ChatGPT

Fiches de révision – Intelligence Artificielle

Pourquoi construire des IA ?

Définition et objectifs de l'IA : L'intelligence artificielle (IA) désigne l'ensemble des théories et techniques visant à créer des machines capables d'accomplir des tâches requérant normalement l'intelligence humaine (raisonnement, apprentissage, perception, etc.). Les objectifs de l'IA incluent l'automatisation de tâches complexes, l'aide à la décision, la résolution de problèmes à grande échelle et la modélisation du fonctionnement de l'intelligence pour mieux la comprendre. En construisant des IA, on cherche à **augmenter les capacités humaines** (par exemple analyser de vastes données, contrôler des robots en environnements hostiles) et à **comprendre les mécanismes de la cognition** en les reproduisant artificiellement. Des exemples courants d'applications de l'IA sont les assistants vocaux, les systèmes de recommandation, les véhicules autonomes ou encore les programmes jouant aux échecs.

IA faible vs IA forte : On distingue deux conceptions clés de l'IA : l'**IA faible** (ou *IA étroite*) et l'**IA forte** (liée à l'idée d'IA *générale*). L'IA faible correspond à des systèmes conçus pour des tâches spécifiques et limitées. Ces programmes **simulent l'intelligence** sans posséder de conscience ni de compréhension réelle. Ils suivent des algorithmes prédéfinis et n'agissent qu'à l'intérieur d'un domaine restreint (par ex. un assistant vocal qui comprend des commandes mais n'a pas de « sens commun »). Au contraire, l'IA forte désigne une intelligence artificielle **générale et autonome**, capable de comprendre et d'apprendre n'importe quelle tâche intellectuelle au même titre qu'un humain. Une IA forte aurait une forme de **conscience** ou de véritable compréhension sémantique de ce qu'elle fait. À ce jour, l'IA forte reste hypothétique et n'existe pas dans la pratique – les systèmes actuels relèvent tous de l'IA faible, aussi sophistiqués soient-ils.

Le test de Turing (agir « comme un humain ») : Historique et conceptuellement, Alan Turing proposa en 1950 un critère opérationnel pour juger de l'intelligence d'une machine. Le **test de Turing** consiste à faire converser, via une interface texte, un juge humain avec un interlocuteur caché qui est soit un humain soit une machine. Si le juge **n'arrive pas à distinguer** la machine de l'humain à l'issue de la conversation, alors on considère que la machine a fait preuve d'intelligence (elle **"imite" l'intelligence humaine** de manière convaincante). Ce test place l'accent sur la capacité d'une IA à *agir comme un humain* du point de vue comportemental. Il a fortement influencé la vision de l'IA dans les débuts : réussir le test de Turing était perçu comme un objectif pour prouver qu'une machine peut penser. Toutefois, ce test ne dit rien sur le fonctionnement interne de la machine (il ne requiert pas qu'elle *pense comme un humain*, seulement qu'elle agisse de façon indiscernable).

Débats et critiques (Searle, French) : La question « **une machine peut-elle vraiment penser ?** » a suscité des débats philosophiques. Le philosophe **John Searle** a proposé en 1980 l'argument de la *chambre chinoise* pour contester l'idée d'une IA forte. Il imagine qu'un homme, ne comprenant pas le chinois, suit un manuel de règles pour répondre à des messages en chinois de façon indiscernable d'un locuteur natif. De l'extérieur, il **donne l'illusion de comprendre** le chinois, alors qu'en réalité il ne fait que manipuler des symboles syntaxiquement. Par analogie, Searle affirme qu'un ordinateur exécutant un programme peut sembler intelligent (répondre correctement) sans **aucune compréhension sémantique**. Pour Searle, les ordinateurs actuels traitent des symboles (syntaxe) mais sont dépourvus d'intentionnalité et de conscience (sémantique), si bien qu'**ils ne « pensent » pas réellement**. Il conclut que l'IA forte (une machine *qui comprend et pense*) est hors de portée en se basant uniquement sur des algorithmes formels.

Un autre chercheur, **Robert French**, a critiqué le **test de Turing** en tant que mesure de l'intelligence. Il souligne que ce test ne vérifie qu'une **ressemblance à l'intelligence humaine**. French donne l'analogie suivante : si un peuple ne connaît qu'une espèce d'oiseau (la mouette) et juge qu'une machine vole uniquement si elle est impossible à distinguer d'une mouette, alors un avion ou un hélicoptère échouerait à ce *« test de la mouette »* bien qu'ils volent bel et bien. De même, une IA pourrait être très intelligente sans forcément adopter un comportement humain, ou inversement imiter l'humain sans être intelligente. **Conclusion de French** : le test de Turing est **un test de similitude avec l'humain, non un véritable test d'intelligence** universelle [1] . Ces critiques invitent à distinguer la *performance extérieure* d'une IA (ce qu'elle affiche) de sa *compréhension interne*, et à reconnaître que l'intelligence peut prendre des formes non humaines.

En résumé, **construire des IA** répond à des motivations scientifiques et pratiques : résoudre des problèmes complexes, accroître l'efficacité dans de nombreux domaines (médecine, transport, communication…), et explorer la nature de l'intelligence. Cependant, cela soulève des enjeux conceptuels (jusqu'où une machine peut-elle penser ? qu'est-ce que comprendre ?), ainsi que des considérations éthiques et sociétales (impact sur le travail, responsabilité des décisions prises par des IA, etc., bien que ces aspects dépassent le cadre de cette fiche). Les distinctions entre IA faible et forte, ainsi que les critiques de Searle et French, rappellent qu'il y a une différence entre **simuler l'intelligence** sur des tâches précises et **réellement posséder** une intelligence ou une conscience comparable à celle de l'humain.

Les agents intelligents et leurs environnements

Notion d'agent intelligent : En IA moderne, la notion centrale est celle *d'agent*. Un **agent intelligent** est **une entité capable de percevoir son environnement et d'y agir de façon autonome**. Plus formellement, **un agent perçoit** l'état de son environnement au moyen de **capteurs** (ou entrées) et **agit** sur cet environnement au moyen d'**effecteurs** (ou sorties, actionneurs). Cette interaction sensorimotrice forme le cycle *perception* → *décision* → *action*. Par exemple, un agent humain a des yeux, oreilles, etc. comme capteurs, et des mains, jambes, voix… comme effecteurs; un agent logiciel peut percevoir des données d'un serveur et agir en envoyant des commandes. On appelle **percept** la perception à un instant donné (les informations que reçoit l'agent à un instant t), et **séquence de percepts** l'historique complet de tout ce que l'agent a perçu. L'**agent** choisit ses actions en fonction de ces percepts (présents et passés) dans le but d'accomplir **une tâche ou un objectif** fixé. On peut associer à un agent une **fonction de performance** qui mesure à quel point ses actions sont efficaces pour atteindre ses objectifs dans l'environnement.

