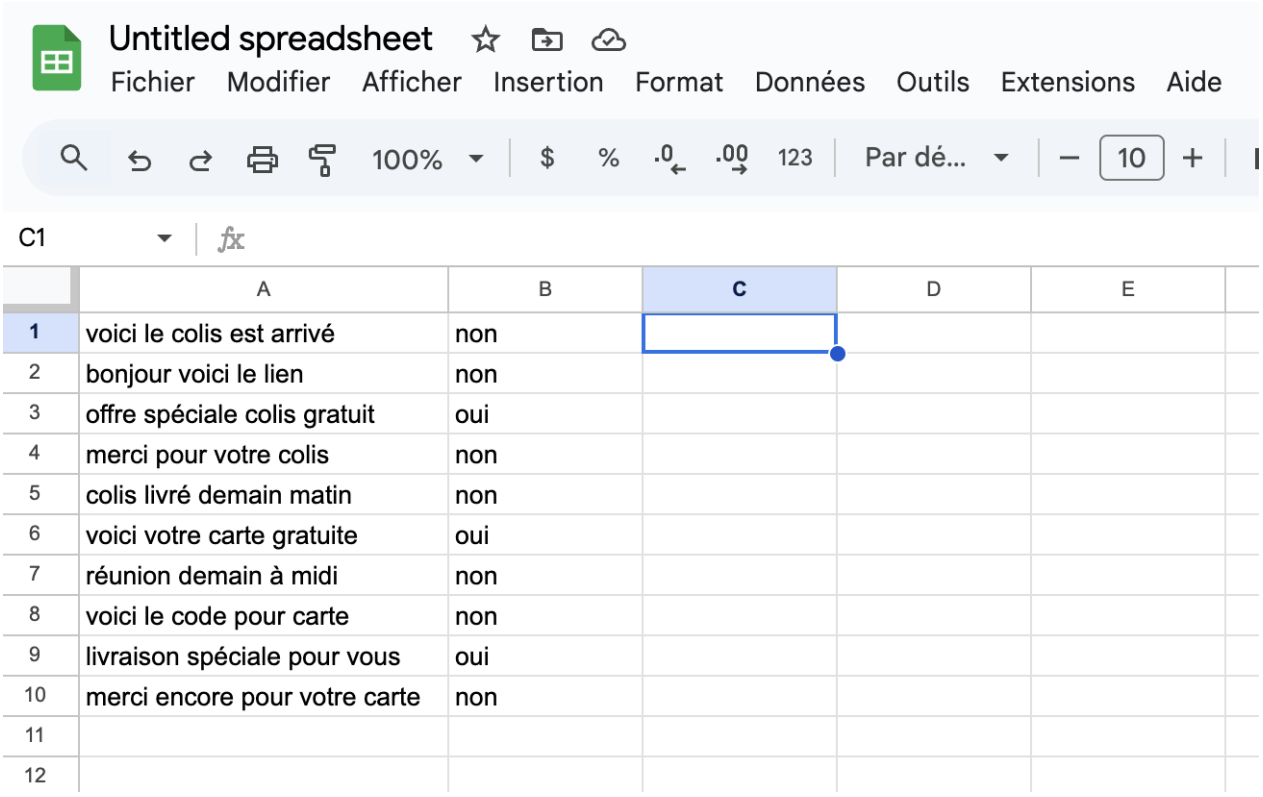
Pour avoir un aperçu de la tâche d'étiquetage des données (qui, dans un scénario réel, peut être très coûteuse et laborieuse), nous vous proposons d'abord de catégoriser vous-mêmes les courriels dans la colonne `B`, en utilisant la valeur `oui` si vous considérez qu'il s'agit d'un pourriel, ou `non` (ce n'est pas un pourriel) sinon.

Si vous préférez ne pas faire cet exercice maintenant, vous pouvez copier ces valeurs (dans la colonne `B`) :

```
non
non
oui
non
non
oui
non
non
oui
```

`non`

À ce stade, votre feuille devrait ressembler à ceci :



	A	B	C	D	E
1	voici le colis est arrivé	non			
2	bonjour voici le lien	non			
3	offre spéciale colis gratuit	oui			
4	merci pour votre colis	non			
5	colis livré demain matin	non			
6	voici votre carte gratuite	oui			
7	réunion demain à midi	non			
8	voici le code pour carte	non			
9	livraison spéciale pour vous	oui			
10	merci encore pour votre carte	non			
11					
12					

Calculez d'abord, dans la colonne `C`, la probabilité *a priori* qu'un courriel quelconque ne soit pas un pourriel (`C1`) ou qu'il en soit un (`C2`), sans tenir compte des mots pour le moment. Dans la cellule `C1` :

```
=COUNTIF(B1:B10, "non") / COUNTA(B1:B10)
```

et dans la cellule `C2` :

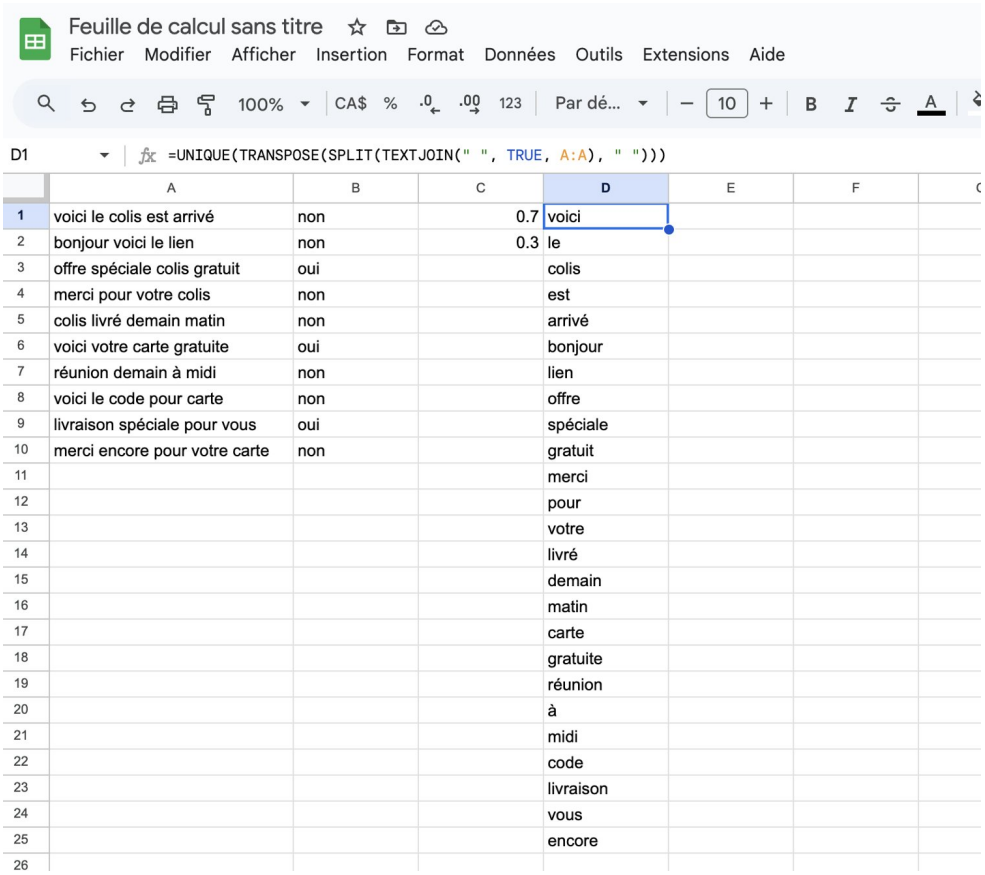
```
=COUNTIF(B1:B10, "oui") / COUNTA(B1:B10)
```

> Si vous obtenez une erreur avec la formule à ce stade, il est très possible que les paramètres linguistiques de votre Google Sheets ne soient pas correctement configurés.

Ces probabilités *a priori* serviront plus loin. Définissez ensuite la colonne `D` avec cette formule :

```
=UNIQUE(TRANSPOSE(SPLIT(TEXTJOIN(" ", TRUE, A:A), " ")))
```

La colonne `D` devrait maintenant contenir le vocabulaire des courriels :



	A	B	C	D	E	F
1	voici le colis est arrivé	non	0.7	voici		
2	bonjour voici le lien	non	0.3	le		
3	offre spéciale colis gratuit	oui		colis		
4	merci pour votre colis	non		est		
5	colis livré demain matin	non		arrivé		
6	voici votre carte gratuite	oui		bonjour		
7	réunion demain à midi	non		lien		
8	voici le code pour carte	non		offre		
9	livraison spéciale pour vous	oui		spéciale		
10	merci encore pour votre carte	non		gratuit		
11				merci		
12				pour		
13				votre		
14				livré		
15				demain		
16				matin		
17				carte		
18				gratuite		
19				réunion		
20				à		
21				midi		
22				code		
23				livraison		
24				vous		
25				encore		
26						

La colonne `E` doit ensuite contenir le nombre de courriels valides (ceux marqués `non`, qui ne sont pas des pourriels) dans lesquels apparaît chaque mot de la colonne `D` :

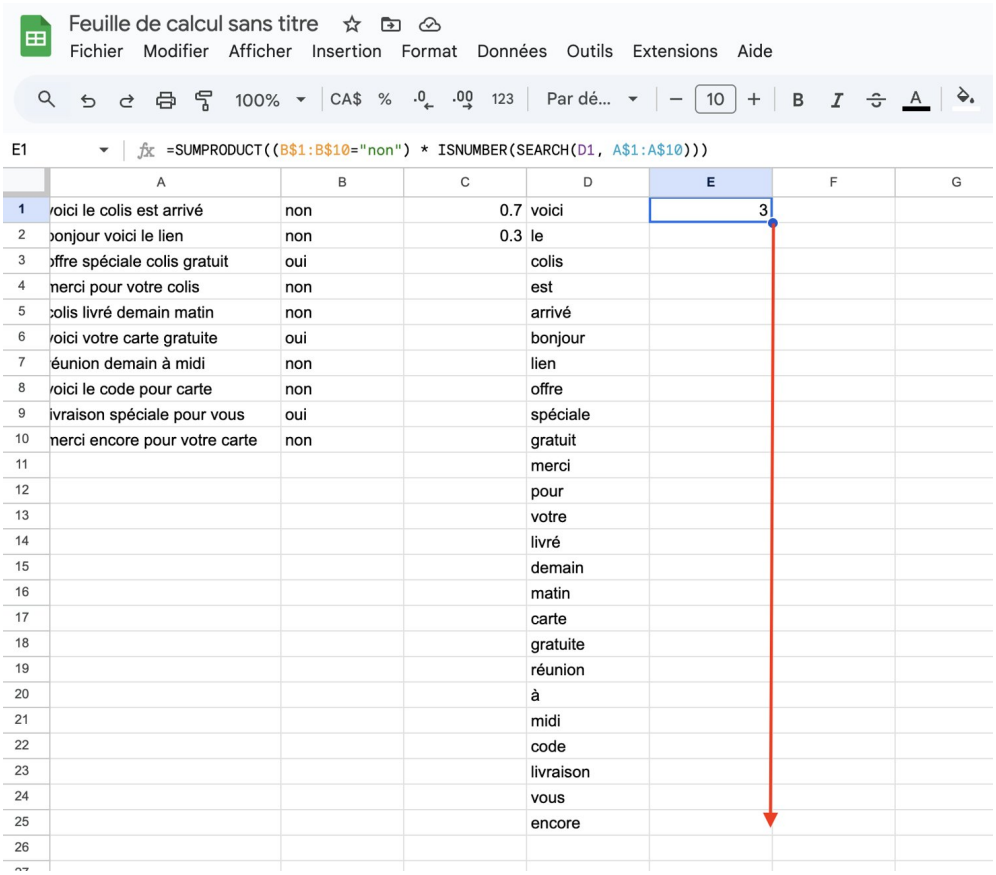
```
=SUMPRODUCT((B$1:B$10="non") * ISNUMBER(SEARCH(" " & D1 & " ", " " & A$1:A$10 & " ")))
```

La formule entoure le mot et chaque courriel d'espaces avant de chercher l'un dans l'autre. Sans cette précaution, le mot « le » serait trouvé à l'intérieur de « spéciale », et « gratuit » à l'intérieur de « gratuite ».

De la même manière, la colonne `F` contient, pour chaque mot, le nombre de courriels marqués `oui`, qui sont des pourriels, dans lesquels il apparaît :

```
=SUMPRODUCT((B$1:B$10="oui") * ISNUMBER(SEARCH(" " & D1 & " ", " " & A$1:A$10 & " ")))
```

> Notez que les colonnes `E` et `F` doivent avoir le même nombre d'éléments que la colonne `D`. Il faut donc utiliser la fonction de remplissage automatique. Le plus simple est de glisser (*drag*) la première cellule vers le bas, une fois qu'elle a été calculée, ou de double-cliquer sur le petit carré bleu qui apparaît en bas à droite de la cellule sélectionnée. Google Sheets propose parfois de lui-même ce remplissage (« Remplissage automatique suggéré ») : vous pouvez l'accepter.



	A	B	C	D	E	F	G
1	voici le colis est arrivé	non	0.7	voici	3		
2	bonjour voici le lien	non	0.3	le			
3	offre spéciale colis gratuit	oui		colis			
4	merci pour votre colis	non		est			
5	colis livré demain matin	non		arrivé			
6	voici votre carte gratuite	oui		bonjour			
7	réunion demain à midi	non		lien			
8	voici le code pour carte	non		offre			
9	livraison spéciale pour vous	oui		spéciale			
10	merci encore pour votre carte	non		gratuit			
11				merci			
12				pour			
13				votre			
14				livré			
15				demain			
16				matin			
17				carte			
18				gratuite			
19				réunion			
20				à			
21				midi			
22				code			
23				livraison			
24				vous			
25				encore			
26							

Untitled spreadsheet

File Edit View Insert Format Data Tools Gemini Extensions Help

F1 =SUMPRODUCT((B$1:B$10="oui") * ISNUMBER(SEARCH(" " & D1 & " ", " " & A$1:A$10 & " ")))

	A	B	C	D	E	F	G	H	
1	voici le colis est	non	0.7	voici	3	1	0.07142857143	0.05405405405	voici
2	bonjour voici le li	non	0.3	le	3	0	0.07142857143	0.02702702703	
3	offre spéciale co	oui		colis	3	1	0.07142857143	0.05405405405	
4	merci pour votre	non		est	1	0	0.03571428571	0.02702702703	
5	colis livré demai	non		arrivé	1	0	0.03571428571	0.02702702703	
6	voici votre carte	oui		bonjour	1	0	0.03571428571	0.02702702703	
7	réunion demain	non		lien	1	0	0.03571428571	0.02702702703	
8	voici le code pou	non		offre	0	1	0.01785714286	0.05405405405	
9	livraison spéciale	oui		spéciale	0	2	0.01785714286	0.08108108108	
10	merci encore po	non		gratuit	0	1	0.01785714286	0.05405405405	
11				merci	2	0	0.05357142857	0.02702702703	
12				pour	3	1	0.07142857143	0.05405405405	
13				votre	2	1	0.05357142857	0.05405405405	
14				livré	1	0	0.03571428571	0.02702702703	
15				demain	2	0	0.05357142857	0.02702702703	
16				matin	1	0	0.03571428571	0.02702702703	
17				carte	2	1	0.05357142857	0.05405405405	
18				gratuite	0	1	0.01785714286	0.05405405405	
19				réunion	1	0	0.03571428571	0.02702702703	
20				à	1	0	0.03571428571	0.02702702703	
21				midi	1	0	0.03571428571	0.02702702703	
22				code	1	0	0.03571428571	0.02702702703	
23				livraison	0	1	0.01785714286	0.05405405405	
24				vous	0	1	0.01785714286	0.05405405405	
25				encore	1	0	0.03571428571	0.02702702703	

À partir de ces fréquences de mots pour chaque classe (`oui` ou `non`), on peut maintenant calculer la probabilité conditionnelle de chaque mot du vocabulaire, selon que le courriel est un pourriel (`oui`) ou non (`non`). La colonne `G` correspond à la probabilité des mots sachant que le courriel n'est pas un pourriel (`non`) :

```
=(E1 + 1) / (SUM(E:E) + COUNTA(D:D))
```

De la même manière, la colonne `H` contient la probabilité des mots sachant que le courriel est un pourriel (`oui`) :

```
=(F1 + 1) / (SUM(F:F) + COUNTA(D:D))
```

Les colonnes `G` et `H` doivent elles aussi avoir la même taille que le vocabulaire (colonne `D`). Il faut donc utiliser le remplissage automatique décrit plus haut.

Untitled spreadsheet

File Edit View Insert Format Data Tools Gemini Extensions Help

H1 =(F1 + 1) / (SUM(F:F) + COUNTA(D:D))

	A	B	C	D	E	F	G	H	
1	voici le colis est	non	0.7	voici	3	1	0.07142857143	0.05405405405	voici
2	bonjour voici le li	non	0.3	le	3	0	0.07142857143	0.02702702703	
3	offre spéciale co	oui		colis	3	1	0.07142857143	0.05405405405	
4	merci pour votre	non		est	1	0	0.03571428571	0.02702702703	
5	colis livré demai	non		arrivé	1	0	0.03571428571	0.02702702703	
6	voici votre carte	oui		bonjour	1	0	0.03571428571	0.02702702703	
7	réunion demain	non		lien	1	0	0.03571428571	0.02702702703	
8	voici le code pou	non		offre	0	1	0.01785714286	0.05405405405	
9	livraison spéciale	oui		spéciale	0	2	0.01785714286	0.08108108108	
10	merci encore po	non		gratuit	0	1	0.01785714286	0.05405405405	
11				merci	2	0	0.05357142857	0.02702702703	
12				pour	3	1	0.07142857143	0.05405405405	
13				votre	2	1	0.05357142857	0.05405405405	
14				livré	1	0	0.03571428571	0.02702702703	
15				demain	2	0	0.05357142857	0.02702702703	
16				matin	1	0	0.03571428571	0.02702702703	
17				carte	2	1	0.05357142857	0.05405405405	
18				gratuite	0	1	0.01785714286	0.05405405405	
19				réunion	1	0	0.03571428571	0.02702702703	
20				à	1	0	0.03571428571	0.02702702703	
21				midi	1	0	0.03571428571	0.02702702703	
22				code	1	0	0.03571428571	0.02702702703	
23				livraison	0	1	0.01785714286	0.05405405405	
24				vous	0	1	0.01785714286	0.05405405405	
25				encore	1	0	0.03571428571	0.02702702703	

Le modèle est maintenant entièrement entraîné et prêt à être utilisé.

Utilisation du modèle (inférence)

Nous allons maintenant utiliser le modèle pour déterminer si un nouveau courriel (qui n'a pas servi à l'entraînement) est un pourriel ou non. Dans la colonne `I`, entrez un courriel à tester :

```
voici votre carte spéciale
```

Extrayez les mots du courriel dans la colonne `J` :

```
=TRANSPOSE(SPLIT(I1, " "))
```

Il faut ensuite, dans la colonne `K`, la probabilité des mots de ce courriel de test dans l'hypothèse où ce n'est pas un pourriel (`non`) :

```
=IFERROR(XLOOKUP(J1, D:D, G:G), 1E-5)
```

De la même manière, la colonne `L` contient la probabilité des mots du courriel dans l'hypothèse où il s'agit d'un pourriel (`oui`) :

```
=IFERROR(XLOOKUP(J1, D:D, H:H), 1E-5)
```

Les colonnes `K` et `L` doivent avoir la même taille que la colonne `J`. Utilisez donc le remplissage automatique. Calculez ensuite, dans la colonne `M`, la probabilité que le courriel ne soit pas un

pourriel (`non`) :

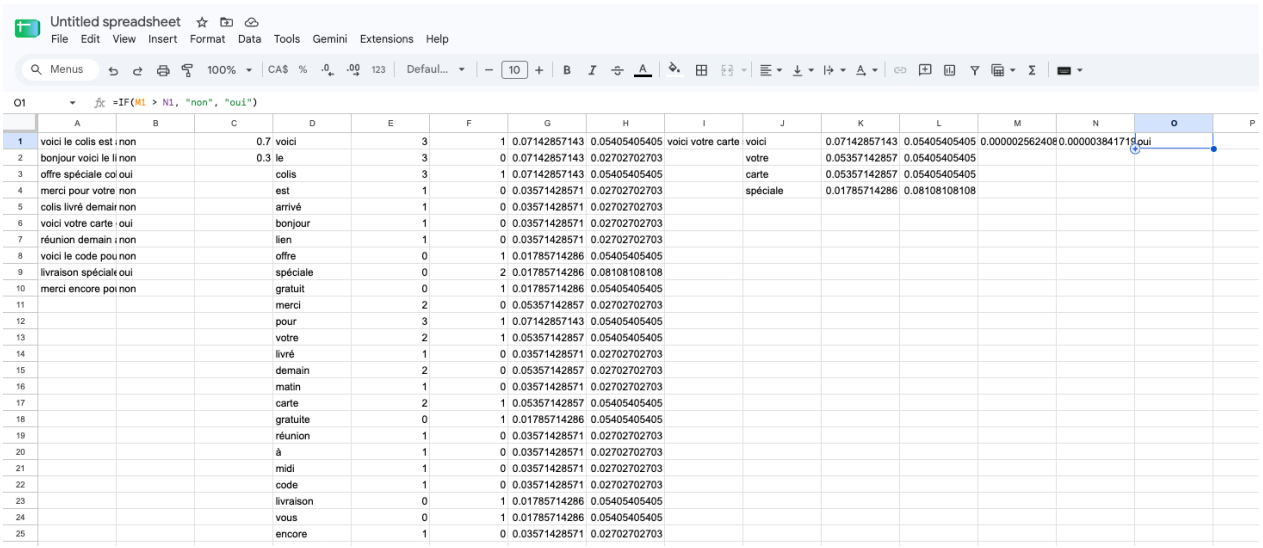
`=PRODUCT(K:K) * C1`

Dans la colonne `N`, calculez la probabilité que le courriel soit un pourriel (`oui`) :

`=PRODUCT(L:L) * C2`

La classification finale se trouve dans la colonne `O` :

`=IF(M1 > N1, "non", "oui")`



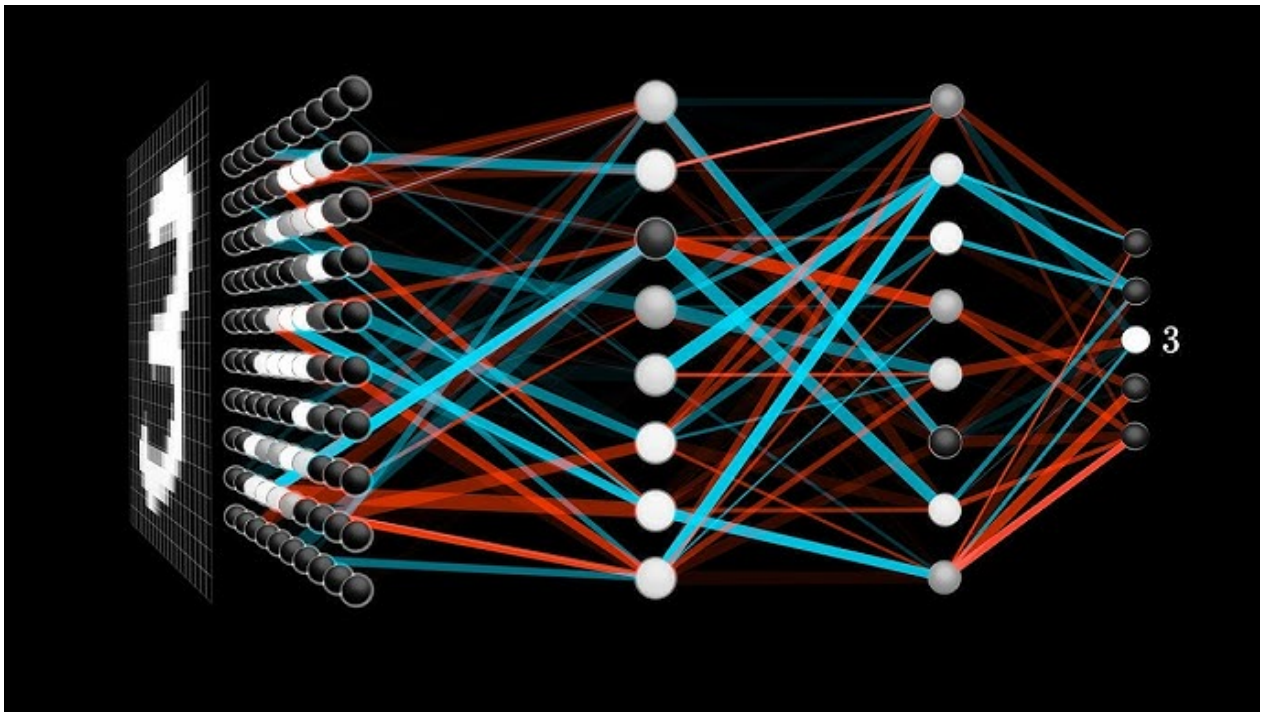
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	voici le colis est	non	0.7	voici	3	1	0.07142857143	0.05405405405	voici votre carte	voici	0.07142857143	0.05405405405	0.000002562408	0.000003841719	oui
2	bonjour voici le li	non	0.3	le	3	0	0.07142857143	0.02702702703		votre	0.05357142857	0.05405405405			
3	offre spéciale coi	oui		colis	3	1	0.07142857143	0.05405405405		carte	0.05357142857	0.05405405405			
4	merci pour votre	non		est	1	0	0.03571428571	0.02702702703		spéciale	0.01785714286	0.08108108108			
5	colis livré demair	non		arrivé	1	0	0.03571428571	0.02702702703							
6	voici votre carte	oui		bonjour	1	0	0.03571428571	0.02702702703							
7	réunion demain	non		lien	1	0	0.03571428571	0.02702702703							
8	voici le code pou	non		offre	0	1	0.01785714286	0.05405405405							
9	livraison spéciale	oui		spéciale	0	2	0.01785714286	0.08108108108							
10	merci encore poi	non		gratuit	0	1	0.01785714286	0.05405405405							
11				merci	2	0	0.05357142857	0.02702702703							
12				pour	3	1	0.07142857143	0.05405405405							
13				votre	2	1	0.05357142857	0.05405405405							
14				livré	1	0	0.03571428571	0.02702702703							
15				demain	2	0	0.05357142857	0.02702702703							
16				matin	1	0	0.03571428571	0.02702702703							
17				carte	2	1	0.05357142857	0.05405405405							
18				gratuite	0	1	0.01785714286	0.05405405405							
19				réunion	1	0	0.03571428571	0.02702702703							
20				à	1	0	0.03571428571	0.02702702703							
21				midi	1	0	0.03571428571	0.02702702703							
22				code	1	0	0.03571428571	0.02702702703							
23				livraison	0	1	0.01785714286	0.05405405405							
24				vous	0	1	0.01785714286	0.05405405405							
25				encore	1	0	0.03571428571	0.02702702703							

Questions d'interprétation

1. Que se passe-t-il si vous remplacez le mot « spéciale » par le mot « livrée » dans le courriel de test de la cellule `I1` ?
2. Après ce changement, expliquez les probabilités qu'on trouve aux cellules `K4` et `L4` associées au nouveau mot « livrée ». D'où proviennent ces nouvelles valeurs, et pourquoi a-t-on besoin d'y avoir recours dans le calcul ?
3. Est-ce que ce modèle est paramétrique ou non ? Expliquez pourquoi.
4. S'il s'agit d'un modèle paramétrique, quels sont les paramètres du modèle (quelles colonnes) ?
5. Quelles colonnes constituent la partie *générative* du modèle ? Expliquez pourquoi.
6. Quelles colonnes constituent la partie *discriminative* du modèle ? Expliquez pourquoi.
7. Quelle est la signification des nombres dans les cellules `G11` et `H11`, associées au mot « merci » ? Comment peut-on les interpréter ?
8. Quelles sont les probabilités non conditionnelles (*a priori*) ? À quoi servent-elles ?
9. Est-ce qu'il serait possible d'utiliser seulement ces probabilités non conditionnelles pour faire un modèle de classification ? Quelles conséquences cela entraînerait-il ?
10. De quelle manière peut-on dire que ce modèle généralise ?

11. Est-ce que l'ordre des mots joue un rôle dans les décisions de ce modèle ? Expliquez pourquoi.
12. Si l'ordre des mots ne joue pas de rôle, comment pourrait-on modifier le modèle pour qu'il en joue un ?
13. Est-ce que certains mots aident particulièrement le modèle ? Si oui, pourquoi ?
14. Est-ce que certains mots sont moins utiles ? Si oui, pourquoi ?

Module 3 — Réseaux de neurones et apprentissage profond



Un réseau qui reconnaît un 3 manuscrit (image tirée de la série de vidéos de 3Blue1Brown sur les réseaux de neurones).

Le retour d'une idée ancienne

Le Module 1 a présenté deux paris sur la nature de l'intelligence, nés presque en même temps dans les années 1950. Le pari symbolique voit l'esprit comme de la logique, des symboles et des règles. Le pari connexionniste le voit comme un cerveau, un réseau qui s'ajuste à l'expérience. Le perceptron de Rosenblatt, en 1958, en était la première machine.

Ce second pari a connu une longue éclipse. En 1969, Minsky et Papert montrent qu'un perceptron ne peut pas apprendre une fonction aussi simple que le XOR, et la recherche sur les réseaux de neurones s'arrête presque. Elle reprend au début des années 1980, et surtout en 1986, avec une méthode pour entraîner des réseaux dotés d'une couche cachée, la rétropropagation. Mais pendant les vingt-cinq années suivantes, les progrès de l'apprentissage automatique viennent surtout d'autres méthodes : les machines à vecteurs de support (SVM) à partir du milieu des années 1990, le *boosting*, les forêts aléatoires au début des années 2000, et les modèles probabilistes comme le classifieur bayésien du Module 2. Ces méthodes s'appuient sur une théorie solide, s'entraînent de façon fiable et donnent d'excellents résultats. Les réseaux de neurones paraissent alors difficiles à régler et peu fiables, et beaucoup de chercheurs s'en détournent.

La situation change en 2012, quand un réseau profond dépasse de loin toutes les autres méthodes de reconnaissance d'images. En quelques années, les réseaux de neurones deviennent la méthode dominante pour les images, la parole et le texte. Ils sont aujourd'hui au cœur de la reconnaissance d'images et de la parole, de la traduction automatique, et des grands modèles de langage comme ChatGPT. Les autres méthodes n'ont pas disparu pour autant, et la différence tient à la nature des données. Dans une image, un son ou un texte, un élément isolé ne veut presque rien dire : un pixel n'est qu'un point de couleur, et le sens naît de l'agencement de milliers d'éléments voisins. Dans un tableau, comme les dossiers d'une banque ou d'un hôpital, chaque colonne a déjà un sens, comme

l'âge, le revenu ou la pression artérielle. On peut d'ailleurs mélanger l'ordre des colonnes d'un tableau sans rien perdre, alors que mélanger les pixels d'une photo la détruit. Sur les premières données, les réseaux de neurones l'emportent. Sur les secondes, les forêts aléatoires et le *gradient boosting* restent souvent les plus efficaces.

Un tableau

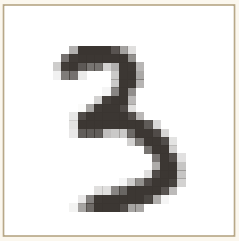
superficie (m²)	année	distance (km)	prix (k$)
130	1972	16	310
150	1980	21	350
180	1995	8	420
220	2010	9	580

mélanger les colonnes

prix (k$)	distance (km)	superficie (m²)	année
310	16	130	1972
350	21	150	1980
420	8	180	1995
580	9	220	2010

Rien n'est perdu : chaque colonne garde son nom et son sens.

Une image



mélanger les pixels

Le chiffre a disparu, alors que les 784 valeurs sont toujours là.

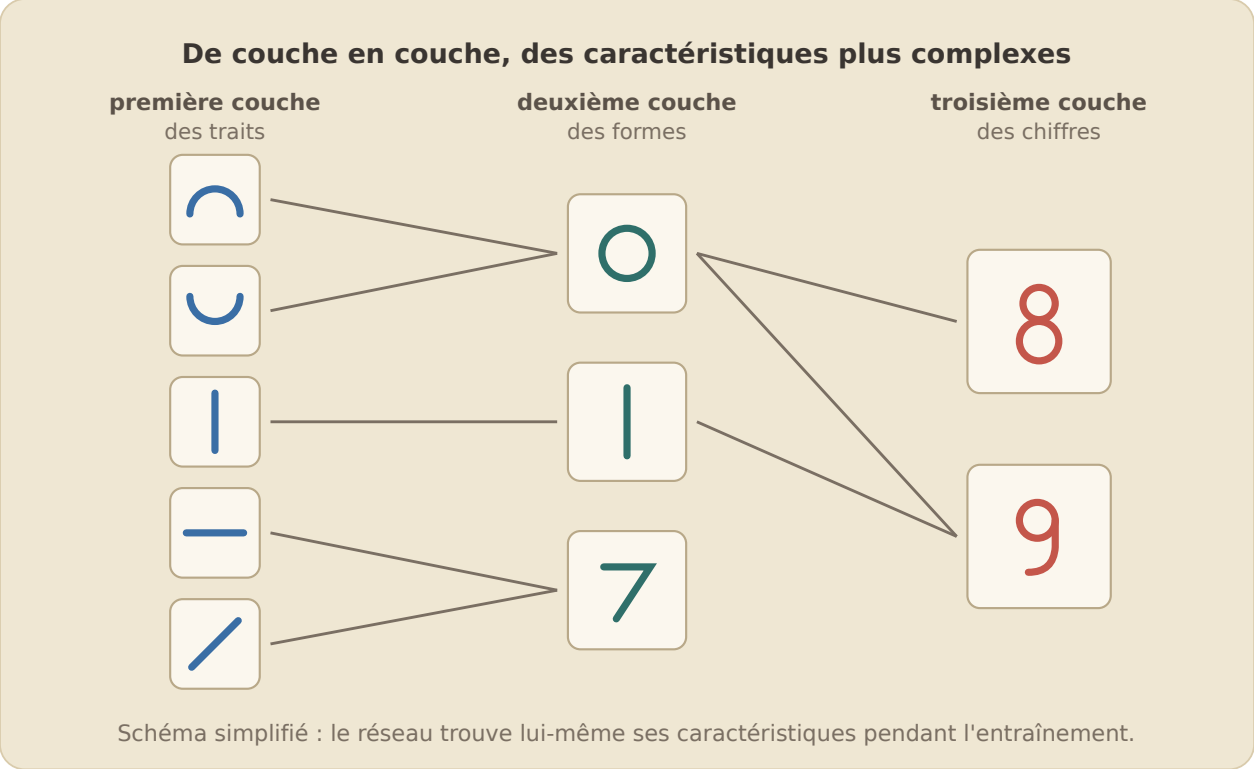
Mélanger les colonnes d'un tableau ne change rien à ce qu'il dit. Mélanger les pixels d'une image efface ce qu'elle montre.

