Apprentissage séquentiel budgétisé pour la classification extrême et la découverte de hiérarchie en apprentissage par renforcement

Aurélia Léon

Apprentissage séquentiel budgétisé pour la classification extrême et la découverte de hiérarchie en apprentissage par renforcement.

Aurélia Léon

thèse soutenue le 10 mai 2019

Sorbonne Université

Ecole doctorale Informatique, Télécommunication et Electronique (Paris)
dans l'équipe Machine Learning and Deep Learning for Information Access
Laboratoire d'Informatique de Paris 6

Apprentissage séquentiel budgétisé pour la classification extrême et la découverte de hiérarchie en apprentissage par renforcement.

Par Aurélia Léon

Thèse de doctorat en Informatique

Dirigée par Ludovic Denoyer

Présentée et soutenue publiquement le 10 mai 2019

Devant un jury composé de :

Jérémie Mary	*rapporteur*
Cecile Capponi	*rapporteuse*
Stéphane Doncieux	*président*
Yves Grandvalet	*examinateur*
Aurélie Beynier	*examinatrice*
Ludovic Denoyer	*directeur*

Résumé

Cette thèse s'intéresse à la notion de budget pour étudier des problèmes de complexité (complexité en calculs, tâche complexe pour un agent, ou complexité due à une faible quantité de données). En effet, l'objectif principal des techniques actuelles en apprentissage statistique est généralement d'obtenir les meilleures performances possibles, sans se soucier du coût de la tâche. La notion de budget permet de prendre en compte ce paramètre tout en conservant de bonnes performances.

Nous nous concentrons d'abord sur des problèmes de classification en grand nombre de classes : la complexité en calcul des algorithmes peut être réduite grâce à l'utilisation d'arbres de décision (ici appris grâce à des techniques d'apprentissage par renforcement budgétisées) ou à l'association de chaque classe à un code (binaire). Nous nous intéressons ensuite aux problèmes d'apprentissage par renforcement et à la découverte d'une hiérarchie qui décompose une tâche en plusieurs tâches plus simples, afin de faciliter l'apprentissage et la généralisation. Cette découverte se fait ici en réduisant l'effort cognitif de l'agent (considéré dans ce travail comme équivalent à la récupération et à l'utilisation d'une observation supplémentaire). Enfin, nous abordons des problèmes de compréhension et de génération d'instructions en langage naturel, où les données sont disponibles en faible quantité : nous testons dans ce but l'utilisation jointe d'un agent qui comprend et d'un agent qui génère les instructions.

Table des matières

1 Introduction

1.1 Motivations

Les techniques d'apprentissage statistique sont aujourd'hui utilisées pour un grand nombre de tâches, et obtiennent en particulier de très bons résultats en vision (classification d'images, reconnaissance d'objets, ...) et en traduction. L'objectif principal est d'obtenir les meilleures performances possibles sur la tâche définie, sans forcément se soucier de la *complexité* en temps et/ou en calcul. La complexité peut également concerner la quantité de données nécessaires pour l'apprentissage. La prise en compte de cette complexité est pourtant cruciale : un algorithme qui donnerait de bons résultats tout en prenant des jours à les fournir pourra être moins utile qu'un autre qui ne nécessiterait que quelques heures pour des résultats à peine inférieurs.

Nous utilisons dans cette thèse une notion de *budget*, ou *coût*, pour étudier ces problèmes de complexité. Le coût peut être le temps d'exécution (si on souhaite un algorithme plus rapide), le nombre de calculs (ou dans un réseau de neurones profond le nombre de couches utilisées), le nombre de données utilisées. Cependant, cette réduction de coût peut entraîner des performances moindres ou un apprentissage plus difficile, ce qui entraîne la nécessité d'avoir un compromis entre *performance* et *coût*.

Nous nous intéressons dans cette thèse à trois problèmes budgétisés. Nous nous concentrons d'abord sur des problèmes de classification en grand nombre de classes, où la complexité en calcul des algorithmes devient vite problématique. Nous nous intéressons ensuite aux problèmes d'apprentissage par renforcement et à la découverte d'une hiérarchie afin de faciliter l'apprentissage et la généralisation à d'autres tâches, en réduisant l'effort cognitif de l'agent. Enfin, nous abordons des problèmes de compréhension et de génération d'instructions en langage naturel, où les données sont disponibles en faible quantité.

1.2 Structure de la thèse

1.2.1 Méthodes du gradient de la politique pour la classification

Notre première contribution concerne la classification en grand nombre de classes. En effet, dans ce domaine, la classification a un coût élevé en calculs, tant en phase d'entraînement qu'en phase de test : si l'entrée est de dimension N et le nombre de classes C, la complexité est en $O(N \times C)$ pour des algorithmes "directs" comme un SVM ou perceptron. Il est possible de réduire la complexité liée à la dimension de l'entrée (et d'améliorer les résultats) grâce à des réseaux de neurones à plusieurs couches (la complexité devient alors en $O(N \times M) + O(M \times C)$ avec $M \ll N$ et $M \ll C$) mais d'autres solutions sont nécessaires pour réduire la complexité liée au nombre de catégories.

Plusieurs méthodes ont été développées dans ce but. En particulier, nous nous intéressons aux méthodes hiérarchiques, ou arbres de décision, où plusieurs décisions successives sont prises. Le but est de simplifier le problème au fur et à mesure pour que les décisions soient à chaque fois (et en particulier au stade final, dans les feuilles de l'arbre), plus simples. Une autre idée est d'associer un code à chacune des classes. A partir d'une entrée, le but est ensuite de prédire directement un code (de taille inférieure au nombre de classes afin de diminuer la complexité) qui donnera la classe de l'entrée.

Nous proposons les modèles Reinforced Decision Trees (RDT) et Reinforced Error-Correcting Output Codes (RECOC) dans ces catégories de modèles. Dans les deux cas, nous utilisons des techniques d'apprentissage par renforcement afin d'apprendre en même temps la structure (comment sont organisées les classes dans les feuilles de l'arbre pour RDT, et les associations entre les codes et la classes pour RECOC) et les classifieurs (c'est-à-dire comment les données parcourent les branches, respectivement quels sont les codes associés aux données). Les travaux sur l'utilisation de techniques d'apprentissage par renforcement pour les arbres de décision ont donné lieu à un poster dans un workshop et à une publication dans une conférence européenne :

— Aurélia Léon and Ludovic Denoyer, "Reinforced Decision Trees", in *European Workshop on Reinforcement Learning (EWRL) 2015*

— Aurélia Léon and Ludovic Denoyer, "Policy-gradient methods for Decision Trees", in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN) 2016*

Les travaux sur les codes binaires appris grâce à des techniques de renforcement ont été repris par Thomas Gerald, et nos travaux ont donné lieu à une publication dans une conférence internationnale :

— Thomas Gerald, Aurélia Léon, Nicolas Baskiotis and Ludovic Denoyer, "Binary Stochastic Representations for Large Multi-class Classification", in *International Conference on Neural Information Processing (ICONIP) 2017*

1.2.2 Découverte d'une hiérarchie en apprentissage par renforcement

Notre deuxième contribution s'intéresse à l'apprentissage par renforcement hiérarchique et à la découverte d'options. En apprentissage par renforcement, les tâches complexes sont en effet difficilement résolubles sans d'abord apprendre à résoudre d'autres tâches intermédiaires, plus simples. Il est possible de définir manuellement ces tâches à résoudre avant les tâches plus complexes (par exemple, apprendre un robot à casser un oeuf avant d'apprendre directement à faire un gâteau) ; mais nous souhaiterions que l'agent découvre par lui-même une *hiérarchie* dans les tâches à effectuer, et divise par lui-même une tâche complexe en plusieurs tâches plus simples (en apprentissage par renforcement, ces sous-tâches peuvent être appelées *options*).

Nous nous inspirons pour cela de travaux en neurosciences. En effet, des études récentes suggèrent qu'une hiérarchie peut émerger à partir des *habitudes*. Un animal est d'abord en interaction constante avec l'environnement, observe autour de lui, et choisit ainsi quelles actions effectuer afin d'atteindre son but. Après avoir atteint ce but plusieurs fois avec des trajectoires proches, il n'a plus besoin d'utiliser autant les informations provenant de l'environnement et ne "réfléchit plus" à ce qu'il doit faire : il s'agit d'un suivi d'habitudes qui lui demande un effort cognitif moindre. Il est suggéré que le temps de réflexion inférieur (dans le cas des actions habituelles) permet de résoudre la tâche plus rapidement et donc d'optimiser la récompense.

Nous nous basons sur cette idée afin de proposer l'architecture Budgeted Hierarchical Neural Network (BHNN). Dans cette architecture, le coût cognitif est assimilé à la récupération et à l'utilisation des informations de l'environnement. Nous cherchons à réduire ce coût afin d'imiter le passage progressif de l'animal à une utilisation d'habitudes : dans ce but, nous utilisons un apprentissage budgétisé afin que l'agent observe de moins en moins souvent son environnement.

Ce travail a donné lieu à un poster dans un workshop puis à une publication dans une conférence internationale :

— Aurélia Léon and Ludovic Denoyer, "Options Discovery with Budgeted Reinforcement Learning", in *Deep Reinforcement Learning Workshop 2016*

— Aurélia Léon and Ludovic Denoyer, "Budgeted Hierarchical Reinforcement Learning", in *International Joint Conference on Neural Networks (IJCNN) 2018*

1.2.3 Interaction en langage naturel

Enfin, nous nous intéressons aux problématiques d'interaction en langage naturel, et plus particulièrement à la compréhension et à la génération d'instructions. L'un des problèmes afin d'apprendre à communiquer en langage naturel est de recueillir assez de données afin d'apprendre correctement. Nous cherchons des méthodes afin d'améliorer l'apprentissage tout en n'ayant pas accès à de nombreuses données. Pour ce faire, nous apprenons d'abord en apprentissage supervisé, à partir des données disponibles, la politique d'un agent acteur (qui doit interagir dans l'environnement selon des instructions) et d'un agent locuteur (qui doit générer une instruction afin qu'il soit possible pour un humain d'effectuer la tâche souhaitée). Nous testons ensuite une interaction entre l'acteur et le locuteur afin d'améliorer les résultats. Nous étendons également les modèles à des cas hiérarchiques, où l'acteur doit suivre plusieurs instructions à la suite.

Ces travaux sont encore en cours et n'ont pas donné lieu à des publications.

2 Apprentissage par renforcement

L'apprentissage par renforcement est un domaine de l'apprentissage automatique où un *agent* apprend quelles actions effectuer dans un *environnement* afin de maximiser une *récompense*. Ce chapitre en introduit les concepts clefs, puis présente plus formellement ces idées. Nous exposerons ensuite sommairement les différentes catégories d'algorithmes utilisés en apprentissage par renforcement, puis nous détaillerons plus précisément ceux utilisés dans les chapitres suivants. Nous ferons également une présentation des travaux existants en apprentissage par renforcement hiérarchique en lien avec le chapitre 4, et nous exposerons des travaux qui utilisent des techniques d'apprentissage par renforcement pour de la classification en lien avec le chapitre 3.

2.1 Rappels généraux

L'apprentissage par renforcement consiste à apprendre quoi faire (quelles actions effectuer à partir d'une situation) afin de maximiser une récompense fournie par l'environnement. Il n'est jamais dit explicitement à l'agent (qui apprend) quelles actions il doit effectuer : il doit tester des séquences d'actions afin de découvrir lesquelles amènent à une récompense plus importante. Des problèmes d'apprentissage par renforcement peuvent être : un joueur qui apprend à joueur aux échecs ; un robot qui apprend à marcher ; un robot qui apprend à se diriger dans une maison ; un groupe d'ascenceurs qui cherche à satisfaire au mieux les utilisateurs...

L'environnement utilisé en apprentissage par renforcement est habituellement modélisé comme un processus de décision markovien (MDP) que nous allons décrire plus loin. L'idée de base de cette modélisation est que l'agent apprenant doit pouvoir interagir avec l'environnement afin d'atteindre son but, et a besoin de récupérer une certaine quantité d'informations de l'environnement afin de choisir ses actions. Nous avons donc un *agent* qui évolue au sein d'un *environnement*, et qui peut recevoir une *observation* de l'état actuel de l'environnement afin de choisir quelle action effectuer pour atteindre

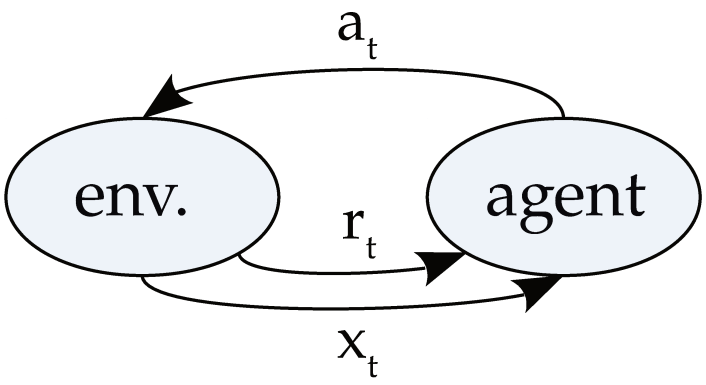


Figure 2.1: Interactions entre l'agent et l'environnement dans un cadre d'apprentissage par renforcement au temps t. L'agent reçoit une observation x_t de l'environnement, qui lui permet de choisir une action a_t et de l'effectuer dans l'environnement. Il reçoit ensuite une récompense r_{t+1}.

son *but*. Le but est défini par la *récompense* que l'agent cherche à maximiser, que nous pouvons rapprocher dans un système biologique par l'expérience de plaisir (récompense élevée) ou de douleur (récompense faible ou négative). On peut voir en figure 2.1 une représentation des interactions entre l'agent et l'environnement. Il faut cependant noter que l'agent cherche à optimiser la somme de toutes les récompenses (et donc également les récompenses futures) et non uniquement la récompense immédiate.

L'apprentissage par renforcement est un domaine de l'apprentissage automatique bien distinct de l'apprentissage supervisé. L'apprentissage supervisé apprend à partir de données étiquetées dîtes *ensemble d'entraînement* (les étiquettes correspondent souvent à des classes). D'un point de vue renforcement, les données sont des descriptions de situations, associées à un label qui correspond à l'action que devrait prendre le système dans cette situation. En apprentissage supervisé, l'ensemble d'entraînement permet d'apprendre et de généraliser les liens entre les données et les étiquettes afin de pouvoir trouver les étiquettes d'autres données. Dans des problèmes interactifs, il est cependant souvent difficile d'obtenir un ensemble d'entraînement (i.e. des exemples de comportements attendus) où les données soient à la fois correctes et représentatives de toutes les situations où l'agent pourrait se retrouver. En renforcement, si l'agent arrive dans une situation non déjà connue, il doit être capable d'apprendre de sa propre expérience (ce qui n'est pas possible en apprentissage supervisé).

L'apprentissage non-supervisé, qui cherche généralement à déterminer des structures dans des ensembles de données non labellisées, est également différent du renforcement, même si aucun de ces deux domaines n'utilisent des exemples de comportement souhaité. L'apprentissage non-supervisé découvre des structures, qui peuvent être utiles en apprentissage par renforcement, mais ne permet pas de maximiser une récompense.

L'apprentissage par renforcement se distingue par deux problématiques principales : d'une part la nécessité *d'explorer* l'environnement afin de découvrir les actions les plus efficaces ; d'autre part *l'exploitation* des connaissances afin d'obtenir une récompense plus importante. C'est le compromis entre ces deux problématiques qui rend souvent l'apprentissage difficile. Un robot humanoïde qui doit apprendre à avancer et qui a déjà appris comment ramper risque de s'en contenter, alors qu'il sera moins efficace que celui qui aura appris à marcher sur deux jambes (mais qui aura peut-être appris plus lentement). L'une des difficulté du renforcement est que l'agent doit optimiser la somme des récompenses obtenues au cours d'un épisode et non uniquement la récompense ponctuelle, et qu'une action qui parait mauvaise à un moment peut finalement être bénéfique plus tard. Un exemple peut être un agent qui doit récupérer une clef pour ouvrir un coffre, mais la clef se trouve derrière un monstre très difficile à combattre. Si la défaite du monstre diminue la récompense (car l'agent s'est épuisé à cause du combat), il risque de ne plus jamais récupérer la clef et ne parviendra pas à ouvrir le coffre. Nous approchons ici le problème de la définition de la fonction de récompense : plusieurs définitions pour la récompense sont possibles pour décrire le but "ouvrir le coffre", et ces définitions rendront l'apprentissage (et le compromis exploration/exploitation) plus ou moins facile. Le retour peut être très riche et faciliter l'apprentissage (par exemple, la récompense augmente dès que l'agent s'approche de la clef, qu'il tue le monstre, qu'il récupère la clef puis s'approche du coffre une fois qu'il a la clef) ou rendre l'exploration beaucoup plus difficile (si la récompense n'est positive que lorsque l'agent ouvre le coffre, et qu'elle est négative s'il prend des dégâts face au monstre : la fonction de récompense ne fournit alors aucune information intermédiaire sur la résolution de la tâche et peut même la rendre plus difficile). Ainsi, la fonction de récompense fait bien partie de la définition du problème.

Dans la suite, nous allons voir plus en détail le formalisme utilisé en apprentissage par renforcement.

2.2 Présentation formelle

Nous allons maintenant formaliser les concepts utilisés en apprentissage par renforcement.

Les *processus de décision markovien* (MDP) permettent de formaliser les problèmes de décisions séquentielles, où le choix d'une action peut influer, non seulement la récompense immédiate, mais également les situations futures et à travers elles les futures récompenses. L'agent et l'environnement interagissent à chaque pas de temps d'une séquence discrète $t = 0, 1, 2....$ A chaque pas de temps t, l'agent reçoit une représentation x_t de l'état $s_t \in \mathcal{S}$ de l'environnement, et choisit une action $a_t \in \mathcal{A}$ en fonction. Au pas de temps suivant, l'agent reçoit la récompense $r_{t+1} \in \mathbb{R}$ de l'environnement qui est dans un nouvel état s_{t+1}, la probabilité que l'environnement se retrouve dans l'état s_{t+1} à partir de s_t dépendant de a_t. On dit que le processus satisfait la *propriété de Markov* si l'état s_{t+1} ne dépend que de s_t et a_t et non des états précédents. Cette interaction donne ainsi une trajectoire $x_0, a_0, r_1, x_1, a_1, r_2, x_2, a_2, ...$

On modélise ainsi un MDP par un quadruplet $(\mathcal{S}, \mathcal{A}, \mathcal{P}_a, \mathcal{R}_a)$ où :

— $\mathcal{S}$ est un ensemble d'états

— $\mathcal{A}$ est un ensemble d'actions (parfois, on considère $\mathcal{A}_s$ l'ensemble des actions disponibles depuis l'état $s \in \mathcal{S}$)

— $T_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ est la fonction de transition : c'est la probabilité que l'action a prise dans l'état s mène à l'état s'

— $R_a(s, s')$ est la récompense immédiate reçue après être passé de l'état s à l'état s' dû à l'action a

Dans les problèmes d'apprentissage par renforcement, l'agent n'a cependant pas accès directement à ces informations (ce qui les distingue des problèmes de planification, où il est possible de calculer la solution à partir du MDP).