Environnements de tâches : L'**environnement** est le monde (réel ou virtuel) dans lequel l'agent opère. La nature de l'environnement influence fortement la conception de l'agent. On caractérise un environnement selon plusieurs propriétés :

- **Complètement observable vs partiellement observable :** Si les capteurs de l'agent peuvent obtenir **à tout instant l'état complet** de l'environnement pertinent, il est *complètement observable* (ex : aux échecs, l'agent voit tout l'échiquier). Sinon, il est *partiellement observable* (ex : dans la conduite automobile, l'agent n'a qu'une vision partielle limitée par ses caméras, angles morts…).
- **Déterministe vs stochastique :** Dans un environnement **déterministe**, l'état suivant est entièrement déterminé par l'état courant et l'action de l'agent (il n'y a pas de place pour le hasard). Par exemple, un jeu de dames est déterministe. Un environnement **stochastique**

comporte une part de **hasard ou d'incertitude** dans l'évolution des états (ex : la météo pour un drone est imprévisible, ou un jeu incluant des dés).

- **Épisodique vs séquentiel :** Si la tâche peut être décomposée en **épisodes indépendants** (chaque perception-action est sans influence sur les suivantes), l'environnement est épisodique. Par exemple, identifier des images présentées une par une est épisodique (chaque image est un cas indépendant). **Séquentiel** signifie que les expériences s'enchaînent avec des dépendances : l'action courante influence les situations futures. La plupart des problèmes (conduire, jouer à un jeu stratégique…) sont séquentiels.
- **Statique vs dynamique :** Un environnement **statique** n'évolue pas pendant que l'agent décide ; s'il évolue (ou que d'autres processus/agents le font changer) *en même temps* que l'agent réfléchit, il est **dynamique**. Exemples : un puzzle posé sur table est statique (il ne change pas tout seul), mais la conduite routière est dans un environnement dynamique (la circulation continue d'évoluer pendant que l'agent délibère).
- **Discret vs continu :** Si l'état de l'environnement et le set d'actions sont définis par un nombre fini de valeurs distinctes, l'environnement est **discret** (ex : échiquier avec cases finies, mouvements précis). S'il peut varier de façon continue (infinie de valeurs possibles), on parle d'environnement **continu** (ex : position d'un robot dans un espace 2D est continue en coordonnées réelles).
- **Monoagent vs multi-agents :** S'il n'y a qu'un seul agent autonome au sein de l'environnement (en plus éventuellement de phénomènes non contrôlés), le problème est monoagent. En présence d'autres **agents actifs** (par exemple plusieurs robots ou plusieurs joueurs), on est en environnement multi-agents, ce qui introduit des interactions pouvant être compétitives ou coopératives.

Architecture des agents : La manière dont un agent prend ses décisions peut être décrite en plusieurs **architectures types**, du plus simple au plus élaboré :

- **Agent réflexe simple :** Il **agit uniquement en réaction aux percepts courants**, en appliquant des règles de type condition-action (*si percept X alors action Y*). Il ne tient pas compte de l'historique des percepts. Ce type d'agent convient pour des environnements simples et entièrement observables. *Exemple :* un thermostat est un agent réflexe (si température perçue < seuil, alors activer chauffage, sinon l'éteindre).
- **Agent réflexe basé sur un état interne :** Il maintient en mémoire un **état interne** qui représente ce qu'il croit de l'environnement (une sorte de modèle du monde) mis à jour à partir des percepts. Cela lui permet de **gérer un environnement partiellement observable** en conservant l'historique pertinent. Ses décisions se basent sur l'état interne + le percept courant. *Exemple :* un aspirateur robot garde en mémoire les zones déjà nettoyées (état interne) pour décider vers où aller ensuite.
- **Agent basé sur des buts :** Il intègre explicitement des **objectifs à atteindre**. En plus de l'état du monde, il connaît la condition de réussite (but) et peut planifier des actions pour l'atteindre. Ce type d'agent fait de la **recherche de plan** (voir fiche suivante) pour choisir une action non seulement réactive, mais orientée vers un résultat futur souhaité. *Exemple :* un GPS routier est un agent à but : il calcule une séquence d'actions (itinéraire) pour atteindre la destination voulue.
- **Agent basé sur l'utilité :** Il se dote d'une **fonction d'utilité** qui, pour un état donné (ou une situation potentielle), fournit une valeur numérique représentant le **degré de satisfaction** ou de préférence de cet état pour l'agent. Cela permet de comparer différents états ou plans selon leur mérite, même si plusieurs aboutissent au but. L'agent cherche à **maximiser son utilité attendue**. Ce cadre est utile quand il y a des **choix optimaux** à effectuer sous incertitude ou avec des objectifs multiples. *Exemple :* un agent de trading financier peut avoir une fonction d'utilité combinant profit espéré et risque ; il choisit les actions maximisant cette utilité.
- **Agent apprenant :** Cet agent possède en plus des mécanismes d'**apprentissage** qui lui permettent d'améliorer son comportement avec l'expérience. Il comporte généralement quatre

modules : 1) un module d'apprentissage qui propose des améliorations (en testant de nouvelles actions ou en ajustant des paramètres), 2) un module critique (qui évalue la performance de l'agent par rapport au standard ou aux retours/récompenses), 3) le module de décision (l'agent lui-même qui choisit les actions), et 4) éventuellement un module de performance initial (pour son comportement de base). L'agent apprenant utilise le feedback reçu de l'environnement (récompenses, succès/échecs) pour modifier sa façon d'agir dans le futur. *Exemple :* un joueur d'échecs automatique qui affine sa stratégie en jouant de nombreuses parties et en apprenant de ses défaites/victoires est un agent apprenant.