Ce module raconte ce retour, et explique comment ces réseaux fonctionnent.

Une seule grande idée : apprendre ses propres caractéristiques

Au Module 2, un modèle recevait des données décrites par des **caractéristiques** (*features*), c'est-à-dire des grandeurs mesurables qui décrivent chaque exemple : la superficie d'une maison, son année de construction, la présence d'un mot dans un courriel. C'est un humain qui choisissait ces caractéristiques, et le modèle ne voyait les données qu'à travers elles. Quand un problème résistait, comme le XOR, c'était encore à l'humain d'inventer une nouvelle caractéristique, le produit des deux entrées, qui rendait le problème facile.

Cette nouvelle caractéristique est une **combinaison** de deux autres. C'est une idée importante : une caractéristique peut être construite à partir d'autres caractéristiques, elles-mêmes construites à partir d'autres, sur plusieurs étages. Pour une image de chiffre, les caractéristiques de départ sont les pixels, qui ne disent presque rien isolément. En combinant des pixels voisins, on obtient de petits traits. En combinant des traits, on obtient des courbes et des coins. En combinant des courbes, on obtient une boucle fermée ou une barre verticale. C'est en combinant ces formes qu'on reconnaît un 8 ou un 9. Concevoir à la main tous ces étages, pour toutes les façons d'écrire chaque chiffre, est presque impossible.



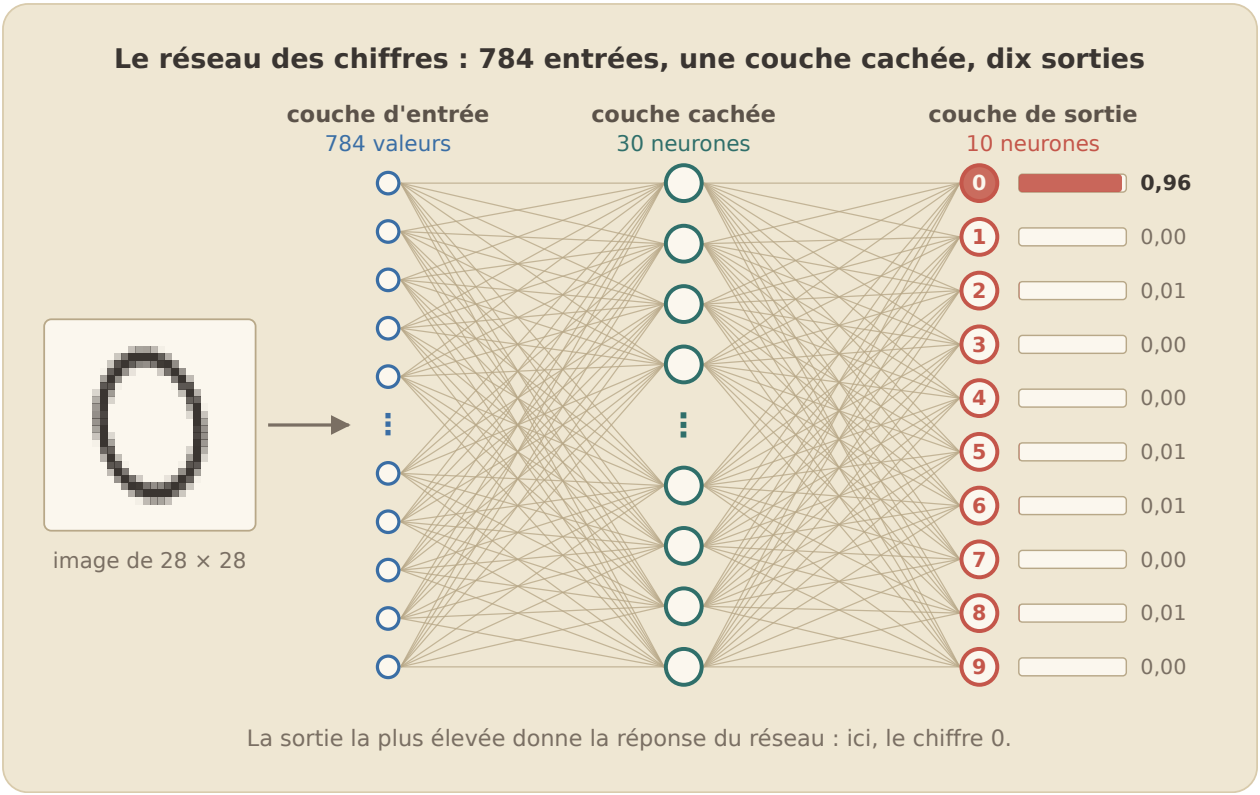
Une hiérarchie de caractéristiques : des traits, puis des formes, puis des chiffres.

Un réseau de neurones fabrique lui-même ses caractéristiques. Chacune de ses couches combine les caractéristiques de la couche précédente pour en former de nouvelles, plus utiles. Un réseau qui reconnaît des chiffres manuscrits reçoit seulement des pixels. Il apprend seul à y repérer des traits, puis des formes, puis des chiffres. Personne ne lui indique quoi chercher.

Pour le reste, rien ne change. Un réseau s'entraîne avec le même schéma qu'au Module 2 : des données, un modèle réglable, une erreur à réduire, et la descente de gradient pour la réduire. La différence est l'échelle. Une droite avait deux paramètres. Un réseau en a des milliers, des millions, et parfois des centaines de milliards.

Un fil conducteur : les chiffres manuscrits

Un même exemple revient tout au long du module : reconnaître un chiffre écrit à la main, sur une image de 28 pixels sur 28. On le retrouve dans le premier neurone, dans le réseau à une couche cachée, dans les réseaux convolutifs de Yann Le Cun, qui lisaient les chèques dans les années 1990, et dans les exemples adverses qui trompent un réseau.



Le réseau des chiffres, qui sert d'exemple tout au long du module.

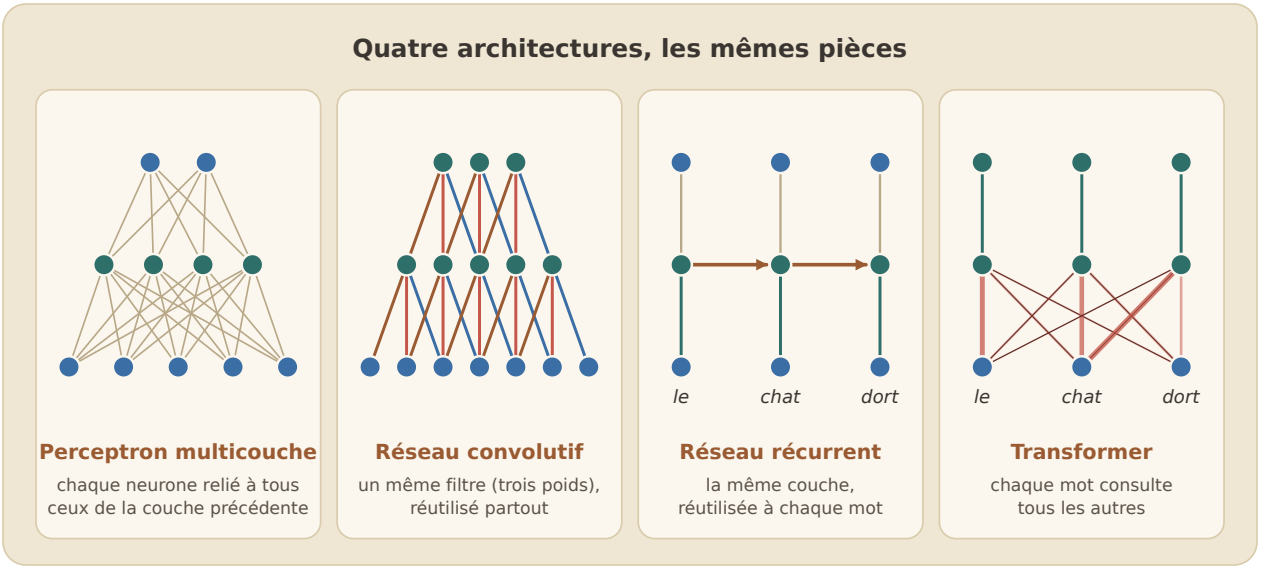
Ce que ce module n'est pas

Trois précisions évitent des malentendus fréquents.

- **Un réseau de neurones n'est pas un cerveau.** Il s'en inspire, mais un neurone artificiel est une formule simple. Le module y revient deux fois, dans les pages *Un neurone* et *L'apprentissage profond*.
- **Les réseaux de ce module reconnaissent et prédisent.** Ils associent une réponse à une entrée : quel chiffre, quel mot traduit, quel coup jouer. La génération de textes et d'images est le sujet du Module 4, qui s'appuie sur les architectures présentées ici, en particulier le Transformer.
- **Aucune formule n'est nécessaire.** Les mécanismes sont expliqués par des figures et des applets. Les calculs, pour qui veut les voir, sont dans des blocs dépliables « Pour aller plus loin ».

Des variations sur un même thème

Les réseaux de ce module portent des noms différents : perceptron multicouche, réseau convolutif, réseau récurrent, Transformer. Ils reposent pourtant tous sur les mêmes pièces. Ce sont des neurones qui calculent des sommes pondérées, des fonctions d'activation et des poids ajustés par une même méthode, la rétropropagation. D'un réseau à l'autre, seule change la façon de relier ces pièces, c'est-à-dire l'architecture. Un réseau convolutif réutilise le même petit filtre partout dans l'image. Un réseau récurrent réutilise la même couche à chaque mot. Un Transformer laisse chaque mot consulter tous les autres.



Des neurones, des poids, une activation. Seule change la façon de les relier.

Ces pièces s'assemblent comme celles d'un jeu de construction. On peut les empiler, les combiner, ou brancher un réseau à la suite d'un autre, et l'ensemble s'entraîne toujours de la même façon. C'est l'une des grandes forces des réseaux de neurones. C'est aussi ce qui explique qu'une seule famille de méthodes se soit imposée pour les images, les sons, les textes et les jeux.

Le parcours du module

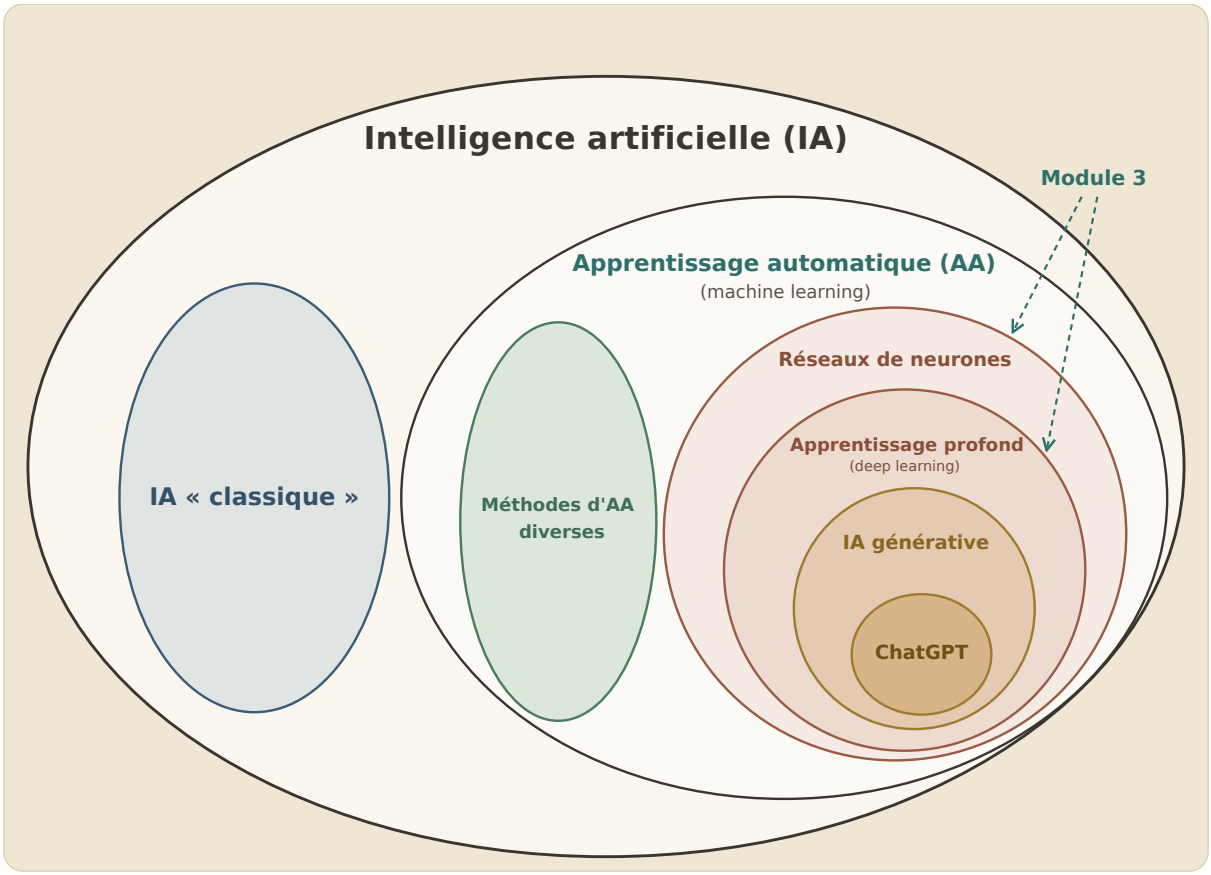
Comme au Module 2, chaque page rencontre une limite qui mène à la suivante.

1. *Un neurone* : le retour du perceptron. Un neurone est une régression logistique, et il ne trace qu'une droite.
2. *Une couche cachée* : quelques neurones reliés résolvent le XOR, en fabriquant eux-mêmes la bonne caractéristique.
3. *Entraîner un réseau* : la rétropropagation, et la question de 1969 réglée en 1986.
4. *L'apprentissage profond* : pourquoi empiler les couches, et pourquoi cela n'a fonctionné qu'à partir de 2012.
5. *Le matériel et les outils* : l'histoire des processeurs graphiques et de la différentiation automatique.
6. *Voir : les réseaux convolutifs* : le filtre qui glisse sur l'image, Yann Le Cun et les chèques, l'apprentissage par transfert.
7. *Lire une séquence : les réseaux récurrents* : un réseau avec une mémoire, le LSTM, la traduction.
8. *L'attention et le Transformer* : regarder toute la séquence à la fois, l'architecture des grands modèles actuels.
9. *Apprendre à jouer : le renforcement profond* : Atari, AlphaGo, AlphaZero.
10. *Tromper un réseau* : les exemples adverses, et la boîte noire.

Le module compte huit applets, dont un neurone à régler, un réseau qui apprend le XOR sous vos yeux, un filtre de convolution, une machine qui apprend le morpion en jouant contre elle-même, et un classifieur à tromper.

Pour situer ce module dans l'ensemble du cours : les réseaux de neurones sont une famille de

méthodes d'apprentissage automatique, devenue la plus importante depuis 2012, et sur laquelle reposent l'IA générative et les grands modèles de langage du Module 4.



La carte de l'IA : le Module 3 porte sur les réseaux de neurones et l'apprentissage profond.

Objectifs

Au terme de ce module, vous devriez être en mesure de :

- expliquer ce qu'est un neurone artificiel, et pourquoi un neurone seul ne trace qu'une frontière droite ;
- expliquer comment une couche cachée permet de résoudre le XOR, en fabriquant de nouvelles caractéristiques ;
- décrire le principe de la rétropropagation, et le distinguer de l'inférence ;
- expliquer ce qu'apporte la profondeur, et les trois conditions du succès de 2012 : les données, le calcul et de meilleures méthodes d'entraînement ;
- décrire l'idée principale des réseaux convolutifs, des réseaux récurrents et du Transformer, et le type de données auquel chacun est adapté ;
- expliquer comment on réutilise un réseau déjà entraîné pour une nouvelle tâche (apprentissage par transfert) ;
- expliquer comment le renforcement profond combine la recherche du Module 1 et l'apprentissage ;
- expliquer ce qu'est un exemple adverse, et pourquoi un réseau profond est une boîte noire ;

- situer ces développements dans l'histoire du connexionnisme, de 1958 à aujourd'hui.

Durée

Quatre semaines, soit environ 36 heures.

Évaluation

Un travail noté (20 % de la note finale) où vous manipulerez des réseaux de neurones dans une application interactive, TensorFlow Playground, avec des questions d'interprétation.

Un neurone

Le chapitre « Deux paris rivaux » a présenté le **perceptron** de Frank Rosenblatt (1958), une machine qui apprend à partir d'exemples au lieu de suivre des règles écrites à la main. Le chapitre « Les hivers et la bascule » a montré comment cette approche a été mise de côté après 1969, quand Minsky et Papert ont démontré qu'un perceptron ne peut pas apprendre certaines fonctions simples, comme le XOR.

Les **réseaux de neurones** (*neural networks*) actuels sont les descendants directs du perceptron. Le plus simple d'entre eux en porte d'ailleurs le nom : le **perceptron multicouche** (*multilayer perceptron*, MLP). Ce module explique comment ils fonctionnent, comment ils ont dépassé la limite de 1969, et pourquoi ils dominent l'intelligence artificielle depuis le début des années 2010. Il commence par l'élément de base, le **neurone artificiel** (*artificial neuron*).

Un neurone est une régression logistique

Vous connaissez déjà le neurone artificiel, sous un autre nom. C'est la régression logistique (*logistic regression*) du Module 2.

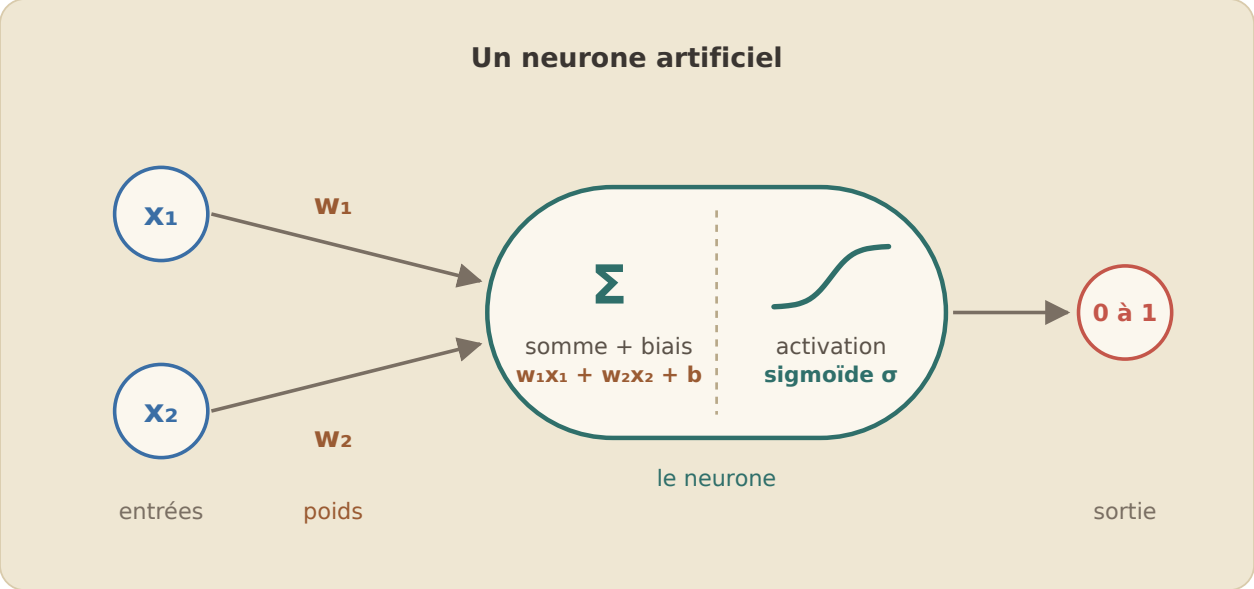
Un neurone reçoit des nombres en **entrée**, et il produit un seul nombre en **sortie**. Il effectue pour cela deux opérations :

1. Il multiplie chaque entrée par un **poids** (*weight*), il additionne les résultats, et il ajoute un dernier nombre, le **biais** (*bias*). On obtient une somme pondérée (*weighted sum*).
2. Il passe cette somme dans une **fonction d'activation** (*activation function*). La plus classique est la sigmoïde, qui ramène n'importe quel nombre à une valeur comprise entre 0 et 1.

Pour un neurone à deux entrées, le calcul s'écrit ainsi :

$$\text{sortie} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

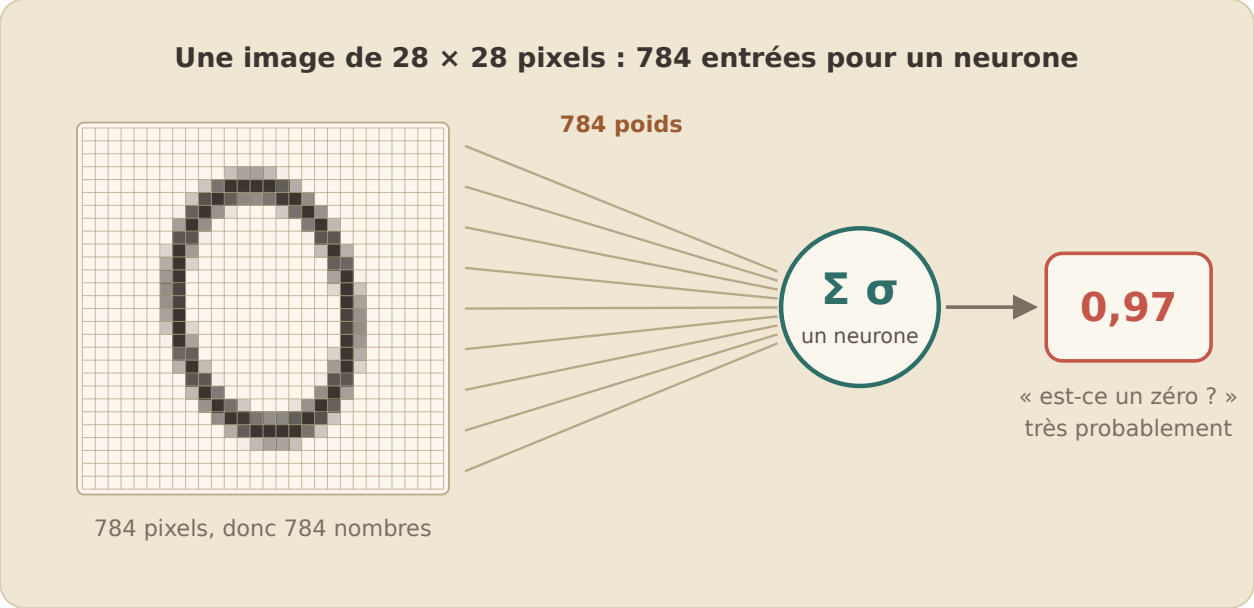
où x_1 et x_2 sont les entrées, w_1 et w_2 les poids, b le biais et σ la sigmoïde.



Un neurone artificiel : une somme pondérée des entrées, puis une fonction d'activation.

Les poids et le biais sont les **paramètres** du neurone, au sens du Module 2. Un poids élevé donne beaucoup d'importance à l'entrée correspondante, un poids proche de zéro la fait presque ignorer, et un poids négatif la fait compter contre la conclusion. Comme pour tous les modèles du Module 2, on ne fixe pas ces paramètres à la main. On les règle par descente de gradient (*gradient descent*), à partir d'exemples.

Un neurone peut avoir beaucoup plus que deux entrées. Prenons l'exemple qui servira dans tout ce module, la **reconnaissance de chiffres manuscrits** (*handwritten digit recognition*). Une image de chiffre de 28 pixels sur 28 est une liste de 784 nombres, un par pixel, comme l'a montré le chapitre « Regarder les données ». Un neurone qui reçoit cette image a donc 784 entrées et 784 poids. Sa sortie peut se lire comme une réponse à une question simple, par exemple « cette image est-elle un zéro ? ».

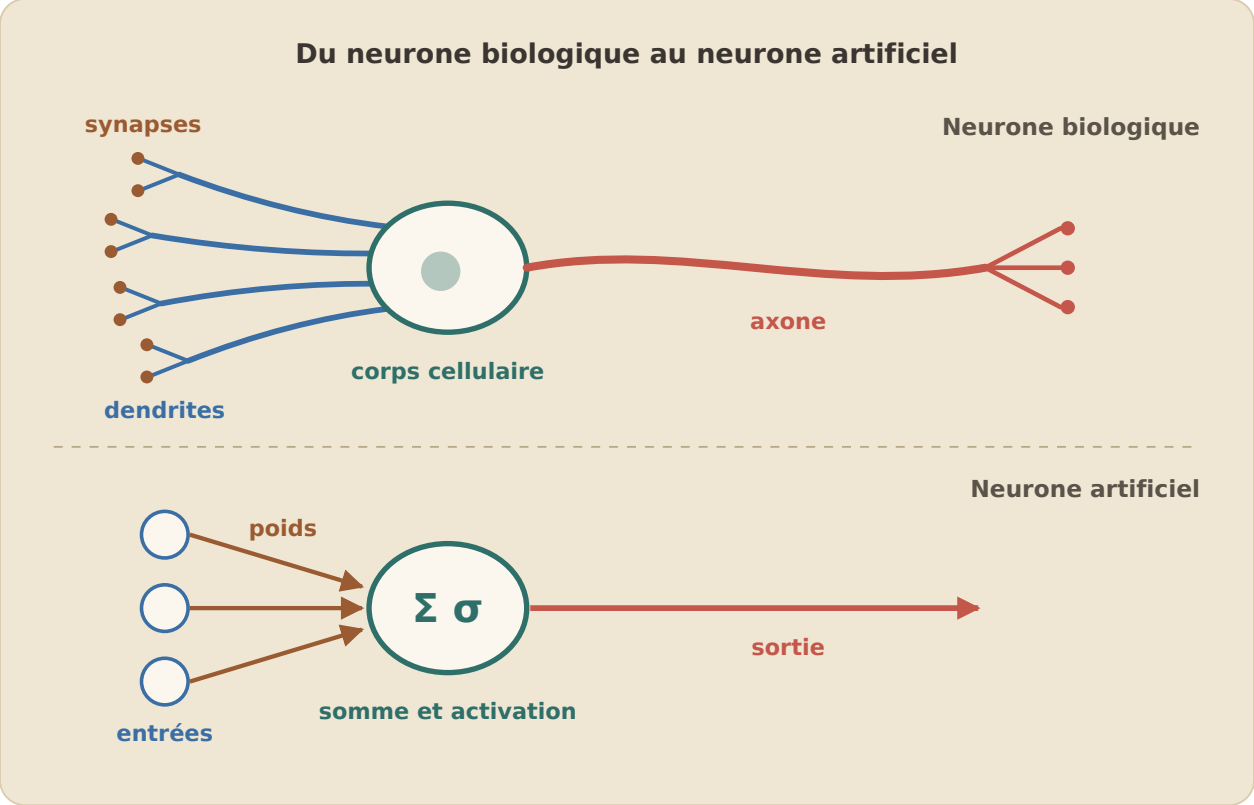


Une image de chiffre manuscrit donne 784 entrées à un neurone, qui répond par un nombre entre 0 et 1.

D'où vient le mot « neurone »

Le terme vient d'une analogie avec le cerveau. Un neurone biologique reçoit des signaux d'autres neurones par ses **dendrites**. Il les combine dans son **corps cellulaire**. Si le signal combiné est assez fort, il envoie à son tour un signal le long de son **axone**, vers d'autres neurones. Les points de contact entre neurones s'appellent des **synapses**, et leur force varie avec l'expérience.

Le neurone artificiel reprend ce schéma en le simplifiant beaucoup. Les entrées jouent le rôle des dendrites, les poids celui des synapses, la somme et la fonction d'activation celui du corps cellulaire, et la sortie celui de l'axone.



Du neurone biologique au neurone artificiel : les entrées pour les dendrites, les poids pour les synapses, le calcul pour le corps cellulaire, la sortie pour l'axone.

Cette analogie a guidé les premiers chercheurs, mais elle est très approximative. Un neurone biologique est beaucoup plus complexe qu'une somme suivie d'une fonction. Le chapitre « L'apprentissage profond » reviendra sur ce que cette comparaison permet de dire, et sur ce qu'elle ne permet pas.

Un neurone à manipuler

Dans l'applet ci-dessous, vous réglez vous-même les entrées et les paramètres d'un neurone, et vous observez sa sortie.

Applet interactive : à utiliser en ligne, sur la page « Un neurone » du site du cours.

Essayez les trois manipulations suivantes :

1. Fixez les deux entrées à 1. Trouvez des poids qui donnent une sortie proche de 1, puis des poids qui donnent une sortie proche de 0.
2. Donnez un poids de zéro à la première entrée. Faites varier cette entrée, et observez que la sortie ne change plus.
3. Donnez un poids négatif à la seconde entrée. Augmentez cette entrée, et observez que la sortie diminue.

Ce qu'un neurone seul ne peut pas faire

Un neurone seul a la même limite que la régression logistique. Sa somme pondérée définit une **frontière droite** : d'un côté, la sortie est supérieure à 0,5, et de l'autre, elle est inférieure. Un neurone ne peut donc séparer deux catégories que si une droite suffit à les séparer.

Ce n'est pas toujours le cas. Le XOR, présenté avec sa table de vérité (*truth table*) au Module 1, en est l'exemple le plus simple. Ses quatre cas occupent les coins d'un carré, et aucune droite ne sépare les deux cas « vrai » des deux cas « faux ». C'est exactement la limite que Minsky et Papert ont démontrée en 1969.

Au Module 2, on contournait cette limite de deux façons : en changeant de modèle, ou en fabriquant à la main une nouvelle caractéristique (*feature*). Les réseaux de neurones apportent une troisième réponse. On relie plusieurs neurones entre eux, et le réseau fabrique lui-même la caractéristique qui manque. C'est le sujet du chapitre suivant, « Une couche cachée ».

Une couche cachée

Le chapitre « Un neurone » s'est terminé sur une limite : un neurone seul ne trace qu'une frontière droite, et il ne peut donc pas apprendre le XOR. Ce chapitre montre comment quelques neurones reliés entre eux dépassent cette limite.

Relier des neurones

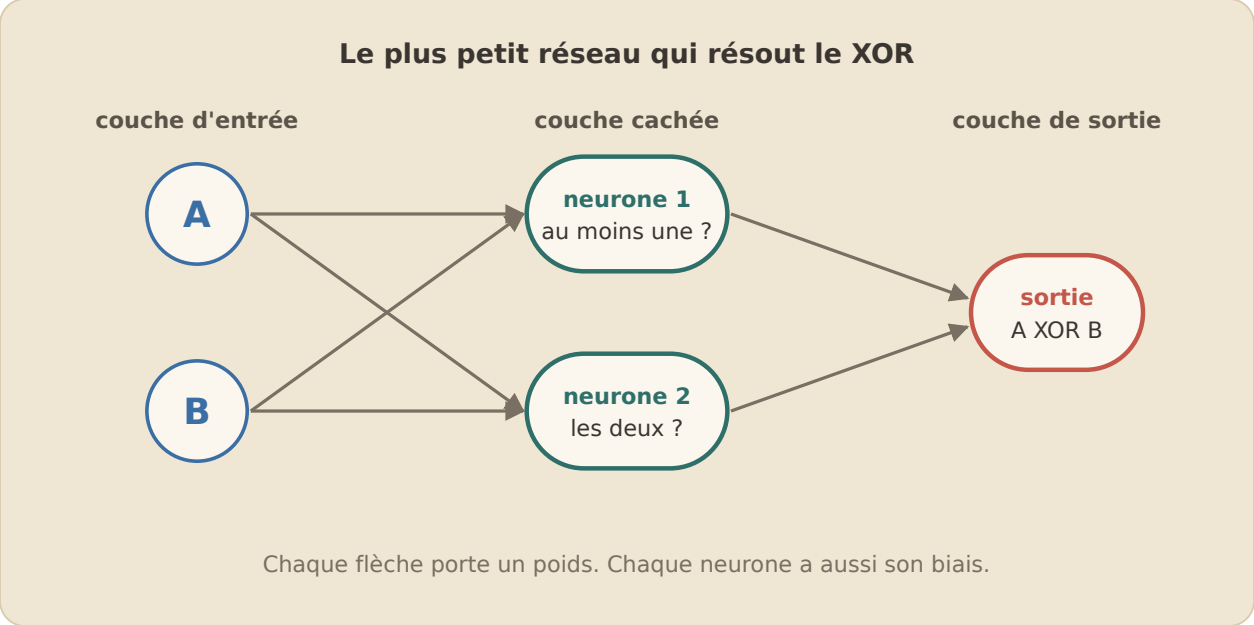
La sortie d'un neurone est un nombre. Rien n'empêche de donner ce nombre en entrée à un autre neurone. On obtient alors un **réseau de neurones**, organisé en **couches** (*layers*) :

- la **couche d'entrée** (*input layer*) contient les données, par exemple les deux entrées du XOR ou les 784 pixels d'une image ;
- la **couche de sortie** (*output layer*) donne la réponse du réseau ;
- entre les deux, une ou plusieurs **couches cachées** (*hidden layers*) font des calculs intermédiaires. On les appelle « cachées » parce que, de l'extérieur, on ne voit que les entrées et la réponse.

Chaque neurone d'une couche reçoit les sorties de tous les neurones de la couche précédente, chacune avec son propre poids.

Un réseau construit de cette façon s'appelle un **perceptron multicouche** (*multilayer perceptron*, MLP). Le nom est un héritage historique, un peu trompeur : ses neurones ne sont pas exactement des perceptrons de Rosenblatt, puisque leur sortie varie de façon continue au lieu de passer brusquement de 0 à 1. Mais l'idée est bien celle-là : des perceptrons empilés en plusieurs couches. C'est le type de réseau le plus simple, et celui de ce chapitre et du suivant.

La forme d'un réseau s'appelle son **architecture** (*architecture*) : le nombre de couches, le nombre de neurones dans chacune, et surtout la façon dont ils sont reliés. Le dessin de ces connexions, où chaque neurone est un point et chaque poids un trait, s'appelle sa **topologie** (*topology*). L'architecture est choisie par un humain avant l'entraînement, alors que les poids sont appris. Le perceptron multicouche relie chaque neurone à tous ceux de la couche précédente. C'est la topologie la plus simple, mais pas la seule. Les chapitres suivants présentent d'autres architectures, adaptées aux images, aux séquences ou au langage. Elles sont toutes faites des mêmes briques, des neurones reliés par des poids et entraînés par la même méthode. Seule change la façon de les relier.



