Complément d'enquête

Les mathématiques de l'apprentissage

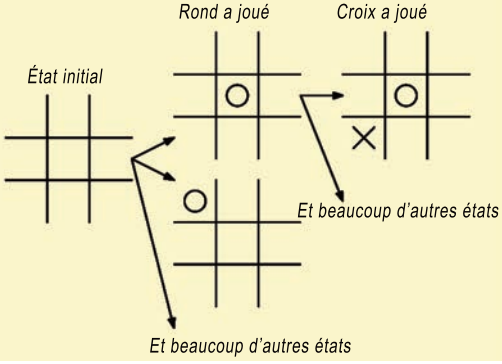
Quentin Labernia

Chercheur en apprentissage automatique
et intelligence artificielle à Corpy&Co

Les années 2000 ont apporté leur lot de nouveautés, et la plus en vogue en ce moment est certainement les réseaux de neurones profonds, dont il a été question dans plusieurs articles précédents. Mais l'IA, ce ne sont pas que des réseaux de neurones, et si vous en avez assez de n'entendre parler que d'eux, suivez-nous ! Découvrons quatre exemples de techniques d'intelligence artificielle (IA) qui ne requièrent pas de réseaux de neurones : rechercher une aiguille dans une botte de foin, gérer le raisonnement avec Prolog, apprendre et interpréter avec des arbres de décision, et laisser un agent renforcer son comportement par lui-même.

Morpion, recherche dans les grands espaces et heuristiques

Savez-vous comment gagner à coup sûr au morpion ? En prenant un papier et un crayon, et un peu de temps, il est possible d'énumérer toutes les combinaisons possiblesqui partent d'une grille vide. Prenez ensuite le chemin qui vous mène à la victoire ! Mais n'essayez pas avec le jeu de go, vous n'aurez pas assez d'atomes dans l'univers pour écrire l'ensemble des combinaisons… Il convient donc de faire preuve de stratégie pour gagner au go ; ces stratégies s'appellent des heuristiques.



L'état initial (à gauche) correspond à une grille vide. Lorsque c'est le tour du joueur Rond, on peut dessiner l'ensemble des états possibles qui partent de l'état initial après avoir dessiné un rond. Chaque grille est un nœud de notre graphe G, et les flèches sont les arcs qui relient les états entre eux.

© Q.L.

Considérons un jeu à deux opposants, comme le morpion (de taille 3×3). Les règles de ce jeu lui confèrent une propriété déterministe : avant de jouer, il est possible de dessiner un graphe orienté $G = (V, E)$ dont les nœuds V représentent l'ensemble des états possibles du plateau, et les arcs, une action entreprise par l'un ou l'autre des joueurs : $E = \{\text{croix, cercle}\} \times \{\text{position1, position2… position9}\}$. Un élément v de V est un vecteur à neuf entrées représentant la grille. Lorsque le jeu débute, le système est dans l'état V_0 où la grille est totalement vide. De cet état, il existe exactement neuf arcs sortants (en considérant avoir déjà choisi le premier joueur). Comme il n'est pas sportif d'effectuer l'action de l'opposant lorsqu'il a le dos tourné, nous allons utiliser le graphe G pour obtenir une description précise du « meilleur coup à jouer » à chaque tour et ainsi gagner la partie. L'objectif est de trouver un nœud *terminal* (qui ne possède aucun arc sortant) gagnant. Le problème du morpion se résume donc à un problème d'exploration de graphe ! Le graphe du morpion est relativement « petit » (il possède exactement cinq mille quatre cent soixante-dix-huit nœuds). L'histoire est tout autre pour un jeu comme le go : pour évaluer de manière exhaustive les états possibles avec quatre coups, il faut explorer plus de 10^{11} états…

L'exploration de vastes espaces mathématiques (« rechercher une aiguille dans une botte de foin ») est à la base de nombreuses techniques d'intelligence artificielle. Les réseaux de neurones explorent un espace vectoriel dont le volume croît exponentiellement avec le nombre de couches considérées. Cet espace est continu, à la différence de notre graphe G, qui lui possède une structure discrète. Il est naturellement possible de se déplacer dans un espace continu et différentiable *presque partout* (c'est-à-dire sauf en un nombre fini de points, comme c'est le cas pour les réseaux de neurones) en utilisant la *rétropropagation du gradient*.