Chaque agent peut être vu conceptuellement comme une **fonction** qui à chaque séquence de percepts associe une action à effectuer. L'objectif en IA est de concevoir l'agent (son programme de décision) qui maximise une mesure de performance dans un environnement donné. Le formalisme *PEAS* (Performance, Environment, Actuators, Sensors) est souvent utilisé pour définir le cadre d'un agent : on précise son critère de performance, l'environnement où il opère, ses actionneurs et ses capteurs. Par exemple, pour un agent aspirateur : Performance = pourcentage de propreté + pénalité si trop de mouvements, Environment = deux pièces avec saleté apparaissant aléatoirement, Actuators = se déplacer gauche/droite, aspirer, Sensors = détecteur de saleté sur la case actuelle, détecteur de position mur. L'analyse *PEAS* aide à clarifier le problème avant de concevoir la solution.

En résumé, penser en termes d'**agents** permet de structurer tout problème d'IA : il s'agit de concevoir une entité autonome rationnelle qui perçoit un environnement aux propriétés données et choisit des actions pour atteindre ses objectifs. Les sections suivantes détaillent comment de tels agents peuvent planifier des actions, représenter des connaissances ou apprendre de l'expérience.

La recherche de solutions par exploration d'états (Recherche en IA)

Problèmes d'IA modélisés en espace d'états : De nombreux problèmes peuvent être formulés comme un **espace d'états à explorer**. Un **état** représente une configuration du problème. L'**état initial** est la situation de départ et l'**état but** (ou un ensemble de buts possibles) représente la/les situation(s) désirée(s). L'agent dispose d'un **ensemble d'actions (opérateurs)** permettant de passer d'un état à un autre : chaque action appliquée dans un état donné entraîne un **état successeur**. Cette modélisation forme un graphe (ou arbre) d'états interconnectés par les actions. **Résoudre le problème** équivaut à **trouver une séquence d'actions** menant de l'état initial à un état but. Par exemple, dans le *jeu du 8-puzzles*, chaque disposition des tuiles est un état, l'état initial est la configuration de départ du plateau et l'état but est la configuration triée. Les mouvements possibles (glissement d'une tuile dans le vide) sont les actions qui font passer d'un état à un autre. Le but est de trouver la suite de mouvements pour atteindre la configuration finale.

Algorithmes de recherche non informée (aveugle) : Ce sont des stratégies d'exploration de l'espace d'états **sans information heuristique** (elles n'utilisent pas de connaissance sur la proximité du but, seulement la structure du graphe). Quelques algorithmes classiques :

- *Recherche en largeur d'abord* (**BFS – Breadth-First Search**) : explore tous les états atteignables en 0 action, puis 1 action, puis 2 actions, etc. Autrement dit, on explore couche par couche (par distance croissante depuis l'état initial). BFS garantit de trouver la solution la plus courte (en nombre d'actions) si elle existe, mais consomme beaucoup de mémoire car la frontière de recherche peut être énorme (exponentielle en la profondeur). **Propriétés** : complète (trouve une solution si elle existe), optimale pour minimiser le nombre d'actions (ou le coût uniforme si

toutes les actions ont même coût), complexité en temps et espace O(b^d) où *b* est le facteur de branchement (nb d'actions possibles par état) et *d* la profondeur de la solution.

- *Recherche en profondeur d'abord* (**DFS – Depth-First Search**) : s'enfonce le long d'un chemin d'états jusqu'à atteindre un état terminal ou but, et recule (backtrack) en cas de cul-de-sac, explorant les branches en profondeur les unes après les autres. DFS utilise peu de mémoire (seulement la pile des états courants) et peut être efficace pour trouver rapidement une solution profonde, mais **ne garantit pas la solution optimale**, et même pas de trouver une solution dans des espaces infinis (il peut s'enliser dans une branche infinie). **Propriétés** : non complète en général (sauf si espace fini ou avec profondeur limitée), non optimale, complexité en temps O(b^m) (m = profondeur maximale explorée), complexité en espace O(b*m).
- *Recherche en profondeur itérative* : combine avantage de BFS (complète, optimale en distance) et de DFS (peu gourmand en mémoire) en lançant des parcours en profondeur limités à une certaine profondeur, puis en augmentant progressivement cette limite. Cela permet de trouver la solution la plus courte sans exploser la mémoire, au prix de redondances en temps.
- *Recherche coût uniforme* : une variante de BFS qui prend en compte des coûts d'actions variables. Elle explore par ordre croissant de coût cumulé depuis le départ (équivalent à Dijkstra). Elle est optimale en coût total minimal, mais peut aussi être lourde en calcul.

Recherche informée (heuristique) : Pour les problèmes complexes, on utilise des **heuristiques** afin de guider la recherche plus efficacement vers le but. Une **heuristique** est une estimation numérique de la distance restante ou du coût restant pour atteindre un état but depuis un état donné (c'est une fonction $h(n)$ qui évalue un état *n*). Une bonne heuristique permet de **prioriser** les états prometteurs. Un algorithme emblématique est **A** *(A-étoile), qui combine le coût déjà accumulé $g(n)$ et l'heuristique $h(n)$ pour évaluer un état via $f(n) = g(n) + h(n)$. A* explore les états dans l'ordre croissant de $f(n)$ (coût estimé total depuis le départ jusqu'au but en passant par *n*).

- Si $h(n)$ est **admissible** (toujours $\le$ au coût réel restant), A *est complète et optimale** (elle trouve un chemin de coût minimal). L'heuristique permet souvent de dramatiquement réduire le nombre d'états explorés comparé à une recherche aveugle.
- *Exemple d'heuristique :* Pour le jeu du 8-puzzles, $h_1(n)$ = nombre de tuiles mal placées, ou $h_2(n)$ = somme des distances de Manhattan de chaque tuile à sa position cible, sont des heuristiques admissibles qui orientent A* plus efficacement vers l'état final.
- A *a une complexité en temps et mémoire qui reste exponentielle dans le pire cas, mais en pratique une bonne heuristique fait la différence entre un problème insoluble et un problème résolu. D'autres variantes existent (comme* IDA – *A itératif en profondeur, ou* algorithmes méta-heuristiques* type Simulated Annealing, recherche localisée, etc.) pour gérer les très grands espaces.