Le plus petit réseau qui résout le XOR : deux entrées, deux neurones cachés, un neurone de sortie.

Le XOR résolu avec trois neurones

Reprenons le XOR, avec ses deux entrées A et B. Le réseau ci-dessus le résout si chaque neurone répond à une question simple, qu'un neurone seul sait traiter :

- le premier neurone caché répond à la question « **au moins une** des deux entrées est-elle vraie ? » ;
- le second neurone caché répond à la question « les **deux** entrées sont-elles vraies ? » ;
- le neurone de sortie combine ces deux réponses. Il répond « vrai » quand le premier neurone dit oui et que le second dit non, c'est-à-dire quand au moins une entrée est vraie, mais pas les deux.

La table suivante reprend les quatre cas :

A	B	Neurone 1 : au moins une ?	Neurone 2 : les deux ?	Sortie : A XOR B
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

La dernière colonne est bien celle de la table de vérité du XOR. Aucun des trois neurones ne fait quelque chose de difficile. C'est leur combinaison qui résout un problème qu'aucun d'eux ne peut résoudre seul.

Pour aller plus loin : les poids de ce réseau
Voici des poids et des biais qui donnent exactement ce comportement. Chaque neurone calcule

une somme pondérée, puis la passe dans la sigmoïde, comme dans le chapitre « Un neurone ».

Neurone	Poids	Biais	Somme calculée
Neurone 1 (au moins une ?)	10 et 10	-5	$10{\cdot}A + 10{\cdot}B - 5$
Neurone 2 (les deux ?)	10 et 10	-15	$10{\cdot}A + 10{\cdot}B - 15$
Sortie	10 et -20	-5	$10{\cdot}(\text{neurone } 1) - 20{\cdot}(\text{neurone } 2) - 5$

Prenons le cas $A = 0$ et $B = 1$:

- le neurone 1 calcule $10{\cdot}0 + 10{\cdot}1 - 5 = 5$, et la sigmoïde donne environ 0,99 ;
- le neurone 2 calcule $10{\cdot}0 + 10{\cdot}1 - 15 = -5$, et la sigmoïde donne environ 0,01 ;
- la sortie calcule $10{\cdot}0{,}99 - 20{\cdot}0{,}01 - 5 = 4{,}7$, et la sigmoïde donne environ 0,99.

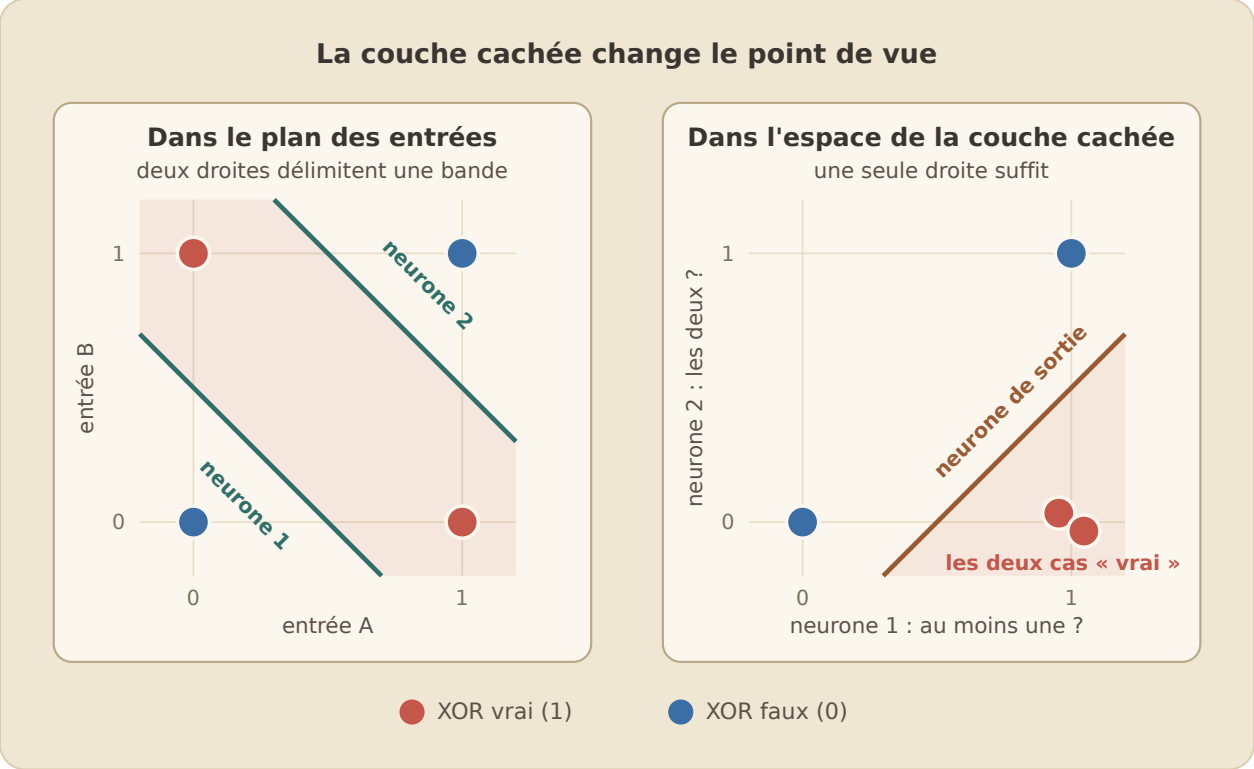
Le réseau répond donc « vrai », comme prévu. Les trois autres cas se vérifient de la même façon. Les poids sont grands (10, 20) pour que la sigmoïde donne des réponses nettes, très proches de 0 ou de 1.

Ce que fait la couche cachée : changer de point de vue

On peut comprendre ce résultat de deux façons.

Première lecture : deux droites au lieu d'une. Chaque neurone caché trace sa propre droite dans le plan des entrées. Avec deux droites, on délimite une bande. Les deux cas « vrai » du XOR sont dans la bande, et les deux cas « faux » sont à l'extérieur, un de chaque côté. Une seule droite ne pouvait pas faire ce découpage.

Seconde lecture : un nouvel espace. La couche cachée remplace les deux entrées, A et B, par deux nouvelles valeurs, les réponses des neurones 1 et 2. Dans ce nouvel espace, les quatre cas ne sont plus aux coins d'un carré. Les deux cas « vrai » arrivent au même endroit, et une seule droite suffit maintenant à les séparer des deux autres. Le neurone de sortie trace cette droite.



Le XOR avant et après la couche cachée : deux droites dans le plan des entrées, une seule droite dans le nouvel espace.

Cette seconde lecture rejoint une idée du Module 2. Dans « Généraliser », on rendait le XOR séparable en ajoutant à la main une caractéristique, le produit des deux entrées. Ce produit vaut 1 seulement quand les deux entrées valent 1. C'est exactement la question à laquelle répond le neurone 2. La différence est que personne n'a besoin de trouver cette caractéristique : avec les bons poids, le réseau la construit lui-même. Une couche cachée est donc une **fabrique de caractéristiques**.

Deux droites à déplacer

Dans l'applet ci-dessous, chaque droite représente un neurone caché. Vous pouvez la déplacer et la faire pivoter, comme dans l'applet de la régression logistique. La zone délimitée par les deux droites est colorée : c'est là que le réseau répond « vrai ». Un point mal classé est entouré d'un cercle orange.

Applet interactive : à utiliser en ligne, sur la page « Une couche cachée » du site du cours.

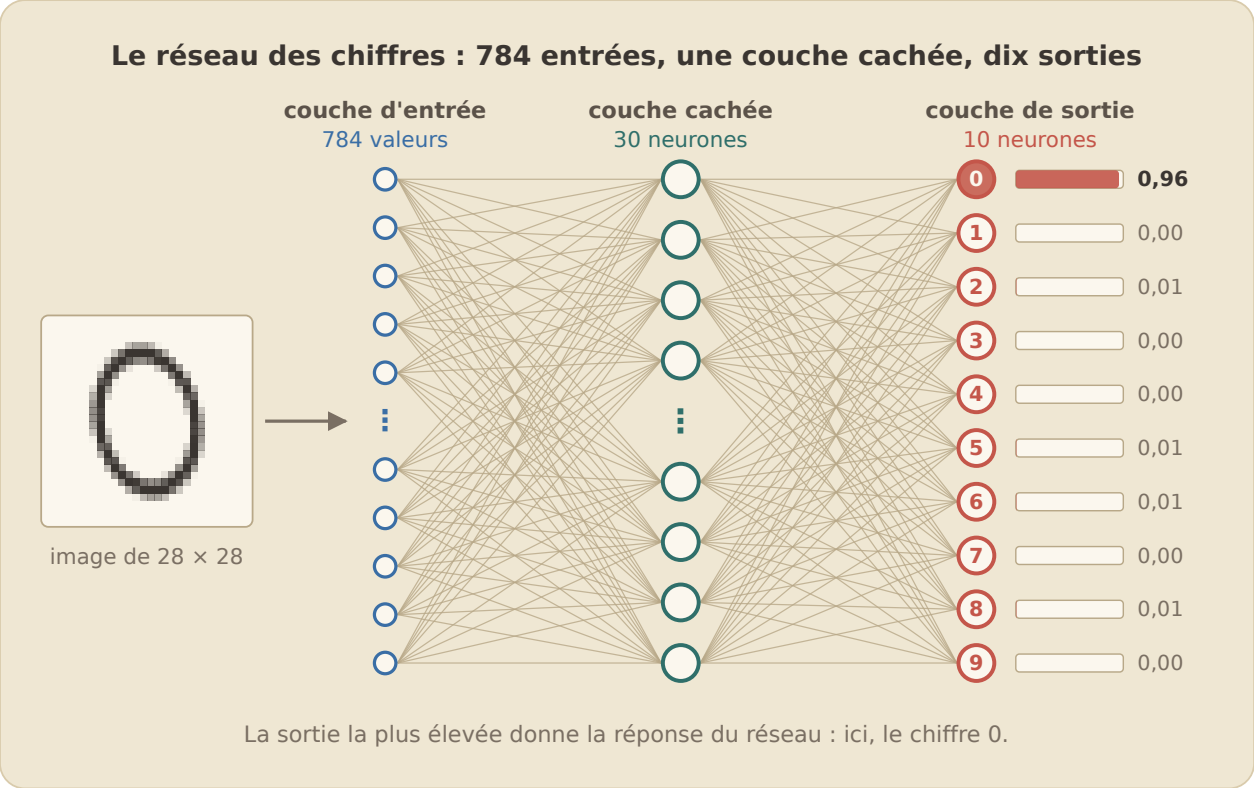
1. Placez les deux droites pour que la zone colorée contienne les deux points rouges, et aucun point bleu.
2. Cherchez une autre solution, avec des droites orientées autrement. Il en existe plusieurs.
3. Choisissez ensuite le second jeu de points, « un groupe entouré », où un groupe rouge est entouré de points bleus. Constatez que deux droites ne suffisent plus.
4. Ajoutez une troisième droite, avec le bouton « 3 », et entourez le groupe rouge.

Plus de neurones, plus de formes

Deux neurones cachés donnent deux droites. Trois neurones en donnent trois, ce qui permet de délimiter un triangle, comme dans la dernière manipulation. Avec un grand nombre de neurones, on peut entourer une zone de forme quelconque, à la précision voulue.

Ce résultat a été démontré en 1989, sous le nom de **théorème d'approximation universelle** (*universal approximation theorem*) : un réseau à une seule couche cachée, avec assez de neurones, peut approcher presque n'importe quelle fonction. Le répertoire de formes d'un réseau de neurones n'a donc pas la limite de celui d'un modèle linéaire.

Un réseau peut aussi avoir plusieurs sorties. Pour reconnaître les chiffres manuscrits, on utilise un réseau à **dix sorties**, une par chiffre. Chaque sortie donne un nombre entre 0 et 1, et le réseau répond par le chiffre dont la sortie est la plus élevée. En pratique, on fait en sorte que les dix sorties totalisent 1. Elles forment alors une distribution de probabilités sur les dix chiffres, comme les probabilités « pourriel » et « courriel » du classifieur bayésien. Le réseau ne répond pas seulement « c'est un 0 », mais « c'est un 0, avec une probabilité de 96 % ».



Un réseau pour les chiffres manuscrits : 784 entrées, une couche cachée de 30 neurones, dix sorties.

Qui règle tous ces poids ?

Pour le XOR, nous avons choisi les poids à la main, parce que le réseau n'en compte que neuf. Le réseau des chiffres en compte des milliers. Avec une couche cachée de 30 neurones, il y a 784 × 30 poids entre l'entrée et la couche cachée, puis 30 × 10 entre la couche cachée et la sortie, soit près de 24 000 paramètres en comptant les biais. Personne ne peut les régler à la main.

Le théorème d'approximation universelle dit qu'il existe de bons poids. Il ne dit pas comment les trouver. Ce problème a bloqué les réseaux de neurones pendant près de vingt ans. On savait depuis les années 1960 qu'une couche cachée dépasserait la limite du perceptron, mais on ne savait pas comment entraîner un tel réseau. La solution, la **rétropropagation du gradient** (*backpropagation*), s'est imposée en 1986. C'est le sujet du chapitre suivant, « Entraîner un réseau ».

Entraîner un réseau

Le chapitre « Une couche cachée » s'est terminé sur un réseau de près de 24 000 poids, que personne ne peut régler à la main. Ce chapitre explique comment un réseau règle lui-même ses poids, à partir d'exemples.

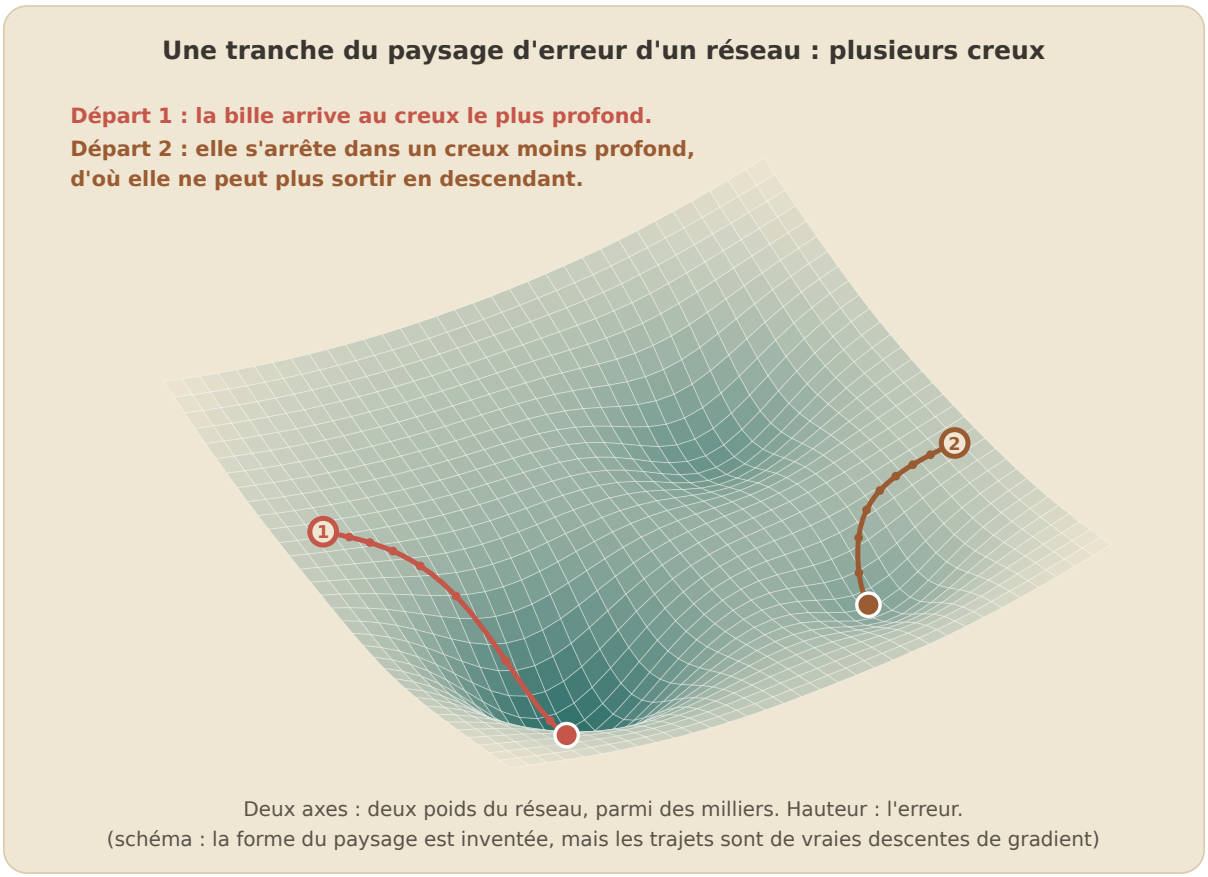
Le même principe qu'au Module 2

La méthode a déjà été présentée dans « Apprendre, c'est descendre la pente ». Elle tient en trois étapes, qu'on répète :

1. on mesure l'**erreur** du modèle sur des exemples dont on connaît la bonne réponse ;
2. on cherche dans quel sens modifier chaque paramètre pour que l'erreur diminue ;
3. on modifie un peu chaque paramètre dans ce sens.

C'est la **descente de gradient**. Pour la droite du Module 2, il y avait deux paramètres, et l'erreur formait un paysage qu'on pouvait dessiner. Pour un réseau, les paramètres sont les poids et les biais. Le paysage a alors des milliers de dimensions et on ne peut plus le dessiner, mais la méthode reste la même : mesurer la pente, puis faire un pas vers le bas.

On peut toutefois en dessiner une tranche, en ne faisant varier que deux poids. Contrairement à la cuvette du Module 2, ce paysage a en général plusieurs creux, et le creux où l'on arrive dépend du point de départ. La section « Un réseau qui apprend le XOR » le montre sur un vrai réseau.



Une tranche du paysage d'erreur d'un réseau, selon deux de ses poids. Selon le point de départ, la descente de gradient mène à des creux différents.

Pour un réseau qui classe, l'erreur mesurée est en général la même que celle de la régression logistique. La réduire revient à rendre les bonnes réponses aussi probables que possible : c'est le maximum de vraisemblance (*maximum likelihood*) présenté au Module 2.

Le problème des couches cachées

La deuxième étape est simple pour un neurone seul. On connaît la bonne réponse, on voit donc directement si sa sortie est trop haute ou trop basse, et on corrige ses poids en conséquence. C'est ce que faisait le perceptron dès 1958.

Dans un réseau, cela ne vaut que pour la couche de sortie. Pour un neurone caché, les exemples ne donnent aucune bonne réponse. Reprenons le réseau du XOR : la table de vérité dit ce que la sortie doit valoir, mais elle ne dit pas que le premier neurone caché doit répondre à la question « au moins une ? ». Quand le réseau se trompe, on ne sait pas quel neurone caché est en cause, ni dans quel sens le corriger.

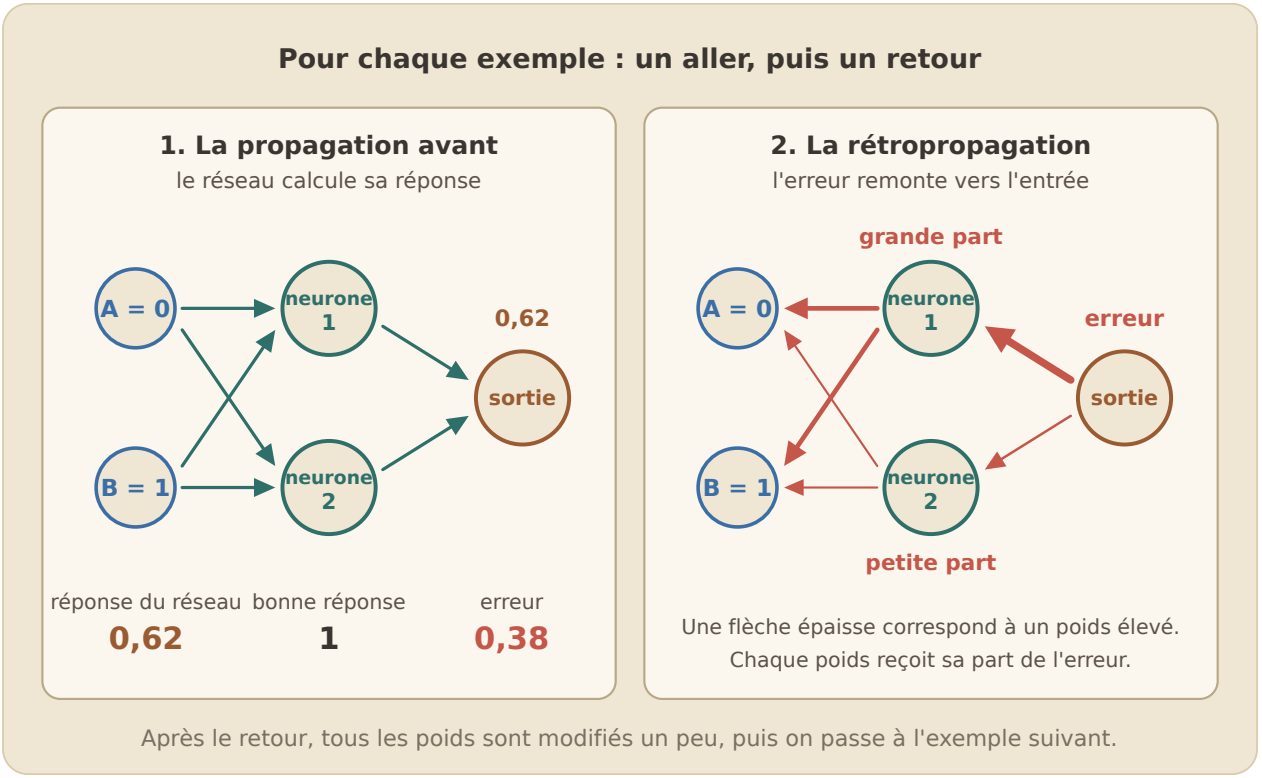
Ce problème s'appelle l'**attribution de la responsabilité** (*credit assignment* en anglais). C'est lui qui a arrêté la recherche sur les réseaux de neurones après 1969 : on savait qu'une couche cachée était nécessaire, et on ne savait pas l'entraîner.

La rétropropagation

La solution consiste à calculer la responsabilité de chaque poids en partant de la sortie et en remontant vers l'entrée. Pour chaque exemple, le calcul se fait en deux temps.

La propagation avant (*forward pass*). L'exemple entre dans le réseau. Chaque couche calcule ses sorties et les passe à la suivante, jusqu'à la réponse du réseau. On compare cette réponse à la bonne, ce qui donne l'erreur. Les poids sont utilisés, mais pas modifiés.

La rétropropagation. L'erreur repart en sens inverse. Le neurone de sortie reçoit l'erreur entière. Il la répartit entre les neurones cachés qui lui ont envoyé un signal, selon le poids de chaque connexion : un neurone caché relié par un poids élevé a beaucoup contribué à la réponse, et il reçoit donc une grande part de l'erreur. Chaque neurone caché sait alors dans quel sens il aurait dû répondre, et on peut corriger ses poids comme ceux d'un neurone seul.



Les deux temps du calcul pour un exemple : la propagation avant donne la réponse et l'erreur, la rétropropagation répartit cette erreur entre les poids.

Une fois la part de chaque poids connue, on modifie tous les poids un peu, dans le sens qui réduit l'erreur. Puis on passe à l'exemple suivant. Le nom complet de la méthode est la **rétropropagation du gradient** (*backpropagation* en anglais).

Avec plusieurs couches cachées, le principe ne change pas : l'erreur remonte d'une couche à la précédente, jusqu'à l'entrée.

Pour aller plus loin : ce que calcule la rétropropagation
Un réseau est une suite de calculs emboîtés. La sortie dépend de la couche cachée, qui dépend elle-même des entrées et des poids. La descente de gradient demande, pour chaque poids, la pente de l'erreur par rapport à ce poids, c'est-à-dire une dérivée. Pour des calculs emboîtés, cette dérivée s'obtient par la règle de dérivation en chaîne (*chain rule*) : on multiplie entre elles les dérivées de chaque étape, de la sortie jusqu'au poids visé.
La rétropropagation est une façon efficace d'appliquer cette règle. En partant de la sortie, elle réutilise les calculs faits pour une couche quand elle traite la couche précédente. Le coût total est à peu près celui de deux propagations avant, quel que soit le nombre de poids. Sans cette économie, il faudrait refaire un calcul complet pour chaque poids, ce qui serait impraticable pour un grand réseau.
Cette méthode exige que chaque étape du calcul ait une dérivée. C'est la raison pour laquelle les neurones utilisent une fonction d'activation lisse comme la sigmoïde, et non le seuil brusque du perceptron d'origine.

Un réseau qui apprend le XOR

L'applet ci-dessous reprend le réseau du XOR, avec ses deux neurones cachés. Les poids de départ sont tirés au hasard. Quand vous lancez l'entraînement, le réseau voit les quatre cas en boucle et corrige ses poids par rétropropagation. Les deux droites sont celles des neurones cachés, comme dans l'applet du chapitre précédent, mais cette fois personne ne les déplace.

Applet interactive : à utiliser en ligne, sur la page « Entraîner un réseau » du site du cours.

1. Lancez l'entraînement. Observez les droites se placer et l'erreur descendre.
2. Repartez de nouveaux poids de départ, plusieurs fois, avec le bouton « Nouveaux poids ». Le réseau ne trouve pas toujours la même solution, et il lui arrive de rester bloqué avec une erreur élevée, environ une fois sur cinq.
3. Changez le taux d'apprentissage. S'il est très petit, l'apprentissage est très lent. S'il est trop grand, l'erreur oscille ou reste élevée, au lieu de descendre.

La deuxième manipulation montre une différence avec la droite du Module 2. Le paysage d'erreur (*loss landscape*) d'un réseau n'est pas une cuvette unique. Il comporte plusieurs creux, et la descente s'arrête dans celui qu'elle rencontre, qui dépend du point de départ. Pour les grands réseaux, on constate en pratique que la plupart des creux donnent des résultats comparables.

1986

La rétropropagation a été découverte plusieurs fois. Paul Werbos la décrit dans sa thèse en 1974, sans que le résultat soit remarqué. En 1986, David Rumelhart, Geoffrey Hinton et Ronald Williams publient dans la revue *Nature* un article, « Learning representations by back-propagating errors », qui montre, expériences à l'appui, qu'un réseau entraîné de cette façon construit lui-même des caractéristiques utiles dans ses couches cachées. C'est cet article qui impose la méthode, et son nom anglais, *backpropagation*, souvent abrégé en *backprop*.

La question ouverte en 1969 est alors réglée. Le XOR, que le perceptron ne pouvait pas apprendre, est appris par un réseau à couche cachée. Les réseaux de neurones redeviennent un sujet de recherche actif, sous le nom de **connexionnisme**. La même année, Rumelhart, James McClelland et leur groupe

de recherche publient *Parallel Distributed Processing*, deux volumes qui présentent cette approche, de ses principes à ses applications en psychologie. L'ouvrage fait connaître le connexionnisme bien au-delà de l'informatique, chez les psychologues, les linguistes et les philosophes.

Ce retour reste limité pendant une vingtaine d'années. Les réseaux de l'époque sont petits, les données rares et les ordinateurs lents. D'autres méthodes progressent plus vite. La plus influente est la machine à vecteurs de support (SVM), présentée au Module 2, dont la forme actuelle date de 1995. Grâce à l'astuce du noyau (*kernel trick*), elle trace des frontières courbes sans avoir à fabriquer de caractéristiques. Son entraînement a un avantage décisif sur celui d'un réseau : il n'y a qu'un seul creux dans le paysage d'erreur, et on est donc sûr de trouver la meilleure solution, quel que soit le point de départ. La deuxième manipulation de l'applet ci-dessus montre que ce n'est pas le cas pour un réseau. Le *boosting* et les forêts aléatoires (*random forests*) complètent cette boîte à outils. Jusqu'au début des années 2010, ces méthodes l'emportent souvent sur les réseaux de neurones, et les grandes conférences du domaine publient peu de travaux sur ces derniers. Le chapitre suivant explique ce qui a changé ensuite.

L'entraînement en pratique

Quelques termes reviennent dans la suite du cours.

- La **descente de gradient stochastique** (*stochastic gradient descent*, SGD). Calculer la pente exacte demanderait de passer sur tous les exemples d'entraînement avant chaque pas, soit des millions pour un grand réseau. En pratique, on estime la pente à partir d'un petit groupe d'exemples tirés au hasard, un **lot** (*batch* ou *mini-batch*), de quelques dizaines à quelques milliers d'exemples, et on fait un pas après chaque lot. Cette pente est approximative, un peu différente d'un lot à l'autre, et la descente zigzague, d'où le mot *stochastique*, qui signifie « au hasard ». Elle est pourtant beaucoup plus rapide, et ce bruit a un avantage inattendu : il aide parfois la descente à sortir d'un creux peu profond, comme celui de la figure plus haut. Des variantes, comme Adam (2014), ajustent en plus la taille des pas pour chaque poids, et ce sont elles qu'on utilise le plus aujourd'hui.
- Une **époque** (*epoch*) est un passage complet sur tous les exemples d'entraînement. Un entraînement compte en général plusieurs époques.
- Le **taux d'apprentissage** (*learning rate*) règle la taille de chaque correction, comme au Module 2.

Il faut aussi distinguer deux phases dans la vie d'un réseau.

- L'**entraînement** (*training*) règle les poids. Il demande beaucoup de calculs, parce qu'il répète l'aller et le retour sur des millions d'exemples. Il est fait une fois.
- L'**inférence** est l'utilisation du réseau entraîné. Les poids ne changent plus, et seule la propagation avant est exécutée. Elle est faite à chaque utilisation.

Cette distinction reviendra au Module 4 : entraîner un grand modèle de langage (*large language model*) prend des semaines ou des mois, alors qu'une réponse est produite en quelques secondes.

Un réseau a beaucoup de paramètres, et il est donc très exposé au sur-apprentissage (*overfitting*). Les remèdes sont ceux de « Généraliser » : un jeu de test (*test set*), et la régularisation. Pour les réseaux, la régularisation prend trois formes courantes. Le *weight decay* pénalise les poids trop grands. Le *dropout* désactive au hasard une partie des neurones à chaque étape de l'entraînement. L'**arrêt précoce** (*early stopping*) interrompt l'entraînement quand l'erreur sur des exemples mis de côté cesse de diminuer.

Pour voir le mécanisme en détail

La chaîne 3Blue1Brown, de Grant Sanderson, propose une série de vidéos sur les réseaux de neurones, construite sur l'exemple des chiffres manuscrits. Les explications sont visuelles et progressives. La vidéo ci-dessous porte sur la rétropropagation.

Vidéo : https://www.youtube.com/watch?v=Ilg3gGewQ5U

Deux autres vidéos de la série complètent ce chapitre : la première présente la structure d'un réseau, et la deuxième la descente de gradient. Les vidéos sont en anglais.

On sait maintenant entraîner un réseau à une couche cachée. Le chapitre suivant, « L'apprentissage profond », examine ce qu'on gagne à empiler plusieurs couches.

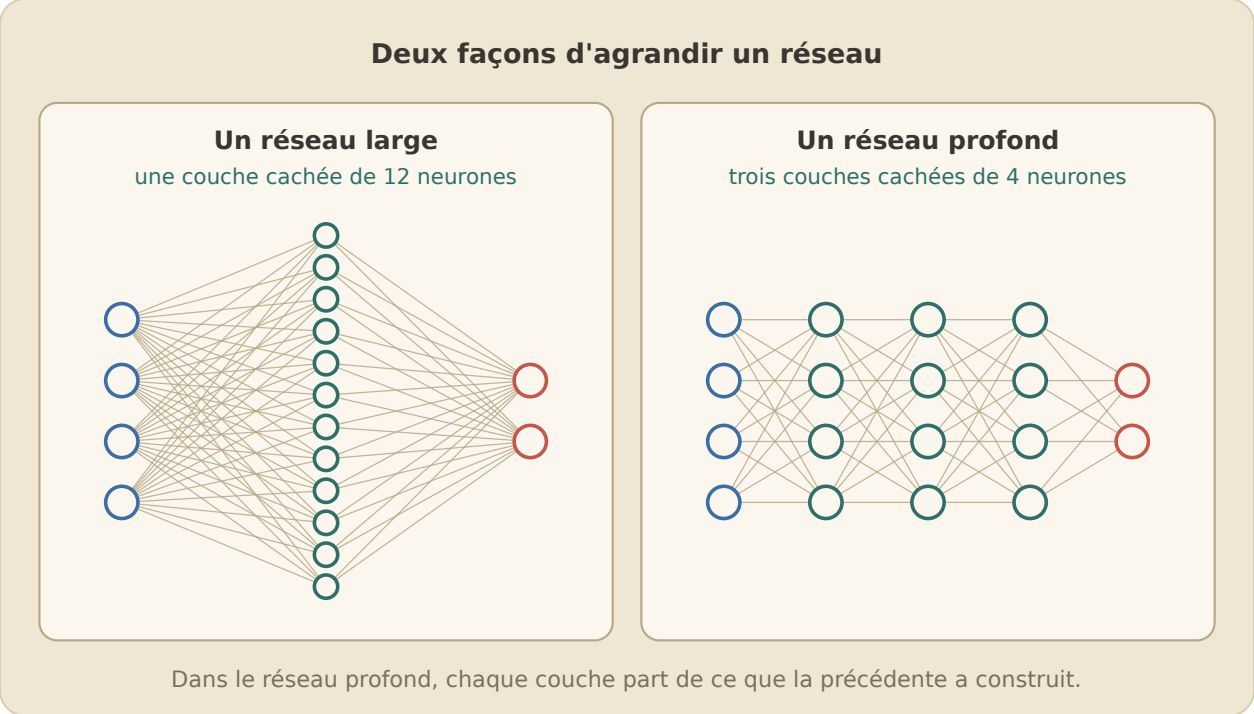
L'apprentissage profond

D'après le théorème d'approximation universelle, une seule couche cachée suffit pour approcher presque n'importe quelle fonction. Pourtant, les réseaux qui ont transformé l'IA à partir de 2012 comptent des dizaines de couches. On les appelle des réseaux **profonds**, et leur entraînement, l'**apprentissage profond** (*deep learning* en anglais). Ce chapitre explique ce qu'on gagne à empiler les couches, et pourquoi cela n'a fonctionné qu'à partir de 2012.