Dans un MDP de base, l'agent a une connaissance complète de l'état actuel de l'environnement : son observation x_t est donc équivalente à l'état s_t. Beaucoup de problèmes d'apprentissage par renforcement actuels considèrent cependant qu'on se place dans des processus de décision markovien *partiellement observables* (POMDP) où l'agent n'observe pas directement l'état s_t de l'environnement mais que son observation x_t est partielle. Dans cette situation, l'agent doit prendre des décisions sans être certain du véritable état

de l'environnement ; mais en intéragissant avec ce dernier et en gardant une mémoire de ses observations, l'agent peut faire une estimation du véritable état s_t. On peut se représenter par exemple un agent dans un labyrinthe, qui voit uniquement à quelques mètres autour de lui, et qui se construit une carte interne de l'environnement au fur et à mesure de son exploration.

La *politique* π définit la manière dont l'agent va agir dans l'environnement. C'est une fonction qui, à partir de ce que sait l'agent sur l'environnement à un moment donné, va renvoyer les actions à effectuer. Généralement, les politiques sont stochastiques et fournissent la probabilité d'effectuer chaque action.

La *fonction de récompense* r d'un MDP définit le but d'un problème d'apprentissage par renforcement. A chaque pas de temps, l'environnement donne à l'agent une *récompense* r_t.

L'objectif de l'agent au temps t est de maximiser la *somme des récompenses* reçues durant le reste de l'épisode (également appelé *retour*) :

$$R_t = \sum_{k=0}^{T} r_{t+k} \tag{2.1}$$

si on considère un épisode qui s'arrête au temps $t = T$. Il s'agit donc d'un objectif à long terme et non immédiat.

Dans le cas où on considère des problèmes à horizon infini, où l'agent continue d'agir sans jamais s'arrêter, l'état final serait $T = \infty$ et le retour pourrait être infini. On définit alors la *somme amortie des récompenses*, ou *retour amorti*, qu'on note également R_t par abus de notations [1] :

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.2}$$

où γ est un paramètre appelé le *facteur d'amortissement*, $0 \leq \gamma < 1$. Ainsi, plus γ est proche de 0, plus l'agent donne d'importance aux récompenses proches dans le temps. Si la récompense est bornée, R_t sera fini.

1. Dans la suite on considérera que R_t est le retour amorti, mais les résultats sont identiques dans le cas de problèmes à horizon fini si on utilise le retour non amorti

Dans la suite, nous utiliserons les notations pour des horizons finis de taille maximale T, mais les notations sont facilement adaptables aux problèmes à horizon infini. Même en horizon fini, nous utilisons un retour amorti avec $0 \leq \gamma \leq 1$, souvent utilisé avec $\gamma \simeq 0.9$, qui permet à l'agent d'optimiser le retour à long terme tout en privilégiant les récompenses immédiates.

La *fonction de valeur* V permet d'estimer à quel point il est "bon" pour l'agent d'être dans un état s_t du monde. Cette estimation est définie à partir des récompenses que l'agent devrait recevoir dans le futur, ou plus exactement, du *retour* attendu. Cette estimation dépend des actions que l'agent va effectuer, et donc de la politique (alors que le retour est simplement défini comme ce qui est effectivement renvoyé par l'environnement lors de l'épisode). Plus exactement, la fonction de valeur correspond à l'espérance de la somme amortie des récompense. Si on note s_t l'état du monde, la valeur de cet état étant donné une politique π est :

$$V_\pi(s_t) = \mathbb{E}_\pi[R_t|s_t] = \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k r_{t+k+1}|s_t] \tag{2.3}$$

où $r_{t+1}, r_{t+2}, ...r_{t+T+1}$ sont les récompenses obtenues d'après une trajectoire tirée par la politique π à partir de l'état s_t.

La *fonction de qualité* est très proche de la fonction de valeur mais prend également en compte l'action choisie afin d'évaluer la qualité de la trajectoire :

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[R_t|s_t, a_t] = \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k r_{t+k+1}|s_t, a_t] \tag{2.4}$$

Résoudre un problème d'apprentissage par renforcement consiste à trouver une politique qui permettra à l'agent d'obtenir un maximum de récompenses. On considère qu'une politique π est meilleure qu'une politique π' si dans n'importe quel état du monde, son retour attendu est plus important que celui de la politique π', c'est-à-dire si $V_\pi(s) \geq V_{\pi'}(s)$ pour tout état s. Il y a toujours au moins une politique qui est meilleure ou égale à toutes les autres : elle est appelée *politique optimale* et elle est notée π^* :

$$\pi^* = argmax_\pi \mathbb{E}_s[V_\pi(s)] \tag{2.5}$$

2.3 Algorithmes utilisés en apprentissage par renforcement

Le but d'un problème d'apprentissage par renforcement est donc de trouver une politique π^* qui permet de maximiser $\mathbb{E}_s[V_{\pi^*}(s)]$. De nombreux algorithmes existent dans ce but, qu'on peut diviser selon différents critères :

Model-based et Model-free Un algorithme *model-based* va apprendre un modèle de l'environnement, et l'utilisera pour simuler l'environnement. Il est possible d'utiliser cette simulation pour choisir les prochaines actions et/ou pour apprendre plus facilement la politique et moins interagir "en vrai". Le problème est d'apprendre un modèle fiable de l'environnement. Au contraire, un algorithme *model-free* n'apprend pas directement un modèle de l'environnement. Il est possible de voir les algorithmes model-based comme cherchant à estimer les éléments du MDP, en particulier les fonctions de transition $T_a(s, s')$ et de récompense $R_a(s, s')$, afin d'utiliser ensuite par exemple des méthodes de planification pour trouver les actions optimales. Les algorithmes model-free ne passent pas par un modèle de l'environnement : par exemple, ils approchent directement les valeurs de chaque état $V(s)$, ou apprennent directement une politique qui permet d'obtenir un meilleur retour de l'environnement. Nous allons en voir des exemples dans la suite.

On-policy et off-policy La politique qui est apprise par une méthode *on-policy* est utilisée, en même temps qu'elle est améliorée, pour prendre des décisions et évoluer dans l'environnement. Une méthode *off-policy* améliore par contre une politique différente de celle utilisée pour générer les trajectoires d'apprentissage. L'inconvénient d'une méthode on-policy est qu'une même politique doit permettre en même temps d'apprendre les actions optimales à effectuer, tout en explorant l'environnement et donc en utilisant un comportement pas forcément optimal. Les méthodes off-policy permettent d'utiliser deux politiques : la politique cible qui est apprise afin d'être optimale, et la politique exploratrice qui permet d'explorer davantage l'environnement et de générer des comportements utilisés pour l'apprentissage. Les méthodes off-policy sont souvent plus lentes à converger et possèdent une plus grande variance à cause de l'utilisation de données venant d'une politique différente, mais permettent des utilisations plus variées car il est possible d'utiliser des

trajectoires venant d'un expert ou d'un autre contrôleur, ou d'utiliser des techniques d'échantillonnage préférentiel (*importance sampling*).

Méthode d'apprentissage Une méthode *value based* va apprendre la fonction de valeur V ou la fonction de qualité Q, et en déduire directement une politique. Une méthode *policy based* va apprendre directement la politique. De manière générale, une méthode *policy based* va converger en moins d'itération, mais chaque itération sera plus complexe qu'une méthode *value based* car il est plus simple d'estimer la fonction de valeur ou de qualité à partir de la récompense fournie par l'environnement. Enfin, les méthodes *actor-critic* estiment la fonction de valeur ou de qualité, et l'utilisent simultanément afin d'apprendre la politique. Il s'agit généralement de méthodes plus puissantes et donnant de meilleurs résultats, mais plus difficiles à faire converger, car l'apprentissage de la politique ne sera pas possible si la fonction de qualité ou de valeur ne donne pas déjà des résultats convenables.

Il faut noter qu'en apprentissage par renforcement, il n'existe pas *une* méthode meilleure que les autres en toute circonstance : les méthodes obtenant les meilleures performances dépendent énormément des environnements (et des fonctions de récompenses) utilisés, et de la forme du problème.

Méthodes tabulaires et méthodes d'approximation Les premiers problèmes d'apprentissage par renforcement étaient tabulaires, avec un nombre fini d'actions et d'états du monde. Ainsi, il était possible de stocker les valeurs pour chaque couple *(action, état)*. Les problèmes plus généraux et plus récents considèrent cependant des espaces d'actions et surtout d'états plus importants, continus, qui ne peuvent plus être stockés dans des tableaux faute d'une mémoire suffisante et nécessitent donc une généralisation. La généralisation est également nécessaire car de très grands espaces d'états (et éventuellement d'actions) nécessiteraient, dans un cadre tabulaire exact, beaucoup de temps et de données afin de remplir les tableaux. Une généralisation permet de prendre des décisions dans des états qui n'ont pas été exactement vus précédemment, mais peuvent être similaires à des exemples déjà observés.

On utilise dans ce but des fonctions d'approximation, qui, à partir d'exemples (les trajectoires), cherchent à généraliser les informations afin d'approcher la ou les fonctions désirées (fonction de valeur, politique...). Une possibilité est l'utilisation de réseaux de neurones profonds. Ceux-ci permettent en effet

Algorithme	Modèle	Politique	Type
Q-learning	model-free	off-policy	tabulaire
Sarsa	model-free	on-policy	tabulaire
DQN (Deep Q Network)	model-free	off-policy	deep nn
DDPG (Deep Deterministic Policy Gradient)	model free	off-policy	deep nn
A3C (Asynchronous Advantage Actor-Critic)	model-free	off-policy	deep nn
TRPO (Trust Region Policy Optimization)	model-free	on-policy	deep nn

Table 2.1: Tableau récapitulatif d'algorithmes en apprentissage par renforcement.

d'obtenir des représentations de l'état du monde et de faire une estimation de la fonction de valeur, de la fonction de qualité, et/ou d'apprendre directement la politique optimale de l'agent.

Les premières utilisations de réseaux de neurones en apprentissage par renforcement datent des années 1990 [Wil92], mais c'est depuis les percées en apprentissage des réseaux de neurones (par exemple en vision [Kri+12] ou en reconnaissance de parole [Gra+13]), puis avec en particulier la sortie du modèle DQN [Mni+15] qui généralise un algorithme de Q-learning aux réseaux de neurones afin d'apprendre à un agent à jouer à des jeux Atari, que la plupart des nouvelles méthodes en apprentissage par renforcement utilisent des réseaux de neurones - avec succès.

On peut voir dans le tableau 2.1 un récapitulatif de principaux algorithmes utilisés en apprentissage par renforcement. Beaucoup de méthodes mixant réseaux de neurones et apprentissage par renforcement existent dorénavant, et nous allons présenter dans cette section uniquement celles que nous utiliserons dans la suite. La liste n'est pas du tout exhaustive et de nombreuses variantes existent ; cette section n'a que pour but d'introduire les méthodes utilisées dans les travaux de cette thèse et non de présenter un panel des méthodes existantes.

Dans la suite de cette thèse, tous les modèles utilisent des réseaux de neurones profonds.

2.3.1 REINFORCE

REINFORCE [Wil92] est une méthode model-free, on-policy, et policy-based. Plus précisément, elle fait partie des méthodes du gradient de la politique, qui estiment directement la politique optimale grâce à des techniques de

gradient. Le gradient utilisé pour apprendre les paramètres de la politique est approché en utilisant plusieurs trajectoires : un autre nom utilisé pour l'algorithme REINFORCE est Monte Carlo Policy Gradient.

On note π_θ la politique, avec θ les paramètres de cette politique (i.e. les paramètres des réseaux de neurones). Typiquement, π_θ est un réseau de neurones ayant en sortie une fonction softmax. Ainsi, $\pi_\theta(a|s)$ est la probabilité de choisir l'actioin a étant donné l'état du monde s.

Le théorème du gradient de la politique permet d'estimer le gradient de la performance $J(\pi) = V_\pi(s_0)$ par :

$$\nabla J(\pi) \approx \sum_s \mu(s) \sum_a Q_\pi(s,a) \nabla_\theta \pi_\theta(a|s,\theta) \tag{2.6}$$

où $\mu(s)$ est la distribution on-policy des états, c'est-à-dire la fraction de temps (normalisée afin de sommer à un) passée dans chaque état lorsqu'on utilise la politique π.

L'algorithme REINFORCE donne une estimation de cette formule. Si on suit la politique π, les états seront rencontrés dans les proportions $\mu(s)$, ainsi on peut réécrire la partie droite de l'équation précédente grâce à une espérance, en notant s_t les états rencontrés lorsqu'on suit la politique :

$$\begin{aligned}
\nabla J(\pi) &= \mathbb{E}_\pi[\sum_a Q_\pi(s_t,a) \nabla_\theta \pi(a|s_t,\theta)] && (2.7)\\
&= \mathbb{E}_\pi[\sum_a \pi(a|s_t,\theta) Q_\pi(s_t,a) \frac{\nabla_\theta \pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}] && (2.8)\\
&= \mathbb{E}_\pi[Q_\pi(s_t,a_t) \frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}] && (2.9)\\
&= \mathbb{E}_\pi[R_t \frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}] && (2.10)
\end{aligned}$$

où a_t sont les actions tirées selon la politique π.

En utilisant un échantilonage de Monte-Carlo de M trajectoires selon la politique π, ce gradient peut être estimé par :

$$\Delta J(\pi) \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t) R_t \tag{2.11}$$

Algorithm 1 REINFORCE

Initialisation des paramètres θ de la politique π
ϵ : pas de gradient
repeat
  Génération d'une trajectoire $s_0, a_0, r_1, ...s_{T-1}, a_{T-1}, r_T$ suivant π
  for $t = 0, ...T-1$ **do**
    $R_t \leftarrow$ retour au temps t, donné par l'environnement
    $\theta \leftarrow \theta + \epsilon \nabla_\theta \log \pi(a_t|s_t) R_t$
  end for
until convergence

L'algorithme 1 montre la procédure d'apprentissage.

REINFORCE avec une baseline

Le problème des approches de Monte Carlo est que l'estimation du gradient peut avoir une variance importante. Une manière de réduire cette variance est d'utiliser une *baseline* b comme introduite en [Wil92] :

$$\Delta J(\pi) \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t)(R_t - b) \tag{2.12}$$

b peut être n'importe quelle fonction tant qu'elle ne dépend pas de a. Un choix naturel est une estimation de la valeur de l'état s_t : on peut utiliser le retour moyen attendu, ou apprendre une baseline dépendant de l'état s_t : la baseline sera alors $b(s_t)$, apprise pour approcher R_t avec par exemple des réseaux de neurones.

REINFORCE récurrent

Les tâches plus complexes requièrent parfois l'utilisation d'une mémoire : l'agent doit retenir les états et/ou les actions passés afin de savoir comment agir dans l'état s_t.

Dans le cas d'une trajectoire $(s_0, a_0, ...s_t)$, l'algorithme REINFORCE récurrent, introduit en [Wie+10], permet d'estimer le gradient de la même manière :

$$\Delta J(\pi) \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_0, a_0, ...s_t)(R_t - b) \tag{2.13}$$

Dans ce cas, la baseline b peut être constante ou dépendre de l'historique de la trajectoire $(s_0, ...s_t)$, typiquement grâce à des réseaux de neurones récurrents.

2.3.2 Asynchronous advantage actor critic

Asynchronous advantage actor critic (A3C) a été introduit dans [Mni+16a]. Cet algorithme acteur-critique utilise une estimation de la fonction de valeur $V(s_t)$ afin de mettre à jour les paramètres de la politique. Il est *asynchrone* car il utilise en même temps plusieurs agents, avec leur propre jeu de paramètres. Le fait d'utiliser plusieurs agents permet d'obtenir dans la plupart des cas un entraînement plus stable (en plus de l'accélérer) car les expériences des agents sont indépendantes des autres et sont donc plus variées.

L'algorithme 2 décrit cette procédure. Les paramètres de la politique sont notés θ (θ' pour les paramètres de chaque thread) et ceux de la fonction de valeur sont notés θ_v (θ'_v pour ceux spécifiques à un thread).

On peut noter A la fonction avantage :

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.14}$$

Intuitivement, elle donne le gain (ou le déficit) sur le retour qu'aura l'agent dans l'état s_t s'il effectue l'action a_t par rapport à la moyenne des retours dans cet état. Une mise à jour des paramètres de la politique et de la fonction de valeur est faite à chaque t_{max} itération, ou quand l'état terminal est atteint. Au temps t, la fonction avantage est approchée par :

$$A(s_t, a_t) \approx \sum_{i=0}^{k-1} \gamma^i r(s_{t+i}, a_{t+i}) + \gamma^k V(s_{t+k}) - V(s_t) \tag{2.15}$$

où $k \leq n$ peut varier selon l'état mais est généralement fixé.

La mise à jour de la politique pour l'algorithme A3C au pas de temps t est alors :

$$\nabla_\theta \log \pi(a_t|s_t) A(s_t, a_t) \tag{2.16}$$

Dans le cas récurrent, on obtient :

$$\nabla_\pi \log \pi(a_t|s_0, a_0, ..., s_t) A(s_t, a_t) \tag{2.17}$$

Algorithm 2 A3C : pseudo-code pour un thread d'acteur

// Les paramètres θ et θ_v sont partagés entre les threads, ainsi que le compteur T=0
// Les paramètres θ' et θ'_v sont spécifiques au thread
Initialisation du compteur d'itérations du thread $t \leftarrow 1$
repeat
  Réinitialisation des gradients : $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
  Synchronisation des paramètres $\theta' = \theta$, $\theta'_v = \theta_v$
  $t_{start} = t$
  Récupération de l'état s_t
  repeat
    Execution de a_t selon la politique $\pi(a_t|s_t, \theta')$
    Reception de la récompense r_t et du nouvel état s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until état terminal s_t ou $t - t_{start} == t_{max}$
  Si s_t est terminal : $R = 0$
  Sinon : $R = V(s_t, \theta'_v)$
  for $i = t - 1, ...t_{start}$ **do**
    $R \leftarrow r_t + \gamma R$
    Accumulation du gradient pour θ' : $d\theta \leftarrow d\theta + \nabla'_\theta \log \pi(a_i|s_i, \theta')(R - V(s_i, \theta'_v))$
    Accumulation du gradient pour θ'_v : $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i, \theta'_v))^2/\partial\theta'_v$
  end for
  Effectuer la mise-à-jour asynchrone de θ en utilisant $d\theta$ et de θ_v en utilisant $d\theta_v$
until $T > T_{max}$

On utilise des acteurs parallèles afin d'accumuler les mises à jour. Le critique V est appris par une simple descente de gradient habituelle pour approcher le retour amorti.