Exemple illustratif : Supposons un agent de livraison qui doit trouver le plus court chemin routier pour aller de la ville A à la ville B. En formulant cela en problème de recherche : chaque **état** est la position actuelle du véhicule; l'**état initial** est A, un **état but** est B; les **actions** sont emprunter un tronçon de route vers une ville voisine. Une recherche en largeur trouverait l'itinéraire en minimisant le nombre d'étapes (peu pertinent ici car les routes ont des distances différentes). Une recherche en coût uniforme trouverait l'itinéraire le plus court en distance totale. Pour accélérer la recherche, on peut utiliser une **heuristique** h estimant la distance restante (par exemple la distance à vol d'oiseau jusqu'à B) : l'algorithme A* explorera en priorité les routes allant globalement vers la destination, trouvant rapidement la route la plus courte. Sans heuristique, le GPS devrait explorer un très grand nombre de routes possibles sans priorité.

Limites de la recherche systématique : La recherche combinatoire souffre souvent de « **l'explosion combinatoire** » – le nombre d'états croît exponentiellement avec la taille du problème. Même avec de bonnes heuristiques, certains problèmes restent inabordables avec les approches de recherche classiques (par exemple les échecs ont un espace d'états astronomique). Cela motive d'autres approches

comme la **recherche locale** (on optimise une configuration initiale par petites modifications, utile pour les problèmes d'optimisation), la **recherche heuristique avancée** et aussi, dans un autre registre, l'utilisation de **connaissances** pour guider la résolution (règles spécifiques, ou apprentissage de stratégies). Néanmoins, le cadre général de la recherche en espace d'états est un fondement théorique essentiel de l'IA, applicable dès qu'on peut formaliser un but et des actions – il apparaît dans la planification, les jeux, la résolution de puzzles, le routage, etc.

La planification automatique

De la recherche à la planification : La **planification** en IA vise à élaborer **à l'avance une séquence d'actions** pour atteindre un but. C'est une forme particulière de recherche où l'on manipule des **actions explicites avec leurs conditions et effets**. Alors que la recherche d'un chemin considère chaque état individuellement et des actions sans structure particulière, la planification introduit souvent un langage formel pour représenter **l'état du monde** (par des variables, des prédicats logiques) et les **actions** (avec des préconditions et des effets). Le système de planification doit trouver une suite d'actions telle que, si on les applique depuis l'état initial, on satisfait finalement la condition d'objectif.

Représentation d'un problème de planification : Classiquement, un problème de planification est décrit par : - Un **état initial**, défini par un ensemble de faits atomiques vrais (par ex. `Position(Robot, A) ∧ Libre(B)` etc.). - Un **but** qui est une condition (ensemble de littéraux) à satisfaire (par ex. `Position(Robot, B) ∧ ¬Libre(B)` signifiant le robot doit être à B en ayant occupé B). - Un ensemble d'**actions** (opérateurs) disponibles. Chaque action est décrite par: - des **préconditions** (les conditions devant être vraies pour pouvoir l'appliquer), - des **effets** (liste de faits ajoutés et retirés de l'état quand l'action est exécutée). - L'exécution d'une action modifie l'état courant selon ses effets (suppression/ajout de faits).

Par exemple, dans un monde de blocs à empiler (*blocks world*), une action pourrait être `Empiler(x, y)` avec comme précondition que le bloc *x* est libre et le bloc *y* est libre, et comme effet que *x* est sur *y*, *x* n'est plus libre, et l'ancienne position de *x* devient libre. Un plan pour atteindre un but (une configuration de blocs souhaitée) serait une suite de telles actions qui conduit de l'état initial à l'état final désiré.

Algorithmes de planification : La planification peut se ramener à une recherche dans l'espace des états (comme vu précédemment) où chaque *action applicable* depuis un état est une transition. Cependant, l'espace est souvent gigantesque. Les systèmes de planification utilisent des techniques dédiées : - **Planification progression vs régression :** La *progression* part de l'état initial et applique des actions en avant jusqu'à atteindre le but (recherche en avant). La *régression* part du but et raisonne à rebours en trouvant quelles actions pourraient précéder le but (on remonte les préconditions, c'est la recherche à rebours). Ces deux approches explorent l'espace d'état, mais la régression peut éviter des chemins inutiles en se focalisant sur la satisfaction du but. - **Planification heuristique :** Comme pour A*, on définit des heuristiques qui estiment la distance d'un état au but (souvent en simplifiant le problème, ex : en ignorant certaines contraintes). Des planificateurs modernes calculent des heuristiques (comme* h *obtenue par un problème relaxé sans certaines contraintes) pour guider la recherche plus efficacement. - Planification partielle (ou PO – Partial-Order Planning) : Cette approche n'impose pas totalement l'ordre des actions dès le départ. On construit un plan partiel (un ensemble d'actions avec des liens de précédence et des contraintes) que l'on raffine progressivement. Cela permet d'introduire des actions "indépendantes" sans décider tout de suite de leur ordre relatif. À la fin, on obtient un ordonnancement complet. Ce type de planification est utile quand les sous-problèmes peuvent être résolus indépendamment et combinés. - Algorithmes spécifiques : Il existe de nombreux algorithmes et systèmes (par ex. GraphPlan, qui construit un* graph de planification

couches par couches alternant états et actions pour trouver un plan minimal, ou les planificateurs SAT *qui traduisent le problème en clauses logiques satisfiables, ou encore* PLANNERS* basés sur des MDP si incertitude, etc.). Pour la planification classique déterministe, des compétitions de planification ont produit des algorithmes très optimisés (en progression avec heuristiques, en SAT, en recherche locale…).

Exemples de planification : - En **robotique mobile**, la planification de trajectoire consiste à trouver un chemin dans l'espace continu pour aller du point A au point B en évitant des obstacles. On modélise l'espace continu en un ensemble d'états atteignables (par ex. en discrétisant l'espace en grille) et on utilise des algorithmes de planification / recherche (souvent A *sur la grille, ou RRT pour de la planification de mouvement continue) pour calculer un chemin praticable. - En logistique (planification de tâches), un planificateur automatique peut organiser les livraisons de colis par une flotte de camions : l'état comprend la localisation de chaque colis et camion, les actions sont « prendre un colis X à l'emplacement Y », « déposer colis X à Z », etc., et le but est tous les colis livrés. Le plan trouvé serait une séquence d'actions pour chaque agent réalisant la livraison optimisée. - Dans les jeux vidéo ou la domotique**, la planification permet à un agent de formuler une suite d'actions pour atteindre un objectif (par ex. un personnage non-joueur planifie une route à travers un niveau, ou une IA de maison intelligente planifie l'ordre d'allumage des appareils pour économiser l'énergie).