Large ou profond

Le théorème dit qu'une couche suffit. Il ne dit pas combien de neurones elle doit contenir. Pour certains problèmes, ce nombre est si grand que le réseau ne peut être ni stocké ni entraîné.

Il y a deux façons d'agrandir un réseau. On peut l'élargir, en ajoutant des neurones à sa couche cachée. On peut l'approfondir, en ajoutant des couches. Les deux augmentent ce que le réseau peut représenter, mais pas au même coût. Dans un réseau profond, chaque couche part de ce que la précédente a déjà construit, et le réutilise. Un réseau large à une seule couche doit tout construire directement à partir des entrées, sans étape intermédiaire. Pour beaucoup de problèmes, le réseau profond obtient le même résultat avec beaucoup moins de neurones.

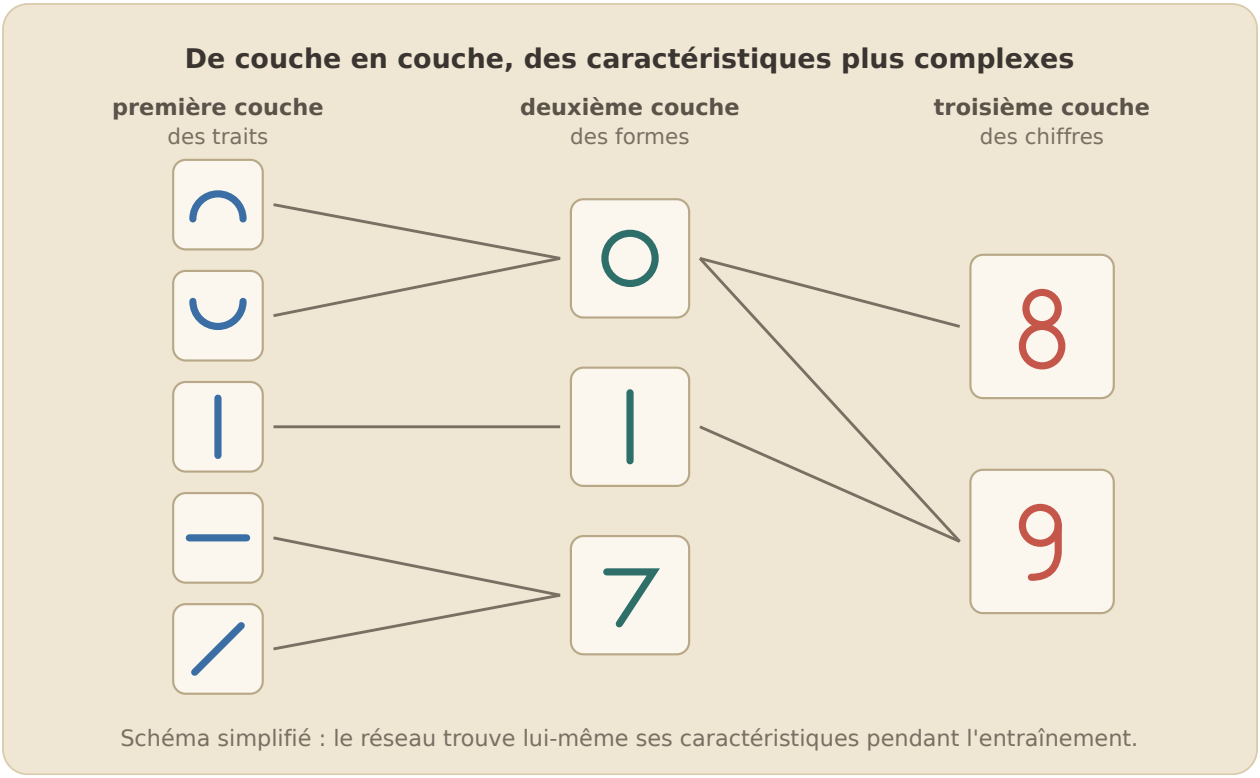


Deux façons d'agrandir un réseau : ajouter des neurones à la couche cachée, ou ajouter des couches.

Une hiérarchie de caractéristiques

Le chapitre « Une couche cachée » a montré qu'une couche cachée est une fabrique de caractéristiques. Dans un réseau profond, chaque couche fabrique les siennes à partir de celles de la couche précédente. Les caractéristiques deviennent plus complexes de couche en couche.

Prenons les chiffres manuscrits. Une première couche peut détecter de petits traits, à différents endroits et selon différentes orientations. Une deuxième couche combine ces traits en formes : une boucle, un angle, une barre verticale. Une troisième combine ces formes en chiffres. Un 9 est une boucle au-dessus d'une barre, un 8 est fait de deux boucles.



Une hiérarchie de caractéristiques : des traits, puis des formes, puis des chiffres.

Cette description est simplifiée. Personne n'indique au réseau quoi détecter dans chaque couche. Il le trouve pendant l'entraînement, et les caractéristiques réelles sont souvent moins nettes que dans ce schéma. Mais quand on examine les couches d'un réseau entraîné sur des images, on observe bien cette progression, de motifs simples vers des motifs complexes.

On peut voir ce que cherche un neurone d'un réseau entraîné. On part d'une image de bruit, et on la modifie pas à pas, par rétropropagation, pour activer le plus possible un neurone choisi. L'image obtenue montre ce à quoi ce neurone réagit. Le même procédé sert à fabriquer les exemples adverses, présentés plus loin, au chapitre « Tromper un réseau ». En 2017, Chris Olah, Alexander Mordvintsev et Ludwig Schubert, chez Google, ont appliqué cette méthode à toutes les couches d'un réseau convolutif entraîné sur ImageNet.

Ce que cherchent des neurones de couches de plus en plus profondes d'un réseau entraîné sur ImageNet (d'après Olah, Mordvintsev et Schubert, Distill, 2017, CC BY 4.0).

La progression décrite plus haut apparaît bien, sans que personne l'ait programmée.

DeepDream
En juin 2015, Alexander Mordvintsev, Christopher Olah et Mike Tyka, ingénieurs chez Google, publient des images produites par un réseau convolutif entraîné sur ImageNet. Ils partent d'une photo ordinaire et la modifient pas à pas, par rétropropagation, pour **renforcer** ce que le réseau croit y voir. Si une tache rappelle vaguement un œil à une couche du réseau, l'image est modifiée pour ressembler davantage à un œil. On recommence, encore et encore. Comme ImageNet contient beaucoup de photos de chiens, des têtes de chiens, des yeux et des pattes finissent par apparaître partout : dans les nuages, sur les visages, dans les feuillages.

Une photo de méduses transformée par DeepDream : à chaque itération, le réseau renforce les formes animales qu'il croit voir (images de Martin Thoma, Wikimedia Commons, CC0).

Les chercheurs publient leur programme, et des milliers de personnes l'appliquent à leurs propres photos. Les images circulent dans le monde entier, et beaucoup y voient pour la première fois ce qui se passe à l'intérieur d'un réseau de neurones. Le phénomène rappelle la **paréidolie** (*pareidolia*), notre tendance à voir des visages dans les nuages ou les prises électriques. Le réseau fait de même avec ce qu'il a appris à reconnaître, mais en l'exagérant à chaque itération. DeepDream a aussi ouvert la voie à des usages artistiques des réseaux de neurones, qui annoncent l'IA générative du Module 4.

La carte n'est pas le territoire
Cette hiérarchie rappelle celle du cerveau. Dans les années 1960, David Hubel et Torsten Wiesel ont montré que le cortex visuel traite l'image par étapes : certains neurones réagissent à des traits orientés, d'autres, plus loin, à des formes. Les réseaux profonds se sont inspirés de ce résultat.
La ressemblance s'arrête là. Comme l'expliquait « D'où vient le mot « neurone » », un neurone artificiel est une formule. Un neurone biologique émet des impulsions électriques dans le temps, et son comportement dépend de nombreux signaux chimiques. Rien n'indique que le cerveau utilise la rétropropagation. Enfin, un enfant reconnaît un chat après en avoir vu quelques-uns, avec un cerveau qui consomme environ 20 watts, alors qu'un réseau a besoin de milliers d'exemples et de beaucoup plus d'énergie.
Un réseau de neurones est un modèle mathématique inspiré du cerveau. Ce n'est pas une copie du cerveau, et son fonctionnement ne nous apprend que peu de choses sur le nôtre.

La fin des caractéristiques fabriquées à la main

Jusqu'en 2012, un système de reconnaissance d'images se construisait en deux étapes. Des spécialistes concevaient d'abord des caractéristiques : des détecteurs de contours, de coins, de textures. Ces caractéristiques étaient ensuite données à un modèle simple, du type de ceux du Module 2. L'essentiel du travail, et de la performance, tenait à la première étape. Elle demandait des années de mise au point pour chaque type de données.

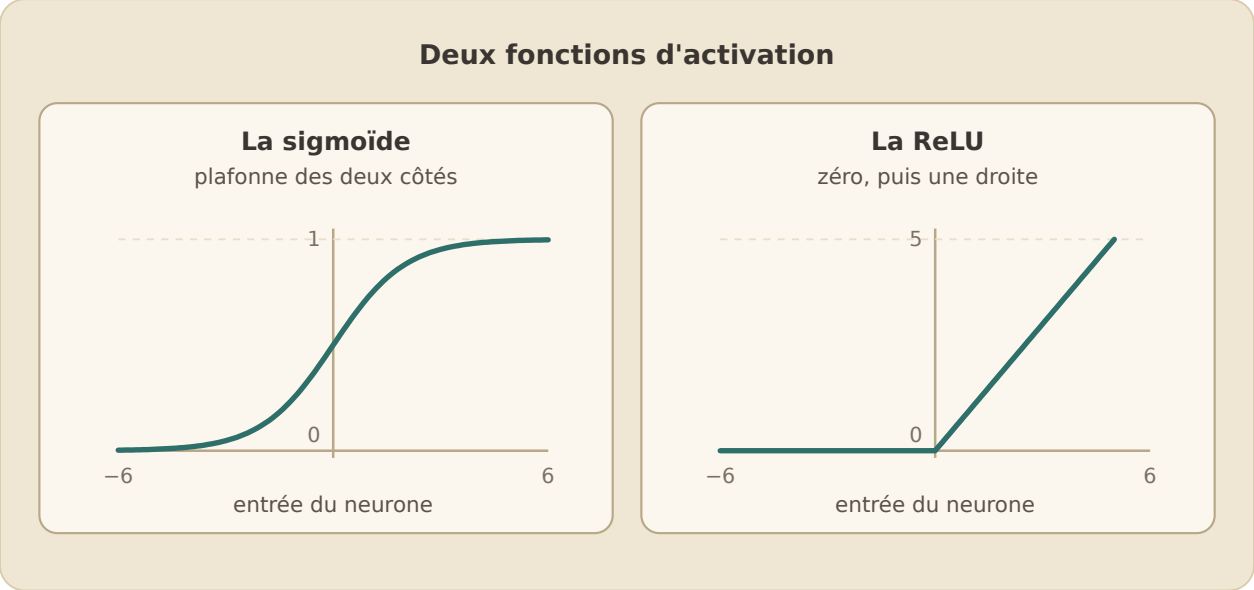
Un réseau profond supprime cette première étape. Il reçoit les pixels bruts, et il apprend à la fois les caractéristiques et la décision. On parle d'apprentissage **de bout en bout** (*end-to-end* en anglais). Le

travail humain se déplace : il ne consiste plus à concevoir des caractéristiques, mais à rassembler des données et à choisir la forme du réseau.

Pourquoi seulement en 2012

L'idée d'empiler des couches est ancienne, et la rétropropagation s'applique à un réseau de n'importe quelle profondeur. Trois obstacles ont pourtant bloqué les réseaux profonds pendant plus de vingt ans.

Le gradient qui s'évanouit. Pendant la rétropropagation, l'erreur remonte de couche en couche. Avec la sigmoïde, elle s'affaiblit à chaque couche traversée. Après quelques couches, il n'en reste presque rien, et les premières couches n'apprennent plus. Ce problème s'appelle l'**évanouissement du gradient** (*vanishing gradient*). Un des remèdes est de remplacer la sigmoïde par une fonction plus simple, la **ReLU** : elle donne 0 quand son entrée est négative, et laisse passer l'entrée telle quelle quand elle est positive. Avec elle, l'erreur traverse les couches sans s'affaiblir autant.



Deux fonctions d'activation : la sigmoïde, qui plafonne des deux côtés, et la ReLU.

Les données. Un réseau profond a des millions de paramètres, et il lui faut des millions d'exemples. Ils n'existaient pas. En 2009, l'équipe de Fei-Fei Li, à Stanford, publie **ImageNet** : 14 millions d'images, classées à la main dans plus de 20 000 catégories. Ce travail a été réalisé par des dizaines de milliers de personnes, sur la plateforme Mechanical Turk. C'est un exemple de l'industrie de l'étiquetage décrite au Module 2.

Quelques centaines d'images d'ImageNet. Elles sont disposées par un réseau profond, qui a placé côte à côte celles qu'il juge semblables (*image : Andrej Karpathy*).

Le calcul. Entraîner un réseau profond sur des millions d'images demande une quantité de calculs hors de portée des processeurs des années 2000. La solution est venue du jeu vidéo, avec les **processeurs graphiques** (*graphics processing units*, GPU), qui entraînent un réseau des dizaines de fois plus vite qu'un processeur ordinaire. Des bibliothèques logicielles ont ensuite automatisé la rétropropagation. Le chapitre « Le matériel et les outils » raconte ces deux histoires.

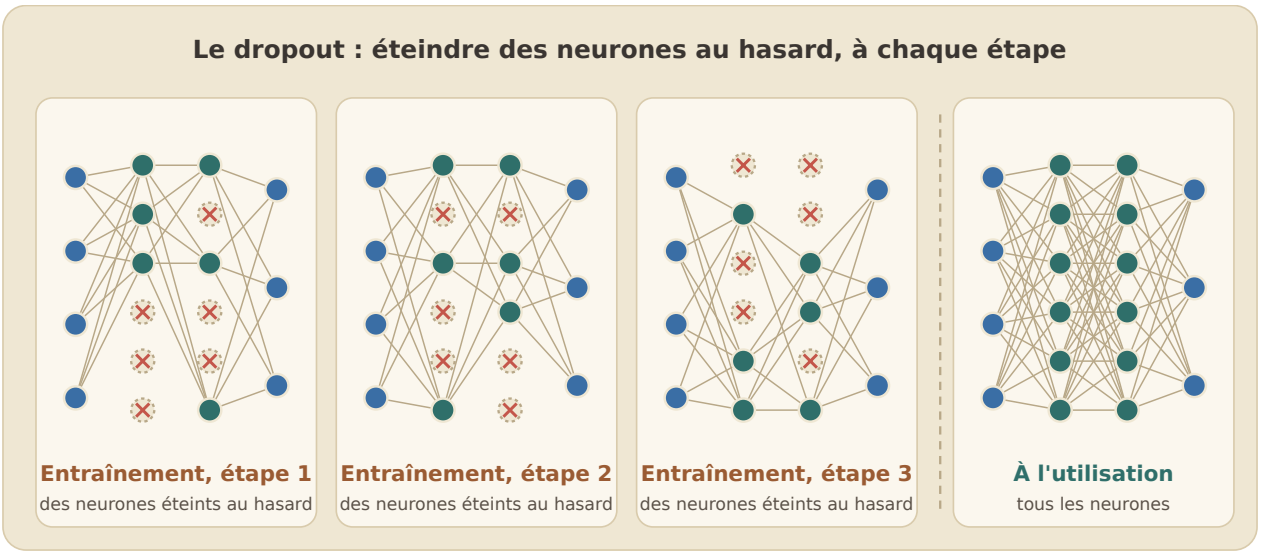
Un premier contournement, en 2006. Avant que ces trois obstacles soient levés, Hinton propose un détour. Au lieu d'entraîner toutes les couches d'un coup, on les entraîne une à une, sans étiquettes : chaque couche apprend à résumer ce que lui transmet la précédente, un peu comme un autoencodeur. Une fois toutes les couches en place, la rétropropagation ajuste l'ensemble. Ce **préentraînement couche par couche** (*greedy layer-wise pre-training*), publié par Hinton et ses collègues en 2006 et repris aussitôt par l'équipe de Bengio, montre qu'on peut enfin entraîner des réseaux à plusieurs couches cachées, et qu'ils font mieux que les réseaux peu profonds sur plusieurs tâches. C'est à cette époque que le nom *deep learning* s'impose. Quelques années plus tard, avec la ReLU, de grandes bases de données et les GPU, le détour devient inutile : AlexNet s'en passe. Mais il avait ramené l'attention sur les réseaux profonds.

2012 : le concours ImageNet

À partir de 2010, ImageNet sert de base à un concours annuel. Il faut classer des images parmi 1 000 catégories, après un entraînement sur 1,2 million d'exemples. Une réponse est comptée comme une erreur si la bonne catégorie ne figure pas parmi les cinq propositions du système.

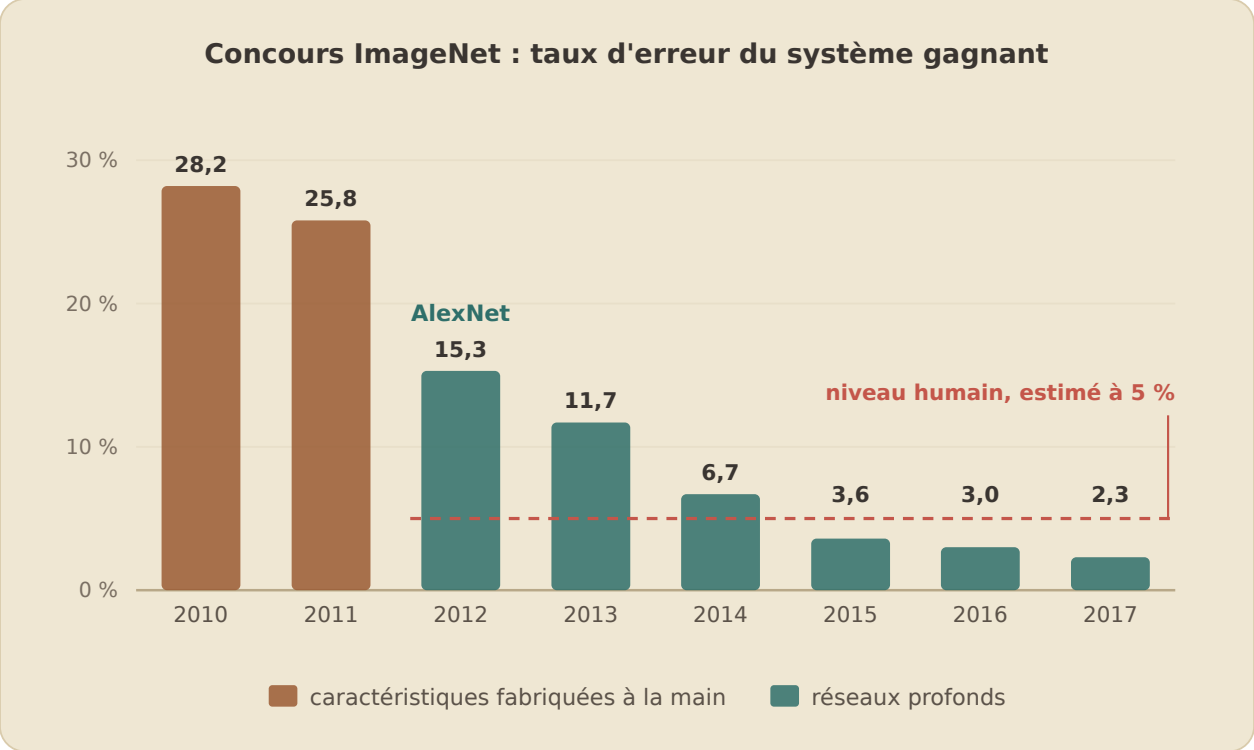
En 2010 et 2011, les meilleurs systèmes reposent sur des caractéristiques fabriquées à la main (*hand-crafted features*), et leur taux d'erreur est de 28 %, puis de 26 %. En 2012, Alex Krizhevsky, Ilya Sutskever et Geoffrey Hinton, de l'Université de Toronto, présentent un réseau profond, **AlexNet**. Il compte huit couches et 60 millions de paramètres, il utilise la ReLU, et il a été entraîné pendant environ une semaine sur deux GPU.

Avec 60 millions de paramètres pour 1,2 million d'images, le risque de sur-apprentissage (*overfitting*) est grand. AlexNet le limite par une forme de régularisation, au sens du Module 2 : le réseau reste riche, mais on l'empêche de trop coller à ses données d'entraînement. Il utilise une méthode toute récente, publiée par l'équipe de Hinton la même année, le **dropout**. À chaque étape de l'entraînement, une partie des neurones, tirés au hasard, est mise hors service. Le réseau ne peut donc pas compter sur un neurone en particulier, et il apprend des caractéristiques plus robustes, un peu comme une équipe où chacun doit pouvoir remplacer un absent. Les images d'entraînement sont aussi recadrées et retournées au hasard, ce qui en multiplie les variantes.



Le dropout : à chaque étape de l'entraînement, d'autres neurones sont éteints au hasard. Une fois l'entraînement terminé, le réseau utilise tous ses neurones.

Le taux d'erreur d'AlexNet est de 15 %. Le deuxième système est à 26 %.



Le taux d'erreur du système gagnant au concours ImageNet, de 2010 à 2017.

L'écart est si grand que le domaine change de méthode en deux ans. Dès 2014, tous les systèmes en compétition sont des réseaux profonds. En 2015, le réseau gagnant compte 152 couches et son taux d'erreur, 3,6 %, est inférieur à celui d'un humain sur la même tâche, estimé à 5 %. La même bascule se produit quelques années plus tard pour la traduction. Pour la reconnaissance de la parole, elle avait même commencé un peu plus tôt. Entre 2009 et 2012, des étudiants de Hinton, avec des équipes de Microsoft, de Google et d'IBM, remplacent une partie des systèmes de reconnaissance vocale par des réseaux profonds, préentraînés couche par couche. En 2012, la recherche vocale des téléphones Android en profite. C'est le premier grand succès industriel de l'apprentissage profond. ImageNet a pourtant davantage frappé les esprits, parce que l'écart y était spectaculaire et mesuré lors d'un concours public.

AlexNet est un réseau d'un type particulier, un réseau convolutif (*convolutional neural network*, CNN), conçu pour les images. Le chapitre « Voir : les réseaux convolutifs » lui est consacré.

Une école canadienne
AlexNet est sorti du laboratoire de Geoffrey Hinton, à l'Université de Toronto. Ce n'est pas un hasard. Pendant les années 1990 et 2000, les réseaux de neurones intéressaient peu, et les organismes qui finançaient la recherche s'en détournaient. En 2004, l'Institut canadien de recherches avancées (CIFAR) lance un programme intitulé « Calcul neuronal et perception adaptative » (*Neural Computation and Adaptive Perception*), dirigé par Hinton. Le programme réunit un petit groupe de chercheurs convaincus, dont Yoshua Bengio, à l'Université de Montréal, et Yann Le Cun, à l'Université de New York. Ils continuent à travailler sur les réseaux profonds quand presque personne d'autre ne le fait.

Le pari réussit en 2012. Hinton, Bengio et Le Cun reçoivent ensemble le prix Turing 2018. En 2024, Hinton reçoit le prix Nobel de physique avec l'Américain John Hopfield, « pour des découvertes et des inventions fondamentales qui permettent l'apprentissage automatique avec des réseaux de neurones artificiels ». Le comité récompense deux réseaux des années 1980 qui empruntent leurs outils à la physique statistique : le réseau de Hopfield (1982), une mémoire qui retrouve une image complète à partir d'un fragment, et la machine de Boltzmann de Hinton, qui tire son nom du physicien Ludwig Boltzmann. Le choix a surpris, et plusieurs physiciens se sont demandé si l'on récompensait encore de la physique. Hinton, qui avait quitté Google en 2023 pour parler librement des risques de l'IA, a profité de la tribune pour les rappeler. Le Module 5 revient sur ces risques. Le laboratoire fondé par Bengio en 1993 devient en 2017 **Mila**, l'Institut québécois d'intelligence artificielle, l'un des plus grands centres de recherche universitaire en apprentissage profond au monde. La même année, le gouvernement du Canada confie au CIFAR la Stratégie pancanadienne en matière d'intelligence artificielle, l'une des premières stratégies nationales consacrées à l'IA. Elle s'appuie sur trois instituts : Mila à Montréal, l'Institut Vecteur à Toronto et l'Amii à Edmonton, où travaille Richard Sutton, l'auteur de la leçon amère.

Geoffrey Hinton, Yoshua Bengio et Yann Le Cun, lauréats du prix Turing 2018 (photos : Ramsey Cardy / Collision via Sportsfile, CC BY 2.0 ; Maryse Boyce, CC BY 4.0 ; Jérémy Barande, CC BY-SA 2.0 ; Wikimedia Commons).

La leçon amère

La fin du Module 2 a présenté l'essai de Richard Sutton, « The Bitter Lesson » (2019). Sa thèse est la suivante. Dans l'histoire de l'IA, les chercheurs ont d'abord tenté d'inscrire leur propre savoir dans les systèmes. Cela aidait à court terme. À long terme, ces systèmes ont été dépassés par des méthodes générales, qui n'utilisent aucun savoir particulier, mais qui tirent parti de la puissance de calcul disponible.

Le concours de 2012 en est l'exemple le plus net. Les caractéristiques d'images mises au point par des experts pendant vingt ans ont été dépassées par un réseau qui apprend les siennes. Le Module 1 en donnait un autre exemple, avec Deep Blue : la recherche à grande échelle l'avait emporté sur les programmes qui tentaient d'imiter le raisonnement des grands maîtres.

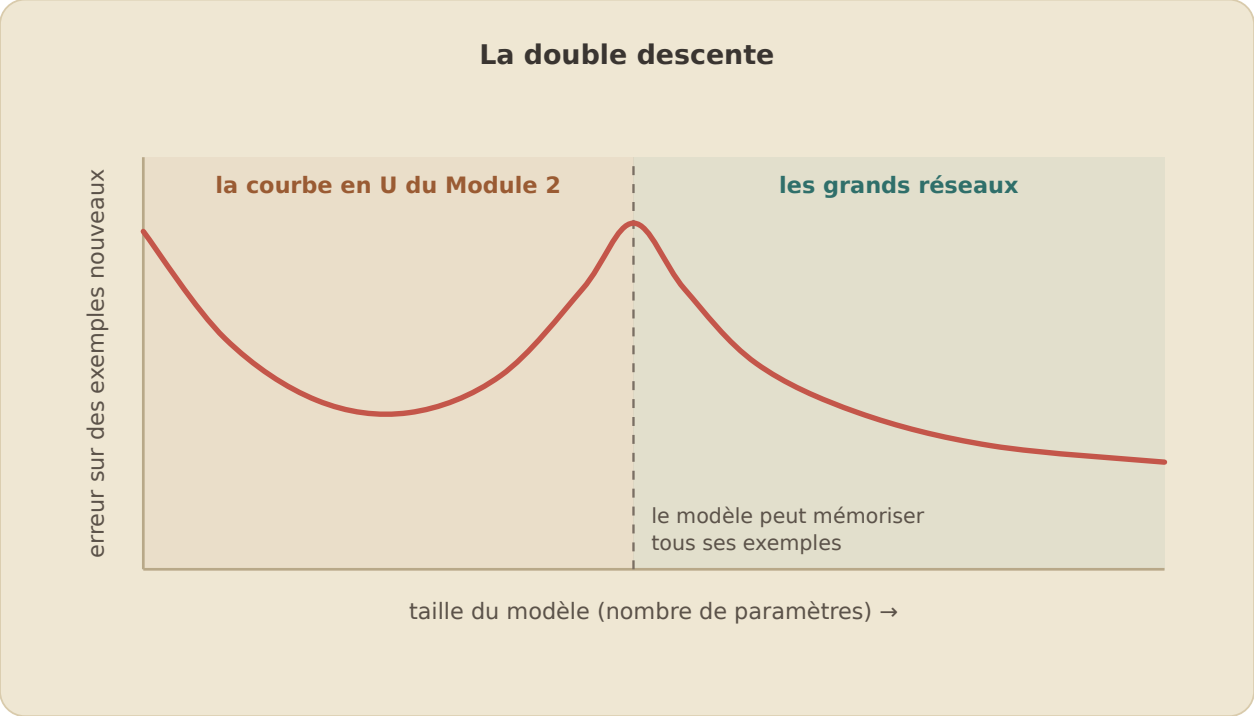
Sutton qualifie cette leçon d'amère parce qu'elle va contre l'intuition des chercheurs : le savoir qu'ils inscrivent dans un système finit par le limiter. Elle a aussi un coût. Un réseau profond ne donne pas ses raisons, alors qu'on pouvait lire les règles d'un système expert ou les questions d'un arbre de décision (*decision tree*). Et ses progrès dépendent de quantités de données et de calcul que peu d'organisations possèdent.

La leçon ne dit pas que le savoir humain est inutile. Il reste présent, mais ailleurs : dans le choix des données, et dans la forme donnée au réseau. Les chapitres sur les réseaux convolutifs, les réseaux récurrents et l'attention en donnent des exemples.

Plus de paramètres que d'exemples

AlexNet a 60 millions de paramètres, pour 1,2 million d'images d'entraînement. D'après « Généraliser », un modèle aussi souple devrait apprendre ses exemples par cœur et échouer sur des images nouvelles. C'est le sur-apprentissage (*overfitting*). Or les réseaux profonds généralisent bien. C'est la question que le Module 2 avait laissée ouverte.

Une partie de la réponse tient à la régularisation, décrite dans « Entraîner un réseau ». Mais elle ne suffit pas à expliquer le phénomène. Quand on fait grossir un réseau, on observe d'abord la courbe en U du Module 2 : l'erreur sur les exemples nouveaux baisse, puis remonte. Si l'on continue au-delà du point où le réseau peut mémoriser tous ses exemples, l'erreur redescend, parfois plus bas qu'avant. Ce phénomène s'appelle la **double descente** (*double descent*).



La double descente : après le pic, un modèle plus grand généralise mieux.

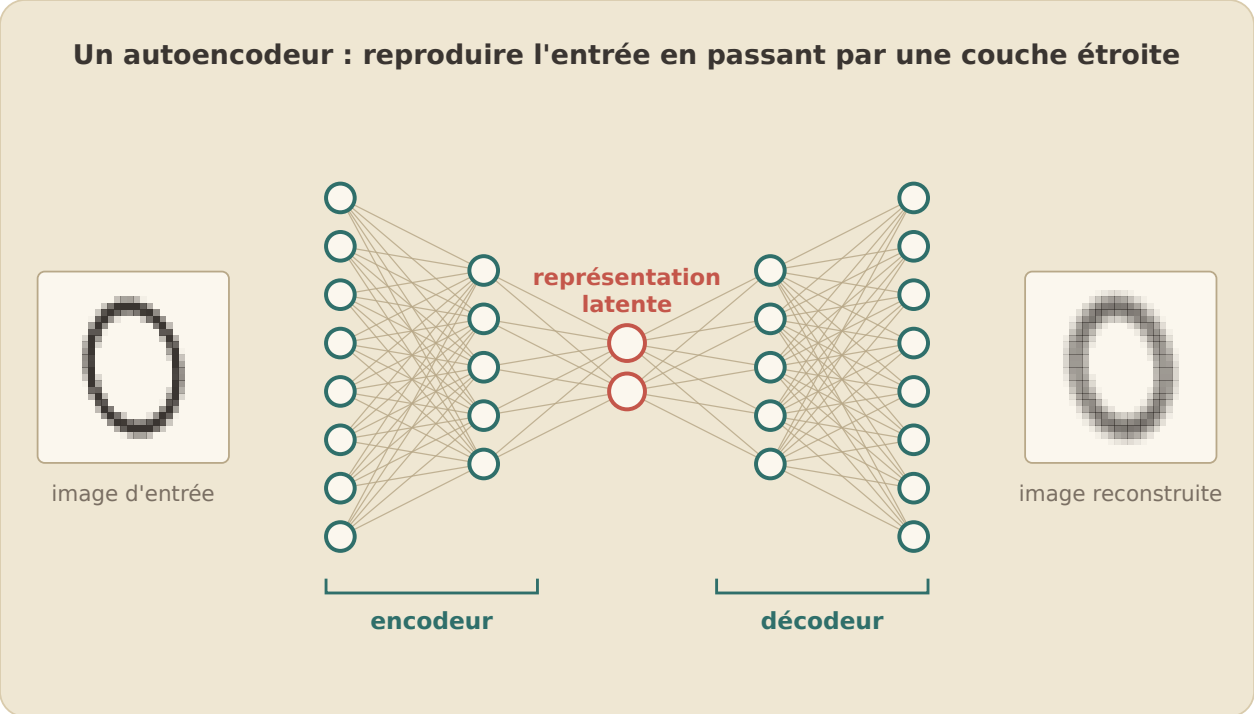
Le phénomène est bien établi, mais son explication n'est pas complète. L'hypothèse la plus courante est la suivante. Un très grand réseau dispose de nombreuses façons de reproduire ses exemples, et la descente de gradient aboutit le plus souvent à l'une des plus simples. C'est un cas où la pratique a précédé la théorie.

Un phénomène voisin a été observé en 2022 par une équipe d'OpenAI, cette fois au fil de l'entraînement plutôt qu'en fonction de la taille. Sur de petites tâches d'arithmétique, un réseau apprend d'abord ses exemples par cœur : il ne fait plus aucune erreur sur eux, mais répond au hasard sur les nouveaux. Si l'on poursuit l'entraînement bien au-delà, parfois des milliers d'étapes plus tard, il se met soudain à généraliser presque parfaitement. Les chercheurs ont appelé ce phénomène ***grokking***, d'après un mot inventé par l'écrivain Robert Heinlein, qui signifie « comprendre en profondeur ».

Apprendre sans étiquettes : l'autoencodeur

Tous les réseaux vus jusqu'ici sont entraînés avec des exemples étiquetés (*labeled examples*). Un réseau peut aussi apprendre des caractéristiques sans étiquettes. L'**autoencodeur** (*autoencoder*) en est l'exemple le plus simple.

Un autoencodeur est un réseau en forme de sablier. On lui donne une image en entrée, et on lui demande de produire la même image en sortie. L'erreur est l'écart entre les deux. La tâche serait sans intérêt si le réseau pouvait simplement recopier son entrée, mais la couche du milieu est très étroite. Les 784 pixels d'un chiffre doivent passer par quelques dizaines de neurones seulement.



Un autoencodeur : l'image doit passer par une couche étroite, puis être reconstruite.