2.4 Apprentissage par renforcement hiérarchique

Nous allons maintenant aborder les problèmes d'apprentissage hiérarchique, que nous traitons dans le chapitre 4.

L'idée d'apprentissage par renforcement hiérarchique est tirée du comportement humain. En effet, des travaux en sciences cognitives ont montré que les humains et les animaux décomposent un comportement complexe en un processus hiérarchique [Bot+09] : le comportement global est vu comme un ensemble de sous-tâches plus simples, et chaque tâche peut à son tour

être décomposée en sous-tâches, etc. Par exemple, faire un gâteau peut être décomposé en plusieurs tâches : réunir les ingrédients, les mélanger, mettre le plat au four...

En informatique, les tâches complexes, nécessitants plusieurs étapes comme par exemple le jeu Montezuma's Revenge des jeux Atari, sont difficilement résolubles en apprentissage par renforcement, faute en partie à une fonction de récompense pas assez informative, comme nous l'avons décrit en fin de partie 2.1. C'est pourquoi des travaux se sont inspirés des processus hiérarchiques des comportements humains ou animaux, pour mener à l'apprentissage par renforcement hiérarchique [DH93 ; Die98 ; PR98]. Ces méthodes permettent un transfert de connaissances entre différentes tâches, et factorisent une tâche complexe en sous-tâches plus simples. Si l'on reprend l'exemple de la cuisine, pour faire un gâteau, *monter les blancs en neige* pourrait être une sous-tâche : il suffit de l'apprendre une fois pour utiliser ces connaissances dans la confection de plusieurs gâteaux différents, plutôt que de devoir ré-apprendre à la résoudre à chaque nouveau gâteau.

Nous allons présenter certains travaux existants dans ce domaine. En particulier, nous nous intéresserons au framework des *options*, qui permet de modéliser les sous-tâches et qui est utilisé par beaucoup d'autres travaux.

2.4.1 Travaux existants

Les premiers algorithmes en apprentissage hiérarchique considèrent que les sous-tâches sont connues *a priori*, du moins en partie. Ainsi, la hiérarchie est pré-établie dans la méthode MAXQ [Die98], et il suffit ensuite d'apprendre à résoudre chaque sous-tâche pour résoudre la tâche globale.

Beaucoup de travaux en apprentissage par renforcement hiérarchique utilisent maintenant le concept d'*options*. Ce concept a été introduit par Sutton et al. dans l'article [Sut+99]. Une option ω, qu'on peut rapprocher d'une sous-politique, consiste en un triplet $(\mathcal{I}_\omega, \pi_\omega, \beta_\omega)$ avec :

— $\mathcal{I}_\omega \subseteq \mathcal{S}$ son ensemble d'états initiaux (la liste des états où peut débuter cette option)

— π_ω sa politique (qui permet de choisir des actions primitives mais peut aussi choisir d'autres options)

— $\beta_\omega : \mathcal{S} \to [0, 1]$ sa fonction terminale (qui donne la probabilité de terminer l'option dans un état donné).

Cette modélisation amène deux problématiques :

— comment choisir l'option la plus adaptée à un moment donné pour résoudre la tâche globale

— ayant déjà choisi une option, quelles sont les actions optimales à effectuer dans l'environnement pour résoudre cette sous-tâche

Nous avons donc un contrôleur haut niveau qui permet de choisir l'option, et un contrôleur bas niveau qui permet ensuite de choisir l'action. Certains travaux se focalisent sur l'apprentissage d'un seul de ces contrôleurs, alors que d'autres tentent d'apprendre les deux en simultané.

Ce concept d'options est utilisé dans plusieurs articles récents afin de définir de nouvelles architectures hiérarchiques. Par exemple, les auteurs de [Kul+16] proposent une extension du framework Deep Q-Learning (DQN) afin d'y intégrer des fonctions de valeur hiérarchiques, en utilisent une motivation intrinsèque afin d'apprendre les politiques des options (voir figure 2.2). Mais les options doivent être manuellement choisies *a priori* et ne sont pas découvertes durant l'apprentissage : on ne fait que apprendre les politiques de chaque option et non quelles sont les tâches résolues par les options. Quelques modèles permettent de découvrir les options (à la fois les politiques de chacune mais aussi la politique qui choisit les options) sans supervision. [Dan+16] traite l'index de l'option utilisée comme une variable latente afin d'utiliser un algorithme Expectation Maximization. Dans [BP15b], l'architecture *option-critic* s'inspire des modèle d'acteurs-critiques : un critique (appris) permet d'apprendre les paramètres des options (paramètres des politiques mais aussi des fonctions terminales) en utilisant le théorème du gradient de la politique (voir 2.3). Dans les deux cas, il faut cependant fixer un nombre discret d'options avant l'apprentissage, ce qui peut poser problème quand on ne sait pas à l'avance combien d'options seront nécessaires.

Enfin, certains articles proposent des politiques hiérarchiques basées sur **différents niveaux d'observations**. Celles-ci nous intéressent particulièrement car c'est le cas de notre contribution du chapitre 4. Ces modèles peuvent être classés en deux catégories : ceux qui n'utilisent parfois aucune observation de l'environnement avant de choisir une nouvelle action, et ceux plus généraux qui utilisent des observations différentes. Nous allons en présenter des exemples.

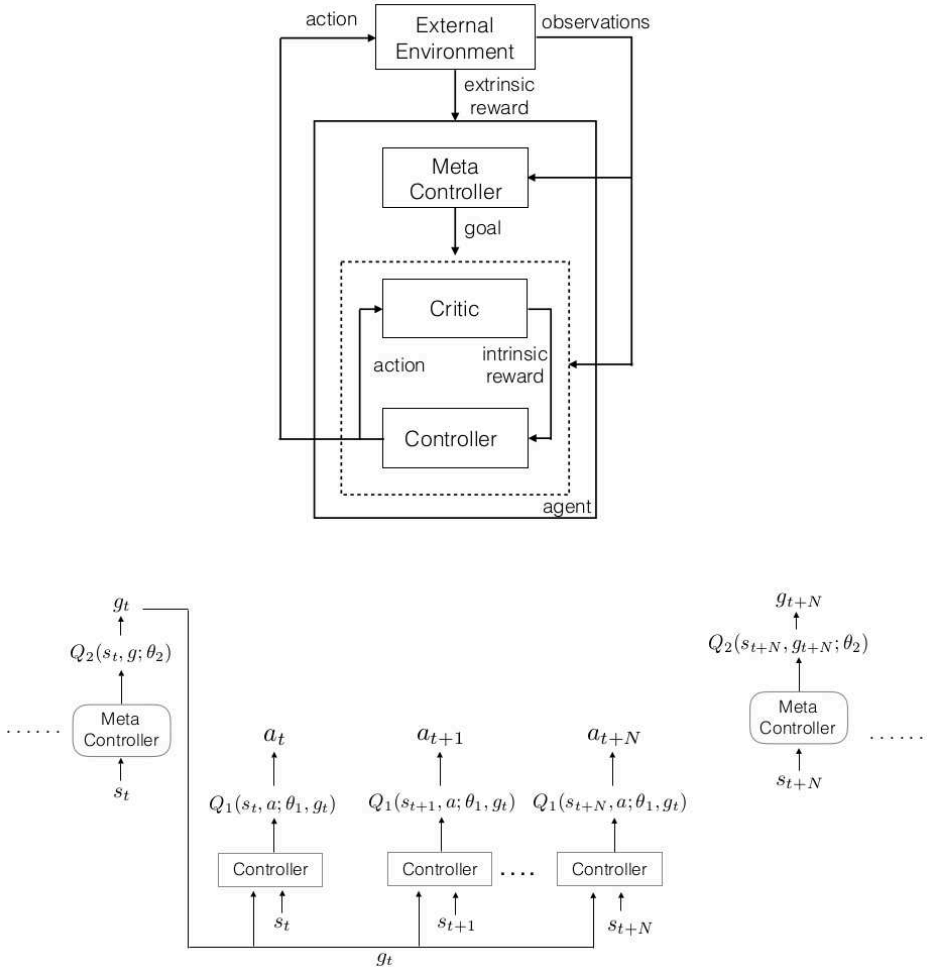


Figure 2.2: *Hierarchical-DQN* ; figure tirée de [Kul+16]. Des réseaux neuronaux profonds sont utilisés pour modéliser deux fonctions de valeur : Q_2 correspond à la politique du contrôleur haut-niveau (meta controler) qui permet de choisir un but (intermédiaire) g_t, et Q_1 correspond à la politique du contrôleur bas-niveau qui permet de choisir une action a_t à partir de l'observation s_t et du but g_t, afin d'atteindre le but g_t.

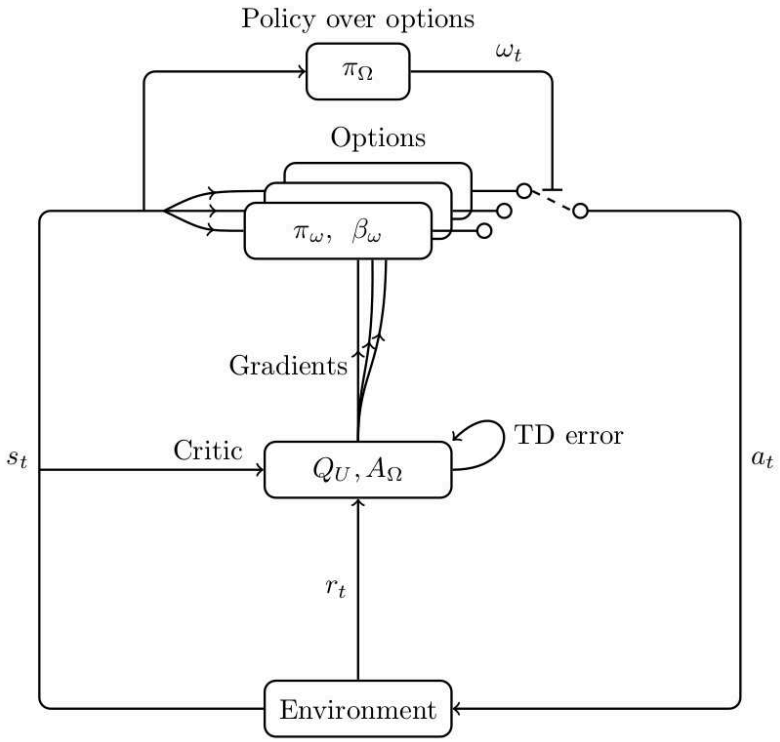


Figure 2.3: Achitecture *Option-Critic* ; figure tirée de [BP15b]. La politique π_Ω permet de choisir une nouvelle option ω qui choisit les actions à effectuer dans l'environnement. Pour l'apprentissage, Q_U renvoie la valeur de l'exécution d'une action selon un couple *état, option*.

On dit qu'un modèle qui prend des décisions sans récupérer des informations de l'environnement est *open-loop* : ainsi, les modèles dont les politiques choisissent parfois les actions en "aveugle" sont considérées en partie open-loop. Par exemple, dans [Han+96], l'agent doit apprendre à ne pas utiliser l'observation à chaque pas de temps car l'acquisition de cette observation a un coût. On dit également que ces modèles apprennent des **macro-actions** [Hau+98 ; Mni+16c] : la politique associe à un état donné une séquence d'actions élémentaires. Le framework FiGAR [Sha+17] en est un cas simplifié qui permet à l'agent de répéter la même action plusieurs fois de suite.

Les modèles plus généraux utilisent des observations différentes, soit selon les options, soit pour le contrôleur haut et bas niveau. Par exemple, le modèle Abstract Hidden Markov Model [Bui+02] utilise des options discrètes, chacune définie sur une certaine région de l'espace ; mais les régions de l'espace d'états doivent être définies manuellement, et cela revient donc à définir à la main les options. Dans [Hee+16], un contrôleur haut-niveau a accès à toutes les informations, alors qu'un contrôleur bas-niveau a uniquement accès aux informations proprioceptives. [Flo+16] factorise également l'espace d'état : les tâches plus faciles utilisent un état toujours disponible et sont pré-apprises pour fournir un ensemble de compétences, puis une tache plus difficile peut ensuite utiliser une observation plus complète et utiliser les compétences déjà apprises.

Ainsi, la découverte automatique d'options est encore loin d'être résolue : dans la plupart des modèles, l'être humain doit donner des *a-priori* sur les options possibles. En effet, la majorité des modèles existants nécessitent que la structure hiérarchique soit contrainte avant l'apprentissage, que ce soit en définissant les sous-tâches, en divisant l'espace des états... Apprendre automatiquement la décomposition d'une tâche, sans supervision, reste une problématique ouverte en apprentissage par renforcement. La difficulté de cet apprentissage est exacerbé par le fait qu'on ne comprend pas encore totalement *comment* ces processus hiérarchiques apparaissent dans les comportements d'êtres vivants. L'apprentissage simultané de la décomposition en sous-tâches et de la résolution de ces sous-tâches est également complexe car il est difficile de savoir si telle sous-tâche est appropriée lorsqu'on ne sait pas encore correctement la résoudre : ainsi, nous risquons d'avoir des problèmes de stabilité lors de l'apprentissage.

2.5 Classification supervisée et renforcement

Dans le chapitre 3, nous utilisons des techniques d'apprentissage par renforcement pour des problèmes de classification. L'avantage de l'apprentissage par renforcement est que la tâche peut être *séquentielle* et que des décisions intermédiaires *discrètes* peuvent être prises. Pour voir la tâche de classification comme un problème de renforcement, il suffit de considérer que les observations de l'agent sont les informations disponibles à propos de la donnée à classer, que la prise de décision finale est la classe, et que la récompense, qui définit la tâche, dépend de l'exactitude de la classification (par exemple, une récompense de $+1$ si la donnée est correctement classée ; 0 sinon).

L'un des premiers travaux qui modélise une tâche de prédiction séquentielle par un MDP est [JC07], qui cherche à résoudre une tâche de classification où chaque sélection de caractéristique a un coût. Le travail plus récent de [GK11] considère une tâche où un ensemble de classifieurs est utilisé : le but est de n'en utiliser qu'une partie pour chaque donnée. Un coût de calcul est associé à chaque classifieur, et le temps de calcul total doit être minimisé.

D'autres travaux plus récents [He+12 ; Rüc+13 ; DA14] formulent la tâche de classification comme un MDP avec des actions pour sélectionner les caractéristiques utilisées et parfois pour la classification.

3 Méthodes de gradient politique pour la classification

Nous présentons ici notre première contribution. Il s'agit d'une nouvelle méthode hiérarchique utilisée pour des problèmes de classification, où l'ensemble du modèle (d'une part la hiérarchie, d'autre part les classifieurs) sont appris en une seule étape, grâce à des algorithmes d'apprentissage par renforcement utilisés en apprentissage statistique.

3.1 Introduction

Les problèmes de classification ont déjà été beaucoup étudiés en apprentissage statistique [Pan+02], mais la complexité en calcul des modèles permettant de les résoudre est fortement reliée au nombre de classes, avec généralement une complexité linéaire par rapport aux nombres de catégories possibles. Si nous considérons la classification d'une entrée de dimension N avec C classes possibles, la complexité en calculs d'un perceptron est en $O(N \times C)$; il est possible d'améliorer cette complexité en $O(N \times M) + O(M \times C)$ avec un réseau de neurones profonds ($M \ll N$ et $M \ll C$) mais face à des problèmes avec un très grand nombre de classes, comme la classification de textes, la reconnaissance d'objets... la complexité par rapport au nombre de classes reste problématique. Ainsi, est apparue la nécessité de développer des modèles spécifiques afin de réduire les coûts de classification, principalement lors de l'inférence.

Diverses méthodes ont été développées ces dernières années ; les sorties y sont généralement organisées grâce à une structure afin de réduire la complexité. On distingue deux principales catégories de modèles qui répondent à cette problématique :

— **les méthodes hiérarchiques** ou arbres de décision, où les classes sont associées aux feuilles d'un arbre. Chaque entrée permet de déterminer une trajectoire jusqu'à une feuille, qui renvoie la classe associée

([Ben+10 ; Liu+13a ; Wes+13]). La prédiction est donc basée sur une stratégie *diviser pour régner*. Ces modèles sont habituellement appris grâce à des algorithmes gloutons basés sur diverses heuristiques qui permettent de construire la structure de l'arbre à partir d'un ensemble d'entraînement.

— **les codes correcteurs d'erreur** (Error-Correcting Output Codes, ECOC) où chaque classe est associée à un code binaire ; la classification consiste ensuite à prédire le code et non directement la classe ([DB95 ; Sch97 ; Cis+11 ; ZX13 ; LB05])

Ces deux groupes de méthodes permettent habituellement d'obtenir une **complexité logarithmique** durant la prédiction plutôt qu'une complexité linéaire par rapport au nombre de classes. Cependant, la plupart nécessitent de construire la structure utilisée (qu'il s'agisse des codes associés aux classes ou de la structure de l'arbre) lors d'une étape préliminaire, avant de construire les classifieurs.

Nos deux contributions permettent de construire en une seule étape la hiérarchie (ou le code) et les classificateurs. Nous allons les présenter brièvement.

Nous proposons d'abord une nouvelle famille d'arbres de décision appelée **Reinforced Decision Trees (RDTs)** qui garde les avantages des arbres de décision (une inférence rapide et de bonnes performances) et qui apprend en une même étape la structure globale et les classifieurs associés. L'idée principale est de considérer le problème de classification comme un *problème séquentiel*, où une politique (apprise) guide une entrée x (la donnée à classer) à travers les branches d'un arbre depuis la racine jusqu'à une feuille. L'algorithme d'apprentissage, inspiré des méthodes de gradient de la politique [BB99] d'apprentissage par renforcement, nous permet d'apprendre à la fois la manière dont les entrées sont guidées à travers l'arbre mais également l'association des classes dans les feuilles de l'arbre. La différence avec les techniques habituelles de classification par arbres de décision est que cette méthode cherche à optimiser directement un objectif global (dérivable) plutôt que d'utiliser un algorithme glouton, où des choix d'optima locaux sont faits successivement. Par rapport aux algorithmes classiques de classification, la structure hiérarchique de ce modèle permet de réduire la complexité en calculs durant la prédiction.

Le second modèle, **Reinforced Error-Correcting Output Codes (RECOC)**, utilise le même procédé pour apprendre des codes binaires, associés aux

catégories, en même temps que les classificateurs qui permettent d'associer un code à l'entrée x à classer. L'avantage du modèle par rapport à RDT est une utilisation plus facile de mini-batchs, ce qui rend une parallélisation plus facile et permet un apprentissage et une inférence beaucoup plus rapide pour un grand nombre de données.