Lien avec la recherche et difficultés : La planification est en théorie un cas particulier de recherche, mais souvent plus complexe : les états sont riches (combinaisons de faits), le nombre d'actions possible très grand, et des défis supplémentaires apparaissent (la fameuse **explosion combinatoire** du nombre de plans, ou le **frame problem** : comment représenter efficacement ce qui ne change pas après une action sans énumérer une multitude de faits inchangés). Les problèmes de planification classiques sont souvent **NP-complets** voire pire (PSPACE-complets), ce qui signifie qu'ils deviennent rapidement intraitables à grande échelle. Les planificateurs doivent donc exploiter au maximum la structure du problème (contraintes, indépendances entre sous-problèmes, heuristiques puissantes). Malgré ces difficultés, la planification est un domaine clé de l'IA, car de nombreuses tâches du monde réel exigent d'**anticiper une séquence d'actions** plutôt que de réagir pas à pas. Dans les agents intelligents, la planification permet à un agent **basé sur des buts** de décider non seulement *quoi* faire, mais *comment* le faire étape par étape.

La représentation des connaissances

Pourquoi représenter des connaissances ? Pour être intelligent de manière générale, un agent doit non seulement réagir ou chercher des plans, il doit aussi **savoir des choses** sur son monde, et pouvoir **raisonner** à partir de ces connaissances. La **représentation des connaissances** est le domaine de l'IA qui étudie comment formaliser des informations et des règles sur le monde de façon exploitable par une machine. Une bonne représentation doit permettre à l'agent de **tirer de nouvelles conclusions (inférence)**, de **prendre des décisions éclairées**, et d'**interpréter ses perceptions** en les reliant à un savoir plus large (par ex. comprendre qu'une scène avec un chat implique un animal, qui a des propriétés connues, etc.). C'est un volet crucial notamment pour les systèmes experts, la compréhension du langage naturel, la vision par ordinateur (reconnaître des objets implique de relier des perceptions à des concepts), etc.

Logiques formelles : Un moyen classique de représenter des connaissances est d'utiliser la **logique**. En IA, on emploie notamment : - La **logique propositionnelle** (ou booléenne) : on y représente des faits par des propositions atomiques (P , Q , etc.) qui peuvent être vraies ou fausses, et on construit des **formules logiques** avec des connecteurs ($\wedge$, $\vee$, $\Rightarrow$, $\neg$). Par exemple, on peut avoir des axiomes

comme `Pluie ⇒ SolMouillé` (s'il pleut alors le sol est mouillé). Les raisonnements en logique booléenne utilisent des règles de déduction (ex : *modus ponens* : de `Pluie ⇒ SolMouillé` et `Pluie` on déduit `SolMouillé`). La logique propositionnelle permet des déductions, mais elle est limitée car elle ne reflète pas la structure interne des faits (pas de variables ni d'objets). - La **logique des prédicats (logique du premier ordre)** : plus puissante, elle introduit des **prédicats** qui prennent des objets en arguments et des **quantificateurs** (`∀, ∃`). On peut ainsi exprimer des connaissances plus générales. Par ex : `∀x Chat(x) ⇒ Mammifère(x)` (tous les chats sont des mammifères), `Chat(Mistigri)` (Mistigri est un chat). De là on peut inférer `Mammifère(Mistigri)`. La logique des prédicats permet de représenter des relations entre objets et de raisonner avec des formules plus complexes (par un processus de **preuve** ou de **résolution** automatique). - Il existe d'autres logiques spécialisées (logique modale, temporelle, floue, etc.) selon les besoins (représenter de l'incertitude, le temps, etc.), mais en base, la logique du premier ordre est un cadre très étudié en IA pour la représentation *déclarative* des connaissances.

Un **moteur d'inférence** (par exemple un système de résolution de clauses, ou un raisonneur à base de chaînes d'inférence) peut opérer sur la base de connaissances logique pour **déduire** de nouveaux faits. Cette approche a donné lieu aux **systèmes experts** dans les années 1970-1980 : des programmes qui encodent le savoir d'un domaine sous forme de règles logiques ou de production, et qui répondent à des questions ou résolvent des problèmes en inférant logiquement. *Exemple :* un système expert médical peut avoir des règles `SymptômeX ∧ TestY ⇒ MaladieZ`. En entrant les faits (symptômes constatés, résultats de tests), le système déduit les maladies plausibles.

Représentations structurées et sémantiques : Outre les logiques formelles, l'IA a développé des structures de représentation plus proches de la façon dont les humains organisent le savoir : - **Réseaux sémantiques** : c'est un **graph** où les nœuds représentent des **concepts** ou entités, et les arcs des **relations sémantiques** entre ces concepts. Par exemple, on peut représenter `Canari` –(est un)→ `Oiseau` –(est un)→ `Animal`. Ce réseau encode qu'un canari est un oiseau, lui-même un animal. D'autres relations peuvent exister : `(Oiseau) –(aPourPropriété)→ (PeutVoler)`. L'inférence dans un réseau sémantique peut se faire par propagation : un canari hérite des propriétés de Oiseau et Animal (il peut voler, il est un être vivant, etc.). Les réseaux sémantiques ont évolué en **ontologies**, où les relations sont typées et où les concepts sont organisés de manière hiérarchique rigoureuse (par ex., l'ontologie WordNet en langue, ou des ontologies médicales). Ce sont des bases de connaissances structurées utiles pour le raisonnement par héritage et pour assurer de la **cohérence** dans les connaissances. - **Frames (cadres)** : introduits par Marvin Minsky, les *frames* sont des structures de données qui représentent un **objet ou une situation type** avec un ensemble d'**attributs (slots)** et de **valeurs** ou contraintes possibles. Par ex., on peut avoir un frame pour un « Restaurant » avec des slots comme `Lieu`, `Menu`, `Serveur`, `Client`, `Plat commandé`, etc., dont certaines valeurs peuvent être par défaut. Un frame peut hériter d'un frame plus général (hiérarchie de classes d'objets, proche de la POO). Les frames sont utiles pour représenter du savoir **par défaut** et traiter les exceptions (on peut remplir ou surcharger les slots). Ils ont été utilisés pour la compréhension de scènes ou de textes (frames de situation). - **Systèmes à base de règles de production** : Ici, les connaissances sont encodées sous forme de règles condition-action (productions) du type `SI conditions ALORS actions`. Ce formalisme est utilisé dans les systèmes experts : la base de connaissances est un ensemble de règles, et un moteur d'inférence effectue une chaîne **avant** (inférence progressive des conséquences à partir des faits connus) ou **arrière** (on part d'une hypothèse et on cherche à la justifier par les règles) pour arriver à une conclusion. Par exemple, un système de diagnostic emploie des règles `SI (fièvre ∧ toux) ALORS hypothèse=grippe`. La **chaîne avant** part des symptômes constatés et déduit toutes les hypothèses possibles. La **chaîne arrière** partirait d'une hypothèse comme « grippe » et vérifie en remontant quelles données confirmeraient cette hypothèse.