Revenons à notre graphe G. La plus simple des techniques est l'énumération exhaustive de l'ensemble des éléments v de V, ce qui n'est réalisable en pratique que lorsque le cardinal de V est « petit ». On utilise alors des *heuristiques* : ce sont des techniques qui réduisent l'espérance du nombre d'éléments rencontrés avant le point d'intérêt (un nœud terminal gagnant). L'*algorithme minmax* ou la *recherche arborescente Monte-Carlo* travaillent dans des espaces discrets et sont de célèbres exemples d'heuristiques. Le premier explore des branches de manière optimisée selon les différentes actions possibles du joueur adverse, alors que la seconde explore aléatoirement des possibilités, joue de manière aléatoire jusqu'à une position terminale et utilise cet état final pour décrire la qualité de l'action prise et ainsi éviter les branches dont l'espérance de gain est faible.

Un moteur d'inférence : l'intelligence en des termes logiques

Explorer, c'est bien. Raisonner, c'est bien aussi. Grâce à un langage de programmation comme Prolog (abréviation de « *programmation logique* ») mis au point en 1972 par Alain Colmerauer et Philippe Roussel, il est possible d'exprimer des raisonnements logiques de manière très naturelle, en les décomposant en faits et en règles : les *faits* sont des concepts de base, figés, alors que les *règles* permettent de mettre en relation les faits. Les nombres entiers (1, 2…) et le signe d'addition, ce sont des faits. Une règle bien connue est la suivante : $1+1=2$. Voilà le cœur de Prolog : comment encoder cette logique.

Considérons la création d'un plan de table à trois personnes (a, b, c) pour illustrer Prolog (les trois sièges forment une ligne droite). Parlons des faits. Dans le cas de notre plan de table, il s'avère que la personne c veut être à la gauche de b : c'est une définition que l'on impose *a priori*. Déclarons donc le fait comme suit : ordre(c, b). Le fait est une fonction *n*-aire (elle peut prendre un nombre quelconque d'arguments) : elle a donc un nom, « ordre », et dans le cas présent deux arguments ($n=2$), c et b. Petit point de rigueur : en Prolog, toutes les lignes se terminent par un point final.

Passons aux règles. Voilà déjà quelques règles de syntaxe : les majuscules en argument correspondent à des variables, alors que les faits sont constitués de minuscules. La virgule dénote le ET logique alors que le point-virgule correspond au OU. Une règle se construit comme suit :

```
nomregle(arguments) :– regle.
```

Définissons la règle R1 :

```
ordre (A, B, C) :– ordre(A, B); ordre(B, C).
```

Ajoutons une autre règle, R2 :

```
plandetable(A, B, C) :– permutation([a, b, c], [A, B, C]), ordre(A, B, C).
```

Nous venons de définir deux règles R1 et R2. *Inférer* une règle, c'est trouver la valeur des variables qui se trouvent dans les arguments à gauche de la règle. Le mécanisme est simple : le moteur d'inférence de Prolog va chercher les faits qui valident les règles. Si l'on demande à Prolog la requête suivante : « plandetable(A, B, C) », il nous renvoie le résultat : A = *a*, B = *c* et C = *b*. La règle R1 peut alors se lire :

« Chercher les valeurs de A, B et C telles que ordre(A, B) existe
ou que ordre(B, C) existe. »

Quant à la règle R2, elle peut s'énoncer :

« Chercher les valeurs de A, B et C telles que nous avons une permutation de [A, B, C]
qui respecte également la règle ordre R1. »

Il est aussi possible de demander à Prolog plandetable(b, B, C), c'est-à-dire de fixer *b* au bout à gauche de notre table. Dans ce cas, Prolog nous retourne B = *b* et C = *a*, ce qui satisfait effectivement notre règle. À l'aide d'un fait et de deux règles, nous avons résolu notre problème de plan de table ! En utilisant une base conséquente de faits et de règles, la logique encodée peut devenir très complexe.

L'intelligence en des termes purement logiques consiste à stocker des faits et établir des règles pour combiner ces faits ensemble. Prolog est un langage naturel pour exprimer cette logique. Pour les fous de programmation, il existe une multitude de bibliothèques qui permettent d'utiliser la logique Prolog intégrée dans d'autres langages de programmation (Pylog pour Python, ...).