Nous présenterons d'abord le modèle Reinforced Decision Trees, puis Reinforced Error-Correcting Output Codes. Nous exposerons les résultats des expériences effectuées avec ces deux modèles, puis nous présenterons les travaux déjà existants, proches de nos modèles.

3.2 Algorithme : Reinforced Decision Tree (RDT)

Nous présentons ici le modèle *Reinforced Decision Tree*, un modèle hiérarchique pour la classification, appris avec des techniques d'apprentissage par renforcement qui permettent un apprentissage simultané de la hiérarchie et des classifieurs.

Nous présenterons d'abord l'architecture, puis les méthodes d'inférence et d'apprentissage des paramètres.

3.2.1 Notations et architecture

On considère un problème de classification multi-classe où chaque entrée $x \in \mathbb{R}^n$ est associée à l'une des K classes possibles. Notons $y \in \mathbb{R}^K$ le label de x, avec $y_i = 1$ si x appartient à la classe i et $y_i = 0$ sinon. On notera $\mathcal{D} = \{(x^1, y^1), ..., (x^N, y^N)\}$ l'ensemble des données d'entraînement.

L'architecture du modèle *Reinforced Decision Tree (RDT)* est très proche des arbres de décision classiques, composés d'une racine, de branches, et de noeuds. Notons un tel arbre $\mathcal{T}_{\theta,\alpha}$, dont les paramètres θ et α seront définis plus loin. Cet arbre est décrit par :

— T noeuds : $noeuds(\mathcal{T}_{\theta,\alpha}) = \{n_1, ...n_T\}$

— une racine n_1

— des branches définies par $parent(n_i) = n_j$: le noeud n_j est le parent de n_i. Chaque noeud a un unique parent, excepté n_1 qui n'en a aucun (étant la racine de l'arbre)

— des feuilles n_i : si n_i est une feuille de l'arbre, on notera $feuille(n_i) = true$ (sinon $false$)

On peut noter qu'on ne pose aucune contrainte sur le nombre de feuilles ou la topologie de l'arbre. Les noeuds de l'arbre sont définis par deux types de paramètres selon qu'ils sont des feuilles ou pas :

— chaque noeud n_i tel que $feuille(n_i) = false$ est associé à un ensemble de paramètres θ_i

— chaque noeud n_i tel quel $feuille(n_i) = true$ est associé à un paramètre $\alpha_i \in \mathbb{R}^K$

Les paramètres $\theta = \{\theta_i\}$ sont les paramètres de la politique qui guide une entrée x depuis la racine de l'arbre jusqu'à une feuille (et permet donc de choisir quelle branche emprunter). Les paramètres α_i correspondent à la prédiction obtenue lorsque l'entrée x atteint la feuille n_i. Ainsi, l'architecture est très proche des arbres de décision classique. La différence majeure est que la prédiction associée à chaque feuille n'est pas directement une classe (discrète) mais un ensemble de paramètres qui sera appris durant l'apprentissage et qui donnera une distribution sur les classes possibles. Dans un problème de classification, ces α_i seront des vecteurs de taille $\mathbb{R}^K$, K étant le nombre de classes, et seront donc associés à une classe donnée. Ces paramètres α_i seront appris durant l'apprentissage et permettront au modèle de choisir comment les différentes catégories seront réparties dans les feuilles. Ainsi, le modèle pourra apprendre simultanément par quelles branches doit passer une donnée (grâce aux paramètres θ_i) et comment sont organisées les catégories dans l'arbre (grâce aux paramètres α_i).

Un exemple d'architecture RDT est donné en figure 3.1.

3.2.2 Inférence

Notons $H = (n_{(1)}, ..., n_{(t)})$ une trajectoire où $n_{(i)} \in nodes(\mathcal{T}_{\theta,\alpha})$ et (i) est l'index du i-eme noeud de la trajectoire. H est donc une séquence de noeuds où $n_{(1)} = n_1$ et $\forall i > 1, n_{(i-1)} = parent(n_{(i)})$ et $feuille(n_{(t)}) = true$.

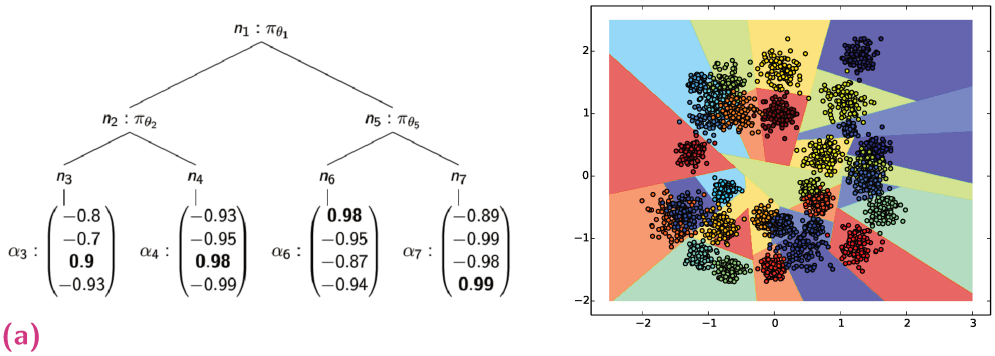


Figure 3.1: (a) Un exemple de RDT. Les fonctions π_θ et les vecteurs α (les valeurs dans les feuilles des arbres) ont été appris en même temps. Les valeurs en gras correspondent à la catégorie prédite pour la feuille associée. (b) Les frontières de décisions apprises par le modèle RDT sur un jeu de données en 2D avec 32 catégories différentes.

Chaque noeud interne de l'arbre n_i est associé à une fonction (ou politique) π_{θ_i} : pour une entrée donnée x, $\pi_{\theta_i}(x, n) = P(n|n_i, x)$ est la probabilité que x passe du noeud n_i au noeud n (avec $P(n|n_i, x) = 0$ si $n \notin children(n_i)$). Dans nos expériences, π_{θ_i} est une fonction linéaire suivie d'une fonction softmax. Contrairement aux arbres de décision classiques, la décision au niveau des noeuds est donc stochastique durant l'apprentissage (on peut toutefois emprunter le chemin avec la plus forte probabilité un fois les paramètres appris). Le fait d'avoir une décision stochastique nous permet d'utiliser des algorithmes de gradient de la politique, tirés de l'apprentissage par renforcement, pour apprendre les paramètres.

La probabilité d'une trajectoire $H = (n_{(1)}, ..., n_{(t)})$ étant donnée une entrée x est :

$$P(H|x) = \prod_{i=1}^{t-1} \pi_{\theta_{(i)}}(x, n_{(i+1)})$$

Une fois la trajectoire obtenue, la prédiction donnée par le modèle dépend de la feuille $n_{(t)}$ atteinte par x. La prédiction est le $\alpha_{(t)} \in \mathbb{R}^K$ associé à $n_{(t)}$ défini précédemment : il fournit un score par classe. Notons cependant que la complexité en inférence de cette étape est $\mathcal{O}(1)$ car le modèle renvoie juste la valeur $\alpha_{(t)}$.

Après entraînement, la sortie est le vecteur $\alpha_{(t)}$ associé au chemin le plus probable, et la catégorie correspondante est celle de plus forte valeur.

3.2.3 Apprentissage

Le but du processus d'apprentissage est d'apprendre simultanément les politiques π_{θ_i} et les paramètres de sortie α_i, afin de minimiser un coût d'apprentissage Δ (qui correspond ici à un coût de classification, c'est-à-dire une erreur quadratique moyenne ou un coût charnière (*hinge loss*)).

L'algorithme d'apprentissage est basé sur une extension de techniques du gradient de la politique utilisées en apprentissage par renforcement, de manière similaire à ce qui est fait en [DG14]. Plus précisémment, la méthode s'inspire de celle proposée en [BB99] à la différence que, au lieu de considérer une récompense (comme utilisée en apprentissage par renforcement), nous utilisons une fonction de coût Δ. Cette fonction permet de connaître la qualité du système et a l'avantage d'être dérivable et donc de donner un retour plus informatif qu'une simple récompense non dérivable, donnant la direction dans laquelle les paramètres doivent être mis à jour.

La performance du système est notée :

$$J(\theta, \alpha) = E_{P_\theta(x,H,y)}[\Delta(F_\alpha(x, H), y)]$$

où $F_\alpha(x, H)$ est la prédiction faite en suivant la trajectoire H - c'est-à-dire la séquence de noeuds choisis par la fonction π. L'optimisation de J peut être effectuée par descente de gradient ; nous devons ainsi calculer le gradient de J, comme fait pour l'algorithme REINFORCE (voir section 2.3.1) :

$$\nabla_{\theta,\alpha} J(\theta, \alpha) = \int \nabla_{\theta,\alpha} \left(P_\theta(H|x)\Delta(F_\alpha(x, H), y)\right) P(x, y) dH dx dy \tag{3.1}$$

qui peut être simplifié par :

$$\begin{aligned}
\nabla_{\theta,\alpha} J(\theta, \alpha) &= \int \nabla_{\theta,\alpha} \left(P_\theta(H|x)\Delta(F_\alpha(x, H), y)\right) P(x, y) dH dx dy \\
&= \int P_\theta(H|x) \nabla_{\theta,\alpha} \left(log P_\theta(H|x)\right) \Delta(F_\alpha(x, H), y) P(x, y) dH dx dy \\
&+ \int P_\theta(H|x) \nabla_{\theta,\alpha} \Delta(F_\alpha(x, H), y) P(x, y) dH dx dy
\end{aligned}$$

En utilisant une approximation de Monte-Carlo, en prenant M trajectoires sur chaque exemple d'entraînement, et en notant $\Delta(F_\alpha(x, H), y) = \Delta(\alpha_{(t)}, y)$, on obtient :

Algorithm 3 Algorithme de gradient stochastique pour RDT

procedure APPRENTISSAGE($(x^1, y^1), ..., (x^N, y^N)$) ▷ l'ensemble d'apprentissage
  # ϵ est le pas de gradient
  # Initialisation des paramètres :
  $\alpha \approx random$
  $\theta \approx random$
  repeat
    Tirage aléatoire d'une donnée d'apprentissage (x^i, y^i)
    Inférer la trajectoire $H = (n_{(1)}, ..., n_{(t)})$ en utilisant les fonctions $\pi_\theta(x^i)$
    $\alpha_{(t)} \leftarrow \alpha_{(t)} - \epsilon \nabla_\alpha \Delta(\alpha_{(t)}, y^i)$ ▷ maj des param. de la feuille associée
    for $k \in [1..t-1]$ **do**
      $\theta_{(k)} \leftarrow \theta_{(k)} - \epsilon \left(\nabla_\theta \log \pi_{\theta_{(k)}}(x^i) \right) \Delta(\alpha_{(t)}, y^i)$ ▷ maj des param. des noeuds de la trajectoire
    end for
  until Convergence
  return $\mathcal{T}_{\theta,\alpha}$
end procedure

$$\nabla_{\theta,\alpha} J(\theta, \alpha) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \frac{1}{M} \sum_{1}^{M} \left[\sum_{j=1}^{t-1} \left(\nabla_{\theta,\alpha} \left(\log \pi_{\theta_{(j)}}(x) \right) \Delta(\alpha_{(t)}, y) \right) + \nabla_{\theta,\alpha} \Delta(\alpha_{(t)}, y) \right]$$

Intuitivement, le gradient est composé de deux termes :

— le premier terme pénalise les trajectoires avec un coût important et agit uniquement sur les paramètres θ, modifiant les probabilités d'aller d'un noeud à l'autre

— le second terme est le gradient du coût final et ne concerne que les valeurs α associées aux feuilles où sont arrivés les entrées x. Le terme modifie ainsi les α prédits et permet de répartir les différentes classes dans les feuilles de l'arbre.

Nous utilisons un algorithme de descente de gradient avec des mini-batchs pour la back-propagation. La version stochastique sans mini-batch (plus directe à écrire) est décrite en algorithme 3.

La complexité du processus d'inférence est linéaire par rapport à la profondeur de l'arbre. Par exemple, dans un problème de classification avec K classes, la profondeur de l'arbre peut être proportionnelle à $\log K$, résultant en une très bonne vitesse d'inférence.

A noter que des fonctions plus complexes peuvent être utilisées pour π tant qu'une descente de gradient est possible, et qu'il n'y a pas de contrainte sur la forme des paramètres α ni sur la fonction de coût Δ, qui ont uniquement besoin d'être dérivables. Ainsi, le modèle peut également être utilisé pour des problèmes de régression ou pour produire des sorties continues.

3.3 Algorithme : Reinforced Error-Correcting Output Codes (RECOC)

Le problème principal du modèle RDT est que chaque donnée parcourt des branches (et arrive dans une feuille) différentes. C'est l'un des avantages des architectures hiérarchiques (les classifieurs sont adaptés aux différentes données), mais, dans le cas de l'utilisation de réseaux de neurones, ce procédé oblige à faire les calculs donnée par donnée. Or, les utilisations récentes de réseaux de neurones profonds se font habituellement avec des mini-batchs afin d'accélérer l'entraînement mais également l'inférence, c'est-à-dire que les calculs sont faits avec plusieurs données en même temps.

La résolution de cet inconvénient nous a amené à l'algorithme Reinforced Error-Correcting Output Codes que nous allons présenter dans cette partie. RECOC apprend à inférer à partir de chaque donnée un code, qui sera associé à l'une des classes. Nous pouvons voir le modèle RECOC comme une version "aplatie" de RDT, où les paramètres de chaque noeud d'un même niveau sont partagés.

Nous allons d'abord détailler l'architecture RECOC, puis ses liens avec le modèle RDT. Nous présenterons ensuite l'apprentissage des paramètres, puis des extensions du modèle en collaboration avec Thomas Gerald.

3.3.1 Notations et architecture

On considère le même problème que dans le cas des RDT : chaque entrée $x \in \mathbb{R}^n$ est associée à l'une des K classes possibles. $y \in \mathbb{R}^K$ est le label de x, avec

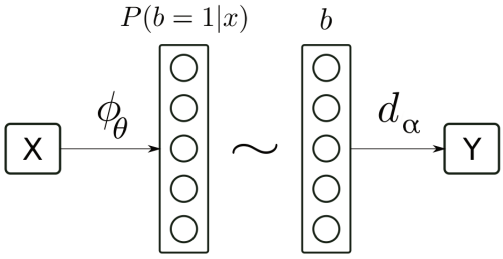


Figure 3.2: Le modèle RECOC (figure tirée de [Ger+17]). ϕ donne la distribution de probabilité sur les codes, b est le code binaire tiré d'après cette distribution, et Y est la classe associée à ce code.

$y_i = 1$ si x appartient à la classe i et $y_i = 0$ sinon. $\mathcal{D} = \{(x^1, y^1), ..., (x^N, y^N)\}$ est l'ensemble des données d'entraînement.

On souhaite associer à chaque entrée x un code binaire b de taille C ($b \in \{0,1\}^C$), qui fournira ensuite la classe correspondante. Le modèle consiste en trois étapes :

— assigner à chaque entrée x une probabilité de distribution sur les codes binaires $P(b|x)$

— tirer un code binaire selon cette distribution

— donner la classe associée à ce code

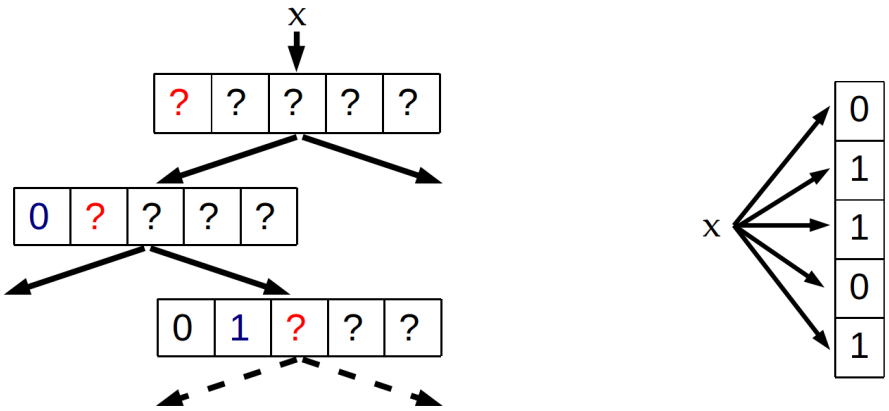
La figure 3.2 illustre ces étapes.

Afin de calculer la probabilité de distribution du code $P(b|x)$, on suppose que tous les bits du code sont indépendants. Ainsi, si l'on note b_i le i-eme bit du code b, la probabilité d'avoir un code b connaissant x se décompose en :

$$P(b|x) = \prod_{i=1}^{C} P(b_i|x)$$

$P(b_i|x)$ peut être modélisée par une distribution de Bernoulli, avec $P(b_i = 1|x) = \pi_i(x)$. Cette fonction π_i sera modélisée par un réseau de neurones, comme dans la section 3.2.

Ainsi, deux fonctions devront être apprises simultanément : $\pi_\theta : \mathbb{R}^n -> [0,1]^C$ qui modélise la probabilité de distribution sur les codes pour une entrée donnée x ; et la fonction de décodage $d_\alpha : \{0,1\}^C -> \{1, ...K\}$ qui associe un code binaire à une classe. Durant l'inférence, plutôt que de tirer un code selon la distribution de probabilité $\pi_\theta(x)$, nous choisirons le code le plus probable selon cette distribution.



(a) RDT

(b) RECOC

Figure 3.3: (a) Modèle RDT avec deux enfants pour chaque noeud : chaque décision est associée à un bit "0" (gauche) ou "1" (droite) (b) Modèle RECOC : les paramètres des noeuds d'un même niveau sont partagés et les choix à chaque noeud sont indépendants des précédents : tous les bits du code peuvent être déterminés en même temps.

3.3.2 Lien avec les RDT

Le modèle RECOC est fortement inspiré du modèle RDT. Il suffit de considérer un arbre de décision RDT où chaque noeud possède deux enfants, et où choisir l'enfant de droite (respectivement de gauche) correspond à choisir le bit "1" (respectivement "0"). Les choix successifs forment le code binaire b, comme illustré sur la figure 3.3a.

Pour RECOC, afin de pouvoir utiliser plusieurs données en même temps pour accélérer les calculs, les paramètres des noeuds d'un même niveau sont partagés et les choix successifs sont indépendants (c'est-à-dire que le choix au niveau d'un noeud ne dépend plus que de l'entrée x et non des choix précédents) comme illustré sur la figure 3.3b. La longueur du code b est la profondeur de l'arbre.

3.3.3 Apprentissage

L'apprentissage de RECOC se fait de la même manière que pour le modèle RDT (voir 3.2.3). On cherche à optimiser la performance du système

$$J(\theta, \alpha) = E_{b\ \pi_\theta(x)}[\Delta(d_\alpha(b^x), y)] \tag{3.2}$$

avec b^x le code binaire tiré d'après la distribution $\pi_\theta(x)$.