Connaissances déclaratives vs procédurales : En IA, on distingue souvent ces deux formes. *Déclarative* : des énoncés sur le monde, utilisables par un moteur générique (ex : des faits et règles logiques, indépendants de l'algorithme). *Procédurale* : du savoir-faire encapsulé dans des procédures ou des réseaux de neurones, etc., où la connaissance est implicite dans les calculs. La représentation des connaissances s'intéresse surtout à la forme déclarative, qui facilite l'**explication** (on peut suivre la chaîne logique d'une conclusion) et la **modification** manuelle du savoir.

Défis et problèmes classiques : Représenter beaucoup de connaissances et en tirer des inférences correctes se heurte à plusieurs obstacles : - **Explosion combinatoire** : plus il y a de faits, plus les combinaisons de règles à appliquer explosent. Des mécanismes de filtrage, d'indexation ou d'heuristiques sont nécessaires dans les moteurs d'inférence pour rester efficaces. - **Le problème du cadrage (*frame problem*)** : en logique, quand une action se produit, comment représenter *tout ce qui ne change pas* sans devoir énoncer une infinité de choses qui restent inchangées ? (Ex : si un robot déplace un objet de A à B, tout le reste du monde reste à sa place, mais il faudrait des axiomes pour chaque fait inchangé, ce qui est ingérable). Des solutions partielles utilisent des *lois d'inertie*, des formalismes spécialisés (logiques de situations, etc.). - **Représentation de l'incertain ou de l'incomplet** : Les logiques classiques ne gèrent pas bien l'incertitude ou la probabilité. Pour cela l'IA fait appel à des modèles probabilistes (réseaux bayésiens, logique floue) – c'est un domaine connexe (*raisonnement incertain*). - **Acquisition des connaissances** : Remplir une base de connaissances riche (*le goulot d'étranglement de la connaissance*) est difficile. Historiquement, le manque de connaissances facilement exploitables a limité les systèmes experts. Aujourd'hui, on combine des méthodes d'apprentissage automatique pour extraire des connaissances de données, et des bases de connaissances manuelles (par ex. les *knowledge graphs* de Google, Wikipedia DBpedia, etc.).

En résumé, la représentation des connaissances vise à donner à l'agent un « **modèle du monde** » exploitable pour le raisonnement. Combinée avec un mécanisme d'inférence, elle permet une **intelligence symbolique** capable de manipuler des concepts abstraits. C'est un pendant complémentaire aux méthodes d'**apprentissage automatique** (qui apprennent à partir des données sans représentation explicite fournie par l'humain). Les approches contemporaines tendent à hybrider les deux : par exemple, utiliser des ontologies pour la cohérence et des réseaux neuronaux pour la perception.

L'apprentissage automatique (Machine Learning)

Concept d'apprentissage automatique : Plutôt que de tout programmer explicitement, on peut concevoir des systèmes qui **apprennent par eux-mêmes** à partir de données ou d'expériences. L'**apprentissage automatique** est la branche de l'IA qui développe des algorithmes capables d'**améliorer leur performance** sur une tâche donnée en **tirant parti des données** disponibles (exemples, expériences passées, interactions). Un agent apprenant ajuste ainsi son modèle interne pour mieux atteindre ses objectifs avec le temps. L'apprentissage automatique permet à l'IA de s'attaquer à des problèmes complexes pour lesquels il serait impossible d'écrire à la main toutes les règles (par ex. reconnaître des images, traduire du texte, prédire un phénomène).

Types d'apprentissages : On distingue plusieurs cadres d'apprentissage selon la nature des données d'entraînement et du feedback :

- **Apprentissage supervisé :** L'algorithme apprend à partir d'exemples **étiquetés**. Chaque exemple du jeu de données est une paire *(entrée, sortie désirée)*. L'objectif est de généraliser un modèle capable de prédire la sortie pour de nouvelles entrées. C'est comme apprendre avec un

professeur qui donne la bonne réponse. Les tâches supervisées typiques sont la **classification** (prédire une catégorie pour une instance, ex : diagnostiquer une maladie à partir de symptômes, reconnaître le contenu d'une image) ou la **régression** (prédire une valeur numérique, ex : prix immobilier en fonction de caractéristiques). L'algorithme va ajuster ses paramètres pour minimiser l'erreur entre ses prédictions et les sorties correctes des exemples d'entraînement. *Exemples d'algorithmes supervisés :* les réseaux de neurones (dont le perceptron, cf fiche suivante), les arbres de décision, les k-plus proches voisins, les machines à vecteurs de support (SVM), etc.