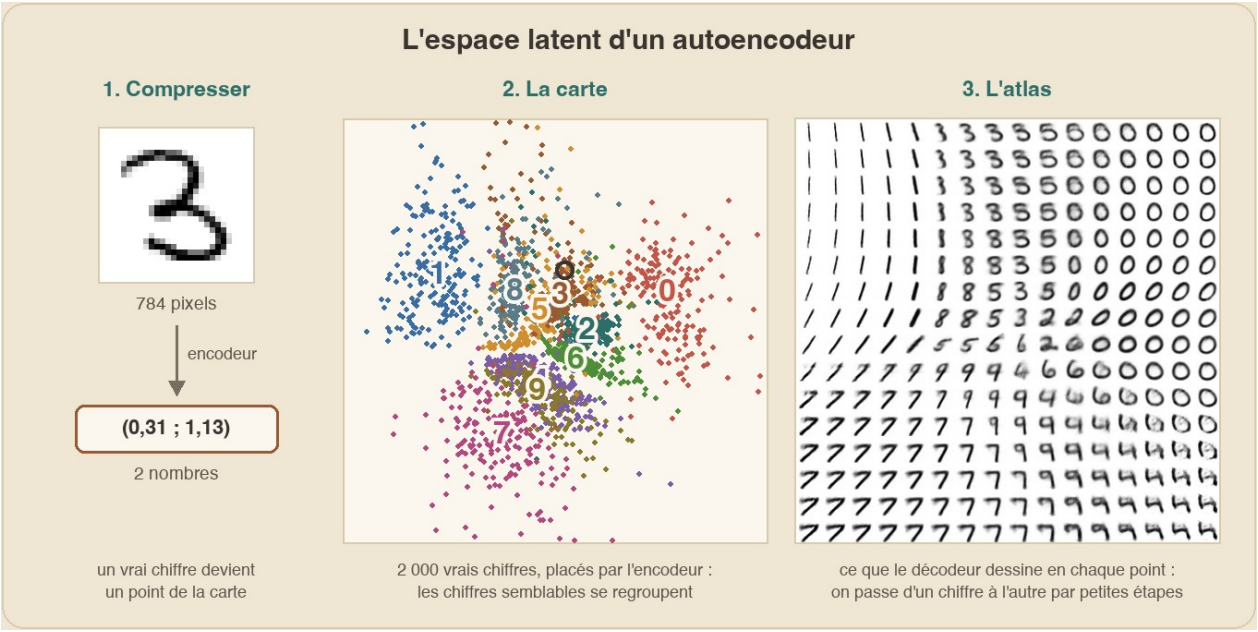
- La première moitié, l'**encodeur**, réduit l'image à quelques nombres.
- La seconde moitié, le **décodeur**, reconstruit l'image à partir de ces nombres.

L'autoencodeur est un premier exemple d'assemblage : deux réseaux, l'un qui comprime et l'autre

qui reconstruit, branchés l'un à la suite de l'autre et entraînés ensemble, comme les pièces du jeu de construction présenté au chapitre suivant.

Pour réussir, le réseau doit garder l'essentiel de l'image dans la couche étroite : quel chiffre, quelle inclinaison, quelle épaisseur de trait. Ces quelques nombres forment une représentation compacte de l'image, qu'on appelle sa **représentation latente** (*latent representation*). C'est un cas d'auto-supervision (*self-supervised learning*), présentée au Module 2 : les données fournissent elles-mêmes la réponse attendue.

L'ensemble des valeurs que peuvent prendre ces quelques nombres forme l'**espace latent** (*latent space*) du réseau. C'est un espace **compressé** : quelques dizaines de dimensions au lieu de 784, où ne reste que ce qui distingue une image d'une autre. Deux images semblables y occupent des points voisins, et l'on peut s'y déplacer d'un chiffre à l'autre par petites étapes. Beaucoup de chercheurs y voient l'endroit où un réseau travaille vraiment : il n'y manipule plus des pixels, mais des propriétés abstraites, comme la forme ou l'inclinaison d'un chiffre. Certains, comme Yann Le Cun, pensent même qu'une intelligence artificielle devrait raisonner et faire ses prévisions dans un tel espace, plutôt que sur les données brutes. Le Module 4 permet d'explorer l'espace latent d'un réseau entraîné sur des chiffres manuscrits.



L'espace latent d'un autoencodeur entraîné sur des chiffres manuscrits. Pour pouvoir le dessiner, ce réseau n'en a que deux dimensions, alors qu'un autoencodeur ordinaire en a quelques dizaines. C'est le même réseau que celui de l'applet du Module 4.

Le décodeur a une autre utilité. Si on lui donne des nombres qui ne viennent d'aucune image réelle, il produit quand même une image. C'est une première façon de générer du contenu, sur laquelle le Module 4 reviendra.

Un réseau profond peut donc apprendre ses propres caractéristiques, à condition d'avoir assez de données et de calcul. Le chapitre suivant, « Le matériel et les outils », raconte comment ce calcul est devenu possible.

Le matériel et les outils

Le chapitre « L'apprentissage profond » a nommé trois conditions du succès de 2012 : une meilleure fonction d'activation, des données, et du calcul. Ce chapitre revient sur la troisième. Deux progrès techniques ont rendu l'apprentissage profond (*deep learning*) praticable : des processeurs capables de faire les calculs, et des bibliothèques logicielles qui écrivent la rétropropagation à la place du programmeur.

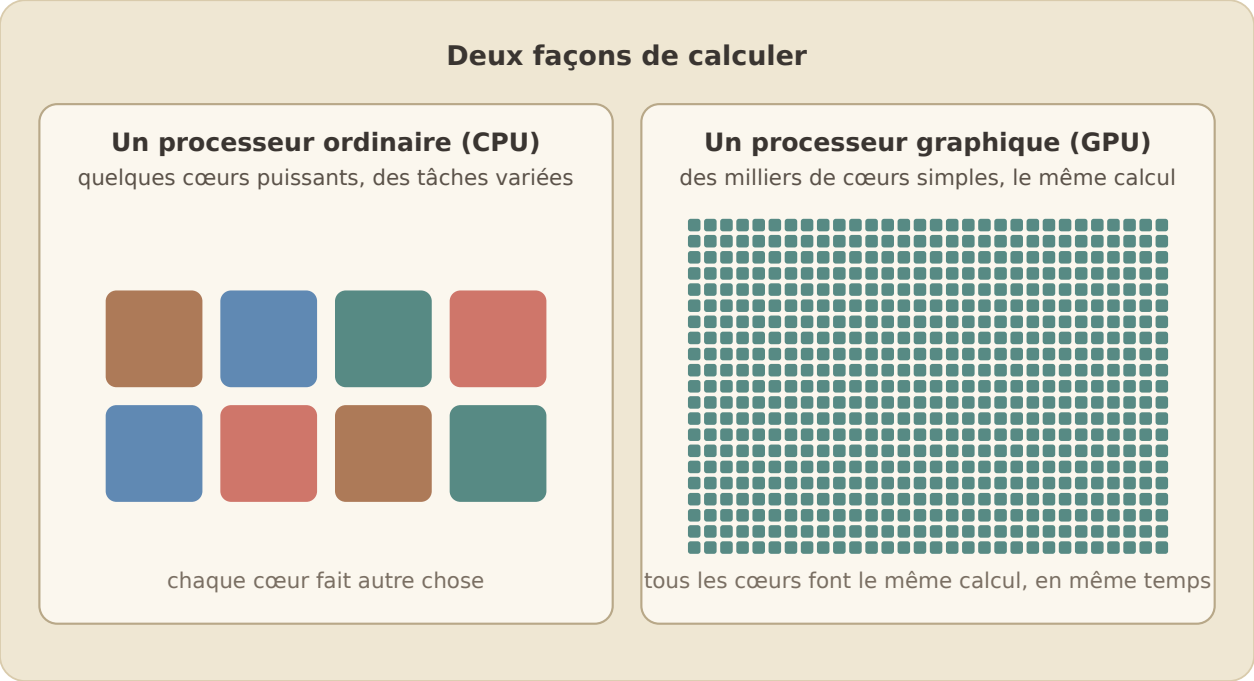
Le matériel et le logiciel de l'apprentissage profond : deux histoires qui se rejoignent en 2012.

Les processeurs graphiques

Un réseau de neurones répète toujours le même calcul : multiplier des entrées par des poids et additionner les résultats, pour chaque neurone, des millions de fois. Ces calculs sont indépendants les uns des autres, et peuvent donc être faits en même temps.

Un processeur ordinaire (CPU) a quelques cœurs puissants, conçus pour exécuter des tâches variées l'une après l'autre. Un **processeur graphique** (GPU) a des milliers de cœurs simples, qui exécutent

tous le même calcul en parallèle. Il a été conçu pour l'affichage des jeux vidéo, où il faut calculer la couleur de millions de pixels des dizaines de fois par seconde, ce qui est un problème du même type.



Un processeur ordinaire a quelques cœurs puissants ; un processeur graphique a des milliers de cœurs simples.

Voici les principales étapes de cette histoire.

- **1993.** Fondation de Nvidia, qui fabrique des cartes graphiques pour le jeu vidéo.
- **1999.** Nvidia lance la GeForce 256 et la présente comme le premier « GPU ».
- **Début des années 2000.** Les cartes deviennent programmables. Des chercheurs s'en servent pour des calculs scientifiques, en présentant leurs calculs comme des images à afficher. La méthode est laborieuse.
- **2007.** Nvidia publie CUDA, un outil qui permet de programmer un GPU directement, pour n'importe quel calcul. L'entreprise investit dans cet usage alors qu'il n'a presque aucun marché.
- **2009.** À Stanford, l'équipe d'Andrew Ng montre qu'un GPU entraîne un réseau de neurones jusqu'à 70 fois plus vite qu'un CPU.
- **2012.** AlexNet est entraîné sur deux cartes de jeu vidéo vendues environ 500 $ chacune, dans la chambre d'Alex Krizhevsky, chez ses parents.
- **2016.** Nvidia vend ses premières machines conçues pour l'apprentissage profond. Google présente ses propres puces spécialisées, les TPU.
- **Années 2020.** Les grands modèles sont entraînés dans des centres de données qui contiennent des dizaines de milliers de GPU. Nvidia fournit la majorité de ces puces. En 2025, elle devient la première entreprise dont la valeur en bourse dépasse 4 000, puis 5 000 milliards de dollars. L'accès à ces puces devient un enjeu politique : depuis 2022, les États-Unis en limitent l'exportation vers la Chine. Le Module 5 reviendra sur ces questions.

Une carte GeForce GTX 580, conçue pour le jeu vidéo. AlexNet a été entraîné sur deux cartes de ce modèle (photo : TheStriker, Wikimedia Commons, CC BY-SA 4.0).

La différentiation automatique

Le second progrès est logiciel. La rétropropagation demande de calculer des dérivées à travers toutes les couches du réseau. Pendant longtemps, ce calcul a été fait à la main : pour chaque nouveau réseau, il fallait établir les formules sur papier, puis les programmer. Le travail prenait des semaines, et les erreurs étaient fréquentes et difficiles à repérer.

La **différentiation automatique** (*automatic differentiation*) supprime ce travail. Le programmeur décrit seulement la propagation avant, c'est-à-dire le calcul qui mène des entrées à l'erreur. La bibliothèque enregistre chaque opération, et en déduit elle-même le calcul de la rétropropagation.

- **1970.** Le Finlandais Seppo Linnainmaa décrit la méthode mathématique générale, dont la rétropropagation est un cas particulier.
- **2002.** Torch, une des premières bibliothèques destinées aux réseaux de neurones. Elle fournit des couches toutes faites, mais les dérivées de chaque type de couche sont encore écrites à la main par ses auteurs.
- **2007.** Theano, développée à l'Université de Montréal dans le laboratoire de Yoshua Bengio. C'est la première bibliothèque largement utilisée qui réunit les deux progrès : elle calcule les dérivées automatiquement, et elle exécute les calculs sur GPU.
- **2013.** Caffe, à Berkeley, conçue pour les réseaux qui traitent des images. Elle se répand vite après le succès d'AlexNet.
- **2015.** Google publie TensorFlow en code ouvert (*open source*). La même année paraît Keras, qui permet de décrire un réseau en quelques lignes.
- **2016.** Facebook publie PyTorch. On y écrit un réseau comme un programme Python ordinaire. Elle devient en quelques années l'outil le plus utilisé en recherche.

Un réseau en quelques lignes

Voici à quoi ressemble aujourd'hui le réseau des chiffres du chapitre « Une couche cachée », écrit en **Python** avec la bibliothèque PyTorch. Python est le langage de programmation le plus utilisé en apprentissage automatique. Il est simple à lire et à écrire, et des bibliothèques comme PyTorch s'y chargent des calculs lourds, programmés dans des langages plus rapides et exécutés sur les processeurs graphiques. Plusieurs de ses traits viennent d'ailleurs de Lisp, le langage de l'IA symbolique, comme le montre *L'héritage invisible* au Module 1. Il n'est pas nécessaire de comprendre le détail du code. Il suffit de constater sa longueur.

```
reseau = nn.Sequential(
    nn.Linear(784, 30),    # 784 entrées vers 30 neurones cachés
    nn.ReLU(),             # la fonction d'activation
    nn.Linear(30, 10),     # 30 neurones cachés vers 10 sorties
)

sorties = reseau(images)                      # la propagation avant
erreur = cross_entropy(sorties, etiquettes)  # l'écart avec les bonnes réponses
erreur.backward()                             # la rétropropagation, faite par la bibliothèque
optimiseur.step()                             # la correction de tous les poids
```

La ligne `erreur.backward()` remplace les semaines de calcul à la main. La conséquence est importante pour la suite. Essayer une nouvelle forme de réseau ne demande plus que quelques heures. Les chercheurs ont donc pu en essayer beaucoup, et c'est ainsi que se sont répandues les architectures des chapitres suivants.

Un jeu de construction

L'exemple précédent assemble trois blocs : une couche, une fonction d'activation, une autre couche. Rien n'oblige à s'en tenir là. On peut empiler d'autres couches, en ajouter de types différents, faire passer les données par deux chemins puis les réunir, ou brancher deux réseaux l'un à la suite de l'autre. Quel que soit l'assemblage, la différentiation automatique calcule comment l'erreur dépend de chaque poids, et le tout s'entraîne d'un seul coup, par la même rétropropagation.

C'est l'une des grandes forces des réseaux de neurones. Ils se composent comme les pièces d'un jeu de construction, et chaque nouvel assemblage s'entraîne sans qu'il faille inventer une nouvelle méthode. En 2018, Yann Le Cun a proposé de parler de **programmation différentiable** (*differentiable programming*) : on écrit un programme fait de blocs ajustables, et ce sont les données qui en règlent les paramètres.

Quelques sortes de briques, toujours les mêmes, et des assemblages sans fin : c'est l'image qu'on donne souvent des réseaux de neurones (photo : Alan Chia, Wikimedia Commons, CC BY-SA 2.0).

Les chapitres suivants présentent les assemblages qui ont le plus compté. Le premier, présenté dans « Voir : les réseaux convolutifs », est conçu pour les images.

Voir : les réseaux convolutifs

Le réseau des chiffres du chapitre « Une couche cachée » reçoit les 784 pixels d'une image comme une simple liste de nombres. Il ne sait pas que ces pixels forment une grille, ni que deux pixels voisins sont liés. Ce chapitre présente une forme de réseau qui tient compte de la disposition des pixels, le **réseau convolutif**.

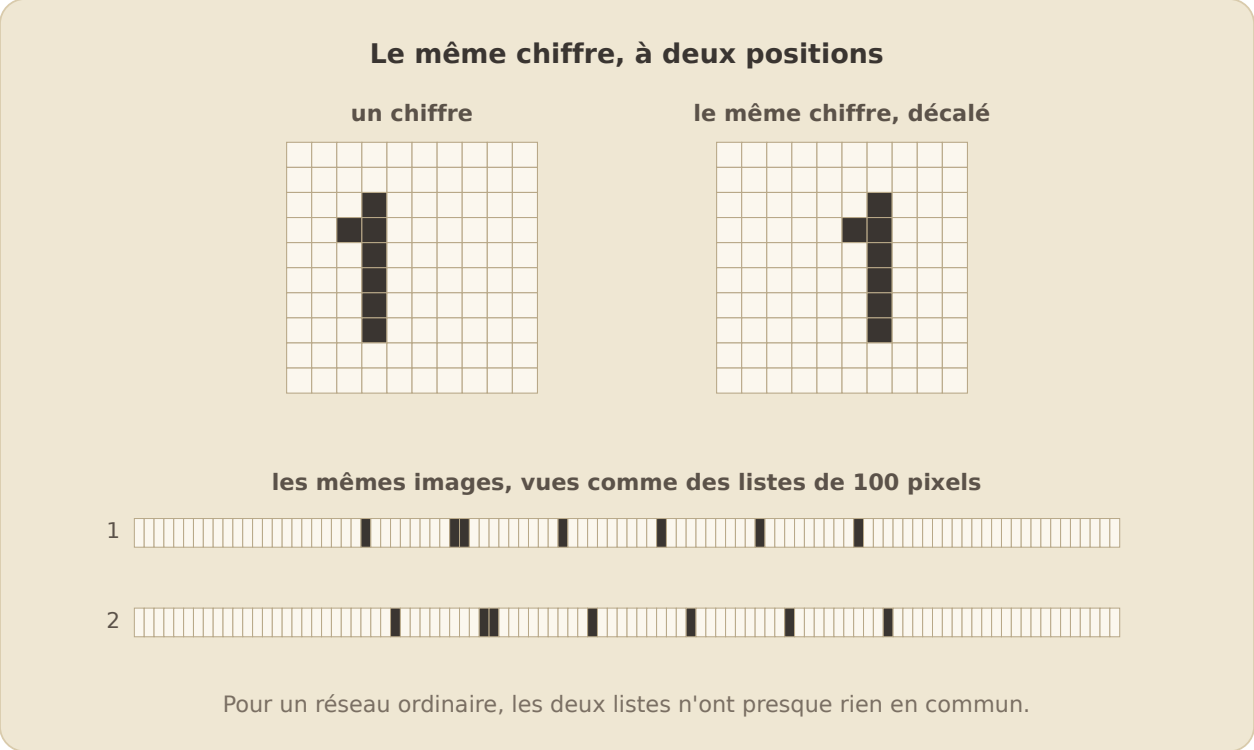
Ce réseau est fait des mêmes briques que le perceptron multicouche : des neurones, des poids, la même rétropropagation. Seule change son architecture, c'est-à-dire la façon dont ses neurones sont reliés. Chaque neurone n'y regarde qu'un petit voisinage de l'image, et les mêmes poids servent partout. Ce choix traduit une hypothèse sur les données : dans une image, ce qui compte est d'abord local, et une même forme peut apparaître n'importe où. Une hypothèse de ce genre, inscrite dans l'architecture avant tout apprentissage, s'appelle un **biais inductif** (*inductive bias*). Comme le biais du Module 2, il restreint ce que le réseau peut apprendre. S'il est juste, il lui permet d'apprendre avec beaucoup moins d'exemples.

Ce qu'un réseau ordinaire ignore dans une image

Un réseau dont chaque neurone est relié à toutes les entrées pose trois problèmes quand ces entrées sont les pixels d'une image.

Il ignore le voisinage. Supposons qu'on mélange les pixels de toutes les images, toujours de la même façon. Pour nous, les chiffres deviennent illisibles. Pour ce réseau, rien ne change : il apprend aussi bien qu'avant. Il n'utilise donc pas le fait que des pixels voisins forment ensemble un trait ou une courbe.

Il ne reconnaît pas un motif déplacé. Si un chiffre est décalé de quelques pixels, il active d'autres entrées, reliées à d'autres poids. Pour le réseau, c'est une image nouvelle. Il doit apprendre séparément le même chiffre à chaque position.



Le même chiffre à deux positions : une fois mises à plat, les deux images n'ont presque aucun pixel allumé en commun.

Il a trop de poids. Une photo en couleur de 1 000 pixels sur 1 000 compte trois millions de valeurs. Avec une couche cachée de 1 000 neurones, il faudrait trois milliards de poids pour cette seule couche.

Le filtre

La solution repose sur un petit élément, le **filtre** (*filter* ou *kernel*). Un filtre est un carré de poids, par exemple de 3 sur 3. On le pose sur un coin de l'image, où il recouvre neuf pixels. Il calcule la somme de ces neuf pixels, chacun multiplié par le poids qui le recouvre. C'est le calcul d'un neurone, limité à neuf entrées.

On fait ensuite glisser le filtre d'un pixel, et on recommence, jusqu'à avoir parcouru toute l'image. Chaque position donne un nombre. Ensemble, ces nombres forment une nouvelle image, presque de la même taille que la première. Cette opération s'appelle une **convolution**.

La nouvelle image indique où se trouve le motif que le filtre détecte. Un filtre dont la colonne du centre a des poids positifs et les colonnes voisines des poids négatifs donne une somme élevée là où l'image contient un trait vertical, et une somme faible ou négative ailleurs.

Pour aller plus loin : le calcul à une position
Prenons le filtre des traits verticaux, et une zone de l'image où passe un trait vertical. Les pixels du trait valent 1, les autres 0.

Filtre			Pixels		
-1	2	-1	0	1	0
-1	2	-1	0	1	0
-1	2	-1	0	1	0

La somme vaut $2 + 2 + 2 = 6$, la valeur la plus élevée possible pour ce filtre. Si la même zone contient un trait horizontal, les pixels de la rangée du milieu valent

1. La somme vaut alors $-1 + 2 - 1 = 0$, et le filtre ne détecte rien.

Un filtre à manipuler

Dans l'applet ci-dessous, l'image d'un chiffre est à gauche, le filtre au centre, et le résultat de la convolution à droite. Plus une case du résultat est foncée, plus le motif du filtre est présent à cet endroit. Les sommes négatives sont ramenées à zéro.

Applet interactive : à utiliser en ligne, sur la page « Voir : les réseaux convolutifs » du site du cours.

1. Choisissez le chiffre 7, puis comparez les filtres « traits verticaux », « traits horizontaux » et « obliques / ». Chacun fait ressortir une partie différente du chiffre.
2. Passez la souris sur l'image. Le carré rouge montre les neuf pixels recouverts par le filtre, et la somme obtenue s'affiche sous le filtre.
3. Déplacez le chiffre avec les flèches. Le résultat se déplace avec lui, sans changer de forme.
4. Modifiez vous-même les neuf poids du filtre, et observez l'effet.

Ce que la convolution apporte

Le filtre répond aux trois problèmes de départ.

- **Le voisinage.** Le filtre ne regarde que neuf pixels voisins à la fois. Il est fait pour détecter des motifs locaux.
- **Le déplacement.** Le même filtre parcourt toute l'image. Un trait vertical est détecté de la même façon en haut à gauche et en bas à droite, comme le montrait la troisième manipulation.
- **Le nombre de poids.** Un filtre de 3 sur 3 a neuf poids, que l'image compte 784 pixels ou trois millions. Les mêmes poids servent à toutes les positions. On parle de **partage des poids** (*weight sharing*).

Du filtre au réseau

Un réseau convolutif est fait de plusieurs éléments, répétés.

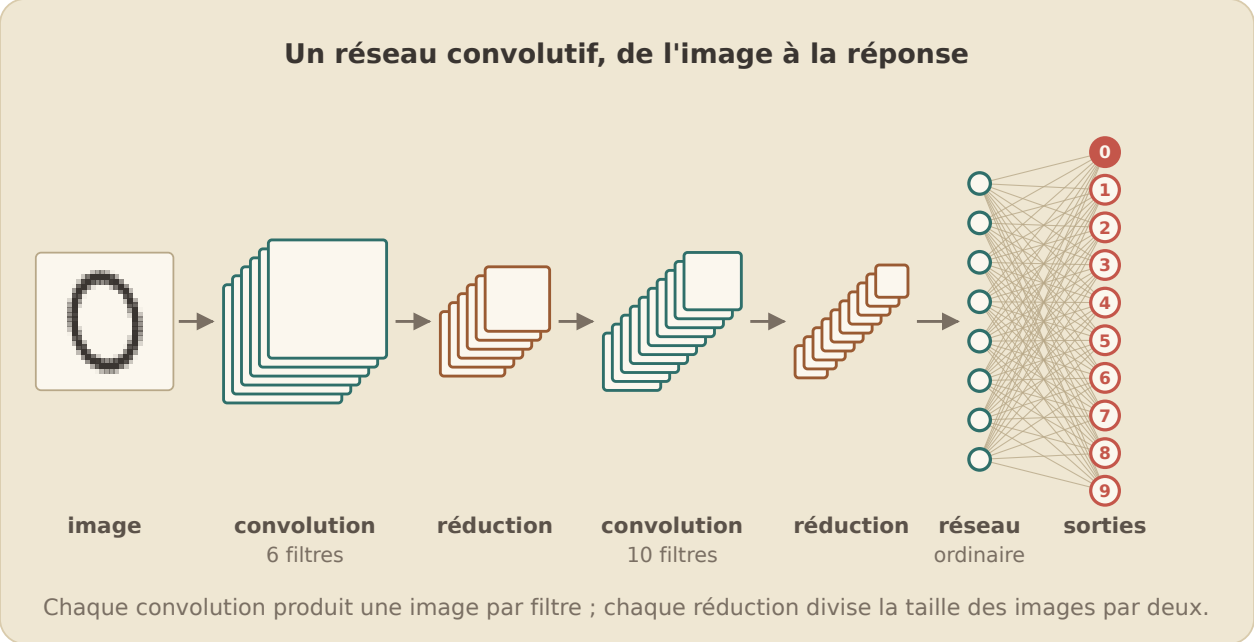
Une **couche convolutive** (*convolutional layer*) contient plusieurs filtres, par exemple 32. Chacun détecte un motif différent et produit sa propre image de résultat. La couche transforme donc une image en 32 images.

Une étape de **réduction** (*pooling* en anglais) diminue ensuite la taille de ces images. La méthode la plus courante découpe l'image en carrés de 2 sur 2 et ne garde que la valeur la plus élevée de chaque

carré. L'image devient deux fois plus petite dans chaque direction. On conserve l'information « ce motif est présent dans cette zone », et on perd sa position exacte.

On répète ces deux étapes plusieurs fois. Les filtres de la deuxième couche ne s'appliquent plus aux pixels, mais aux résultats de la première. Ils combinent donc des traits en formes. Ceux de la troisième combinent des formes en parties d'objets. C'est la hiérarchie de caractéristiques du chapitre « L'apprentissage profond », obtenue ici par construction.

À la fin, les dernières images sont petites et nombreuses. On les donne à un réseau ordinaire, à quelques couches, qui produit la réponse.



L'architecture d'un réseau convolutif : convolutions et réductions en alternance, puis un réseau ordinaire.

Il reste un point essentiel. Dans l'applet, les filtres étaient choisis à la main. Dans un réseau convolutif, personne ne les choisit. Les poids des filtres sont tirés au hasard au départ, puis réglés par rétropropagation, comme tous les autres poids. Le réseau trouve lui-même les motifs utiles pour sa tâche. Entraîné sur des visages, il produit des filtres adaptés aux visages. Entraîné sur des radiographies, il en produit d'autres.

Yann Le Cun et les chèques

L'idée du filtre qui glisse sur l'image vient de la neurophysiologie. En 1979, le chercheur japonais Kunihiko Fukushima s'inspire des travaux de Hubel et Wiesel sur le cortex visuel, décrits dans l'encadré du chapitre précédent. Il propose le **Neocognitron**, un réseau qui alterne déjà des couches de filtres et des couches de réduction. Ses filtres ne sont pas réglés par rétropropagation, qui n'est pas encore connue.

En 1988, le chercheur français Yann Le Cun entre aux Bell Labs, le laboratoire de recherche de l'entreprise de télécommunications AT&T, dans le New Jersey. Il y réunit les deux idées : un réseau

convolutif, dont tous les filtres sont appris par rétropropagation. En 1989, ce réseau lit les codes postaux manuscrits des lettres du service postal américain.

Yann Le Cun en 2018 (*photo : Jérémy Barande, Wikimedia Commons, CC BY-SA 2.0*).

La vidéo ci-dessous, publiée par Le Cun lui-même, montre ce réseau en 1989, lisant en direct des chiffres écrits à la main.

Vidéo : https://www.youtube.com/watch?v=FwFduRA_L6Q

Le travail se poursuit pendant une dizaine d'années, avec Léon Bottou, Yoshua Bengio et Patrick Haffner. Il aboutit en 1998 à **LeNet-5**, un réseau à sept couches, et à un système de lecture automatique des chèques. Vers la fin des années 1990, selon Le Cun, ce système lit entre 10 et 20 % des chèques déposés aux États-Unis. C'est l'une des premières applications commerciales à grande échelle d'un réseau de neurones.

Pour entraîner et comparer ces réseaux, l'équipe constitue une base de chiffres manuscrits, **MNIST** : 60 000 images d'entraînement et 10 000 images de test, de 28 pixels sur 28. C'est l'origine des chiffres qui servent d'exemple tout au long de ce module. MNIST est devenue la base d'essai (*benchmark*) la plus utilisée de l'apprentissage automatique (*machine learning*).

Des chiffres de la base MNIST, une ligne par chiffre (*image : Josef Steppan, Wikimedia Commons, CC BY-SA 4.0*).

Malgré ces succès, les réseaux convolutifs restent une méthode parmi d'autres pendant quinze ans. Ils fonctionnent sur de petites images de chiffres, mais les photos du monde réel demandent des réseaux plus grands, plus de données et plus de calcul. Il faut attendre que ces trois conditions, décrites dans « Pourquoi seulement en 2012 », soient réunies.

Après 2012

AlexNet est un réseau convolutif. Son architecture reprend celle de LeNet, avec trois différences : il est beaucoup plus grand, il utilise la ReLU, et il est entraîné sur GPU. Le Cun partage cette réussite avec Hinton et Bengio, les deux autres membres de l'école canadienne de l'apprentissage profond.

Les réseaux convolutifs deviennent ensuite plus profonds : 8 couches pour AlexNet en 2012, 22 pour GoogLeNet en 2014, 152 pour ResNet en 2015. Ce dernier introduit des **raccourcis** (*skip connections*) qui laissent l'information sauter certaines couches, ce qui limite l'évanouissement du gradient et permet d'entraîner des réseaux très profonds.

Nommer ce que contient une image ne suffit pas toujours. Une voiture qui doit freiner devant un piéton doit aussi savoir où il se trouve. Deux tâches prolongent donc la classification. La **détection d'objets** (*object detection*) entoure chaque objet d'un rectangle et lui donne une étiquette. En 2015, le réseau YOLO (*You Only Look Once*, « on ne regarde qu'une fois ») le fait en une seule passe, assez vite pour traiter une vidéo en temps réel. La **segmentation** (*image segmentation*) va plus loin : elle classe chaque pixel, ce qui dessine le contour exact de chaque objet. La même année, le réseau U-Net, conçu pour des images de microscopie, devient la référence en imagerie médicale, par exemple pour délimiter une tumeur. On le retrouvera au Module 4 : il sert de squelette aux premiers générateurs

d'images par diffusion.

Classer, détecter, segmenter. Les étiquettes, les rectangles et les silhouettes ont été produits par deux réseaux préentraînés : un classifieur entraîné sur ImageNet, qui ne donne qu'une réponse pour toute l'image, et Mask R-CNN, qui détecte et segmente chaque objet (photo de Gabriel Santiago, Unsplash, CC0).

Les réseaux convolutifs servent aussi à bien d'autres tâches :

- **analyser des images médicales**, comme des radiographies ou des images de la rétine ;
- **reconnaître des visages** (*face recognition*), un usage qui pose des questions de surveillance et de vie privée, traitées au Module 5 ;
- **traiter des sons**, en les transformant d'abord en images de leurs fréquences.

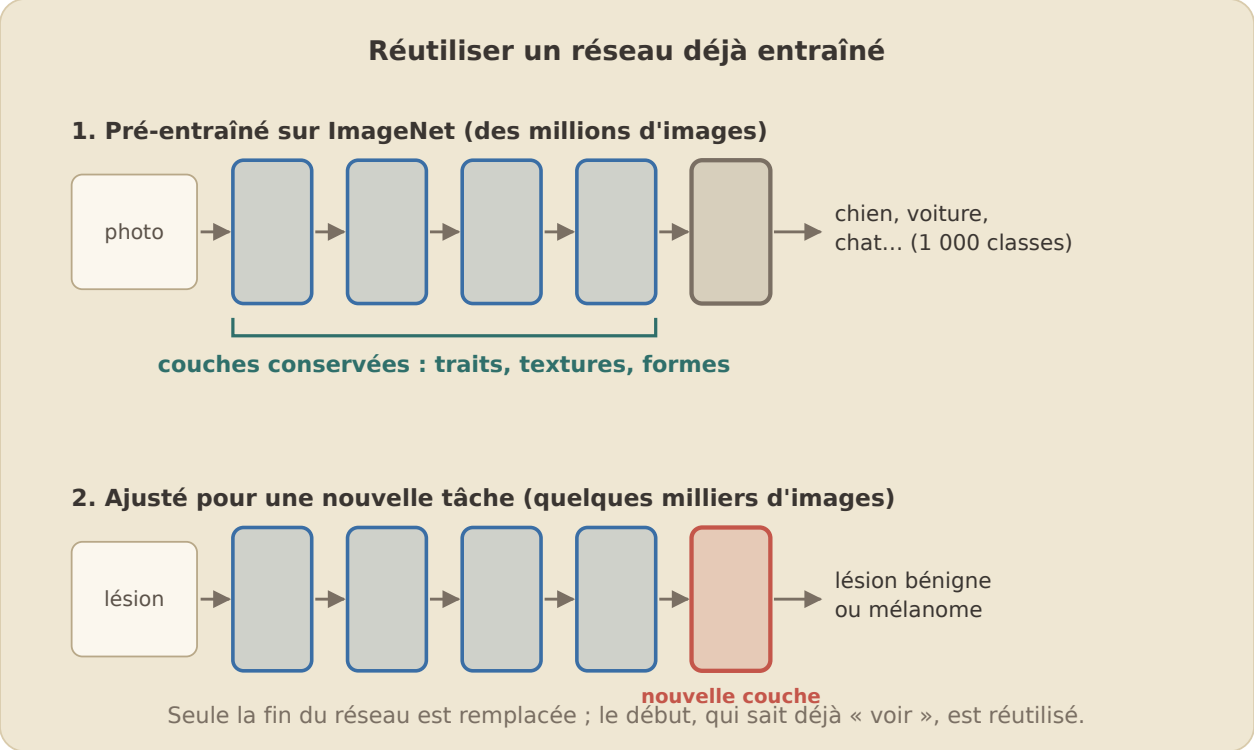
Réutiliser un réseau : l'apprentissage par transfert

Entraîner un grand réseau convolutif demande des millions d'images étiquetées et des jours de calcul sur des GPU. Peu d'équipes disposent de ces moyens, et pour beaucoup de tâches, il n'existe que quelques milliers d'exemples. Un dermatologue ne dispose pas d'un ImageNet de grains de beauté.