On optimise J par descente de gradient en utilisant l'algorithme REINFORCE (voir section 2.3.1), et on obtient le gradient de J comme en section 3.2.3 en utilisant une approximation de Monte-Carlo, en prenant M trajectoires pour chaque exemple d'entraînement :

$$\nabla_{\theta,\alpha} J(\theta, \alpha) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \frac{1}{M} \sum_{1}^{M} \left[\nabla_{\theta,\alpha} \left(\log \pi_\theta(x)\right) \Delta(d_\alpha(b^x), y) + \nabla_{\theta,\alpha} \Delta(d_\alpha(b^x), y)\right]$$

Comme avant, le premier terme permet de mettre à jour les paramètres θ de la distribution de probabilité alors que le second met à jour les paramètres α correspondant à l'association d'une classe à un code.

3.3.4 Extensions

Le problème de l'algorithme REINFORCE est que le temps d'apprentissage est long et qu'il est peu approprié pour des problèmes avec un grand nombre d'actions, ou avec un nombre de décisions séquentielles très important, comme ici pour un code avec un grand nombre de bits. Des codes de grandes tailles (>50) sont pourtant nécessaires pour obtenir de bons résultats sur des problèmes avec un grand nombre de classes. D'autres approches ont été étudiées avec Thomas Gerald ([Ger+17]) pour faire suite à ces travaux. Nous allons dans la suite présenter les extensions étudiées.

Fonction de décodage

Dans la première version, la fonction de décodage fait correspondre à chaque code un vecteur de probabilité (qui une fois la phase d'apprentissage finie

se résume en la classe de plus forte probabilité). Il est ainsi nécessaire de sauvegarder un vecteur de taille K (K le nombre de classes) pour chaque code, ce qui devient très coûteux en mémoire avec un grand nombre de classe *et* un nombre non négligeable de codes (il faut garder $2^C \times K$ valeurs en mémoire). Il est possible de réduire cette complexité durant l'apprentissage en utilisant un réseau de neurones (un simple perceptron peut suffire) pour associer à un code un vecteur de probabilité sur les classes (cette fois, $C \times K$ valeurs suffisent). Cette solution peut encore être améliorée pour la phase de test en utilisant des techniques de plus proche voisin : on sauvegarde uniquement les codes déjà rencontrés, et lorsqu'on rencontre un code non déjà rencontré, on cherche le code plus proche voisin vu durant la phase d'apprentissage, et la classe retournée est la classe de ce code. La méthode de plus proche voisin a de base une complexité en $O(kc)$ (k le nombre de codes vus en entraînement), mais de nombreuses techniques permettent d'améliorer la complexité, comme celles que nous utilisons [Nor+14] qui permet une complexité $O(\frac{c\sqrt{k}}{\log_2 k})$.

Apprentissage

Des méthodes alternatives à REINFORCE ont été développées pour l'apprentissage de tels problèmes de gradients non différentiables. L'estimateur Straight-Through (STE, [Ben+13]) permet d'obtenir de bons résultats dans le cas de problèmes stochastiques binaires. L'idée du STE est que la back-propagation de la fonction de Heaviside (1 si l'argument est positif, 0 sinon) se fait comme s'il s'agissait de la fonction identité. Il s'agit d'une approche biaisée, mais le signe de la dérivée est cohérent lorsqu'on back-propage à travers une couche unique de neurones stochastiques.

Structure de l'espace latent

Une autre amélioration est d'utiliser un espace latent binaire **structuré**, afin que l'espace latent des codes permette une meilleure généralisation. En effet, lors de l'apprentissage, plusieurs codes différents sont associés à une même classe. Nous souhaitons que les codes d'une même classe soient proches les uns des autres dans l'espace latent. Pour atteindre cet objectif, un terme de régularisation est introduit afin de minimiser les distances *intra-classes* et

de maximiser les distances *extra-classes*. On considère les deux ensembles suivants :

$$\mathcal{D}_{intra} = [((x,y),(x',y')) \in \mathcal{D}^2 | y = y']$$
$$\mathcal{D}_{extra} = [((x,y),(x',y')) \in \mathcal{D}^2 | y \neq y']$$

La nouvelle fonction objectif est alors :

$$J(\theta,\alpha) = E_{b\ \pi_\theta(x)}[\Delta(d_\alpha(b^x),y)] \tag{3.3}$$
$$+ \beta \sum_{((x,y),(x',y'))\in\mathcal{D}_{intra}} \|\pi(x) - \pi(x')\|^2 \tag{3.4}$$
$$- \gamma \sum_{((x,y),(x',y'))\in\mathcal{D}_{inter}} \|\pi(x) - \pi(x')\|^2 \tag{3.5}$$

avec $\pi_i(x) = P(b_i = 1|x)$. La minimisation de la distance intra-classe est représentée par le terme 3.4 et la maximisation de la distance extra-classe est représentée par le terme 3.5. β et γ permettent de modérer ces deux régularisations.

3.4 Expériences

Des expériences ont d'abord été faites sur des données jouets afin de visualiser facilement les résultats en deux dimensions, puis sur des données réelles.

Les modèles RDT et RECOC basiques ont été comparés à trois autres modèles : des SVMs linéaires dans une approches un-contre-tous, des arbres de décision classiques, et un réseau de neurones sans couche cachée. Les modèles RDT, RECOC ainsi que les arbres de décision sont en complexité logarithmique par rapport au nombre de classes durant la phase de test, alors que le SVM linéaire un-contre-tous et le réseau de neurones est en complexité linéaire.

Les hyper-paramètres (pas de gradient, taille des mini-batchs et nombre d'itération) ont été optimisés sur un ensemble de validation, et les résultats sont une moyenne de cinq lancements différents. Pour les arbres de décision, nous utilisons l'implémentation de scikit-learn d'une version optimisée de l'algorithme CART.

Nous récapitulons également les expériences effectuées avec Thomas Gerald [Ger+17] sur les extensions sur modèle RECOC. Les extensions ont été évaluées sur plusieurs jeux de données avec grand nombre de classes : DMOZ, ALOI, et IMAGENET.

3.4.1 Données jouet

Il s'agit de données en 2D composées de 16 ou 32 classes, générées selon des distributions gaussiennes. Chaque classe est composée de 100 vecteurs (50 pour l'entraînement, 50 pour le test) échantillonnés suivant la loi $\mathcal{N}(\mu_c, \sigma_c)$ où μ_c a été généré aléatoirement entre $\{(-1,-1)$ et $(1,1)\}$, et σ_c a également été généré aléatoirement. On peut observer en figure 3.4 les frontières de décisions apprises par les modèles. On remarque bien la différence principale entre les frontières obtenues avec les modèles RDT et RECOC : RDT coupe l'espace de manière locale (les choix de coupes dépendent de la zone et donc des décisions précédentes) alors que RECOC coupe tout l'espace (les décisions ne dépendent pas des précédentes).

Les tableaux 3.1 et 3.2 montrent les résultats sur les données jouets dans le cas de 16 et 32 classes. Pour un même nombre d'enfants par noeud, on voit qu'augmenter la profondeur de l'arbre permet bien d'améliorer les résultats de RDT et de RECOC (par exemple pour 16 classes, les performances de RDT sont de 46% avec une profondeur de 3, alors qu'elles augmentent à 83% avec une profondeur de 5), et qu'à profondeur égale les résultats sont très supérieurs à ceux d'un arbre de décision classique (par exemple pour 16 classes, un arbre de décision classique de profondeur 5 obtient un score de 46% ; il doit utiliser une profondeur de 10 pour obtenir un score semblable à RDT de 80%). Ils permettent même d'avoir des résultats proches de ceux d'un réseau de neurones, tout en ayant une complexité moindre en phase de test.

Le modèle RECOC, moins expressif, donne des résultats inférieurs avec une profondeur équivalente aux RDTs : : le tableau 3.1 avec les résultats sur les données avec 16 classes montre que RDT obtient une performance de 83% avec une profondeur de 5, alors que RECOC nécessite une profondeur de 8 pour atteindre un score de 79%. Ce constat rejoint les observations des frontières de décision en figure 3.4, où les frontières de RECOC sont moins riches que celles de RDT. Il est cependant plus facile d'avoir une profondeur

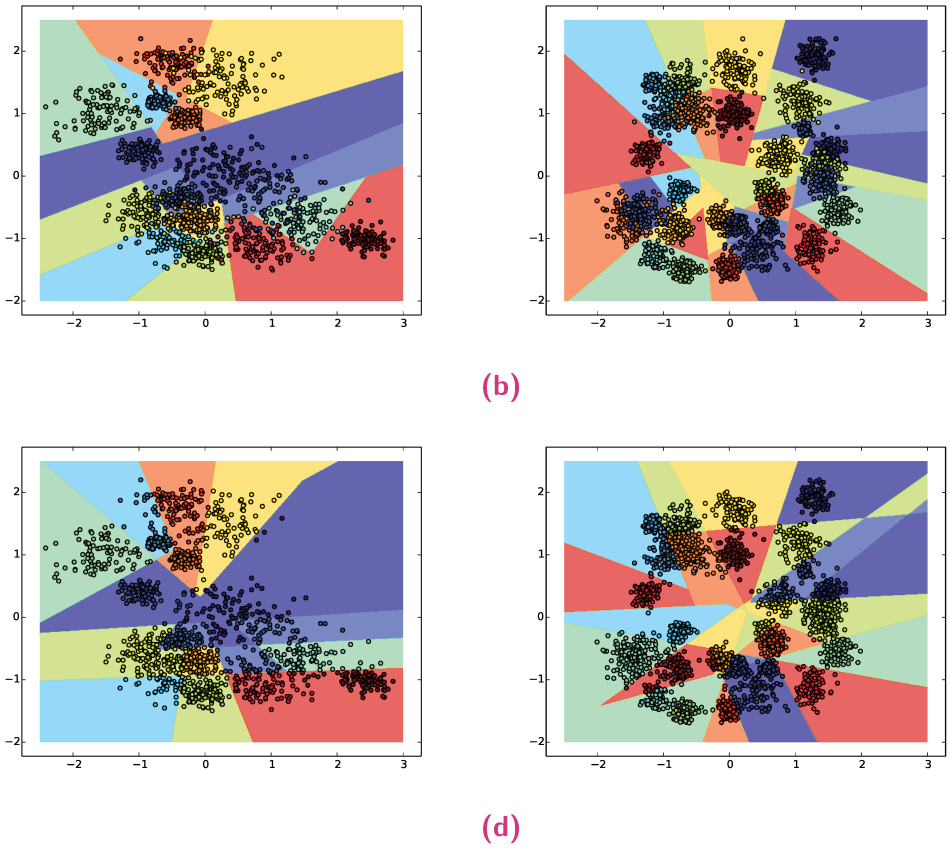


Figure 3.4: Les frontières de décisions apprises par le modèle RDT (a et b) et RECOC (c et d) sur un jeu de données aléatoires en 2D avec 16 (a et c) et 32 (b et d) catégories différentes.

(et donc une taille de code) plus importante sur RECOC - alors que au delà d'une profondeur de 5 (pour $W = 2$), RDT est trop long à entraîner. Les extensions pour lesquelles nous allons présenter les résultats dans la suite permettent d'augmenter encore la taille du code utilisé par RECOC.

3.4.2 Données réelles

Modèle RDT

D'autres expériences avec le modèle RDT ont été effectuées sur différents jeux de données UCI multi-classes. Les détails des jeux de données (nombre d'exemples d'entraînement, nombre de classes...) sont disponibles dans le tableau 3.3, ainsi que l'erreur de classification pour les algorithmes RDT et pour un arbre de décision classique comme dans la section précédente.

			Précision±Variance		
W	D	L	RDT	Random Trees	RECOC
2	3	8	0.46 ± 0.01	0.18 ± 0.01	0.34 ± 0.01
2	4	16	0.70 ± 0.06	0.25 ± 0.02	0.46 ± 0.02
2	5	32	0.83 ± 0.04	0.26 ± 0.06	0.58 ± 0.04
2	6	64	-	-	0.68 ± 0.05
2	7	128	-	-	0.76 ± 0.01
2	8	256	-	-	0.79 ± 0.02
3	2	9	0.51 ± 0.01	0.25 ± 0.01	-
3	3	27	0.75 ± 0.04	0.24 ± 0.02	-

SVM linéaire un-contre-tous :	0.50± 0.01
arbre de décision (depth=5) :	0.46± 0.04
arbre de décision (depth=10) :	0.80± 0.01
arbre de décision (depth=50) :	0.79± 0.01
réseau de neurones :	0.85 ± 0.01

Table 3.1: Résultats pour 16 classes - W est la largeur de l'arbre (ie le nombre d'enfants par noeud), D est la profondeur (la longueur du code pour RECOC) et L est le nombre résultants de feuilles (le nombre de codes différents pour RECOC).

			Précision±Variance		
W	D	L	RDT	Random Trees	RECOC
2	5	32	0.71 ± 0.04	0.10 ± 0.02	0.34 ± 0.00
2	6	64	0.84 ± 0.01	0.11 ± 0.03	0.41 ± 0.02
2	7	128	-	-	0.50 ± 0.02
2	8	256	-	-	0.55 ± 0.02
2	9	512	-	-	0.62 ± 0.03
2	10	1024	-	-	0.72 ± 0.05
3	3	27	0.58 ± 0.04	0.12 ± 0.04	-
3	4	81	0.79 ± 0.04	0.14 ± 0.06	-

SVM linéaire un-contre-tous :	0.54± 0.01
arbre de décision (depth=5) :	0.77± 0.03
arbre de décision (depth=10) :	0.88± 0.02
arbre de décision (depth=50) :	0.86± 0.01
réseau de neurones :	0.88 ± 0.01

Table 3.2: Résultats pour 32 classes - W est la largeur de l'arbre (ie le nombre d'enfants par noeud), D est la profondeur (la longueur du code pour RECOC) et L est le nombre résultants de feuilles (le nombre de codes différents pour RECOC).

	D	pendigits	optdigits	letter	mnist
#train samples		3745	1911	6661	50000
#val samples		3748	1908	6652	10000
#test samples		3497	1796	6687	10000
#features		16	64	16	784
#classes		10	10	26	10
SVM linéaire		17.38 ± 3.62	6.84 ± 0.46	29.98 ± 0.01	8.47 ± 0.02
Arbre de décision	5	22.61 ± 0.08	31.47 ± 0.13	63.24 ± 0.01	32.53 ± 0.01
	8	11.40 ± 0.20	18.89 ± 0.21	37.22 ± 0.08	18.44 ± 0.02
	10	9.99 ± 0.24	16.63 ± 0.35	30.73 ± 0.09	14.09 ± 0.06
	(*)	9.74 ± 0.35	16.00 ± 0.24	17.90 ± 0.14	12.87 ± 0.04
RDT	5	5.35 ± 1.41	7.24 ± 0.83	30.89 ± 1.13	7.73 ± 0.50
	8	4.45 ± 1.25	5.86 ± 0.26	15.81 ± 0.75	6.25 ± 0.57
	10	3.43 ± 0.34	5.35 ± 0.71	12.34 ± 0.44	5.41 ± 0.44

Table 3.3: Erreur de classification (%) de différents modèles et de RDTs. Dans le cas de RDT, chaque noeud a exactement deux enfants. D est la profondeur des arbres construits. Dans le cas de l'arbre de décision (*), les noeuds sont étendus jusqu'à ce que toutes les feuilles contiennent une unique classe ou moins de deux exemples.

Les meilleurs résultats sont obtenus avec le modèle RDT et la profondeur maximale $D = 10$, comme montré en fond grisé : par rapport à des arbres de décision classiques, RDT prend plus de temps à être entraîné mais donne de meilleurs résultats. Les meilleurs résultats doivent être dû aux frontières de décision linéaires mais pas forcément parallèles aux axes, ainsi qu'à l'optimisation d'un objectif global plutôt que l'utilisation d'un algorithme glouton.

Modèle RECOC

Le modèle RECOC basique a également été évalué sur le jeu de données ALOI [1] qui contient 108000 exemples (dont 10% ont été pris en tant que données de test) de dimension 128, répartis en 1000 classes. Les résultats dans le tableau 3.4 sont pour le modèle RECOC, le modèle RDT prenant trop de temps à être entraîné. Les valeurs sont les moyennes sur 3 expériences.

Encore une fois, augmenter la taille du code améliore fortement les résultats (précision de 62% pour une profondeur de 18, au lieu de 38% pour une profondeur de 12), et l'utilisation de la baseline dans l'algorithme REINFORCE pour réduire la variance du gradient est ici utile (précision de 67% pour une

1. téléchargé sur http ://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/multiclass.html#aloi

D	Précision±Variance	
	pas de baseline	baseline moyenne glissante
12	0.38 ± 0.00	0.43 ± 0.00
14	0.47 ± 0.03	0.54 ± 0.01
16	0.50 ± 0.04	0.62 ± 0.01
18	0.62 ± 0.04	0.67 ± 0.01

SVM linéaire un-contre-tous :	0.77
NN 8	0.67
NN 12	0.76
NN 16	0.84
NN 20	0.87

Table 3.4: Résultats pour le modèle RECOC sur les données aloi. Les réseaux de neurones ont une couche cachée suivie d'une sigmoïde : "NN 8" signifie que la couche cachée possède 8 neurones.

profondeur de 12 au lieu de 62%). Les résultats restent cependant inférieurs à un SVM un-contre-tous ou à des réseaux de neurones, mais la complexité en inférence est moindre (logarithmique contre linéaire en le nombre de classes). D'autres modèles récents ([CL14a], [MK15]) avec une complexité logarithmique rapportent des performances de 85% à 90%, mais les variantes du modèle RECOC avec l'estimateur de gradient STE permettent d'utiliser des codes plus conséquents et d'obtenir des résultats comparables, comme nous allons le voir dans la suite.

Modèle RECOC avec extensions

Les expériences qui évaluent les extensions du modèle RECOC ont été effectuées par Thomas Gerald [Ger+17]. Elles ont été évaluées sur trois jeux de données avec un grand nombre de classes, souvent utilisés dans la littérature (on peut voir leurs détails dans le tableau 3.5) :

— *ALOI* [Gal+13] est un jeu de données de 1k classes contenant les caractéristiques SIFT extraites d'images.

— *DMOZ* [Par+15] contient de courtes descriptions textuelles pré-processées en représentation sac-de-mots ; on y trouve 12275 classes. Le jeu de données *DMOZ-1K* désigne un extrait de 1k classes.

— *ImageNet* contient 1k catégories d'images. On utilise comme entrée les caractéristiques du modèle pré-entraîné *resnet-152* [He+15].