- **Apprentissage non supervisé :** Ici, on n'a **pas de sorties correctes fournies** – seulement des données brutes en entrée. L'apprentissage consiste à **découvrir des structures ou des régularités** cachées dans les données. C'est comme apprendre sans professeur, en repérant des motifs. Les tâches typiques incluent la **clustering (regroupement)** où l'on segmente les données en groupes similaires (par ex. segmenter des clients en profils d'achats proches, ou regrouper des documents par thème) et la **réduction de dimension** (trouver comment représenter les données de façon plus simple tout en conservant l'essentiel de l'information). *Exemples de techniques non supervisées :* l'algorithme de k-moyennes pour le clustering, les méthodes de factorisation de matrices et PCA pour la réduction de dimension, les réseaux de neurones auto-encodeurs, etc.
- **Apprentissage par renforcement :** C'est un cadre différent, inspiré de la psychologie behavioriste. Un **agent** apprend en interagissant avec un environnement et en recevant des **récompenses** (ou pénalités) en conséquence de ses actions. Contrairement au supervisé, il n'y a pas de « bonne réponse immédiate » fournie pour chaque situation ; au lieu de cela, l'agent doit **essayer des actions** et apprendre de l'effet (récompense) différé qu'elles produisent. Le but est de **maximiser la récompense cumulée** à long terme. L'agent apprend une **politique** (une stratégie qui indique quelle action prendre dans chaque état) par essai-erreur, grâce à des algorithmes comme Q-learning, SARSA, ou des méthodes par politique. *Exemple classique :* un agent qui apprend à jouer à un jeu vidéo reçoit +1 à chaque victoire et -1 à chaque défaite, sans indication durant le jeu de ce qui est bien ou mal – il doit découvrir quelles séquences d'actions mènent à la victoire. Un autre exemple est la robotique : un robot peut apprendre à marcher en se récompensant quand il avance sans tomber. L'apprentissage par renforcement modélise souvent le problème comme un **Processus de Décision Markovien (MDP)** avec des états, actions, récompenses et probabilités de transition si l'environnement est stochastique. C'est particulièrement utile pour les scénarios séquentiels où chaque décision influence les suivantes et où un feedback n'est disponible qu'en fin de parcours.

Ces trois paradigmes sont les principaux. Il existe aussi l'**apprentissage semi-supervisé** (quand on combine quelques données étiquetées avec beaucoup de données non étiquetées), et l'**apprentissage par transfert** (réutiliser un modèle entraîné sur une tâche comme base pour une nouvelle tâche).

Exemples et succès : L'apprentissage automatique a conduit à des avancées spectaculaires ces dernières années : - En vision par ordinateur, les algorithmes supervisés (notamment les réseaux de neurones profonds) apprennent à reconnaître des objets, des visages, etc. à partir de millions d'images étiquetées. - En traitement du langage naturel, des modèles entraînés sur d'énormes corpus textuels réalisent des traductions automatiques et répondent à des questions (ex : ChatGPT est entraîné de façon supervisée + renforcement). - Les systèmes de recommandation (films, produits) apprennent des préférences des utilisateurs (non supervisé + supervisé). - En robotique, l'apprentissage par renforcement permet à des bras robotiques d'apprendre à manipuler des objets par essais successifs, ou à des IA de jeu (AlphaGo, AlphaZero) d'atteindre des niveaux surhumains en apprenant par auto-jeu.

Apprentissage symbolique vs connexionniste : Historiquement, on oppose parfois l'IA *symbolique* (basée sur des règles explicites, des symboles, cf. représentation des connaissances plus haut) et l'IA *connexionniste* (basée sur des réseaux de neurones qui apprennent des représentations implicites).

L'apprentissage automatique couvre en partie l'IA connexionniste, où le savoir est « engrammé » dans les poids synaptiques d'un réseau de neurones plutôt que dans des symboles logiques. Par exemple, un **réseau de neurones** apprend à catégoriser des images sans jamais manipuler de concepts explicites de « bords », « formes », ou « chat » : il ajuste des milliers de poids pour réaliser la tâche, et le résultat est un modèle dont le fonctionnement interne est difficile à interpréter (boîte noire statistique). À l'inverse, un système symbolique aurait des règles claires mais difficilement apprises automatiquement. Aujourd'hui, l'**apprentissage profond (deep learning)** – utilisant des réseaux neuronaux à plusieurs couches – est une technique dominante du machine learning, grâce à la disponibilité de grandes puissances de calcul et de données. Néanmoins, les deux approches (symbolique et apprentissage) peuvent se compléter (par exemple, apprendre des règles à partir de données, ou contraindre un réseau à respecter des connaissances logiques).

En résumé, l'apprentissage automatique permet à l'IA d'**adapter ses comportements** aux données réelles et de **généraliser** des solutions sans qu'un programmeur ait à tout définir manuellement. La prochaine fiche détaille un algorithme fondateur de l'apprentissage automatique connexionniste : le **perceptron**, premier neurone artificiel apprenant.

Le perceptron (réseau de neurones élémentaire)

Schéma d'un perceptron simple à n entrées. Chaque entrée x_{i} est multipliée par un poids w_{i}, les sommes sont additionnées (signe Σ). Une fonction d'activation f est ensuite appliquée à la somme pondérée pour produire la sortie o.

Le **perceptron** est l'un des plus anciens algorithmes d'apprentissage supervisé, représentant le **neurone artificiel élémentaire**. Inventé en 1957 par Frank Rosenblatt, il modélise de façon simplifiée le comportement d'un neurone biologique. Un perceptron prend plusieurs **valeurs d'entrée** $x_1, x_2, …, x_n$ (par exemple, des pixels d'une image ou des mesures numériques) et calcule une **somme pondérée** de ces entrées : $s = w_1 x_1 + w_2 x_2 + … + w_n x_n + b$. Les w_i sont des **poids synaptiques** associés à chaque entrée et b est un biais (une constante à ajouter, équivalente à un poids connecté à une entrée fixe de valeur 1). Ensuite, le perceptron applique une **fonction d'activation** f à cette somme s pour produire une **sortie** $o = f(s)$. Dans sa forme la plus simple, la fonction d'activation est une **fonction de seuil** (aussi appelée *fonction Heaviside*): par exemple $f(s) = 1$ si $s \ge 0$ et $f(s) = 0$ sinon. Ainsi, le perceptron produit en sortie soit 1 soit 0 (ou -1/1 selon les conventions) – c'est un **classifieur binaire** qui décide d'une classe en fonction que la combinaison linéaire des entrées dépasse le seuil ou non.

Capacité de représentation : Géométriquement, le perceptron **sépare l'espace des entrées en deux régions** par un hyperplan (défini par l'équation $w_1 x_1 + … + w_n x_n + b = 0$). D'un côté de cet hyperplan, le neurone sort 1, de l'autre il sort 0. Cela signifie qu'un perceptron peut apprendre à reconnaître (ou à filtrer) des **patrons linéairement séparables**. *Exemples :* avec deux entrées, il peut apprendre la fonction booléenne ET (AND) ou OU (OR) qui sont linéairement séparables dans $ \mathbb{R}^2$. En revanche, certaines fonctions ne sont pas séparables linéairement, comme le **OU exclusif (XOR)** : il n'existe pas de droite qui sépare parfaitement les points correspondants à vrai/faux du XOR. Cette **limitation** du perceptron simple (incapable de résoudre XOR) a été démontrée par Minsky & Papert en 1969, ce qui a temporairement freiné la recherche sur les réseaux de neurones à l'époque. La solution pour modéliser des fonctions non-linéaires sera d'assembler plusieurs neurones sur plusieurs couches (voir **perceptrons multi-couches**, base du deep learning).