Une observation change la situation. Les premières couches d'un réseau entraîné sur ImageNet ont appris à détecter des traits, des textures et des formes simples. Ces caractéristiques ne sont pas propres aux chiens, aux voitures ou aux autres catégories d'ImageNet. Elles servent pour presque n'importe quelle image. Seules les dernières couches sont spécialisées dans les mille catégories du concours.

D'où la méthode de l'**apprentissage par transfert** (*transfer learning*) :

1. on part d'un réseau déjà entraîné sur un grand jeu de données, un réseau dit **pré-entraîné** (*pre-trained*) ;
2. on remplace ses dernières couches par de nouvelles, adaptées à la tâche visée ;
3. on entraîne le réseau sur les exemples de cette tâche, peu nombreux. On règle surtout les nouvelles couches, et on modifie peu, ou pas du tout, les premières. On parle d'**ajustement** (*fine-tuning*).



L'apprentissage par transfert : on garde les premières couches d'un réseau pré-entraîné, et on remplace la dernière.

Un exemple a marqué les esprits. En 2017, une équipe de l'Université Stanford part d'un réseau pré-entraîné sur ImageNet et l'ajuste sur environ 130 000 images de lésions de la peau. Sur les cas testés, le réseau distingue les mélanomes des lésions bénignes aussi bien que des dermatologues. Le réseau n'avait jamais vu de lésion de la peau avant cet ajustement, mais il savait déjà voir.

L'apprentissage par transfert est devenu la manière normale d'utiliser l'apprentissage profond. C'est aussi l'idée qui fonde les grands modèles de langage du Module 4 : un modèle est d'abord pré-entraîné sur une immense quantité de texte, puis ajusté pour une tâche ou pour la conversation. Le *P* de GPT signifie d'ailleurs *pre-trained*.

La leçon de la convolution

Ce chapitre nuance la leçon amère de Sutton. La convolution est une connaissance humaine, inscrite dans la forme du réseau : dans une image, ce qui compte est d'abord local, et un même motif peut apparaître n'importe où. Le réseau convolutif doit une grande partie de son efficacité à cette connaissance.

Mais c'est une connaissance très générale. Elle porte sur la structure des images en général, et non sur les chats, les chiffres ou les visages. Le contenu des filtres reste entièrement appris. C'est le type de savoir humain qui a survécu à l'apprentissage profond : non pas des règles sur le monde, mais des hypothèses sur la forme des données.

Le chapitre suivant applique la même démarche à un autre type de données. Une image a une taille fixe, et tous ses pixels sont disponibles en même temps. Une phrase ou un enregistrement sonore ont une longueur variable, et l'ordre de leurs éléments compte. Le chapitre « Lire une séquence : les

réseaux récurrents » présente les réseaux conçus pour ces données.

Lire une séquence : les réseaux récurrents

Une image a une taille fixe, et tous ses pixels sont disponibles en même temps. Beaucoup de données n'ont pas cette forme. Une phrase, un enregistrement de parole, un morceau de musique ou le tracé d'un stylo sur une tablette arrivent élément par élément. Leur longueur varie, et l'ordre de leurs éléments compte. On appelle ces données des **séquences**. Ce chapitre présente les réseaux conçus pour les traiter, les **réseaux récurrents** (*recurrent neural networks*, RNN).

Comme le réseau convolutif, le réseau récurrent est fait des mêmes briques que le perceptron multicouche. Il en diffère par son architecture : ses neurones forment une boucle, et les mêmes poids servent à chaque élément de la séquence. Son biais inductif est qu'une séquence se lit dans l'ordre, un élément après l'autre, et qu'une même règle s'applique à chaque pas.

Le problème des séquences

Un réseau ordinaire a un nombre fixe d'entrées. Pour lui donner une séquence, on peut essayer deux solutions simples.

Tout donner d'un coup. On réserve une entrée par élément de la séquence. Mais la longueur varie : une phrase compte trois mots, la suivante quarante. Et si l'on se contente de compter les mots présents, sans tenir compte de leur place, « le chien mord l'homme » et « l'homme mord le chien » deviennent identiques.

Regarder une fenêtre. On ne donne au réseau que les derniers éléments, par exemple les cinq derniers mots. La taille est fixe et l'ordre est conservé. Mais tout ce qui précède la fenêtre est oublié. Dans « Le livre que tu m'as prêté la semaine dernière était… », il faut se souvenir de « livre » pour bien accorder la suite.

Il faut donc un réseau capable de lire une séquence de n'importe quelle longueur, dans l'ordre, en gardant une trace de ce qu'il a déjà lu.

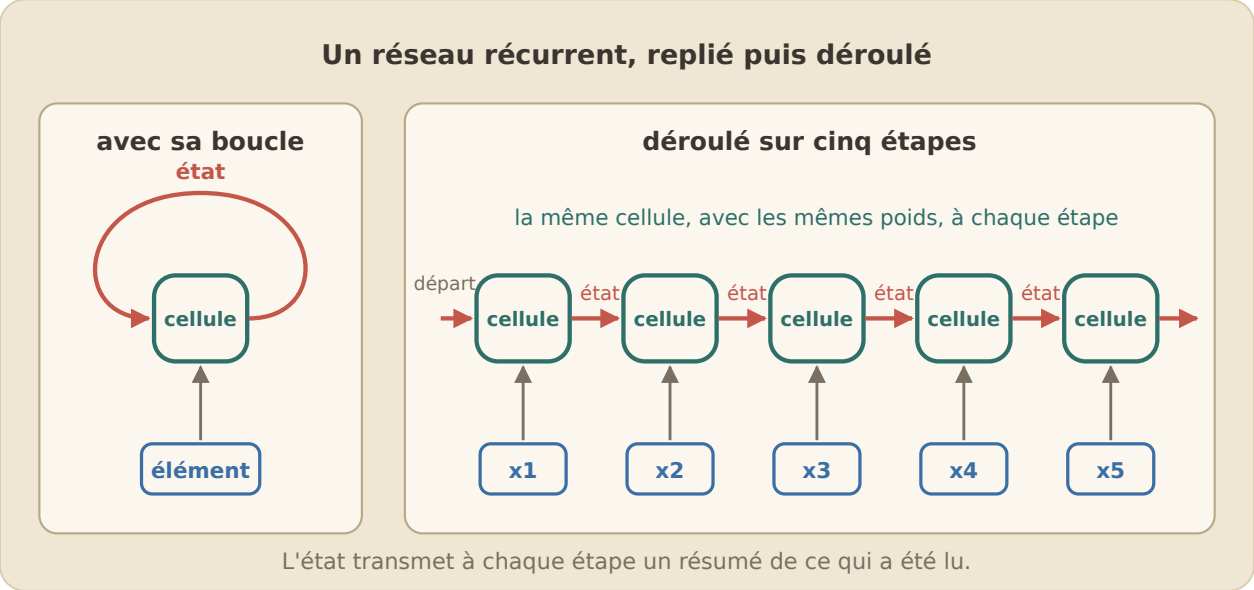
Pour traiter des mots, il faut d'abord les convertir en nombres. Le chapitre « Des mots aux nombres » du Module 4 expliquera comment. Les exemples de ce chapitre portent sur des suites de bits, qui sont déjà des nombres.

Une boucle : la mémoire

Un réseau récurrent lit la séquence un élément à la fois. À chaque étape, il reçoit deux choses : l'élément courant, et son propre **état** (*hidden state*) à l'étape précédente. Il calcule à partir d'elles un

nouvel état, qu'il se transmet à l'étape suivante. Cet état est une liste de nombres qui résume ce que le réseau a lu jusque-là. C'est sa mémoire.

On représente souvent ce réseau avec une boucle, l'état sortant de la cellule pour y revenir. Pour comprendre son fonctionnement, il est plus simple de le **dérouler** (*unroll*) : on dessine une copie de la cellule par étape, chacune passant son état à la suivante.



Un réseau récurrent, avec sa boucle, puis déroulé sur cinq étapes.

Toutes les copies sont la même cellule, avec les mêmes poids. Le réseau applique le même calcul à chaque étape, quelle que soit la longueur de la séquence. C'est le partage des poids des réseaux convolutifs, appliqué dans le temps plutôt que dans l'espace : le filtre glissait sur l'image, la cellule glisse sur la séquence.

Un réseau récurrent à manipuler

Voici une tâche qui demande de la mémoire. On reçoit une suite de bits, comme 1 0 1 1 0, et il faut dire si elle contient un nombre **pair** ou **impair** de 1. Cette question s'appelle la **parité**. Un réseau ordinaire avec une fenêtre ne peut pas y répondre : un seul 1 hors de la fenêtre change la réponse.

Un réseau récurrent la résout avec un état très simple, qui répond à la question « le nombre de 1 lus jusqu'ici est-il impair ? ». À chaque nouveau bit, l'état change si le bit vaut 1, et reste le même s'il vaut 0. C'est exactement le XOR de l'état et du nouveau bit. La cellule de ce réseau est donc le petit réseau du XOR du chapitre « Une couche cachée », avec les mêmes poids, appliqué à chaque étape.

Applet interactive : à utiliser en ligne, sur la page « Lire une séquence : les réseaux récurrents » du site du cours.

1. Avancez d'un pas à la fois et suivez l'état. Il passe près de 1 (en rouge) quand le nombre de 1 lus jusque-là est impair, et revient près de 0 (en bleu) quand il est pair.
2. Passez la souris sur une cellule pour voir le calcul de ses trois neurones.

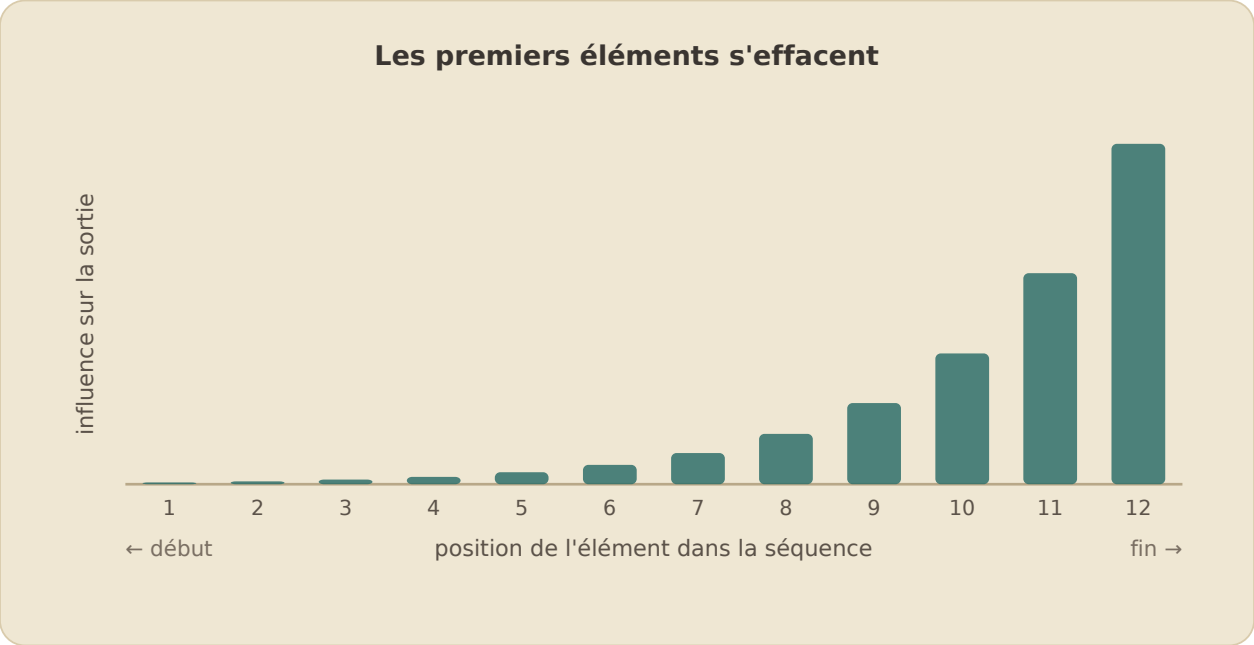
3. Lisez toute la suite, puis cliquez sur le premier bit pour le changer. La réponse finale change aussi : l'information du premier bit a traversé toutes les étapes.

Dans cette applet, les poids sont choisis à la main. Un vrai réseau récurrent les apprend, et son état compte des dizaines ou des centaines de nombres, qui ne correspondent à aucune question aussi lisible que « impair jusqu'ici ? ».

Entraîner un réseau récurrent, et ce qu'il oublie

Un réseau récurrent s'entraîne par rétropropagation, sur sa version déroulée. L'erreur, mesurée à la fin de la séquence, remonte les étapes une à une jusqu'à la première. On parle de **rétropropagation à travers le temps** (*backpropagation through time*).

Une séquence de 100 éléments donne un réseau déroulé de 100 étapes, donc un réseau très profond. On retrouve alors l'évanouissement du gradient : l'erreur s'affaiblit à chaque étape qu'elle remonte. Les premiers éléments de la séquence ont de moins en moins d'influence sur l'apprentissage, et le réseau peine à apprendre les liens entre des éléments éloignés.



Dans un réseau récurrent simple, l'influence d'un élément sur l'apprentissage diminue avec sa distance à la fin de la séquence.

Le langage contient beaucoup de liens de ce type. Dans « Les clés que j'avais posées hier soir sur la table près de la fenêtre **sont** introuvables », le verbe s'accorde avec « clés », placé une douzaine de mots plus tôt. Un réseau récurrent simple se souvient bien des derniers mots, et mal des premiers.

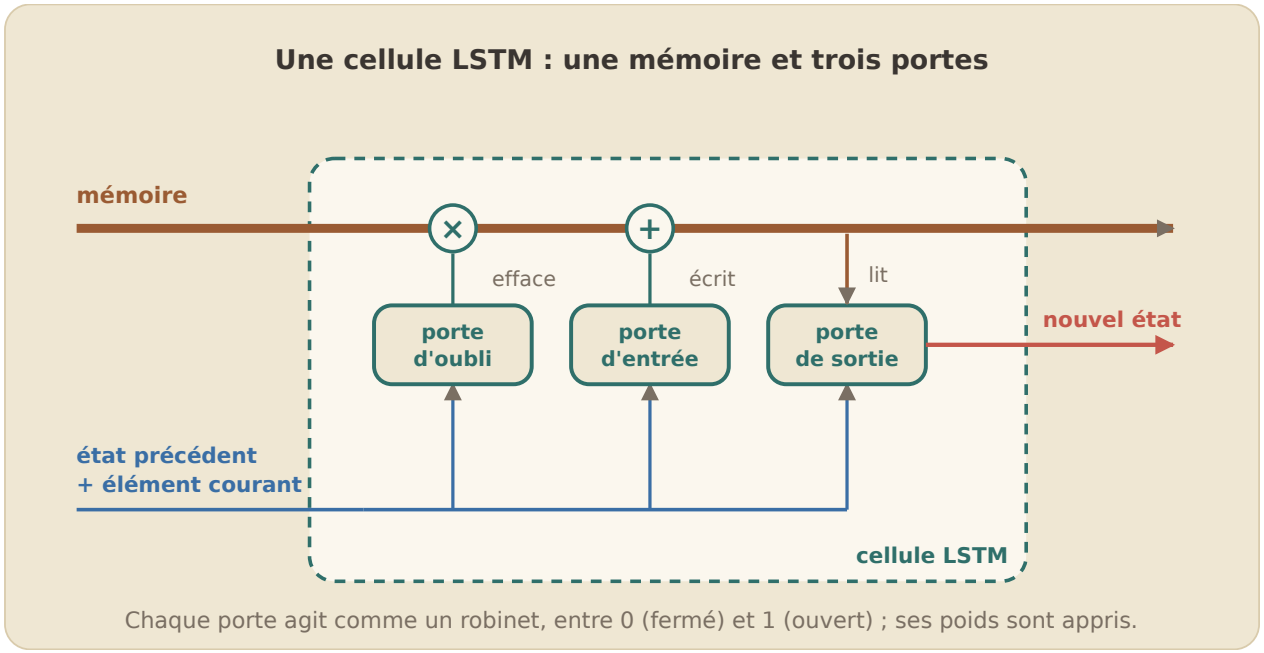
Le LSTM : une mémoire à portes

En 1997, Sepp Hochreiter et Jürgen Schmidhuber, à Munich, proposent une cellule récurrente conçue pour ne pas oublier : la **mémoire à long et court terme**, ou **LSTM** (*long short-term memory*).

La cellule LSTM contient une **mémoire** (*cell state*) séparée de son état, une liste de nombres qui passe d'une étape à la suivante presque sans transformation. À chaque étape, trois **portes** (*gates*) décident de ce qui arrive à cette mémoire :

- la **porte d'oubli** (*forget gate*) décide ce qu'il faut effacer ;
- la **porte d'entrée** (*input gate*) décide ce qu'il faut y écrire à partir de l'élément courant ;
- la **porte de sortie** (*output gate*) décide ce qu'il faut en lire pour produire l'état.

Chaque porte est un petit groupe de neurones, dont la sortie, entre 0 et 1, agit comme un robinet : 0 bloque l'information, 1 la laisse passer. Les portes ont leurs propres poids, appris par rétropropagation comme les autres. Le réseau apprend donc lui-même quand retenir et quand oublier. Dans la phrase des clés, il peut apprendre à écrire « sujet au pluriel » dans sa mémoire en lisant « Les clés », à le garder pendant les mots suivants, puis à le lire au moment du verbe.



Une cellule LSTM, simplifiée : une mémoire qui traverse la cellule, et trois portes qui décident ce qu'on y efface, écrit et lit.

Comme la mémoire passe d'une étape à l'autre sans être recalculée, l'erreur peut aussi la remonter sans s'affaiblir autant. Un LSTM apprend des liens sur des centaines d'étapes, là où un réseau récurrent simple se limite à une dizaine.

Des réseaux avec une mémoire externe
En 2014, Alex Graves et ses collègues de DeepMind vont plus loin. Ils donnent à un réseau récurrent une **mémoire externe**, une sorte de tableau où il peut écrire et lire à n'importe quelle case. Ils l'appellent **machine de Turing neuronale** (*neural Turing machine*), en référence au ruban de la machine de Turing du Module 1. Le réseau apprend ainsi, à partir d'exemples, de petits algorithmes comme copier ou trier une liste. L'idée est restée un sujet de recherche, mais le mécanisme qui choisit où lire dans la mémoire annonce celui du chapitre suivant.

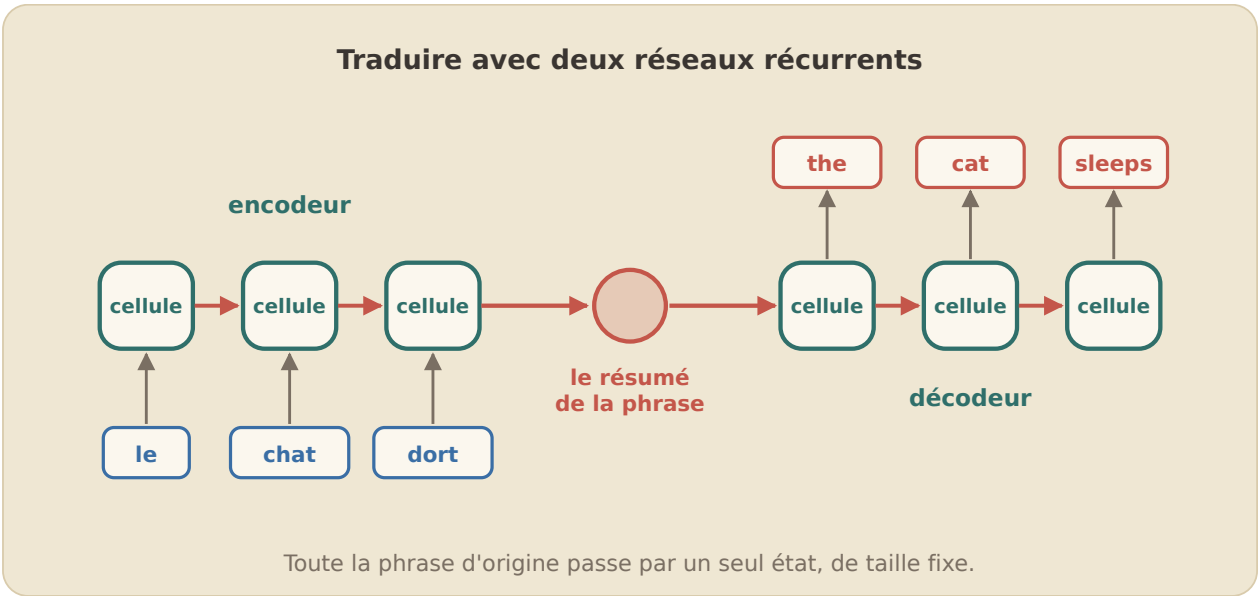
Ce que les réseaux récurrents ont permis

Entre 2009 et 2017, les réseaux récurrents, et surtout les LSTM, deviennent la méthode de référence pour les séquences.

L'écriture manuscrite. En 2009, un LSTM conçu par Alex Graves remporte des concours de reconnaissance d'écriture manuscrite (*handwriting recognition*). Il lit l'écriture comme une séquence, sans découper les mots en lettres à l'avance. Le fil des chiffres manuscrits de ce module se prolonge ainsi des chiffres isolés à l'écriture courante.

La parole. Un enregistrement de parole est une séquence de sons. En 2015, Google adopte les LSTM pour la reconnaissance vocale (*speech recognition*) de ses téléphones, et son taux d'erreur baisse nettement.

La traduction. En 2014, Ilya Sutskever, Oriol Vinyals et Quoc Le, chez Google, proposent l'architecture **encodeur-décodeur** (*encoder-decoder*) pour la traduction. Un premier réseau récurrent, l'encodeur, lit la phrase à traduire et la résume dans son état final. Un second réseau récurrent, le décodeur, part de cet état et écrit la traduction mot par mot. On retrouve l'assemblage de l'autoencodeur : deux réseaux branchés l'un à la suite de l'autre et entraînés ensemble, avec une représentation compacte au milieu. En novembre 2016, Google Traduction adopte cette méthode. Selon le *New York Times*, le gain obtenu d'un seul coup équivaut à peu près à tous les progrès accumulés par l'ancien système depuis son lancement, une dizaine d'années plus tôt.



L'architecture encodeur-décodeur : un réseau lit la phrase, un autre écrit la traduction à partir de son résumé.

Les limites

Les réseaux récurrents ont trois limites, qui motivent le chapitre suivant.

Le goulot (*bottleneck*). Dans l'encodeur-décodeur, toute la phrase d'origine est résumée en un seul état, de taille fixe, qu'elle compte cinq mots ou cinquante. Plus la phrase est longue, plus la traduction se dégrade.

La lenteur. Un réseau récurrent ne peut calculer l'étape 10 qu'après l'étape 9. Le calcul est séquentiel par nature, et ne profite donc pas des milliers de cœurs d'un GPU, qui exigent des calculs simultanés. Cette limite devient décisive quand on veut entraîner sur des milliards de mots.

La mémoire lointaine. Le LSTM a repoussé la limite de l'oubli, sans la supprimer.

En 2014, à Montréal, Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio s'attaquent au premier problème. Ils permettent au décodeur, à chaque mot qu'il écrit, de regarder directement tous les mots de la phrase d'origine, et de choisir ceux qui comptent à ce moment. Ce mécanisme s'appelle l'**attention**. Trois ans plus tard, une équipe de Google montre qu'on peut garder l'attention et supprimer la récurrence. Le chapitre suivant, « L'attention et le Transformer », présente ce résultat.

L'attention et le Transformer

Le chapitre « Lire une séquence » s'est terminé sur trois limites des réseaux récurrents (*recurrent neural networks*, RNN) : toute une phrase passe par un seul état, le calcul avance une étape à la fois, et les éléments lointains s'oublient. Ce chapitre présente le mécanisme qui a levé ces limites, l'**attention**, puis l'architecture construite autour de lui, le **Transformer**. Il traite le Transformer comme une architecture, au même titre que les réseaux convolutifs ou récurrents. Ce qu'il permet de faire avec le langage est le sujet du Module 4.

Le Transformer est lui aussi fait des mêmes briques. Son biais inductif est le plus faible des trois : chaque élément peut consulter tous les autres, quelle que soit leur distance, et l'architecture ne suppose presque rien sur l'ordre ou le voisinage. Le Transformer doit donc presque tout apprendre des données, ce qui demande d'énormes quantités d'exemples, mais il s'adapte ainsi à presque tout : le texte, les images, le son, et même les protéines. On retrouve la leçon amère de Richard Sutton : moins on impose d'hypothèses, plus on dépend des données et du calcul, et plus la méthode est générale.

L'attention dans la traduction

Revenons à l'encodeur-décodeur. Le décodeur écrit la traduction à partir d'un seul état, qui résume toute la phrase d'origine. En 2014, Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio, à Montréal, gardent l'état de l'encodeur après **chaque** mot, et non seulement le dernier. Le décodeur peut alors consulter toute la phrase d'origine à chaque mot qu'il écrit.

Pour chaque mot à écrire, le décodeur procède en trois temps :

1. il compare ce qu'il cherche à chacun des mots de la phrase d'origine, ce qui donne un score par mot ;
2. il transforme ces scores en **poids**, positifs et de total égal à 1 ;
3. il fait un mélange des mots d'origine selon ces poids, et s'en sert pour choisir le mot à écrire.

Les poids indiquent où le décodeur « regarde » au moment d'écrire chaque mot. C'est pourquoi on parle d'**attention**.

L'article de 2014 montre un exemple de traduction de l'anglais vers le français. « The European Economic Area » devient « la zone économique européenne ». L'ordre des mots s'inverse : « European » vient avant « Economic » en anglais, « européenne » vient après « économique » en français. Les poids de l'attention suivent cette inversion. En écrivant « économique », le décodeur regarde surtout « Economic ». En écrivant « européenne », il regarde surtout « European ».

Où regarde le décodeur, pour chaque mot qu'il écrit ?

traduction (français) \ phrase d'origine (anglais)	the	European	Economic	Area
la	0,85	0,05	0,04	0,06
zone	0,05	0,05	0,10	0,80
économique	0,02	0,18	0,75	0,05
européenne	0,02	0,78	0,15	0,05

Chaque ligne donne les poids de l'attention pour un mot écrit ; ils totalisent 1.

Les poids de l'attention pour la traduction de « the European Economic Area ». Les valeurs sont redessinées d'après l'exemple de Bahdanau, Cho et Bengio (2014).

Personne n'a indiqué au réseau quel mot correspond à quel mot. Ces correspondances sont apprises, comme tous les poids, par rétropropagation.

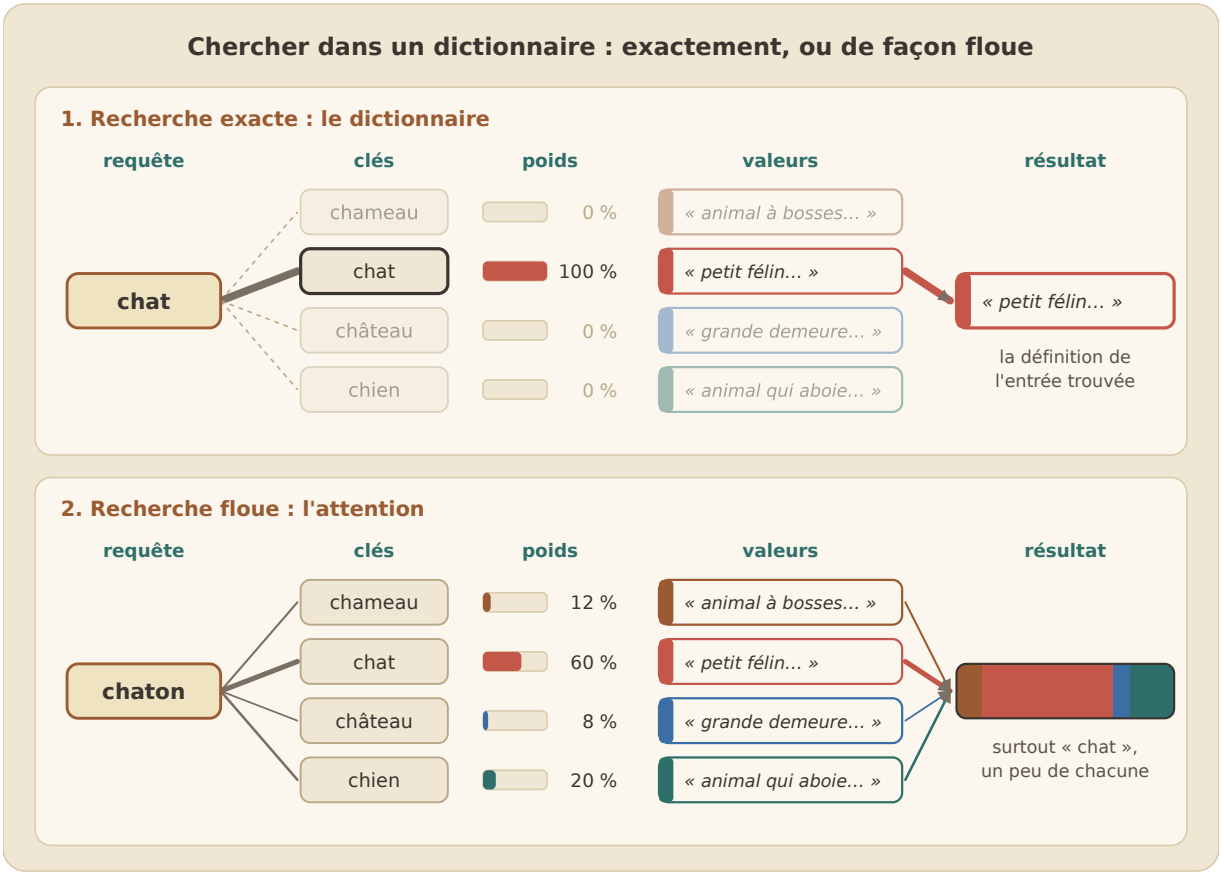
Une recherche floue

L'attention est une forme de recherche. Dans une recherche ordinaire, par exemple dans un dictionnaire, on cherche un mot, on le compare aux entrées, et on récupère la définition de l'entrée qui correspond exactement. Le Module 1 décrivait de la même façon la recherche d'un fait dans la base de faits d'un système expert.

L'attention procède de la même façon, avec une différence : elle ne choisit pas une seule entrée. Elle compare la question à toutes les entrées, et récupère un peu de chacune, en proportion de leur ressemblance avec la question. C'est une recherche floue.

On donne un nom à chacun des trois éléments :

- la **requête** (*query*) est ce que l'on cherche ;
- les **clés** (*keys*) sont ce à quoi on compare la requête, une par entrée ;
- les **valeurs** (*values*) sont ce que l'on récupère, une par entrée.



Une recherche exacte ne retient qu'une entrée. L'attention les retient toutes, chacune selon sa ressemblance avec la requête, et répond même quand aucune entrée ne correspond exactement.

Dans un réseau, les requêtes, les clés et les valeurs sont des listes de nombres calculées par des neurones. Leurs poids sont appris. Le réseau apprend donc à la fois quoi chercher, comment se décrire pour être trouvé, et quoi transmettre.

Pour aller plus loin : le calcul sur un petit exemple
Prenons une requête et trois entrées, chacune décrite par deux nombres.

	Clé	Valeur
Entrée A	$(1, 0)$	10
Entrée B	$(0, 1)$	20
Entrée C	$(1, 1)$	30

La requête est $(2, 0)$. On la compare à chaque clé en multipliant les nombres deux à deux et en additionnant : 2 pour A, 0 pour B, 2 pour C. On transforme ensuite ces scores en poids qui totalisent 1, en favorisant les scores élevés. On obtient environ 0,47 pour A, 0,06 pour B et 0,47 pour C. Le résultat est le mélange des valeurs selon ces poids : $0{,}47 \times 10 + 0{,}06 \times 20 + 0{,}47 \times 30$, soit environ 20. La requête a surtout récupéré les valeurs de A et de C, dont les clés lui ressemblent.

L'auto-attention

Dans la traduction, le décodeur porte son attention sur la phrase d'origine. L'étape suivante consiste à appliquer l'attention à l'intérieur d'une même phrase. Chaque mot émet une requête, et va chercher l'information utile chez les autres mots de la phrase. On parle d'**auto-attention** (*self-attention*).

Prenons la phrase « Le chien n'est pas monté dans le camion parce qu'il était trop fatigué. » Le mot « il » peut désigner le chien ou le camion. Pour le comprendre, il faut aller chercher l'information dans « chien ». Si la phrase se termine par « trop haut », « il » désigne le camion. Le sens d'un mot dépend de son contexte, et l'auto-attention calcule ce lien.

Après une couche d'auto-attention, chaque mot n'est plus représenté seul. Sa représentation contient une part de l'information des mots qu'il a regardés. Le « il » de la première phrase porte maintenant une part de « chien ».

Une attention à manipuler

Dans l'applet ci-dessous, cliquez sur un mot pour voir où il porte son attention. Les arcs le relient aux autres mots, d'autant plus épais que le poids est grand, et les poids s'affichent sous chaque mot. Ces poids ont été choisis pour l'exemple. Dans un vrai Transformer, ils sont calculés par le réseau, et ils sont en général moins nets.

Applet interactive : à utiliser en ligne, sur la page « L'attention et le Transformer » du site du cours.

1. Le mot « il » est choisi. Observez où il porte son attention.
2. Remplacez la fin de la phrase par « … trop haut. » L'attention de « il » se déplace vers « camion ».
3. Cliquez sur « fatigué », puis sur « haut » dans la seconde phrase. Chacun regarde le nom qu'il qualifie.
4. Cliquez sur « monté ». Il regarde surtout « chien » et « camion », son sujet et son complément.