Nom du jeu de données	nombre def classes	nombre d'exemples
DMOZ-1K	1000	$41,846 \pm 5,255$
DMOZ-12K	12275	$155,775$
ALOI	1000	$108,000$
IMAGENET	1000	$14,197,122$

Table 3.5: Caractéristiques des jeux de données

Pour tous les jeux de données, 80% des données sont utilisés pour l'entraînement, 10% pour la validation et les 10% restants sont utilisés en test pour l'évaluation de la précision.

Les expériences dans cette partie sont toutes faites avec l'estimateur STE et Adam [KB14a]. Deux variantes sont évaluées pour le décodeur (qui permet d'obtenir la classe à partir du code binaire) :

— le décodeur linéaire "basique"

— le décodeur "plus proche voisin" noté "nn" décrit en section 3.3.4

Dans les deux cas, nous testons avec et sans la régularisation proposée en section 3.3.4. Lorsque la régularisation (notée "reg") est utilisée, le facteur de régularisation est augmenté ($\times 2$) quand la précision sur l'ensemble de validation s'améliore, et réduit ($\times \frac{1}{2}$) quand la précision sur l'ensemble de validation diminue.

Les pas de gradient et les valeurs initiales du facteur de régularisation sont sélectionnés selon la précision sur l'ensemble de validation.

Nous comparons les résultats à un perceptron multi-couches classique (MLP) avec une couche cachée totalement connectée[2] et à un algorithme ECOC avec des classifieurs linéaires. Des algorithmes un-contre-tous ont été testé avec des résultats similaires aux meilleurs résultats des MLP.

La figure 3.5 montre la précision en fonction de la complexité des algorithmes utilisés sur le jeu de données DMOZ-12k (le modèle noté DSNC est un modèle RECOC avec estimateur STE). Le modèle RECOC avec régularisation possède une complexité similaire aux ECOC tout en permettant de meilleurs résultats, et obtient même des résultats proches des MLP tout en ayant une complexité moindre.

2. la taille du code désigne alors la taille de la couche cachée

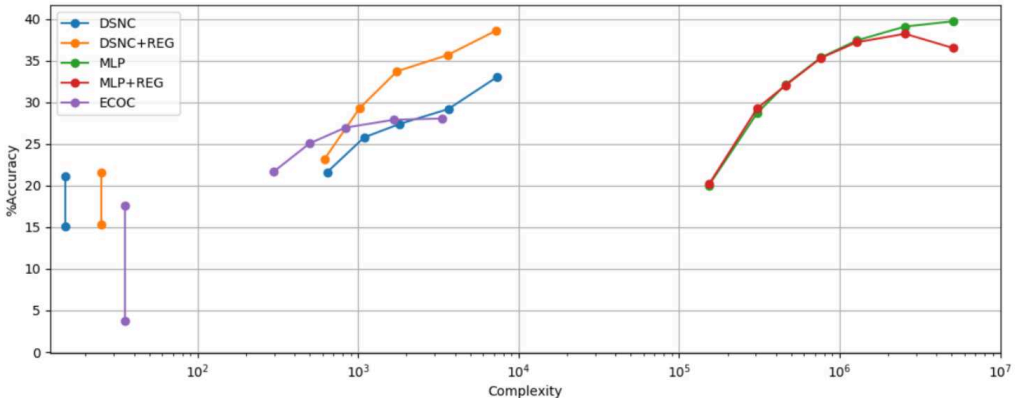


Figure 3.5: Précision en fonction de la complexité sur le jeu de données DMOZ-12k. Le modèle noté DSNC est le modèle RECOC avec un décodeur plus-proches-voisins.

Des résultats plus détaillés sont disponibles dans le tableau 3.6, qui montre les précisions obtenues en test sur les différents jeux de données. On remarque dans un premier temps que les résultats avec le décodeur plus proche voisin ("NN") sont presque équivalents au décodeur linéaire. Ainsi, pour des codes de petites tailles où il est possible de stocker en mémoire tous les codes et leur classe associée, il est plus intéressant d'utiliser le décodeur linéaire (plus rapide et avec de meilleurs résultats). Si le stockage devient trop difficile, les pertes induites par l'utilisation du décodeur plus proche voisin sont toutefois minimes.

On remarque également qu'en dehors du jeu de données IMAGENET, notre modèle obtient de meilleures performances que la baseline ECOC et très proches du modèle MLP tout en ayant une complexité moindre. La régularisation permet dans certains cas d'améliorer légèrement les résultats, en particulier pour un code de grande taille. La régularisation, également testé avec les MLP, semble par contre dans ce cas ne pas améliorer les résultats. Le modèle proposé permet ainsi un bon compromis entre les résultats (meilleurs que les ECOC) et la complexité (meilleure que les autres modèles testés).

3.5 Autres travaux pertinents

Les modèles hiérarchiques peuvent se diviser en deux catégories. Une première concerne les données où une catégorisation hiérarchique est déjà connue ; dans ce cas, cette hiérarchie des classes peut être utilisée directement pour former une hiérarchie sur les classifieurs.

	model	DSNC				MLP		ECOC
dataset	code size	linear	NN	linear+Reg.	NN+Reg.		Reg.	
DMOZ-1K	12	31.772	22.425	33.156	23.492	39.204	39.716	10.738
	24	39.736	36.779	41.326	37.028	48.496	48.748	22.144
	36	42.928	41.208	45.414	42.776	51.48	51.716	27.589
	60	46.84	44.734	48.742	47.41	53.532	54.084	33.850
	100	49.164	46.807	50.984	49.888	54.954	55.658	38.212
	200	51.058	49.065	53.052	52.355	56.262	56.908	41.33
DMOZ-12k	12	15.09	15.2	15.34	15.24	20.05	20.18	3.8
	24	21.15	18.79	21.64	19.27	28.74	29.3	17.591
	36	24.83	22.21	25.95	24.17	32.14	32.05	21.71
	60	28.36	25.97	29.71	29.96	35.41	35.36	25.079
	100	30.42	27.6	31.98	33.94	37.45	37.23	27
	200	32.22	29.32	33.95	35.95	39.12	38.25	27.91
	400	31.96	33.02	33.6	38.65	36.71	39.75	28.08
ALOI	12	34.918	34.84	33.328	33.366	82.04	81.992	1.53
	24	67.92	63.66	66.064	64.19	88.174	87.27	4.21
	36	76.73	74.19	75.84	79.94	89.91	88.18	5.81
	60	83.478	83.19	82.66	81.74	92.22	89.78	9.03
	100	88.014	88.58	87.606	88.72	99.96	90.79	13.8
	200	91.288	91.88	90.542	91.26	95.15	92.41	22.4
IMAGENET	12	1.5	0.82	1.49	0.805	14.6	13.11	12.32
	24	15.6	7.705	9.01	4.595	53.19	53.46	32.42
	36	53.82	36.46	45.74	25.14	59.11	58.85	45.03
	60	60.07	48.61	63	53.655	60.83	63.66	56.5
	100	68.66	63.405	68.81	63.515	65.9	66.81	64.5
	200	70.67	66.935	71.61	68.54	-	67.3	69.76

Table 3.6: Précision du modèle RECOC avec l'estimateur STE, et deux baselines, sur les différents jeux de données. Le fond grisé désigne un temps de décodage constant.

Une seconde catégorie cherche à construire une hiérarchie grâce aux données d'entraînement. Généralement, les algorithmes construisent la hiérarchie dans une étape préliminaire avant d'apprendre à classifier les données. La construction de la hiérarchie peut se faire avec des techniques de clustering (comme l'utilisation d'un clustering spectral sur la matrice de confusion dans [Ben+]), d'un arbre probabiliste sur les labels dans [Liu+13b], ou des techniques d'optimisation sur les partitions [Wes+].

Le modèle RDT, lui, permet d'apprendre en une seule étape la hiérarchie et le classifieur grâce à une unique fonction de coût. D'autres travaux cherchent à apprendre hiérarchie et classifieur en une étape. [JJ94] apprend une mixture hiérarchique d'experts grâce à un algorithme EM, et [BB96] construit un arbre de décision globalement optimal et peut également optimiser un arbre de décision déjà existant. Dans [CL14b], la hiérarchie est apprise grâce à des algorithmes d'apprentissage online et la construction de l'arbre se fait pendent l'apprentissage. Plus récemment, [Nor+15] utilise également un arbre de décision et optimise les fonctions de séparation et les paramètres des feuilles ensembles grâce à un objectif global et un algorithme de descente de gradient stochastique.

Parmi les familles de méthodes plus proches du modèle RECOC, on trouve les codes correcteurs d'erreur [DB ; Sch ; Cis+] , mais leurs performances sont habituellement inférieures aux méthodes classiques un-contre-tous, et très dépendantes de la facilité de séparer les classes. Les fonctions de hachage, qui transforment des données de taille variable (par exemple une représentation continue) en une représentation de taille fixe (par exemple un code binaire), peuvent aussi s'approcher de notre modèle. Ces algorithmes ont donné de bons résultats pour la récupération d'information [Wan+18], mais pas pour la classification, car les codes dans l'esace de hachage permettent de conserver certaines métriques mais pas de différencier les classes, ni de permettre une généralisation à des données inconnues.

3.6 Conclusion et pistes

Le modèle Reinforced Decision Trees est un arbre de décision construit avec des réseaux de neurones, capable d'apprendre en même temps la structure de l'arbre et comment classer les entrées. Il s'agit d'un modèle séquentiel avec lequel une prédiction peut se faire en utilisant $\mathcal{O}(\log C)$ classifieurs, ce qui rend la méthode appropriée pour des problèmes avec un grand nombre de classes. Qui plus est, le modèle peut s'adapter facilement à des problèmes de régression uniquement en changeant la fonction de coût.

Le modèle Reinforced Error Correcting Output Code est lui aussi construit avec des réseaux de neurones. Il apprend conjointement une fonction qui donne pour une entrée donnée un code binaire, et un décodeur qui associe à chaque code une catégorie. Par rapport à RDT, ce modèle permet d'apprendre plus facilement avec un grand nombre de données, et le modèle étendu avec l'estimateur Straigh-Through peut utiliser des codes plus longs afin d'obtenir de meilleurs résultats. Le modèle permet un bon compromis entre la précision et la complexité des calculs.

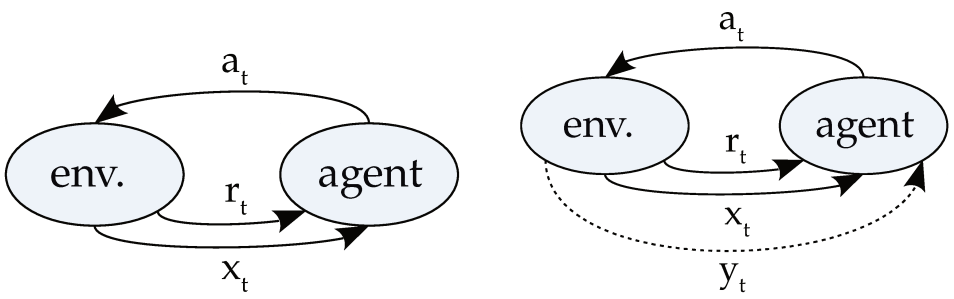
4 Découverte d'une hiérarchie en apprentissage par renforcement

Notre deuxième contribution aborde le problème de l'apprentissage par renforcement hiérarchique. Le framework des options, décrit en 2.4.1, modélise les sous-politiques qui permettent de diviser une tâche complexe en plusieurs sous-tâches plus simples. Nous proposons le modèle *Budgeted Hierarchical Neural Netxwork* (BHNN) inspiré des sciences cognitives pour découvrir et apprendre des options à partir de zéro. Les options émergent naturellement afin de minimiser l'*effort cognitif* de l'agent, qui ici correspond à la quantité d'informations récupérées et utilisées par l'agent.

4.1 Introduction

Cette section traite de l'apprentissage par renforcement hiérarchique, décrit plus en détails dans la section 2.4. En modélisant les sous-tâches par des *options*, nous espérons qu'une tâche complexe soit plus facilement résoluble. Mais la plupart des travaux existants ont besoin d'informations *a priori*, données par un expert, afin de définir les sous-tâches. L'une des difficultés principales de la découverte automatique d'options, avant même leur apprentissage, est de comprendre comment elles émergent.

Plusieurs études en neurosciences étudient la structure hiérarchique des comportements. En particulier, des études récentes suggèrent qu'une hiérarchie peut émerger à partir des *habitudes* [DB13]. Les travaux [Ker+11] et [DB12] font en effet la distinction entre les choix d'actions dirigés vers un but (où l'agent utilise de manière active les informations de l'environnement afin de trouver le comportement optimal) et ceux qui découlent d'habitudes (où l'agent n'utilise pas ou très peu les informations de l'environnement et ne "réfléchit plus" à ce qu'il doit faire). L'idée sous-jacente est que l'agent doit



(a) (b)

Figure 4.1: (a) Une configuration habituelle en RL, où l'agent reçoit une récompense r_t et une observation x_t de l'environnement à chaque pas de temps, puis execute une action a_t. (b) La configuration utilisée pour le modèle BHNN, où l'agent peut demander une observation supplémentaire y_t.

d'abord chercher comment atteindre son objectif, puis passe progressivement à un suivi d'habitudes qui demandent un coût cognitif moindre. Les études suggèrent que le temps de réflexion inférieur dans le cas des actions habituelles permet de résoudre la tâche plus rapidement, et donc d'obtenir une récompense plus importante.

Nous nous basons sur cette idée afin de proposer l'architecture BHNN (Budgeted Hierarchical Neural Network). Nous assimilons le coût cognitif à la récupération et à l'utilisation des informations de l'environnement. Afin de reproduire l'apprentissage d'habitudes qui cherchent à réduire ce coût, nous utilisons un apprentissage budgétisé afin que l'agent agisse plus vite au fil de son apprentissage.

Plus précisément, dans le framework que nous proposons, l'agent a accès à différentes informations provenant de l'environnement. A chaque pas de temps t, l'agent peut utiliser une information basique x_t qui sera utilisée par le contrôleur bas-niveau comme dans les problème d'apprentissage par renforcement classiques. L'agent peut également choisir d'acquérir une information supplémentaire y_t, plus complète, comme illustré dans la figure 4.1. Cette nouvelle information sera utilisée par le contrôleur haut-niveau qui disposera ainsi d'une information plus complète que le contrôleur bas-niveau, mais nous considérons que l'acquisition et l'utilisation de cette observation a un coût cognitif plus important. Cette situation avec deux observations différentes peut se retrouver dans des cas pratiques. Par exemple, un robot qui utilise les informations transmises par sa caméra (x_t) pourrait parfois décider d'effectuer un scan complet de la pièce (y_t) ; un utilisateur conduisant une voiture (observations x_t) peut choisir de consulter une carte ou un GPS

(y_t) ; un agent qui prend des décisions dans un monde virtuel d'après ce qu'il connaît de l'environnement (x_t) peut demander des instructions ou des informations supplémentaires à un humain (y_t), etc.

Nous supposons que des options vont émerger naturellement afin de réduire l'effort cognitif global de l'agent. Formulé autrement, BHNN va apprendre une structure hiérarchique en réduisant son effort cognitif et donc la quantité d'informations haut-niveau utilisées. Nous montrons également que la structure de cette politique hiérarchique peut alors être vue comme une séquence d'options intrinsèques [Gre+16], c'est-à-dire que chaque option est représentée par un vecteur dans un espace latent.

Nous présenterons d'abord l'architecture BHNN, puis nous exposerons les résultats obtenus avec ce modèle dans différentes configurations pour en montrer l'utilité et pour comprendre les options découvertes. Nous exposerons ensuite des travaux similaires avant de conclure.

4.2 L'architecture BHNN

Dans la configuration d'un MDP habituel, l'agent a accès à une observation x_t à chaque pas de temps. Dans le cas de l'architecture BHNN, l'agent a de même accès en permanence à une observation bas niveau x_t, mais peut également acquérir une observation haut-niveau y_t, plus informative, comme illustré dans la figure 4.1. Un cas particulier (appelé dans la suite *blind setting*) est celui où l'observation basique x_t est inexistante, et l'agent doit apprendre quand est-ce qu'il a besoin d'observer l'environnement y_t. En sciences cognitives, cette configuration correspond au choix entre une habitude (sans observation) et un comportement dirigé vers un but [Ker+11 ; DB12].

Nous exposerons dans un premier temps les principes généraux de l'architecture, puis nous la détaillerons. Nous présenterons ensuite les techniques d'apprentissage budgétisé des paramètres. Enfin, nous étendrons le modèle afin d'utiliser des options discrètes.

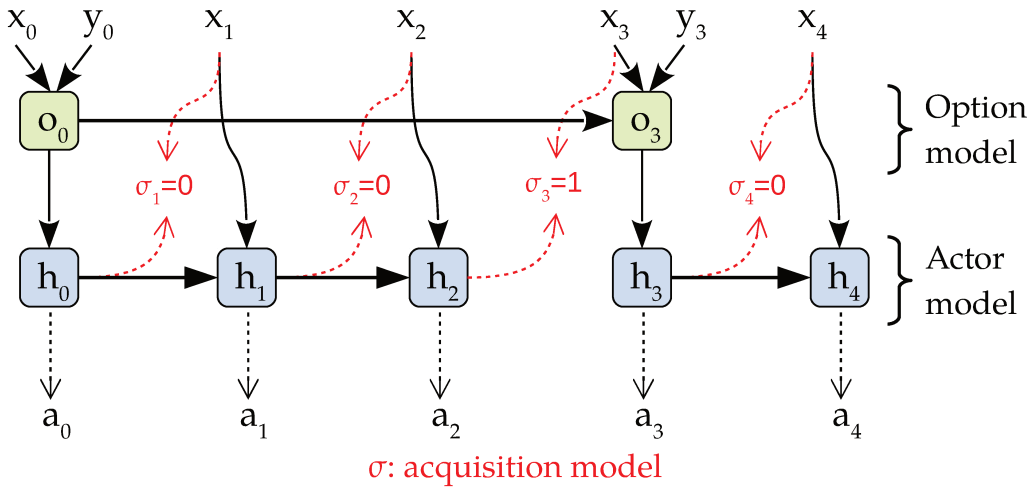


Figure 4.2: L'architecture de BHNN. Les flèches représentent les dépendances, les flèches en pointillés sont des valeurs tirées au hasard. Dans cet exemple, $\sigma_3 = 1$ donc le modèle acquière y_3 et calcule une nouvelle option o_3. Quand $\sigma_t = 0$ (pour $t \in \{1, 2, 4\}$ ici), le modèle n'utilise pas y_t et continue avec la même option.

4.2.1 Notations et principes

La structure de BHNN est semblable à un réseau neuronal récurrent hiérarchique avec deux couches cachés, d'états cachés o_t et h_t. Il est constitué de trois parties. Le **modèle d'acquisition** doit choisir s'il faut acquérir l'observation y_t ou pas. Si l'agent choisit de l'utiliser, le **modèle d'option** utilise alors les deux observations x_t et y_t afin de calculer une nouvelle option définie par un vecteur o_t. Enfin, le **modèle d'acteur** met à jour l'état de l'acteur h_t et permet de choisir quelle action effectuer.