Interprétabilité, arbres, forêts aléatoires et industrie musicale

Les réseaux de neurones sont des êtres compliqués : il est terriblement difficile de comprendre pourquoi ils ont pris telle ou telle décision. C'est là un de leurs grands défauts. Les arbres de décisions, sont, eux, beaucoup plus terre à terre, puisqu'un humain peut déchiffrer sans souci chaque étape qui a mené à leur décision en suivant un *diagramme de décision*. On peut également combiner ces arbres de décisions en organisant un vote, telles les présidentielles (mais entre sorties d'arbres de décisions) : il s'agit de *forêts d'arbres décisionnels*. Elles offrent un avantage statistique indéniable. Mais quel est le rapport avec l'industrie musicale ?

Les programmes logiques créés avec le langage Prolog sont de remarquables exemples d'algorithmes de décision dits « interprétables ». Des secteurs comme les assurances, les banques ou le domaine médical ont besoin d'interprétabilité : étant donné le vecteur de paramètres d'entrée *x* d'un algorithme A, on dit que A est *interprétable* lorsqu'il est décomposable en une suite de décisions, que l'on peut comprendre simplement. Les réseaux de neurones sont difficilement interprétables : à chaque couche, l'espace est déformé sur la base d'une combinaison linéaire des descripteurs en entrée. Il n'est pas rare qu'à cette étape, le réseau de neurones mélange des choux et des carottes, sans se soucier de leur sémantique. Il est ensuite très difficile d'apporter une quelconque interprétation au résultat obtenu !

Les arbres de décision sont un modèle d'apprentissage automatique supervisé qui, au contraire, font preuve d'une interprétabilité naturelle. Un *arbre de décision* D = (V, E) est un graphe acyclique orienté, où l'on associe à chaque nœud *v* de l'ensemble V une fonction de décision. Une fonction de décision prend en entrée une donnée $x = (x_1, x_2 \dots x_N)$, où chaque élément est un *descripteur* (variable contenant une information), et décide ensuite du nœud suivant où aller *via* un arc *e* de l'ensemble E. Typiquement, les inéquations de la forme $f(x) > t$ sont souvent utilisées dans les arbres de décision *binaires*

(pour lesquels il existe deux arcs sortants de chaque nœud non terminal). La valeur *t* est également mise à jour au cours de l'apprentissage. Chaque donnée d'entrée *x* va alors cheminer dans le graphe D jusqu'à rencontrer un nœud terminal : la valeur de ce nœud est la réponse souhaitée ! Un exemple d'algorithme de classification supervisé très fameux est C4.5 (Ross Quilan, 1993), autrement appelé J48 et très utilisé dans l'industrie du disque.

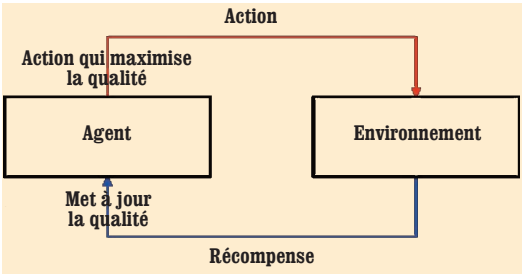
Comme l'union fait la force, des chercheurs ont eu l'idée de considérer un ensemble d'arbres de décision, et donc de créer *une forêt d'arbres décisionnels*. Chaque arbre de décision est entraîné sur un sous-ensemble des données (X, Y) qui sont tirées aléatoirement avec remise, et un sous-ensemble de descripteurs, tirés aléatoirement sans remise. Lorsqu'une donnée *x* arrive en entrée de la forêt, elle est évaluée par l'ensemble des arbres de décision selon les modalités de chaque arbre. Le nombre de réponses y_i est égal au nombre d'arbres. Un vote est organisé et la réponse majoritaire est retournée. Cette technique est appelée *apprentissage ensembliste*. Elle possède un effet de masse qui évite souvent les réponses farfelues grâce aux tirages aléatoires qui génèrent chaque arbre.