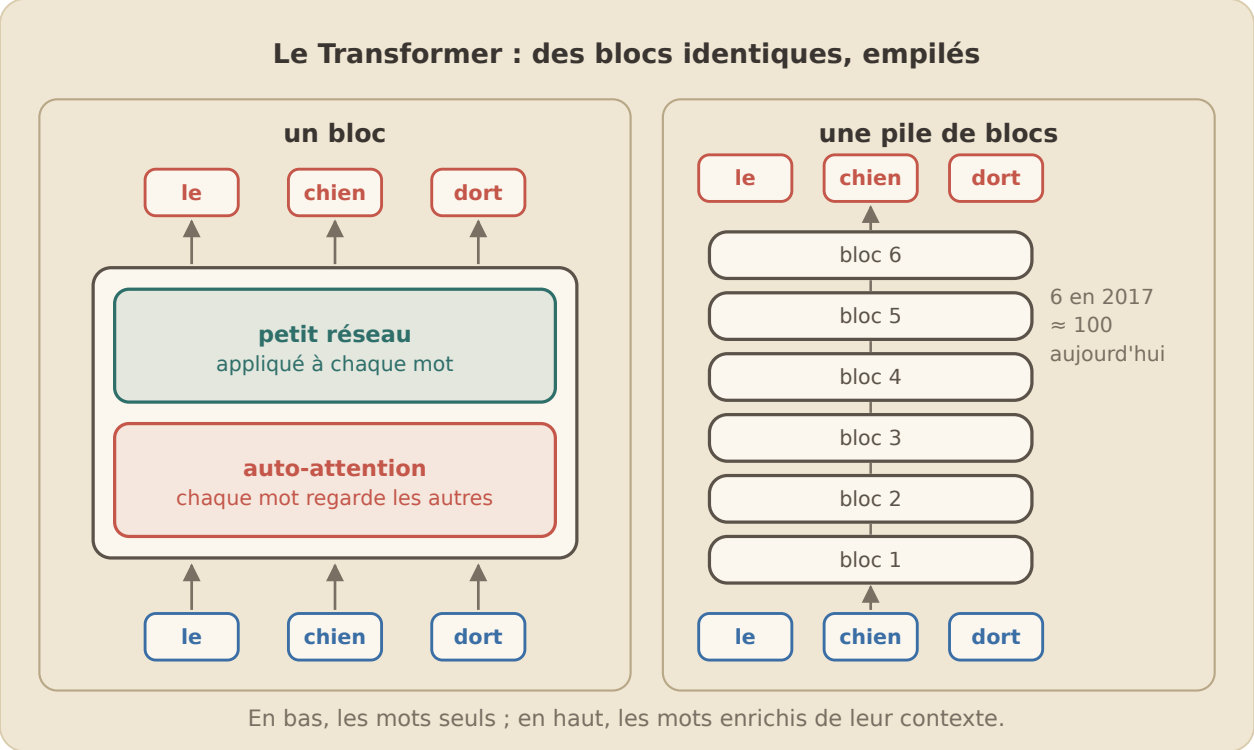
Le Transformer

En 2017, huit chercheurs de Google, dont Ashish Vaswani, publient un article intitulé « Attention Is All You Need » (« L'attention suffit »). L'idée tient dans le titre. On supprime la récurrence et on ne garde que l'attention. L'architecture qui en résulte s'appelle le **Transformer**.

Un Transformer est fait de **blocs** identiques, empilés. Chaque bloc contient deux étapes :

1. une couche d'**auto-attention**, où chaque mot va chercher l'information utile chez les autres mots ;
2. un petit réseau ordinaire, appliqué à chaque mot séparément, qui transforme l'information recueillie.

Le texte traverse les blocs l'un après l'autre. À chaque bloc, la représentation de chaque mot s'enrichit de son contexte. L'article de 2017 empile six blocs. Les grands modèles actuels en empilent une centaine.



Le Transformer : chaque bloc combine l'auto-attention et un petit réseau ; on empile les blocs.

Deux précisions complètent ce schéma.

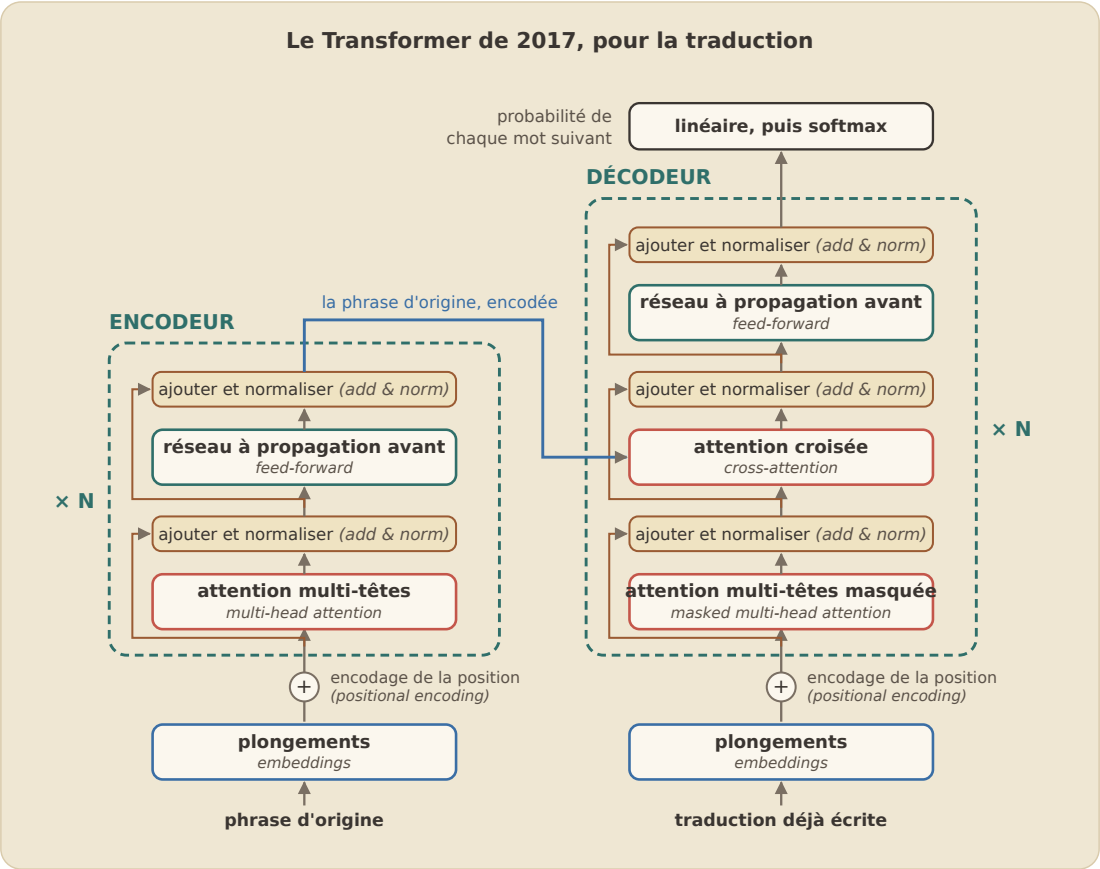
Plusieurs têtes. Chaque couche d'auto-attention contient plusieurs mécanismes d'attention en parallèle, appelés **têtes** (*attention heads*), par exemple huit. Chacune a ses propres requêtes, clés et valeurs, et peut apprendre un type de lien différent : une tête relie un verbe à son sujet, une autre un pronom au nom qu'il désigne, une autre un mot à son voisin.

La position. Un réseau récurrent connaissait l'ordre des mots, puisqu'il les lisait un à un. L'auto-attention, elle, traite tous les mots en même temps et ne connaît pas leur ordre. Pour elle, « le chien mord l'homme » et « l'homme mord le chien » seraient identiques. On ajoute donc à chaque mot, à l'entrée du réseau, une indication de sa position dans la phrase (*positional encoding*).

On retrouve aussi dans le Transformer les raccourcis de ResNet, qui laissent l'information sauter une étape et facilitent l'entraînement d'une pile très profonde.

Le schéma de l'article de 2017 rassemble toutes ces pièces. Il a deux colonnes, parce que le Transformer a d'abord été conçu pour la traduction.

- À gauche, l'**encodeur** lit la phrase d'origine. Chacun de ses blocs contient l'auto-attention à plusieurs têtes, puis le petit réseau ordinaire, appelé ici « réseau à propagation avant » (*feed-forward*). Chaque étape est suivie de « ajouter et normaliser » (*add & norm*) : on ajoute à sa sortie l'entrée qui a pris le raccourci, puis on ramène les nombres à une échelle stable.
- À droite, le **décodeur** écrit la traduction, un mot à la fois. Son auto-attention est **masquée** : chaque mot ne voit que les mots déjà écrits, pas ceux qu'il doit justement prédire. Vient ensuite l'**attention croisée** (*cross-attention*), où les mots de la traduction vont chercher l'information dans la phrase d'origine encodée. C'est la grille de traduction du début du chapitre.
- En haut, une dernière couche et un softmax donnent la probabilité de chaque mot suivant.



Le Transformer de 2017, d'après la figure de l'article « Attention Is All You Need ». Chaque colonne répète son bloc N fois (six dans l'article).

Les deux colonnes peuvent aussi servir séparément. Le Module 4 montre ce que chacune est devenue.

Pourquoi il a tout changé

Le Transformer règle les trois limites des réseaux récurrents, et en ajoute une.

Le parallélisme. Tous les mots d'un texte sont traités en même temps, et non l'un après l'autre. Les calculs de l'attention sont des multiplications répétées, le type de calcul pour lequel les GPU sont conçus. Un Transformer s'entraîne donc beaucoup plus vite qu'un réseau récurrent de même taille.

La distance. Dans un réseau récurrent, l'information du premier mot doit traverser toutes les étapes pour atteindre le centième. Dans un Transformer, le centième mot regarde directement le premier, en une seule étape. La mémoire lointaine n'est plus un problème de principe.

L'échelle. Ces deux propriétés permettent d'entraîner des réseaux beaucoup plus grands, sur beaucoup plus de texte. Les Transformers des années 2020 comptent des centaines de milliards de poids, entraînés sur une grande partie du texte disponible sur le Web. Ce changement d'échelle est celui que décrivait la leçon amère.

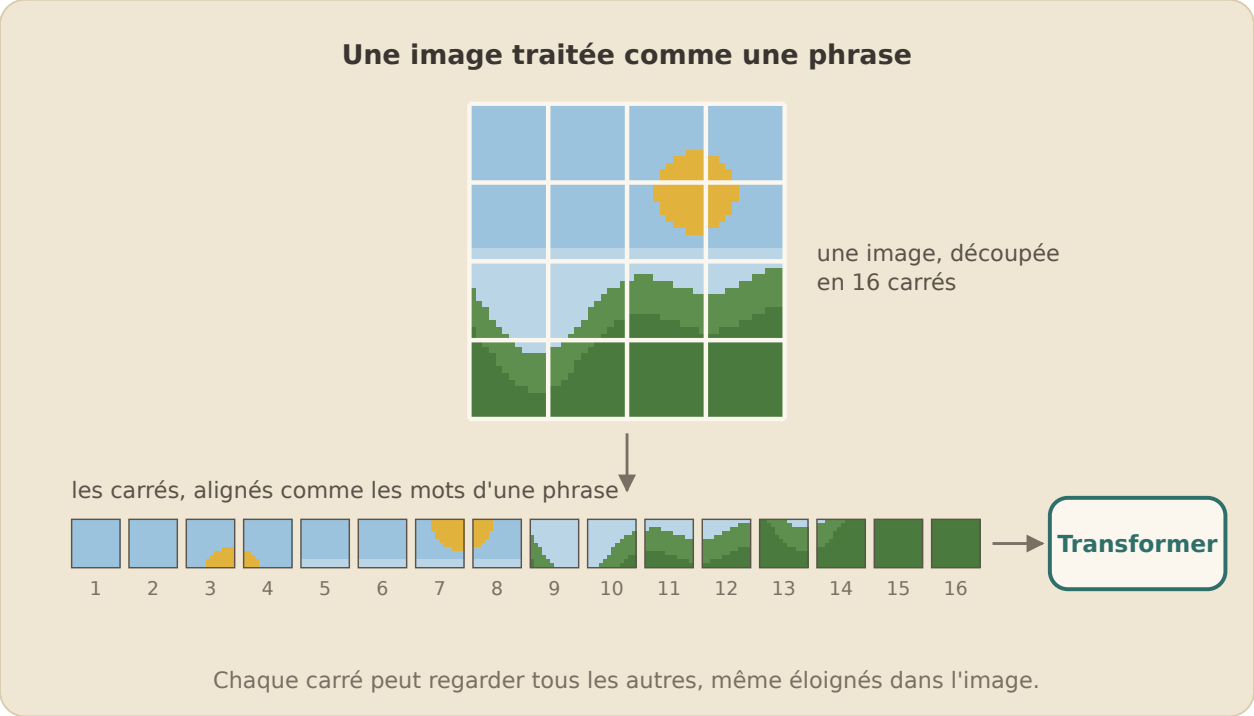
Un coût. Dans l'auto-attention, chaque mot regarde tous les autres. Pour un texte de 1 000 mots, cela fait un million de comparaisons. Pour 2 000 mots, quatre millions. Le calcul croît avec le carré de la

longueur du texte. C'est pourquoi un Transformer ne peut traiter qu'un texte de longueur limitée à la fois, sa **fenêtre de contexte** (*context window*). Le Module 4 reviendra sur cette notion.

Au-delà du texte

Le Transformer a été conçu pour la traduction, mais rien dans son fonctionnement n'est propre au langage. Il suffit de découper les données en éléments, et de laisser chaque élément regarder les autres.

Les images. En 2020, une équipe de Google découpe une image en petits carrés de 16 pixels sur 16 et les traite comme les mots d'une phrase. Ce *Vision Transformer* n'a pas la connaissance inscrite dans un réseau convolutif : il ne sait pas d'avance que les pixels voisins sont liés, et doit l'apprendre. Avec peu d'images, il fait moins bien qu'un réseau convolutif. Avec des centaines de millions d'images, il fait mieux. C'est un nouvel exemple de la leçon amère, et une nuance à la leçon de la convolution : une connaissance inscrite dans le réseau aide quand les données sont rares, et peut gêner quand elles sont abondantes.



Le Vision Transformer découpe une image en carrés et les traite comme les mots d'une phrase.

La biologie. Une protéine est une chaîne d'acides aminés qui se replie dans l'espace, et sa forme détermine son rôle. Prédire cette forme à partir de la chaîne était un problème ouvert depuis cinquante ans. En 2020, le système **AlphaFold 2**, de DeepMind, le résout pour la plupart des protéines, avec un mécanisme d'attention au cœur de son architecture. Ce travail a valu à Demis Hassabis et John Jumper une moitié du prix Nobel de chimie 2024.

La parole, la musique, le code. Tous les types de séquences se prêtent à la même méthode.

Vers le Module 4

Dès 2018, deux usages du Transformer apparaissent. **BERT**, chez Google, lit un texte en entier pour en produire une représentation, utile par exemple aux moteurs de recherche. **GPT**, chez OpenAI, est entraîné à une tâche plus simple : prédire le mot suivant d'un texte. Agrandi de version en version, ce second type de modèle est à l'origine des grands modèles de langage, comme ChatGPT. Le Module 4 leur est consacré.

Pour voir le mécanisme en détail

La vidéo ci-dessous, de la série de 3Blue1Brown déjà présentée dans « Entraîner un réseau », explique l'attention pas à pas, avec ses requêtes, ses clés et ses valeurs. Elle est en anglais.

Vidéo : https://www.youtube.com/watch?v=eMlx5fFNoYc

Le chapitre suivant, « Apprendre à jouer : le renforcement profond », quitte les séquences. Il revient à l'apprentissage par renforcement (*reinforcement learning*) du Module 2 et montre ce qu'il devient avec les réseaux profonds.

Apprendre à jouer : le renforcement profond

Au Module 2, un agent apprenait à sortir d'une petite grille par essais et erreurs. Il remplissait une table, avec une valeur pour chaque case, et la récompense finale se propageait peu à peu vers les cases de départ. Ce chapitre montre ce qui se passe quand on remplace cette table par un réseau profond, et comment cette combinaison a permis à une machine de battre les meilleurs joueurs de go.

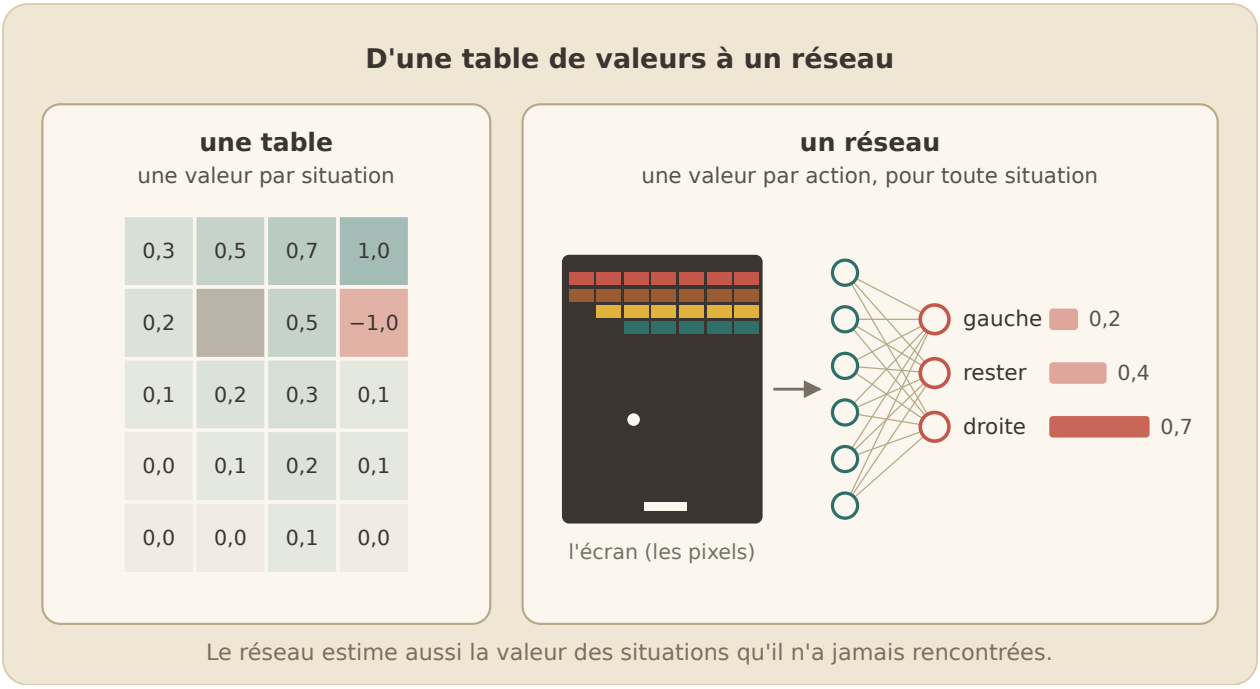
Les limites de la table

Rappelons le vocabulaire du Module 2. Un **agent** choisit des **actions**. Il reçoit parfois une **récompense** (*reward*), souvent longtemps après les actions qui l'ont produite. Il apprend la **valeur** de chaque situation, c'est-à-dire la récompense qu'il peut en espérer, et doit trouver un équilibre entre **explorer** de nouvelles actions et **exploiter** celles qu'il connaît.

La grille du Module 2 comptait une vingtaine de cases, et la table des valeurs tenait sur un écran. Les situations réelles sont beaucoup plus nombreuses. Une image d'un jeu vidéo peut prendre un nombre astronomique d'états différents. Au go, le nombre de positions possibles dépasse le nombre d'atomes de l'univers observable. On ne peut pas remplir une table de cette taille. Et même si on le pouvait, on ne reverrait presque jamais deux fois la même situation : la table n'apprendrait rien d'utile.

Remplacer la table par un réseau

La solution consiste à remplacer la table par un réseau de neurones. Le réseau reçoit la situation, par exemple l'image de l'écran, et estime la valeur de chaque action possible.



La table du Module 2 donne une valeur par situation ; un réseau estime la valeur de chaque action, pour n'importe quelle situation.

Le réseau apporte ce que la table ne pouvait pas offrir : la généralisation. Deux situations semblables reçoivent des valeurs semblables, même si l'une d'elles n'a jamais été rencontrée. L'agent peut donc profiter de son expérience dans des situations nouvelles.

L'idée n'est pas nouvelle. Au début des années 1990, Gerald Tesauro, chez IBM, crée **TD-Gammon**, un réseau de neurones qui apprend le backgammon en jouant contre lui-même, au cours d'environ un million et demi de parties. Il atteint presque le niveau des meilleurs joueurs du monde, et certains de ses choix modifient la façon dont les experts humains jouent certaines positions. Mais la méthode semble alors propre au backgammon, et elle est peu reprise pendant vingt ans.

Atari : jouer à partir des pixels

En 2013, la jeune entreprise londonienne DeepMind présente **DQN** (*deep Q-network*). En 2015, une version améliorée paraît dans la revue *Nature*. Le même réseau, un réseau convolutif, apprend à jouer à 49 jeux vidéo de la console Atari 2600. Il ne reçoit que les pixels de l'écran et le score. Personne ne lui explique les règles, ni le but du jeu. Sur une bonne partie de ces jeux, il atteint ou dépasse le niveau d'un joueur humain expérimenté.

L'exemple le plus connu est *Breakout*, où il faut détruire un mur de briques avec une balle. Après quelques centaines de parties, le réseau découvre une stratégie que ses concepteurs ne lui avaient pas indiquée : creuser un tunnel sur le côté du mur, pour envoyer la balle derrière, où elle détruit les briques toute seule.

Vidéo : https://www.youtube.com/watch?v=TmPfTpjtdgg

Le réseau échoue en revanche à *Montezuma's Revenge*, un jeu d'exploration où il faut traverser plusieurs salles avant d'obtenir le moindre point. En jouant au hasard, l'agent ne reçoit presque jamais

de récompense, et n'a donc rien à renforcer. Le compromis entre explorer et exploiter (*exploration-exploitation trade-off*), vu au Module 2, reste une difficulté centrale.

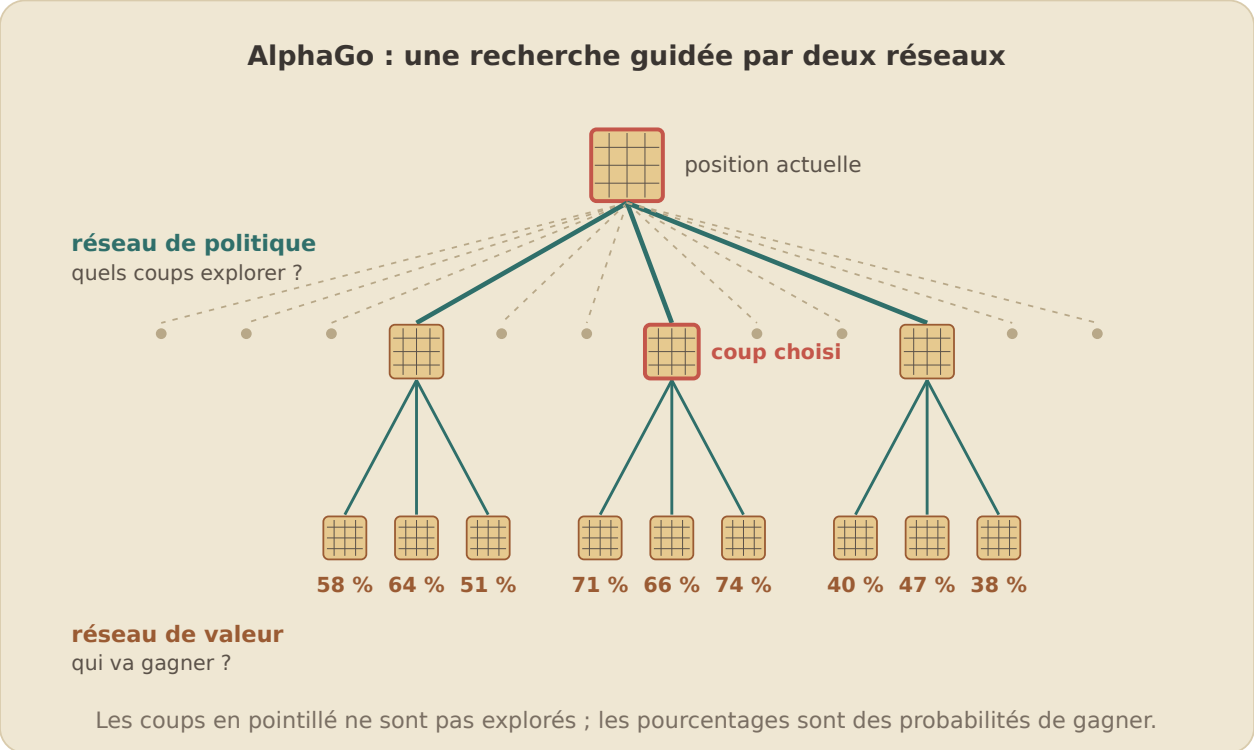
AlphaGo

Le go se joue sur un plateau de 19 lignes sur 19, le goban. Deux joueurs y posent tour à tour des pierres noires et blanches, et cherchent à entourer le plus grand territoire. Les règles sont simples, mais le jeu est d'une grande profondeur. En 2015, les meilleurs programmes n'atteignaient que le niveau d'un bon amateur.

La méthode de Deep Blue, présentée au Module 1, ne suffisait pas, pour deux raisons. À chaque tour, un joueur de go a environ 250 coups possibles, contre environ 35 aux échecs. L'arbre des coups (*game tree*) grandit donc beaucoup trop vite. Surtout, personne ne savait écrire une bonne fonction d'évaluation (*evaluation function*) pour une position de go. Les meilleurs joueurs eux-mêmes jugent une position à l'intuition, sans pouvoir l'expliquer en règles.

AlphaGo, de DeepMind, combine trois éléments :

- un **réseau de politique** (*policy network*), qui regarde la position et propose les coups prometteurs. Il évite d'explorer les 250 coups possibles ;
- un **réseau de valeur** (*value network*), qui regarde une position et estime qui va gagner. Il remplace la fonction d'évaluation que personne ne savait écrire ;
- une **recherche dans l'arbre des coups** (*tree search*), comme au Module 1, guidée par ces deux réseaux.



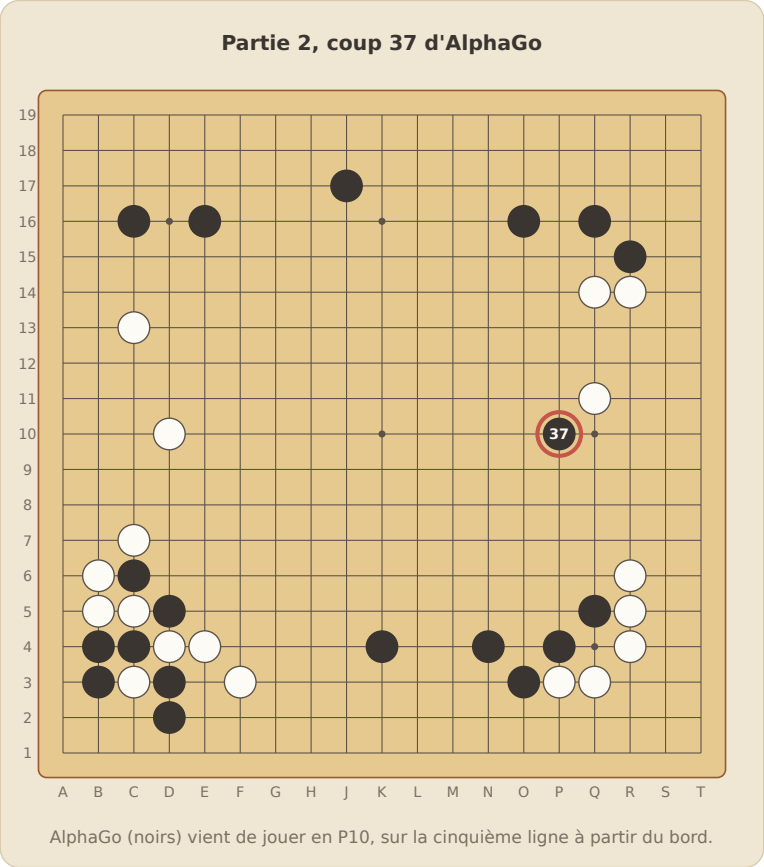
AlphaGo : le réseau de politique choisit les coups à explorer, le réseau de valeur évalue les positions atteintes.

C'est la rencontre des deux traditions de ce cours : la recherche du Module 1, et l'apprentissage

des Modules 2 et 3. AlphaGo montre aussi que l'assemblage ne se limite pas aux réseaux : on peut brancher des réseaux de neurones sur un algorithme classique, ici une recherche, et chacun fait ce qu'il fait le mieux. Les deux réseaux sont d'abord entraînés sur des parties de joueurs humains, en apprentissage supervisé (*supervised learning*) : le réseau de politique apprend à prédire le coup qu'un expert jouerait. Ils sont ensuite améliorés par renforcement, AlphaGo jouant des millions de parties contre lui-même.

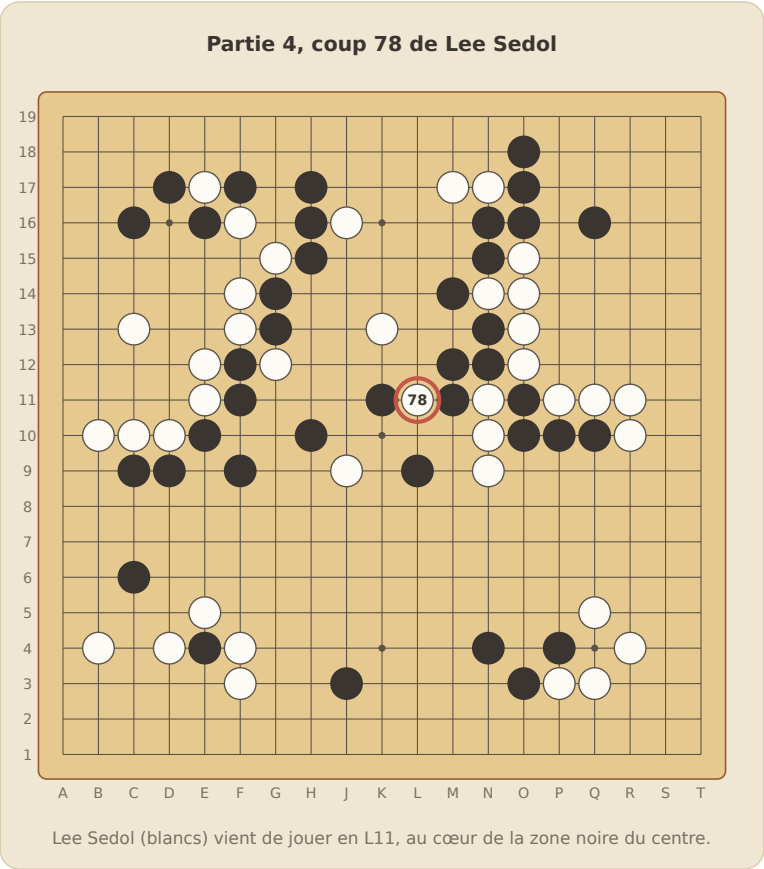
En mars 2016, à Séoul, AlphaGo affronte Lee Sedol, l'un des meilleurs joueurs du monde, en cinq parties. Il gagne 4 à 1.

Au 37e coup de la deuxième partie, AlphaGo pose une pierre sur la cinquième ligne à partir du bord, un coup que les commentateurs prennent d'abord pour une erreur. Selon DeepMind, AlphaGo estimait lui-même qu'un joueur humain n'aurait joué ce coup qu'une fois sur dix mille. La suite de la partie montre qu'il était excellent.



Le coup 37 de la deuxième partie (positions relevées d'après les diagrammes de Wikimedia Commons).

Lee Sedol remporte la quatrième partie grâce à son 78e coup, au centre du goban, que les commentateurs ont qualifié de « coup divin ». AlphaGo ne l'avait pas envisagé, et il joue mal pendant plusieurs coups ensuite.



Le coup 78 de la quatrième partie (positions relevées d'après une image de Axd, Wikimedia Commons, CC BY-SA 4.0).

Le documentaire *AlphaGo* (2017), mis en ligne gratuitement par DeepMind, raconte ce match. Il dure environ une heure et demie, en anglais.

Vidéo : https://www.youtube.com/watch?v=WXuK6gekU1Y

AlphaZero : sans parties humaines

En 2017, DeepMind présente **AlphaGo Zero**. Il ne voit aucune partie humaine. Il ne connaît que les règles du go, et apprend uniquement en jouant contre lui-même, en partant de coups joués au hasard. En trois jours, il joue près de cinq millions de parties contre lui-même, puis bat la version qui avait vaincu Lee Sedol par 100 parties à 0. Les joueurs qui ont étudié ses parties y ont trouvé des ouvertures que les humains n'avaient jamais jouées en plusieurs siècles.

La même année, **AlphaZero** applique la même méthode, sans modification, aux échecs et au shogi, les échecs japonais. Après quelques heures d'entraînement pour chacun, il dépasse les meilleurs programmes de ces jeux.

Le contraste avec Deep Blue est instructif. La fonction d'évaluation de Deep Blue avait été réglée pendant des années avec l'aide de grands maîtres, et Deep Blue examinait environ 200 millions de positions par seconde. AlphaZero a appris seul sa fonction d'évaluation, et il examine environ 80 000 positions par seconde, plus de deux mille fois moins, parce que son réseau de politique lui

indique où chercher. C'est l'illustration la plus nette de la leçon amère : une méthode générale, l'apprentissage par auto-jeu (*self-play*), dépasse le savoir humain inscrit à la main.

Une machine qui apprend en jouant contre elle-même

L'applet ci-dessous applique ce principe au morpion. La machine ne connaît que les règles. Elle tient une table : pour chaque position, une valeur entre -1 et $+1$, qui indique si cette position mène plutôt à la défaite ou à la victoire du joueur qui vient de jouer. À chaque partie d'entraînement contre elle-même, elle joue le coup qui mène à la meilleure valeur connue, sauf de temps en temps où elle essaie un coup au hasard. À la fin de la partie, elle rapproche la valeur de chaque position jouée du résultat obtenu.

Le morpion ne compte que quelques milliers de positions. Une table suffit donc, et il n'est pas nécessaire d'utiliser un réseau. Mais le principe de l'auto-jeu est celui d'AlphaZero.

Applet interactive : à utiliser en ligne, sur la page « Apprendre à jouer : le renforcement profond » du site du cours.

1. Avant tout entraînement, mesurez le niveau de la machine. Elle joue au hasard, et perd la plupart de ses parties contre un joueur parfait.
2. Entraînez-la par étapes, 100 parties, puis 1 000, puis plusieurs fois 10 000, et observez son niveau progresser. Le « joueur parfait » de l'applet calcule tous les coups possibles jusqu'à la fin de la partie, avec la méthode minimax du Module 1. Contre lui, le mieux possible est la partie nulle.
3. Jouez contre elle après chaque étape. Cochez « voir ce qu'elle pense de chacun de vos coups » pour voir les valeurs de sa table.

Quelques dizaines de milliers de parties suffisent pour qu'elle ne perde plus. Personne ne lui a montré de bon coup.