Un schéma de l'architecture de BHNN est donné en figure 4.2, et la procédure d'inférence est donnée dans l'algorithme 4.

4.2.2 Architecture détaillée

Nous allons maintenant détailler davantage ces trois composants. Dans un soucis de simplicité, on considère que la dernière action effectuée a_{t-1} est inclue dans l'observation bas niveau x_t, comme il est couramment fait en apprentissage par renforcement, et nous éviterons d'écrire explicitement dans les équations cette dernière action choisie. Des représentations pour x_t et y_t sont apprises grâce à des réseaux de neurones – des modèles linéaires dans

Algorithm 4 Le pseudo code pour le modèle BHNN.

procedure INFERENCE(s_0) ▷ s_0 is the initial state
    initialize o_{-1} and h_{-1}
    for $t = 0$ to T **do**
        Acquérir l'observation bas-niveau x_t
        Modèle d'acquisition : tirer $\sigma_t \in \{0, 1\}$
        if $\sigma_t == 1$ **then**
            option level : Acquérir l'observation haut-niveau y_t et mettre à jour l'option o_t
            actor level : Initialiser l'état de l'acteur h_t
        else
            actor level : Mettre à jour l'état de l'acteur h_t
        end if
        actor level : Choisir l'action a_t w.r.t h_t
        Exécuter l'action choisie
    end for
end procedure

notre cas, bien qu'on puisse utiliser des réseaux plus complexes, par exemple si les observations sont des images. Dans la suite, par soucis de simplicité, les notations x_t et y_t désignent directement ces représentations. Elles sont utilisées comme entrées par l'architecture BHNN.

Modèle d'acquisition Le modèle d'acquisition permet de décider si une nouvelle observation haut niveau y_t doit être acquise. Il tire aléatoirement $\sigma_t \in \{0, 1\}$ d'après une distribution de Bernoulli avec $P(\sigma_t = 1) = sigmoid(f_{acq}(h_{t-1}, x_t))$. Si $\sigma_t = 1$, l'agent va utiliser y_t pour calculer une nouvelle option (voir le paragraphe suivant), sinon il va uniquement utiliser l'observation bas-niveau x_t pour décider quelle action choisir, étant donné l'option courante.

Modèle d'option Si le modèle d'acquisition a tiré $\sigma_t = 1$, le modèle d'option calcule une nouvelle option étant donné les observations bas-niveau et haut-niveau x_t et y_t, et o_{last} la dernière option choisie avant le pas de temps t : $o_t = gru_{opt}(x_t, y_t, o_{last})$. gru_{opt} représente une cellule GRU [Cho+14].

Modèle d'acteur Le modèle d'acteur met à jour l'état de l'acteur h_t et calcule l'action suivante a_t. La mise à jour de h_t dépend de σ :

— si $\sigma_t = 0$: $h_t = gru_{act}(x_t, h_{t-1})$

— si $\sigma_t = 1$: $h_t = o_t$

L'action suivante a_t est alors tirée selon la distribution $softmax(f_{act}(h_t))$. gru_{act} représente une cellule GRU alors que f_{act} est dans nos expérience un simple perceptron.

4.2.3 Liens avec les options

Les options, décrites plus précisément en 2.4.1, sont souvent utilisées en apprentissage hiérarchique. Une option équivaut à une sous-tâche, et est constituée d'un ensemble d'états initials (où peut débuter cette option), d'une politique, et d'une fonction terminale (qui donne la probabilité que l'option se termine).

Chaque fois que le modèle BHNN choisit d'acquérir une observation haut niveau y_t, il calcule une nouvelle *option intrinsèque* o_t [Gre+16] : il s'agit d'un vecteur qui défini l'option dans un espace latent. De là vient le nom "modèle d'option" du second composant de BHNN. Une option intrinsèque est encodée grâce à un vecteur continu o_t à chaque fois que $\sigma_t = 1$ et qu'une observation haut niveau est récupérée. La politique agit ensuite comme une politique classique, en utilisant la dernière option calculée o_{last} jusqu'à ce qu'une nouvelle option soit calculée. En d'autres mots, quand le modèle acquiert une nouvelle observation haut-niveau, une nouvelle sous-politique est choisie en fonction de celle-ci (ainsi que de l'observation bas niveau et de l'option précédente). L'état de l'option o_t peut être vu comme un vecteur latent qui représente l'option choisie au temps t, alors que σ_t représente ce qui est habituellement appelé la fonction terminale dans les modèles d'options. Habituellement, les modèles d'options utilisent aussi un ensemble d'états initiaux qui déterminent les états du monde où chaque option peut débuter, mais dans le cas de BHNN ce n'est pas nécessaire car l'option est choisie directement selon l'état de l'environnement.

4.2.4 Apprentissage budgétisé

S'inspirant des sciences cognitives [KB14b], on considère que la découverte d'une hiérarchie a pour but de réduire l'effort cognitif de l'agent. Dans notre cas, l'effort cognitif est associé à la quantité d'observations haut niveau y_t qui sont récupérées puis utilisées par le modèle afin de résoudre la tâche, et donc par le nombre de vecteurs d'option o_t calculés par le modèle au

cours d'un épisode. En contraignant le modèle à trouver un compromis entre l'efficacité de la politique et le nombre d'observations haut-niveau acquises, BHNN permet de découvrir quand cette information supplémentaire est indispensable et doit être acquise.

Notons $C = \sum_{t=0}^{T-1} \sigma_t$ le coût de l'acquisition des observations pour un épisode donné. Nous pouvons intégrer ce coût d'acquisition C (qui correspond à l'effort cognitif dont nous parlions plus tôt) dans l'objectif d'apprentissage. Cela rejoint le paradigme d'apprentissage budgetisé déjà exploré dans des problèmes d'apprentissage par renforcement [Con+16 ; DA+12]. Nous définissons ainsi une nouvelle récompense qui inclut ce coût :

$$r^*(s_t, a_t, \sigma_t) = r(s_t, a_t) - \lambda\sigma_t \tag{4.1}$$

où λ permet de contrôler le compromis entre l'efficacité de la politique et la charge cognitive, i.e. le nombre d'observations haut-niveau au cours de l'épisode. Le facteur d'amortissement associé au temps t sera noté R^*_t, et ainsi R^*_0 sera le nouvel objectif à maximiser. N'importe quel algorithme d'apprentissage par renforcement peut permettre de maximiser cet objectif, en notant toutefois que l'apprentissage de BHNN nécessite l'apprentissage à la fois du modèle d'acquisition (c'est-à-dire de la politique qui choisit d'acquérir l'observation y_t ou pas) et des modèles d'option et d'acteur (c'est-à-dire le choix de l'action effectuée par l'agent dans l'environnement).

Dans le cas d'un algorithme du gradient de la politique (voir section 2.3.1), étant donné une trajectoire $x_0, a_0, ...x_T$, le nouveau gradient est :

$$\sum_{t=0}^{T-1} (\nabla_\pi \log P(a_t|h_t) + \nabla_\pi \log P(\sigma_t|h_{t-1}, x_t)) \, (R^*_t - b^*_t) \tag{4.2}$$

où h_t est l'état récurrent qui encode la trajectoire passée $x_0, a_0, ..., x_t$ comme défini en 4.2.2.

Le cas de l'algorithme A3C (voir section 2.3.2) est similaire, et la mise à jour qui en résulte au temps t est :

$$\nabla_\pi \log P(a_t|h_t) A^*(h_t) + \nabla_\pi \log P(\sigma_t|h_{t-1}, x_t) A^*(h_{t-1}, a_t) \tag{4.3}$$

où A^* est une estimation de la nouvelle fonction avantage, associée à la récompense mise à jour r^*.

4.2.5 Version discrète du modèle

Dans le modèle BHNN, nous considérons que l'option est représentée par un vecteur continu o_t dans un espace latent $\mathbb{R}^O$. Nous n'avons ainsi pas besoin de définir à l'avance le nombre d'options différentes, et nous espérons que les options semblables aient des représentations proches.

Habituellement, les options sont cependant discrètes : l'agent dispose d'un catalogue de sous-tâches possibles dont le nombre maximal est défini à l'avance. Nous proposons ici une variante du modèle BHNN qui permet d'apprendre un ensemble discret et fini d'options, comme c'est fait habituellement.

Notons K le nombre d'options, défini manuellement. Chaque option sera associée à un vecteur de représentation o^k dont les paramètres seront appris. Le modèle d'option choisira quelle option doit être utilisée au pas de temps t, à l'aide d'un modèle stochastique qui tirera l'index i_t d'une option d'après une distribution multinomiale.

L'apprentissage doit alors prendre en compte ce choix stochastique lorsqu'une nouvelle option est utilisée. Dans le cas d'un algorithme du gradient de la politique (comme utilisé dans les expériences), le gradient devient :

$$\sum_{t=0}^{T-1} \left(\nabla_\pi \log P(a_t|h_t) + \nabla_\pi \log P(\sigma_t|h_{t-1}, x_t) + \nabla_\pi \log P(i_t|h_{t-1}, x_t, y_t)\right) (R_t^* - b_t^*) \tag{4.4}$$

4.3 Experiences

Plusieurs environnements ont été utilisé afin d'évaluer BHNN. Nous présenterons d'abord les configurations utilisées pour évaluer différents aspects du modèle, puis les détails de l'architecture. Nous exposerons ensuite les différents résultats.

4.3.1 Configurations

La politique a été optimisée grâce à un algorithme du gradient de la politique (voir section 2.3.1) dans tous les environnements, excepté pour les jeux Atari

où A3C (voir section 2.3.2) a été utilisé (comme il est usuellement fait dans la littérature). Pour toutes les expériences, l'optimiseur utilisé est ADAM [KB14a] avec un clipping de gradient. Une recherche par grille a permis d'optimiser les pas de gradient et le coût λ.

Nous avons effectué des expériences dans plusieurs configurations, que nous allons décrire en détail dans les sections suivantes. Sans entrer dans les détails, nous avons :

— la configuration *aveugle* (4.3.3) où BHNN apprend à utiliser une observation le moins souvent possible : l'observation bas niveau est vide, alors que l'observation haut niveau est l'observation habituelle

— la configuration générale, avec à la fois une observation bas-niveau et une observation haut-niveau. Les expériences dans des environnements stochastiques permettent de montrer les avantages d'avoir une observation bas-niveau (peu coûteuse) plutôt que de n'utiliser aucune observation, dans le cas où l'observation haut niveau n'est pas utilisée.

— la configuration où l'observation haut-niveau est donnée par un oracle, et correspond à une instruction

— la découverte d'*options discrètes* grâce à la variante de BHNN

4.3.2 Détails de l'architecture

En dehors des jeux Atari, nous calculons des représentations pour les observations x_t et y_t grâce à des modèles linéaires avec des couches cachées de taille respective N_x et N_y, suivies d'une fonction d'activation $relu$, et les cellules GRU sont de taille N_{gru} (qui dépend de l'environnement) [Cho+14].

Afin qu'un bon compromis soit découvert entre la qualité de la politique, i.e. sa précision, et la fréquence d'observation des observations haut-niveau (qu'on veut faible), le coût de l'observation y_t est d'abord de 0 puis augmente lentement jusqu'à atteindre λ. Des expériences préliminaires avec un λ initial non nul ou un λ trop élevé ont montré que le modèle apprend alors à ne jamais utiliser l'observation haut niveau et n'arrive pas à améliorer cet optimum local.

Figure 4.3: Illustrations des environnements CartPole, LunarLander et 2×2-rooms.

4.3.3 Configuration aveugle

Dans la configuration aveugle, l'agent ne récupère en temps normal aucune observation de l'environnement (il est aveugle). Il doit choisir quand observer autour de lui. Formellement, nous considérons que l'observation bas-niveau x_t est vide et que l'observation haut-niveau y_t est l'observation habituelle transmise par l'environnement. Ce cas rejoint celui décrit en neuroscience qui met en opposition le contrôle suivant un but (utilisant les observations et les connaissances) avec les habitudes (qui effectue les actions habituelles sans observer à nouveau l'environnement, d'où des erreurs si celui-ci a changé). En informatique, cette configuration correspond également au framework des macro-actions (stochastiques dans notre cas) : à chaque nouvelle observation, l'agent effectue une séquence déterminée d'actions (suivant une certaine distribution de probabilité dans le cas stochastique).

Environnements

Plusieurs environnements, dont nous pouvons voir une illustration en figure 4.3, ont été utilisés pour évaluer BHNN dans cette configuration :

— **CartPole :** Il s'agit de l'environnement classique "cart pole", implémenté dans la plate-forme OpenAI Gym [Bro+16] [1] . Il s'agit d'une voiture qui peut bouger à droite et à gauche, sur laquelle est fixé un pendule inversé. Le pendule est au départ verticale et le but est de ne pas le faire tomber. Dans OpenAI, les observations sont *(position, angle, speed, angular speed)* et les actions possibles sont *droite* et *gauche*. La

1. https ://gym.openai.com/

récompense est de $+1$ pour chaque pas de temps avant que la tâche n'échoue.

Dans cet environnement, les tailles des réseaux utilisés sont $N_y = 5$ et $N_{gru} = 5$.

— **LunarLander :** Cet environnement est celui proposé par OpenAI Gym. L'observation décrit la position, la vélocité, l'angle de l'agent et si il est en contact avec le sol ou pas. Les actions possibles sont *ne rien faire, utiliser le moteur droit, utiliser le moteur gauche, utiliser le moteur principal*. La récompense est $+100$ s'il attérit, $+10$ pour chaque pied à terre, -100 s'il tombe au sol et -0.3 à chaque fois qu'il utilise le moteur principal.

Ici $N_y = 10$ et $N_{gru} = 10$.

— $k \times k$**-rooms :** Cet environnement est un labyrinthe constitué de $k \times k$ pièces avec des portes entre chacune d'entre elles (voir figure 4.5a). L'agent commence toujours à la même position (le coin en haut à gauche) alors que le but final est aléatoire et peut être n'importe où, dans n'importe quelle pièce. La position du but change à chaque épisode, l'agent ne peut donc pas apprendre par coeur à aller à un endroit donné. La récompense est -1 pour chaque mouvement, et $+20$ lorsque l'agent atteint le but. Les actions sont *haut, bas, droite, gauche*. L'observation est la position de l'agent dans la pièce, les positions des portes dans la pièce (si elles sont présentes ou pas) ainsi que la position du but si il est dans la pièce où est actuellement l'agent. L'agent n'observe ainsi que la pièce où il se trouve.

Il est à noter que cet environnement s'inspire des autres problèmes avec des environnements de quatre pièces (introduits par [Sut+99]) mais est bien plus difficile. En effet dans ce dernier, le but ne peut se trouver qu'en une ou deux positions différentes alors qu'il peut être n'importe où dans notre cas. Qui plus est, dans notre cas plus réaliste, l'agent n'observe que la pièce courante.

Ici $N_y = 20$ et $N_{gru} = 10$.

— **Pong :** Il s'agit de l'environnement Pong, venant des jeux Atari. Les jeux Atari étant plus complexes, et l'observation étant ici une image, l'architecture de BHNN doit être légèrement adaptée. Nous utilisons donc la variante LSTM de l'algorithme A3C [Mni+16b], et la représentations des observations y_t (les images tirées du jeu) viennent d'un réseau de convolution (quatre couches de convolution, chacun avec 32 filtres de taille 3×3 avec stride 2). Les couches du LSTM sont de taille 256.

Comme fait habituellement dans le cas des jeux Atari, chaque action est répétée quatre fois après avoir été choisie. Les résultats avec l'algorithme FiGAR (décrit plus loin) viennent directement de l'article concerné [Sha+17].

Dans le cas des environnements où l'observation au temps t contient toutes les informations nécessaires comme CartPole et LunarLander, le modèle d'option n'a pas besoin d'être récurrent. Ainsi, o_t est uniquement dépendant de x_t et de y_t.

Algorithmes utilisés pour comparaison

Nous comparons BHNN a un algorithme du gradient de la politique récurrent (noté R-PG) avec des cellules GRU. A noter que cette méthode R-PG a accès à toutes les observations (x_t et y_t) à chaque pas de temps, et peut trouver les politiques optimales dans ces environnements ; alors que BHNN doit apprendre à utiliser l'observation le moins possible tout en conservant une politique raisonnable.

Nous comparons également BHNN au framework FiGAR (Finer Grained Action Repetition [Sha+17]). FiGAR permet à l'agent de choisir une action et le nombre de fois où cette action va être répétée avant de devoir choisir une nouvelle action. Ainsi, FiGAR utilise l'observation de l'environnement uniquement quand il faut décider d'une nouvelle action. Nous avons réimplémenté FiGAR en utilisant une architecture semblable à R-PG et à BHNN, en fixant le nombre de répétitions maximales à 10 pour CartPole et LunarLander et à 5 pour 3×3-rooms (d'autres nombres ont été testé mais donnaient de moins bons résultats). Afin de comparer équitablement les méthodes, nous avons également testé FiGAR avec une récompense augmentée r^* qui inclut le coût λ.

Résultats

Le tableau 4.1 contient les résultats. Deux versions de chaque environnement ont été utilisées : une déterministe ($\epsilon = 0$) et une stochastique ($\epsilon = 0.25$), où

		$\epsilon = 0$		$\epsilon = 0.25$	
		R	%obs	R	%obs
Cartpole	R-PG	200	1	196.0	1
	FiGAR $\lambda = 0$	200.0	0.38	185.3	0.44
	FiGAR $\lambda = 1$	199.3	0.23	184.2	0.3131
	BHNN $\lambda = 0.5$	199.7	0.06	181.6	0.26
	BHNN $\lambda = 1$	190.3	0.05	172.2	0.20
Lunar Lander	R-PG	227.3	1	109.3	1
	FiGAR $\lambda = 0$	227.5	0.33	119.6	0.31
	FiGAR $\lambda = 0.5$	220.6	0.20	108.89	0.14
	BHNN $\lambda = 0.5$	221.2	0.16	91.6	0.07
	BHNN $\lambda = 5$	210.5	0.06	90.4	0.04
3×3-rooms	R-PG	-3.3	1	-14.9	1
	FiGAR $\lambda = 0$	-5.02	0.68	-15.4	0.98
	FiGAR $\lambda = 1$	-6.9	0.44	-22.1	0.52
	BHNN $\lambda = 1$	-7.4	0.36	-16.3	0.61
Pong	LSTM-A3C	21	0.25	-	-
	FiGAR $\lambda = 0$	20.32	0.077	-	-
	BHNN $\lambda = 0.01$	19.9	0.02	-	-

Table 4.1: Valeurs des récompenses et des pourcentages d'observations de y_t pour les différents environnements, avec différentes valeurs de coût λ et différents niveaux de stochasticité de l'environnement ϵ.

le mouvement de l'agent peut échouer avec la probabilité ϵ. Dans ce cas, une transition aléatoire est appliquée.