Apprentissage (règle du perceptron) : Le perceptron apprend de façon supervisée, c'est-à-dire qu'on lui fournit des exemples d'entraînement où l'on connaît la sortie désirée. L'**algorithme d'apprentissage du perceptron** ajuste les poids w_i et le biais b de manière itérative pour améliorer les résultats. Le principe de base est : 1. Initialiser les poids (souvent aléatoirement petits). 2. Pour chaque exemple d'entraînement $(\mathbf{x}, y_{\text{vrai}})$ (où y_{vrai} est la classe attendue 0 ou 1), calculer la sortie prédite $o = f(w \cdot x + b)$. 3. Mettre à jour les poids en fonction de l'erreur commise. La règle de mise à jour du perceptron (pour un perceptron à sortie 0/1) est : pour chaque poids w_i, $ $w_i \leftarrow w_i + \alpha \times (y_{\text{vrai}} - o) \times x_i,$$ et $$b \leftarrow b + \alpha \times (y_{\text{vrai}} - o),$$ où α est le **taux d'apprentissage** (un petit coefficient qui détermine l'amplitude des ajustements).

En mots, si le perceptron a fait une erreur sur cet exemple (sa sortie o est différente de la sortie désirée y_{vrai}), on **corrige les poids** en ajoutant une fraction de l'entrée (positive si on voulait une sortie 1 mais on a eu 0, négative si l'inverse). Si l'exemple est bien classé ($o = y_{\text{vrai}}$), on ne change rien. 4. Répéter sur de nombreux exemples, éventuellement plusieurs passes (époques) sur le jeu de données, jusqu'à ce que l'erreur globale devienne faible voire nulle.

Cette règle d'apprentissage est un cas particulier de la **règle de Hebb modifiée** par l'erreur : "ajuster les connexions proportionnellement à la corrélation entre entrée et erreur". On peut montrer mathématiquement le **théorème de convergence du perceptron** : si les données sont linéairement séparables, il existe un nombre fini d'itérations après lequel le perceptron aura trouvé un jeu de poids qui classe parfaitement tous les exemples. En revanche, si le problème n'est pas linéairement séparable, l'algorithme ne converge pas (il oscille).

Exemple simple : Supposons un perceptron à deux entrées qui doit apprendre la fonction ET logique. On a des exemples : $(x_1=0,x_2=0) \to y=0$, $(0,1)\to0$, $(1,0)\to0$, $(1,1)\to1$. En entraînant le perceptron, il peut converger vers des poids tels que $w_1 = 0.6, w_2 = 0.6$ et $b = -0.8$ (valeurs possibles parmi d'autres). Ainsi la somme $s = 0.6 x_1 + 0.6 x_2 - 0.8$ donnera : pour $(1,1)$ $s=0.4$ (seuil dépassé donc sortie 1 correcte), et pour toute autre combinaison, $s \le -0.2$ (sortie 0). On a donc appris la table de vérité de ET. Si on essaye d'**apprendre XOR** avec un seul perceptron, aucun ensemble de poids ne permettra d'avoir les sorties correctes pour tous les exemples : l'erreur ne descendra jamais à 0.

Importance historique et actualité : Le perceptron a marqué le début de l'**IA connexionniste**. Après un creux dans les années 1970 (suite aux limites montrées par Minsky), l'idée a ressurgi avec l'invention de l'**algorithme du rétropropagation du gradient** dans les années 1980, qui permet d'entraîner des **réseaux de neurones à couches multiples** (empilement de perceptrons). Ces réseaux multi-couches ont levé la barrière du XOR et autres problèmes non linéaires, inaugurant la vague des réseaux neuronaux modernes. Aujourd'hui, les **perceptrons multicouches** (ou *réseaux neuronaux profonds*) sont au cœur de nombreuses applications d'IA (vision, NLP, jeux). Le perceptron simple demeure un **modèle pédagogique** pour introduire les concepts de classification linéaire et d'apprentissage supervisé. Il est aussi à la base d'autres algorithmes comme les **SVM linéaires** (qui cherchent une séparation linéaire optimale) – on peut voir un SVM comme un perceptron entraîné avec des critères particuliers pour maximiser la marge de séparation.

Limites du perceptron : En plus de l'incapacité à résoudre les problèmes non-linéaires (d'où la nécessité de passer à plusieurs couches), un perceptron simple ne fournit qu'une sortie binaire. Pour des tâches à multiples classes, on peut en combiner plusieurs (par exemple *n* perceptrons pour classer en *n* classes différentes, chaque perceptron apprenant à reconnaître une classe contre toutes les autres). De même, les perceptrons originaux utilisent une fonction de seuil discontinue ; les réseaux modernes préfèrent des fonctions d'activation continues différentiables (sigmoïde, ReLU, etc.) qui

facilitent l'optimisation par gradient. Néanmoins, la logique fondamentale du **neurone formel** introduite par le perceptron est restée la brique de base de l'**apprentissage connexionniste**.

En résumé, le perceptron illustre comment une machine peut **apprendre à classifier** des exemples en ajustant ses paramètres automatiquement. C'est un algorithme simple à comprendre, qui converge sur des problèmes simples linéairement séparables, et dont l'échec sur des cas plus complexes a motivé l'évolution vers des architectures plus sophistiquées. Il relie les notions vues précédemment : agent apprenant (le perceptron s'ajuste en fonction du feedback d'erreur), recherche (il recherche un vecteur de poids solution dans l'espace des paramètres) et représentation (il calcule une fonction linéaire, ce qui représente un certain *modèle* de la décision). Aujourd'hui encore, comprendre le perceptron est fondamental pour qui veut approfondir l'IA, car il préfigure de nombreux concepts utilisés dans le machine learning moderne.

---

[1] Le test de Turing - PoBot
https://pobot.org/Le-test-de-Turing.html