L'apprentissage par renforcement pour les agents intelligents

Les arbres ne vous plaisent pas ? Imaginez-vous alors sur une île déserte… Vous ne vous étiez pas préparés à une telle aventure et il est maintenant nécessaire de survivre. Vous êtes un *agent* : un système autonome qui évolue dans son environnement. Après avoir mangé des crabes délicieux et des serpents venimeux, vous vous rendez compte que vous n'êtes pas malade après avoir mangé les crabes, *a contrario* des serpents. Vous apprenez de vos erreurs et essayez de maximiser la qualité de votre état de santé : c'est tout le principe de l'*apprentissage par renforcement* ! Cette méthode est très utilisée pour la mise au point de systèmes robotiques ou d'agents « intelligents » : un robot devant se déplacer de manière autonome dans un labyrinthe, un capteur pivotable qui suit le soleil pour maximiser la quantité d'énergie reçue…

L'apprentissage par renforcement est une technique supervisée où l'agent « apprend » selon les informations provenant de l'environnement. En résumé, l'agent est autonome et la supervision intervient par la prise en compte des retours de l'environnement et de leur corrélation avec les actions entreprises. Cet environnement peut exister dans la vie réelle ou bien être simulé.

Explorons le *Q-learning*. Dans la même veine que les algorithmes génétiques ou les réseaux de neurones, il existe une forte inspiration provenant de la biologie. En l'occurrence, le principe du *Q-learning* repose sur la notion de *qualité* Q. Cette qualité est définie à chaque instant pour l'ensemble des actions A réalisables par l'agent dans des conditions S. Chaque action octroie une certaine récompense R à l'agent. La qualité mesure la récompense espérée



L'agent évolue dans son environnement. Il choisit de réaliser une action qui maximise la qualité (que représente son acquis) qui va avoir un impact sur l'environnement. Il reçoit en retour une récompense, qu'il utilise pour mettre à jour la qualité de l'action choisie (était-ce une action adaptée à la situation ?).

© *Q.L.*

en effectuant une action spécifique. La fonction Q prend en argument l'état du système et une potentielle action, puis calcule un nombre réel qui mesure la qualité. L'agent a alors intérêt à effectuer l'action qui maximise la qualité (par exemple, manger un crabe délicieux plutôt qu'un serpent venimeux).

Prenons l'exemple d'un robot à quatre pattes, l'agent, qui se trouve sur une table, l'environnement. On définit les actions possibles pour l'agent : $A = \{\text{bouger patte 1}\ldots\ \text{bouger patte 4}\}$. Les états du système sont $S = \{\text{avance, recule, sur-place, à gauche, à droite}\}$. La fonction de récompense est fixée arbitrairement, constante, égale à une valeur Q_0.

Au temps t_0, on choisit une action parmi celles qui maximisent la récompense, ici, la vitesse du robot (ou encore, votre état de santé). Au temps t, après avoir choisi une action $a(t)$ dans A, l'agent observe l'état du système $s(t)$ dans S et met à jour la qualité en se basant sur la formule suivante :

$$Q_{new}(s(t), a(t)) = Q(s(t), a(t)) + \text{Facteur}_{\text{apprentissage}} \times (R + \text{Estimation}_{\text{qualité}}(t+1) - Q(s(t), a(t))).$$

La qualité Q est mise à jour avec la récompense effectivement reçue et une estimation de la future valeur de qualité. En pratique, la fonction Q est une simple table d'association dont les lignes sont les états possibles S du système, et les colonnes sont les différentes actions A que l'agent peut effectuer. L'algorithme est indépendant du modèle qui décrit la récompense Q. À la place d'une table d'association, on pourrait par exemple utiliser… un réseau de neurones ! Dans ce cas, la fonction, définie sur un domaine continu, peut généraliser à des paires (a, s) non précédemment rencontrées.

Ces quelques exemples illustrent que le paysage de l'IA ne devrait plus être uniquement constitué de réseaux de neurones. De nombreux autres modèles d'apprentissage supervisé sont utilisés, comme les machines à vecteurs de support. Des méthodes d'apprentissage non supervisées permettent également de compresser l'information, de découvrir des tendances ou des groupes de données similaires, d'encoder la sémantique des mots dans des phrases… En multipliant les approches et les outils, on augmente le champ des possibles !

Q. L.