Au-delà des jeux, et les limites

Les jeux sont un terrain d'essai commode : les règles sont claires, la récompense est nette, et l'on peut jouer des millions de parties. Le renforcement profond (*deep reinforcement learning*) a aussi trouvé des usages hors des jeux.

- En 2016, Google l'utilise pour piloter le refroidissement de ses centres de données, et annonce une réduction d'environ 40 % de l'énergie consacrée au refroidissement.
- En 2022, DeepMind et l'École polytechnique fédérale de Lausanne l'utilisent pour contrôler la forme du plasma dans un réacteur expérimental de fusion nucléaire.
- En robotique, des robots apprennent à marcher ou à manipuler des objets, d'abord dans une simulation, puis dans le monde réel.

La méthode a aussi des limites importantes.

Elle demande énormément d'essais. AlphaGo Zero a joué près de cinq millions de parties contre lui-même en trois jours. Un joueur humain de haut niveau en joue quelques dizaines de milliers dans toute sa vie. Dans le monde réel, où chaque essai prend du temps ou peut causer des dégâts, ce besoin est un obstacle.

Elle optimise exactement la récompense qu'on lui donne. En 2016, OpenAI entraîne un agent sur un jeu de course de bateaux, où l'on gagne des points en passant sur des cibles. L'agent découvre qu'il peut tourner en rond autour de trois cibles qui réapparaissent, et accumuler plus de points qu'en finissant la course. Il fait exactement ce qu'on lui a demandé, et pas du tout ce qu'on voulait. Ce problème de la récompense mal définie concerne tous les systèmes qui optimisent un objectif. Le Module 5 y reviendra, sous le nom d'**alignement**.

Le renforcement a enfin une place importante dans les grands modèles de langage, comme l'annonçait le Module 2. Il sert à ajuster leurs réponses selon les préférences d'évaluateurs humains, et à les entraîner à raisonner sur des problèmes dont la réponse peut être vérifiée. Le Module 4 présentera ces méthodes.

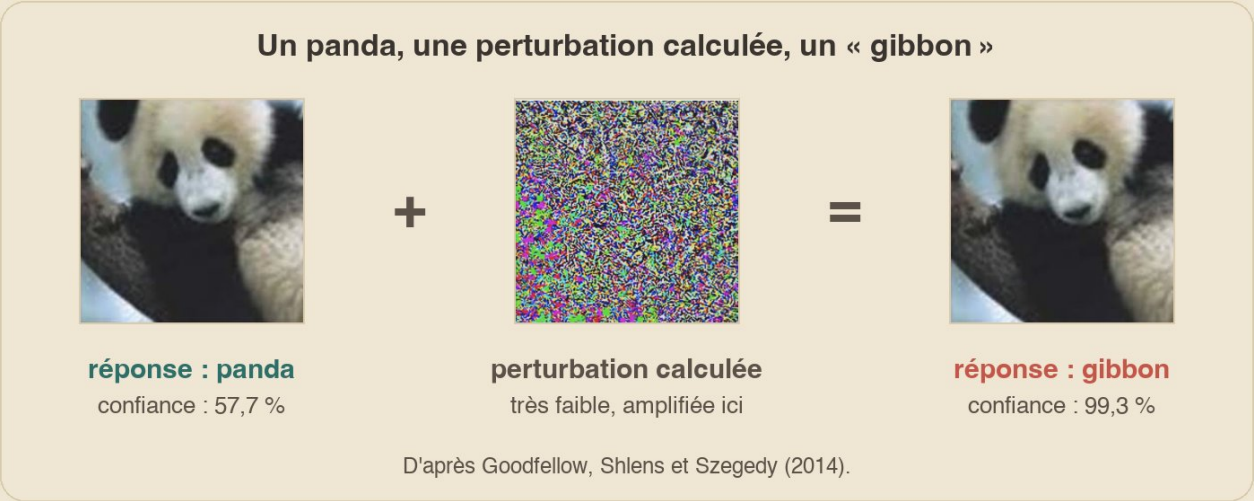
Les réseaux profonds voient, lisent et jouent désormais mieux que nous dans plusieurs domaines. Le dernier chapitre, « Tromper un réseau », montre qu'ils peuvent pourtant être trompés de façon surprenante.

Tromper un réseau

Les chapitres précédents ont montré des réseaux qui reconnaissent des images, traduisent des phrases et battent les meilleurs joueurs de go. Ce dernier chapitre montre une faiblesse surprenante de ces systèmes : une modification invisible pour nous peut leur faire dire n'importe quoi. Il explique comment c'est possible, et ce que cela révèle de leur fonctionnement.

Le panda devenu gibbon

En 2014, Ian Goodfellow, Jonathon Shlens et Christian Szegedy, chez Google, publient l'exemple suivant. Un réseau convolutif (*convolutional neural network*, CNN) entraîné sur ImageNet reconnaît un panda, avec 58 % de confiance. Les chercheurs ajoutent à l'image une perturbation très faible, calculée avec soin. À l'œil, l'image modifiée est identique à l'originale. Le réseau y voit maintenant un gibbon, avec 99 % de confiance.



L'exemple du panda : une perturbation invisible fait passer la réponse de « panda » à « gibbon » (d'après Goodfellow, Shlens et Szegedy, 2014).

Une image modifiée de cette façon s'appelle un **exemple adverse** (*adversarial example*). Le phénomène n'est pas propre à ce réseau ni à ce panda. On peut fabriquer un exemple adverse pour presque n'importe quelle image et n'importe quel réseau.

Comment on le fabrique

La méthode utilise la rétropropagation, mais à l'envers.

À l'entraînement, on garde l'image fixe et on modifie les poids, dans le sens qui réduit l'erreur. Pour tromper un réseau déjà entraîné, on garde les poids fixes et on modifie l'image. La rétropropagation indique, pour chaque pixel, dans quel sens le modifier pour augmenter la probabilité d'une réponse choisie, par exemple « gibbon ». On déplace alors chaque pixel d'une quantité minuscule, toujours dans le sens le plus utile.

Aucun pixel ne change de façon visible. Mais une image de 224 pixels sur 224 en couleur compte environ 150 000 valeurs. Des changements minuscules, tous orientés dans le même sens, finissent par s'additionner et faire basculer la réponse.

Tromper un réseau à manipuler

Dans l'applet ci-dessous, un classifieur de chiffres a été entraîné sur des chiffres dessinés par l'ordinateur. C'est un modèle simple, une régression logistique à dix sorties, sans couche cachée, et non un réseau profond. Le principe de l'attaque est le même.

Applet interactive : à utiliser en ligne, sur la page « Tromper un réseau » du site du cours.

1. La force est à zéro, et le classifieur reconnaît le 3. Augmentez lentement la force. Observez la perturbation, et le moment où la réponse bascule vers 8.
2. Regardez l'image modifiée au moment où la réponse bascule. Le 3 reste parfaitement lisible pour nous.
3. Essayez d'autres paires de chiffres. Certaines demandent une force beaucoup plus grande que d'autres. Par exemple, faire dire 1 à un 6 est difficile.

Pourquoi c'est possible

Plusieurs explications se complètent.

La dimension. Une image est un point dans un espace qui compte autant de dimensions que de valeurs de pixels, des centaines de milliers. Dans un espace aussi grand, une frontière de décision (*decision boundary*) passe presque toujours tout près de chaque image, dans une direction ou une autre. L'attaque trouve cette direction.

Le réseau ne voit pas comme nous. Il utilise tous les indices qui l'aident à réduire son erreur d'entraînement, y compris des indices que nous ne remarquons pas ou que nous jugerions sans importance. En 2019, Robert Geirhos et ses collègues montrent que les réseaux entraînés sur ImageNet se fient surtout à la **texture** des objets, alors que nous nous fions surtout à leur forme. Une silhouette de chat couverte d'une texture de peau d'éléphant est classée « éléphant ».

(a) Texture image

81.4%	**Indian elephant**
10.3%	indri
8.2%	black swan

(b) Content image

71.1%	**tabby cat**
17.3%	grey fox
3.3%	Siamese cat

(c) Texture-shape cue conflict

63.9%	**Indian elephant**
26.4%	indri
9.6%	black swan

Une texture, un chat, et un chat à texture d'éléphant : le réseau suit la texture plutôt que la forme (*Geirhos et collègues, 2019*).

Le réseau peut réussir pour de mauvaises raisons. En 2016, Marco Ribeiro, Sameer Singh et Carlos Guestrin présentent une méthode pour expliquer les réponses d'un classifieur. Pour la tester, ils entraînent volontairement un classifieur biaisé, qui distingue les loups des huskies : dans ses photos d'entraînement, les loups sont presque toujours sur de la neige. Le classifieur atteint une bonne précision, et leur méthode révèle qu'il regarde surtout l'arrière-plan. Il est devenu un détecteur de neige. Avant de voir cette explication, une partie des personnes interrogées faisaient confiance au classifieur. Après, presque aucune. La neige est **corrélée** aux loups dans ces photos, mais elle n'en est pas la cause, une distinction présentée au Module 2 dans « Corrélation n'est pas causalité ».

Hans le Malin
Au début des années 1900, à Berlin, un cheval nommé Hans semblait savoir compter. On lui posait une addition, et il frappait du sabot le bon nombre de fois. En 1907, le psychologue Oskar Pfungst montre que Hans ne compte pas. Il observe la personne qui l'interroge, et s'arrête quand il perçoit chez elle une réaction involontaire, un léger mouvement de tête, au moment où il atteint le bon nombre. Hans réussissait vraiment, mais pour une autre raison que celle qu'on croyait. On parle aujourd'hui d'**effet Hans le Malin** (*Clever Hans effect*) pour un modèle d'apprentissage qui réussit en s'appuyant sur un indice imprévu, comme la neige derrière les loups.

Hans le Malin et son public, à Berlin, vers 1904 (photo du domaine public, Wikimedia Commons).

Dans le monde réel

Un exemple adverse sur une image numérique demande de modifier chaque pixel, ce qui suppose un accès direct à l'image. Les chercheurs ont montré que les attaques fonctionnent aussi dans le monde physique.

- **Le panneau d'arrêt.** En 2018, une équipe universitaire colle quelques bandes noires et blanches sur un vrai panneau d'arrêt. Pour un humain, ce sont des autocollants sans importance. Le réseau qui lit les panneaux, photographiés sous différents angles et à différentes distances, y voit le plus souvent une limite de vitesse.
- **La tortue.** En 2017, une équipe du MIT imprime en 3D une tortue dont la texture a été calculée pour tromper un réseau. Sous presque tous les angles, le réseau y voit un fusil.
- **Les lunettes.** En 2016, des chercheurs montrent qu'une monture de lunettes imprimée avec un motif particulier peut empêcher un système de reconnaissance faciale (*facial recognition*) de reconnaître une personne, ou lui faire croire qu'il s'agit de quelqu'un d'autre.
- **La voix.** Des commandes vocales peuvent être cachées dans un enregistrement de musique ou de parole, sans qu'un humain les remarque, et être comprises par un assistant vocal.

La tortue imprimée en 3D, qu'un réseau prend pour un fusil sous presque tous les angles (photo : LabSix, MIT, 2017).

Illustration de l'attaque du panneau d'arrêt : quelques autocollants suffisent à tromper la machine, pas l'humain.

Se défendre

La défense la plus utilisée s'appelle l'**entraînement adverse** (*adversarial training*). On fabrique des exemples adverses pendant l'entraînement, et on les ajoute aux données avec leur bonne réponse. Le réseau apprend à ne plus se laisser tromper par ces perturbations. Il devient plus robuste, mais il reste vulnérable à des attaques plus fortes ou d'un autre type, et sa précision sur les images normales baisse souvent un peu.

Depuis 2014, de nombreuses autres défenses ont été proposées. La plupart ont été contournées, parfois quelques mois après leur publication. C'est une course entre attaques et défenses, sans solution complète à ce jour.

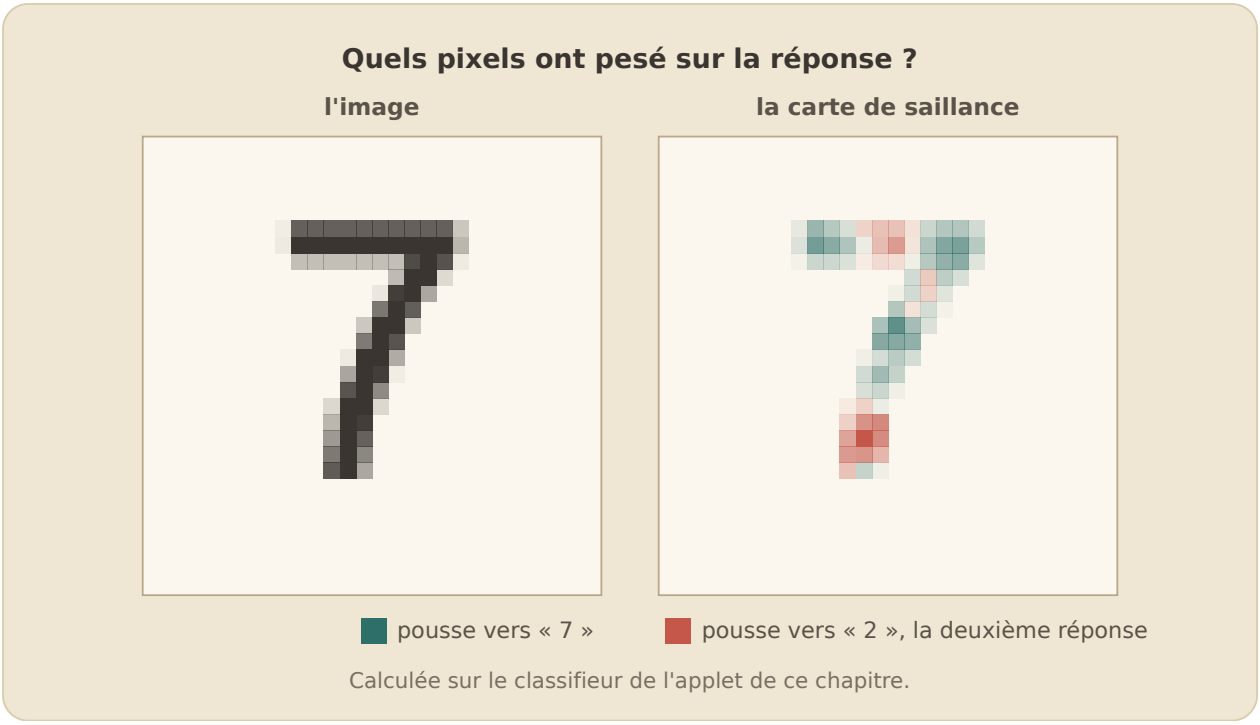
Le problème ne se limite pas aux images. Les grands modèles de langage ont leurs propres attaques : des formulations conçues pour leur faire ignorer leurs consignes, ou des instructions cachées dans un document qu'on leur demande de lire. Le Module 4 y reviendra.

Une boîte noire

Les exemples adverses et l'effet Hans le Malin ont une cause commune. On ne sait pas exactement ce qu'un réseau a appris.

Le contraste avec les systèmes des modules précédents est net. On pouvait lire les règles d'un système expert du Module 1 et suivre son raisonnement. On pouvait lire les questions d'un arbre de décision du Module 2. Un réseau profond compte des millions de poids, et aucun d'eux n'a de signification isolée. Il donne une réponse, souvent très bonne, sans donner ses raisons. On dit que c'est une **boîte noire** (*black box*).

L'**IA explicable** (*explainable AI*, XAI) cherche à retrouver ces raisons. Une de ses méthodes produit une **carte de saillance** (*saliency map*) : pour une image donnée, elle indique quels pixels ont le plus pesé sur la réponse. C'est ainsi que l'on découvre qu'un classifieur de loups regarde la neige, ou qu'un classifieur de radiographies s'appuie sur une marque propre à l'hôpital qui a pris l'image plutôt que sur les poumons du patient.



La carte de saillance d'un 7, calculée sur le classifieur de l'applet : en vert ce qui plaide pour « 7 », en rouge ce qui plaide pour « 2 ».

Ces méthodes donnent des indices, pas une explication complète. La question de savoir ce qu'un réseau a vraiment appris reste en grande partie ouverte. Le Module 5 reviendra sur ses conséquences : peut-on confier une décision importante, un diagnostic ou un prêt bancaire, à un système qui ne peut pas expliquer sa réponse ?

Ce que ce module a montré

Ce module est parti d'un neurone, qui n'est qu'une régression logistique. En reliant des neurones en couches, on obtient un réseau qui fabrique ses propres caractéristiques, et qui résout le XOR, le problème qui avait arrêté le perceptron en 1969. La rétropropagation permet d'entraîner ces réseaux. Avec des données et du calcul en quantité suffisante, des réseaux profonds de formes adaptées ont dépassé toutes les autres méthodes pour les images, les séquences et les jeux.

Ces réseaux ne sont pas des inventions séparées. Le perceptron multicouche, le réseau convolutif, le réseau récurrent, le Transformer et des assemblages comme AlphaGo sont des variations sur un même thème. Ce sont les mêmes pièces, reliées autrement et entraînées par la même rétropropagation. Cette facilité à composer explique en bonne partie leur succès. Le Module 4 en tire parti à son tour, par exemple en branchant un réseau qui lit une image sur un modèle de langage.

Ce module a aussi montré leurs limites. Ces réseaux ne voient pas, ne lisent pas et ne jouent pas comme nous. Ils peuvent être trompés par des perturbations invisibles, réussir pour de mauvaises raisons, et ils n'expliquent pas leurs réponses.

Tous les réseaux de ce module **reconnaissent** et **prédisent** : ils associent une réponse à une entrée. Le Module 4 montre comment les mêmes architectures, et en particulier le Transformer, apprennent

à **produire** des textes, des images et des sons.

Apprivoiser un réseau de neurones (travail noté 3)

Ce travail utilise TensorFlow Playground, une application en logiciel libre que nous avons adaptée pour ce cours. Elle entraîne sous vos yeux un petit réseau de neurones à classer des points de deux couleurs, et montre à chaque instant la frontière qu'il trace.

Lire l'application

- **À gauche**, le jeu de données. Chaque point est orange ou bleu. Le curseur « Bruit » mélange les deux groupes.
- **Au centre**, les **caractéristiques** (*features*) données au réseau. X_1 et X_2 sont les deux coordonnées d'un point, et les autres ($X_1{}^2$, X_1X_2...) sont calculées à partir d'elles. À leur droite, les **couches cachées** (*hidden layers*), qu'on ajoute ou retire avec les boutons + et −.
- **À droite**, la réponse du réseau : la couleur du fond montre la classe qu'il prédit en chaque point. La **perte** (*loss*) est l'erreur du réseau, au sens de la fonction d'erreur du Module 2. Plus elle est basse, mieux le réseau classe les points.
- **En haut**, le bouton de lecture lance l'entraînement. Le bouton de réinitialisation, la flèche circulaire, tire de nouveaux poids de départ au hasard. Le compteur indique le nombre d'**époques** (*epochs*), au sens de « Entraîner un réseau ».

Si l'application est coupée sur votre écran, ouvrez-la en pleine page.

Consignes

1. Faites les expériences proposées dans les questions ci-dessous, avec l'application.
2. Répondez aux questions dans un fichier PDF (**aucun autre format ne sera accepté**). Les réponses doivent être claires et précises.

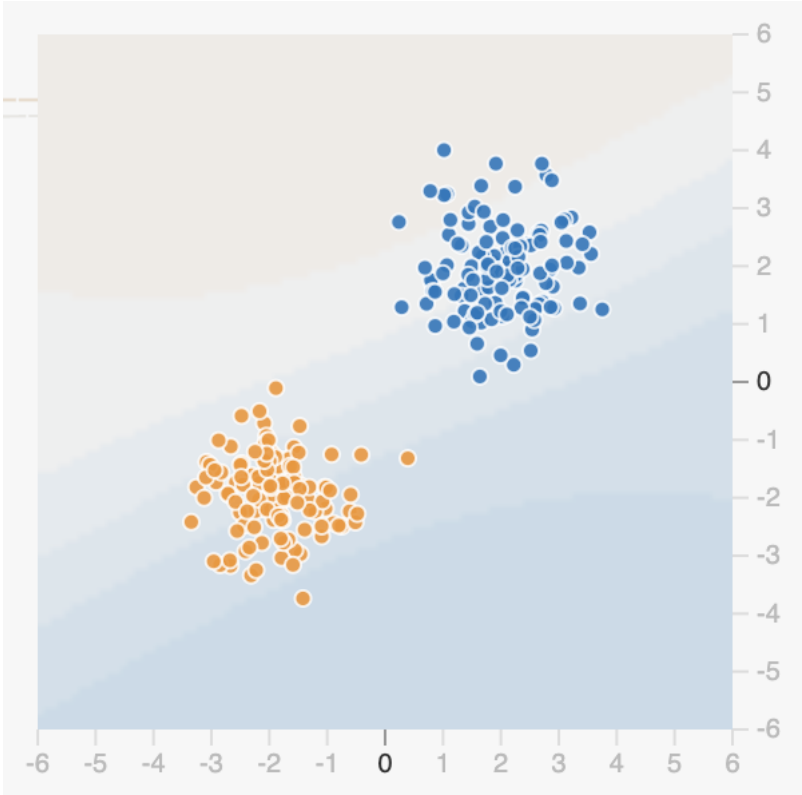
Applet interactive : à utiliser en ligne, sur la page « Travail noté 3 » du site du cours.

Questions

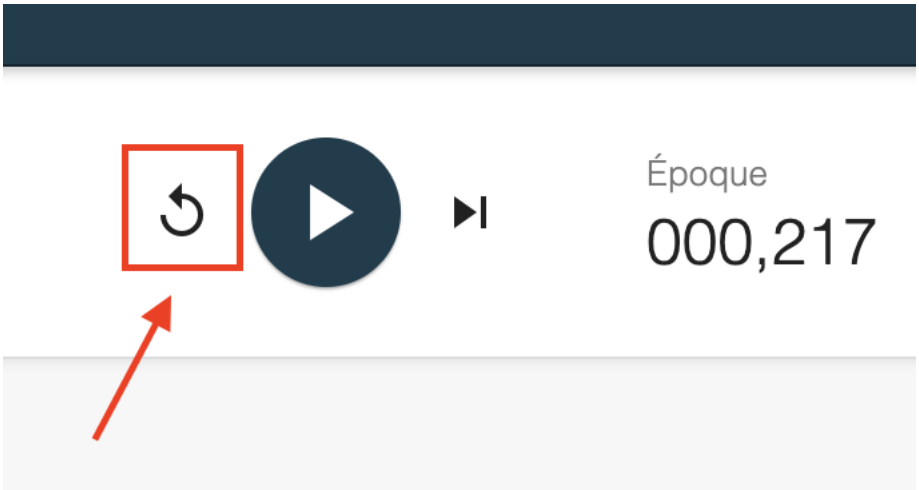
Un problème facile

1. Parmi les quatre jeux de données proposés, lequel vous semble le plus facile à classer, en séparant les points orange des points bleus, pour un algorithme d'apprentissage automatique en général, et pas seulement pour un réseau de neurones ? Expliquez pourquoi.

2. Choisissez le jeu des deux groupes de points disposés en diagonale.

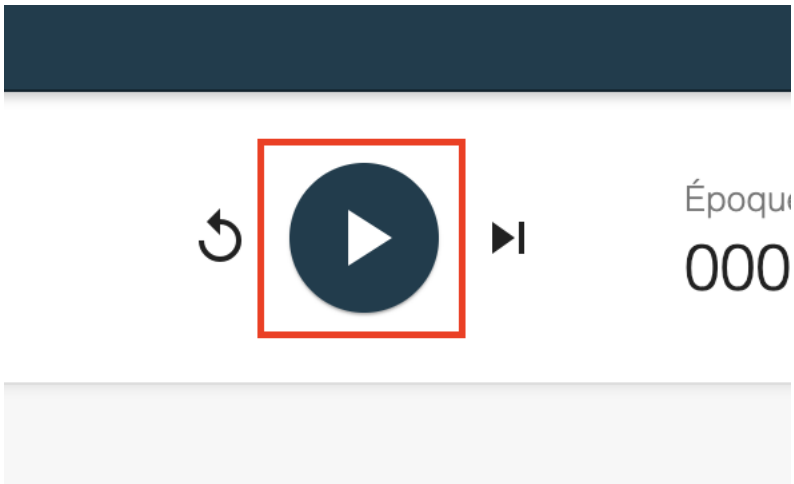


Vérifiez qu'il n'y a aucune couche cachée et que seules les caractéristiques X_1 et X_2 sont activées. Quelle est la perte d'entraînement ? Appuyez plusieurs fois sur le bouton de réinitialisation et observez comment varie cette perte initiale, avant tout entraînement.



Que signifient ces variations ? Pourquoi la perte initiale est-elle parfois haute et parfois basse, et comment le voit-on sur le graphique ? (Voir « Un réseau qui apprend le XOR ».)

3. Sans couche cachée, ce réseau ne contient qu'un seul neurone de sortie. Lancez l'entraînement avec ce bouton :

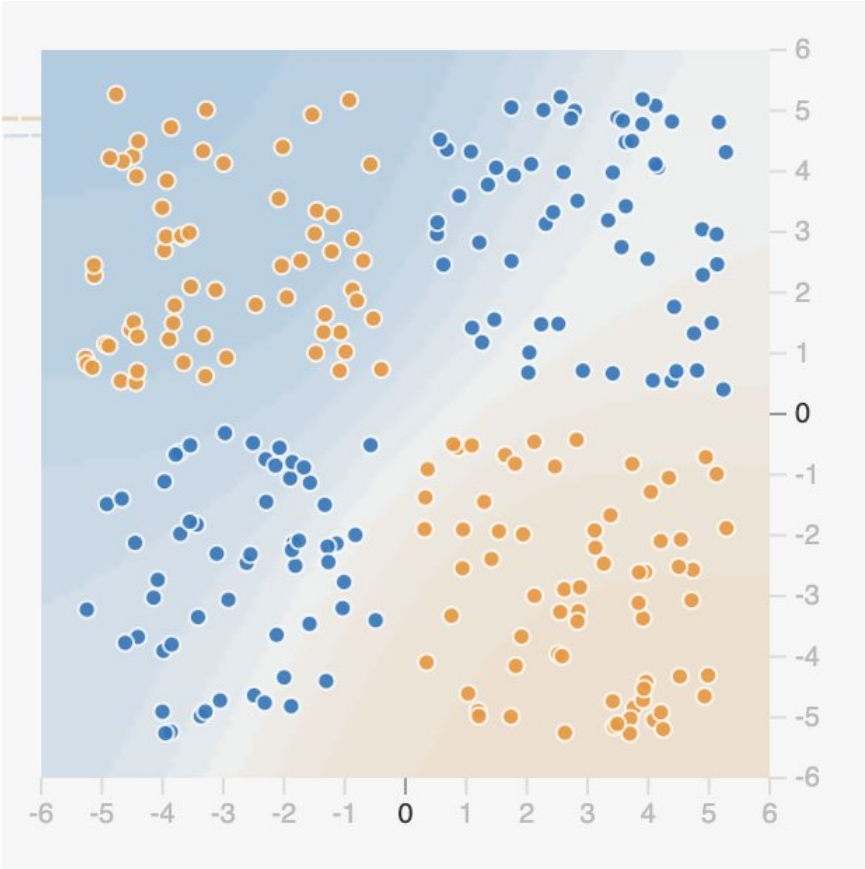


Quel modèle du Module 2 retrouve-t-on ainsi, et pourquoi ? (Voir « Un neurone est une régression logistique ».)

4. Réglez le bruit à 25, puis appuyez plusieurs fois sur « Régénérer ». Ce changement rend-il la classification plus facile ou plus difficile pour un algorithme d'apprentissage ? Expliquez pourquoi.

Le XOR

5. Remettez le bruit à 0 et choisissez le jeu du XOR, avec quatre groupes de points aux coins.

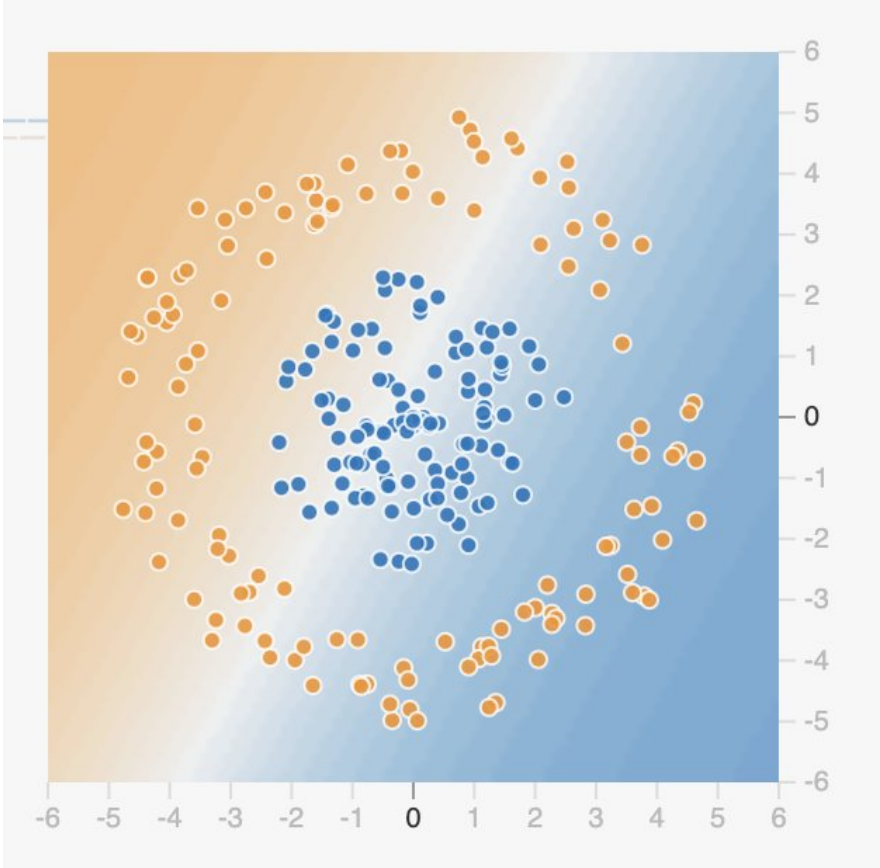


A priori, le réseau de la question précédente, sans couche cachée, peut-il classer ces points ? Expliquez pourquoi, puis faites l'essai. Que se passe-t-il ? (Voir le XOR au Module 1 et « Linéaire ou non-linéaire » au Module 2.)

6. Ajoutez une couche cachée de trois neurones, puis lancez l'entraînement. Quel est l'effet ? Comment l'expliquer ? (Voir « Une couche cachée ».)
7. Recommencez l'entraînement plusieurs fois, en appuyant sur le bouton de réinitialisation entre deux essais. Le réseau arrive-t-il toujours à la même frontière et à la même perte ? Comment l'expliquer ? (Voir « Un réseau qui apprend le XOR ».)
8. Ajoutez une deuxième couche cachée de deux neurones, et faites quelques entraînements. Qu'observez-vous par rapport à la question précédente ?
9. Retirez toutes les couches cachées, et n'activez que la caractéristique X_1X_2. Entraînez le modèle plusieurs fois. Qu'observez-vous, et comment l'expliquer ? Comparez avec la question 6 : dans chaque cas, qui a construit la caractéristique qui rend le problème séparable ? (Voir « Linéaire ou non-linéaire ».)

Le cercle

10. Choisissez le jeu du cercle, avec un groupe de points au centre entouré d'un anneau.



Sans couche cachée, et avec seulement X_1 et X_2, est-il possible de résoudre ce problème ? Pourquoi ?

11. Remplacez X_1 et X_2 par $X_1{}^2$ et $X_2{}^2$. La situation change-t-elle ? Comment l'expliquer ? Indice : pensez à l'équation d'un cercle. Quel lien faites-vous avec l'astuce du noyau du Module 2 ? Si la perte reste élevée, réinitialisez et relancez : il arrive que l'entraînement parte dans une mauvaise direction.

12. Revenez à X_1 et X_2 seulement. Peut-on réduire la perte en ajoutant des couches cachées ? Comparez avec la question précédente.

Entraîner trop longtemps

L'application ci-dessous est réglée autrement. Le jeu des deux groupes est très bruité, seuls 20 % des points servent à l'entraînement, et le réseau compte trois couches cachées de huit neurones. Les autres points forment le **jeu de test** (*test set*). Ils sont affichés avec un contour, et leur perte, la « perte de test », est indiquée au-dessus de la perte d'entraînement. Ne modifiez pas le ratio des données d'entraînement.

Applet interactive : à utiliser en ligne, sur la page « Travail noté 3 » du site du cours.

13. Lancez l'entraînement et laissez-le tourner longtemps, plusieurs milliers d'époques. Observez les deux pertes et la frontière. Que se passe-t-il ? Quel nom donne-t-on à ce phénomène ? Comment pourrait-on l'éviter ? (Voir « Généraliser » et « L'entraînement en pratique ».)

Pour aller plus loin

Le podcast MLST

Le podcast Machine Learning Street Talk (en anglais seulement) est un excellent mélange de discussions philosophiques, scientifiques et techniques à propos de ce qui se passe en IA :

Vidéo : https://www.youtube.com/playlist?list=UUMLtBahI5DMrt0NPvDSoIRQ

Présentations diverses

Cette présentation (en anglais seulement) fait un excellent survol des liens profonds entre les mathématiques, la philosophie et l'intelligence artificielle :

Vidéo : https://www.youtube.com/watch?v=Lqt7TgYk8rU

Transcription traduite en français

En gros, on y présente 4 paradigmes mathématiques et leur correspondance dans le monde de l'intelligence artificielle :

1. L'algèbre booléenne et l'intelligence "classique" (GOFAI)
2. Les espaces vectoriels et l'apprentissage automatique
3. Les "variétés" (manifolds) et les espaces latents
4. La symétrie et l'invariance

Conclusion

Merci d'avoir participé à ce cours d'initiation à l'IA !

Au fil des modules, vous avez découvert les fondements de l'intelligence artificielle, ses principaux paradigmes, les réseaux de neurones, l'apprentissage automatique, l'IA générative et les grands modèles de langage. Vous avez exploré les enjeux techniques, philosophiques et éthiques qui accompagnent le développement de ces technologies.

L'IA transforme déjà de nombreux domaines : santé, éducation, industrie, arts, communication… Elle ouvre des perspectives fascinantes, mais soulève aussi des questions sur la responsabilité, la transparence et l'impact social.

Nous espérons que ce parcours vous a permis de mieux comprendre les mécanismes de l'IA, d'en saisir les opportunités et les limites, et de développer un regard critique et informé sur ses usages.

L'aventure ne fait que commencer : continuez à explorer, à questionner et à apprendre !