Dans les environnements déterministes ($\epsilon = 0$), le modèle BHNN réalise des performances semblables aux baselines, alors que le modèle n'utilise l'observation que très peu au cours d'un épisode. Dans le cas de CartPole par exemple, nous voyons ligne 4 que l'agent n'utilise l'observation que 6% du temps (quand $\lambda = 0.5$) pour des résultats comparables au modèle du gradient de la politique (qui utilise l'observation en permanence) ou au modèle FiGAR (qui peut n'utiliser l'observation que environ un quart du temps). Ces résultats montrent clairement que dans le cas d'environnements déterministes suffisamment simples, recevoir des observations en permanence n'est pas nécessaire : étant donné une situation initiale, l'agent peut apprendre quelle séquence d'actions est optimale à partir d'un état initial donné, et cette séquence sera toujours la même pour cet état. Lorsque nous comparons BHNN avec FiGAR, nous voyons que BHNN permet d'obtenir à peu près les mêmes performances que FiGAR tout en observant beaucoup moins l'environnement. Là encore, le résultat est attendu : FiGAR répète la même action plusieurs fois alors que BHNN permet de découvrir des macro-actions, qui mélangent des actions différentes et permettent plus de possibilités.

L'hyper-paramètre du coût λ permet d'ajuster le compromis entre la performance et la quantité d'observations utilisées. Un coût plus élevé incite le

modèle à observer moins souvent, en dégrandant aussi peu que possible les performances.

Dans les environnements stochastiques ($\epsilon = 0.25$), les observations doivent être utilisées plus souvent et les performances sont moindres : pour l'environnement 3×3-rooms ligne 14 par exemple, nous voyons que BHNN utilise des observations 61% du temps dans le cas stochastique (contre 36% dans le cas déterministe) tout en ayant une performance moindre (-16.3 contre -7.4). Ces résultats sont dus à la stochasticité : il n'est dans ce cas pas possible pour l'agent de déduire sa position et de prévoir l'évolution du monde en connaissant uniquement l'action qu'il a choisi. Ainsi, l'agent doit observer autour de lui plus souvent. Cela démontre les limites des politiques open-loop (qui n'utilisent pas d'informations provenant de l'environnement) dans des environnements non prévisibles, et justifie l'utilisation d'une observation bas-niveau x_t dans la suite : cette nouvelle observation permettra de faire face à la stochasticité, tout en n'étant pas aussi coûteuse que l'observation haut-niveau y_t.

4.3.4 Utilisation d'observations bas-niveau et haut-niveau

Comme montré précédemment, le problème des politiques open-loop (c'est-à-dire des politiques qui décident d'une action sans utiliser aucun retour de l'environnement) est que, si l'environnement est stochastique, l'agent ne peut pas prévoir son évolution sans en recevoir des informations.

Nous étudions donc maintenant une configuration où l'environnement délivre en permanence une information peu coûteuse et minimale x_t, alors que l'observation haut-niveau et plus complète y_t n'est délivrée que sur demande. Le raisonnement est que x_t peut permettre de mieux décider quelle action effectuer tout en demandant un effort cognitif moindre par rapport à l'utilisation de l'observation complète y_t.

Nous utilisons dans ce but une nouvelle version de l'environnement $k \times k$-rooms. Cette fois, x_t contient la position de l'agent dans la pièce actuelle, alors que y_t contient le reste de l'information (c'est-à-dire la position des portes, et la position du but s'il est dans la pièce actuelle). Ainsi, la différence avec la version précédente est que l'agent a accès en permanence à sa position

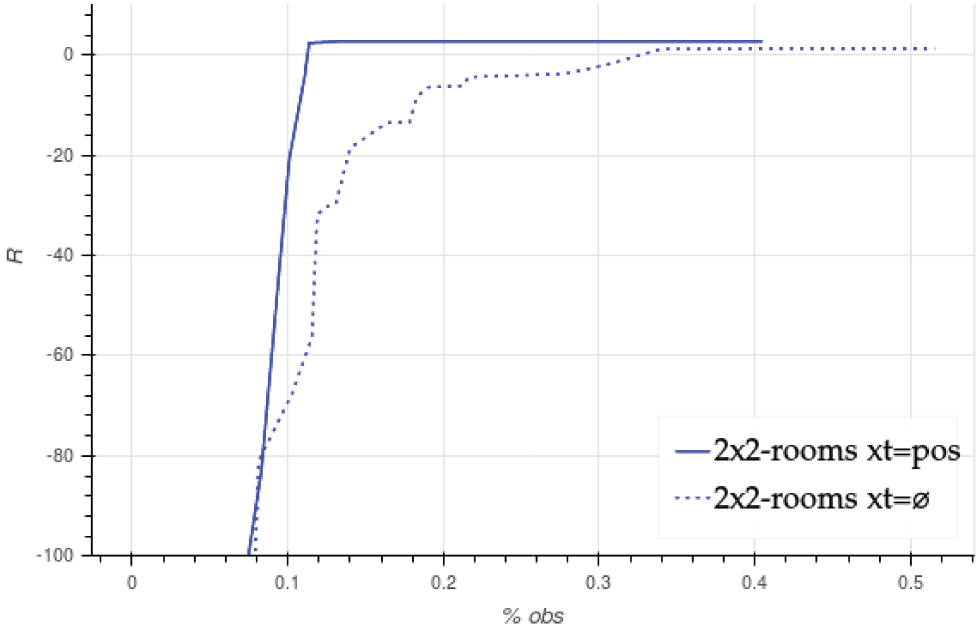


Figure 4.4: Courbes des récompenses en fonction du coût, pour l'environnement 2×2-rooms, pour une stochasticité $\epsilon = 0.25$.

plutôt que ne la connaître que lorsqu'il utilise l'observation y_t. Dans ce cas, nous utilisons $N_x = 10$, $N_y = 10$ et $N_{gru} = 10$.

Dans un environnement stochastique avec $\epsilon = 0.25$, BHNN utilise l'observation haut-niveau y_t uniquement 16% du temps (contre 60% dans le cas "aveugle") tout en conservant une récompense totale semblable (-18).

Les courbes de la figure 4.4 nous montrent la récompense totale obtenue en fonction du coût (ces courbes ont été obtenues en calculant le front de Pareto de modèles BHNN avec des pourcentages d'observations différents ; ces différences d'observation ont été obtenues en diminuant progressivement le coût λ). La courbe signifie par exemple que pour 20% d'observations (0.2 en abscisse), le modèle "aveugle" obtient une récompense d'environ -8 (courbe en pointillé), alors que le modèle qui utilise deux niveaux d'observations conserve une récompense supérieure à 0 (courbe continue). Les expériences ont été faites dans l'environnement 2×2-rooms, plus rapides que celles dans l'environnement 3×3-rooms. En comparant la courbe du modèle en configuration aveugle à celle du modèle avec deux observations (où $x_t = position$), on constate que la chute de performance dans le cas $x_t = position$ survient avec un coût inférieur, c'est-à-dire que l'utilisation d'une observation bas-niveau permet de conserver des performances raisonnables tout en ayant un coût cognitif moindre. En effet, avec $x_t = position$, l'agent connaît en permanence

sa position dans la pièce et peut davantage compenser la stochasticité de l'environnement. Tout comme dans le cas déterministe aveugle, l'agent peut découvrir des options pertinentes et ne récupère l'information haut niveau qu'une seule fois par pièce (voir section 4.3.5).

Cette expérience montre l'utilité d'utiliser en permanence une observation bas-niveau qui demande un coût cognitif moindre (que ce soit par le coût d'acquisition ou le coût d'utilisation de cette observation), en contraste avec les politiques temporairement open-loop qui n'utilisent parfois aucun retour de l'environnement.

4.3.5 Analyse des options découvertes

La figure 4.5a illustre les trajectoires prises par l'agent dans l'environnement 3×3-rooms. Les points représentent les positions où l'agent a utilisé une observation haut-niveau et donc où de nouvelles options ont été générées. Nous constatons que l'agent apprend à observer y_t uniquement une fois par pièce (il n'y a qu'un point par pièce pour chaque trajectoire) puis utilise l'option choisie jusqu'à ce qu'il arrive dans une nouvelle pièce. Ainsi, l'agent déduit de y_t s'il doit bouger jusqu'à une autre pièce (et par où), ou s'il doit rejoindre le but (s'il se trouve dans la pièce actuelle). Précisons que l'agent ne peut pas se diriger directement dans la pièce où se trouve le but car le but peut changer de pièce à chaque nouvelle trajectoire et l'agent ne peut avoir aucune information à ce sujet, avant de se trouver dans la pièce concernée. Etant donné qu'il ne voit que ce qu'il y a dans la pièce actuelle, il les explore toutes les unes après les autres.

Nous pouvons également observer les vecteurs latents des différentes options en utilisant l'algorithme t-SNE (figure 4.5b). Les options représentées par des couleurs identiques (par exemple tous les points verts) correspondent à des observations où les buts sont dans des positions similaires. Nous observons que toutes les options vertes (ou toute autre couleur) sont proches dans la projection de l'espace latent, ce qui signifie que l'espace latent des options capture effectivement de l'information pertinente au sujet des options semblables. En effet, l'agent doit effectuer un parcourt similaire pour des buts qui sont proches les uns des autres.

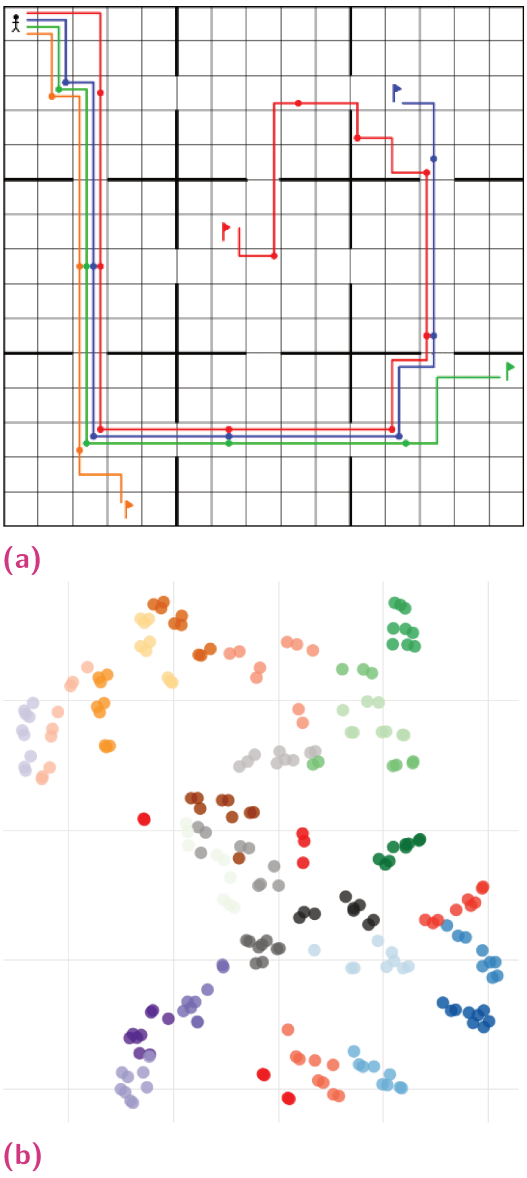
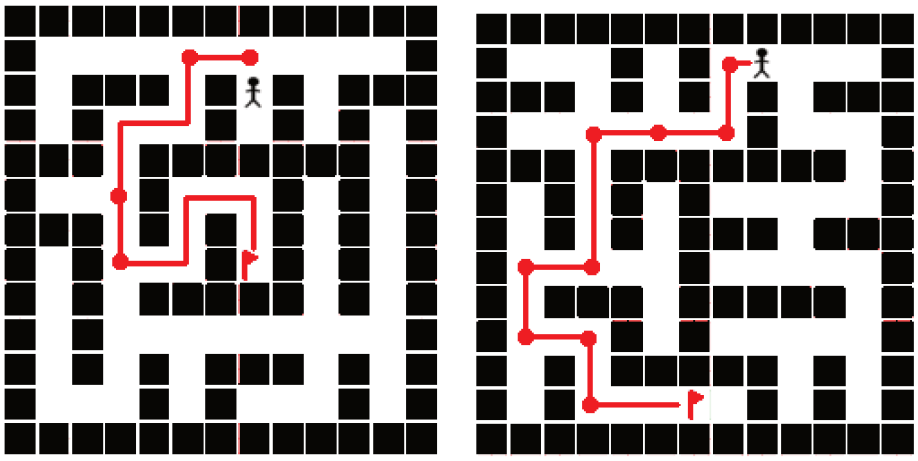


Figure 4.5: (a) 3×3-rooms : exemple de trajectoires (chacune en couleur différente) générée par l'agent. Les points sont les positions où l'agent acquiert une observation y_t et génère une nouvelle option : l'agent utilise seulement une option par pièce. (b) Les vecteurs latents des options visualisés grâce à l'algorithme t-SNE. Les couleurs proches correspondent à des situations semblables, quand le but est dans une même partie de la pièce ; les points rouges correspondent à des options où l'agent passe par une porte pour rejoindre la pièce suivante, ou quand le but est proche d'une porte.



(a) (b)

Figure 4.6: Exemple de trajectoires dans des labyrinthes. Chaque point correspond à une position où l'agent a décidé d'acquérir y_t et donc de générer une nouvelle option. (a) Un apprentissage correct où y_t est uniquement utilisé à chaque intersection. (b) Apprentissage moins optimal où l'agent acquiert y_t à chaque changement de direction.

4.3.6 Demande d'instructions

Nous considérons finalement une configuration où y_t est une information donnée par un oracle, alors que x_t est une observation classique. L'idée sous-jacente est que l'agent peut soit agir comme d'habitude en utilisant les informations fournies par l'environnement, soit utiliser des informations données par un modèle avec un coût plus élevé ou par un humain afin de choisir plus facilement quelle action effectuer.

Afin d'étudier ce cas particulier, nous considérons un labyrinthe généré aléatoirement pour chaque épisode (ainsi l'agent ne peut pas apprendre par coeur la carte de l'environnement). Le but et la position initiale de l'agent sont également aléatoire. L'observation x_t correspond aux 9 cases entourant l'agent (il voit donc autour de lui mais n'a pas connaissance du labyrinthe complet). L'observation y_t est donné par un algorithme de planification qui a accès au labyrinthe complet et peut calculer la trajectoire jusqu'au but : y_t contient l'action que doit effectuer l'agent au moment t sous forme d'un vecteur de 0 et de 1. Le calcul de y_t peut être coûteux dans des cas complexes, d'où l'idée que l'agent ne doit pas l'utiliser à chaque pas de temps et qu'il correspond à un coût cognitif plus élevé. Les paramètres du modèle sont ici $N_x = 10$ et $N_{gru} = 5$ (aucune représentation n'est utilisée pour y_t).

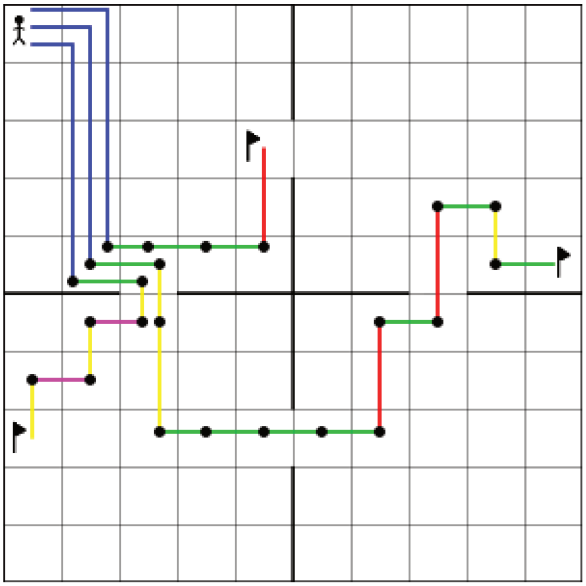


Figure 4.7: Exemple de trajectoires générées avec la version discrète de BHNN. Chaque point représente un changement d'option, et chaque couleur représente une option différente.

Deux exemples de trajectoires générées sont visibles sur la figure 4.6. La figure 4.6a montre une politique optimale apprise avec des hyper-paramètres appropriés. Ici, l'agent a appris à demander une instruction (l'observation haut-niveau y_t) uniquement aux intersections, quand il ne peut pas savoir dans quelle direction aller. Entre les intersections, l'agent suit simplement les couloirs. La figure 4.6b montre par contre une politique non optimale où le contrôleur a appris à demander une instruction à chaque changement de direction en plus des intersections, même quand ce n'est pas indispensable, comme pour les deux premiers points de la trajectoires. En début d'apprentissage, quand l'agent ne sait pas du tout quelle action effectuer (et ne sait par exemple pas qu'il ne sert à rien de foncer dans un mur), l'agent peut en effet se rendre compte que l'instruction lui dit directement quelle est la bonne action à effectuer. Dans les coins du début de la trajectoire 4.6b, plutôt que d'utiliser l'observation pour trouver quelle action effectuer afin de suivre le chemin, il demande directement y_t afin d'obtenir l'information. Une fois cette politique apprise, il est très difficile d'améliorer cet optimum local. Le choix du coût λ (il doit être assez important pour inciter à utiliser le moins possible y_t, mais pas trop pour que l'agent l'utilise de temps en temps) et du pas de gradient sont ainsi cruciaux.

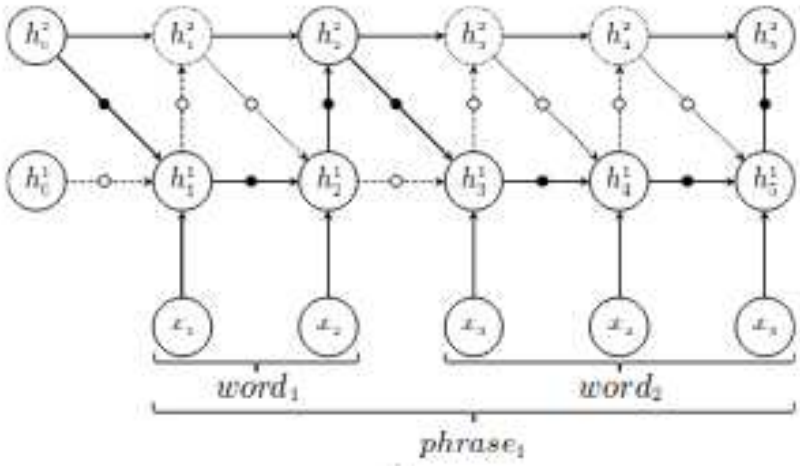


Figure 4.8: Architecture *Hierarchical Multiscale Recurrent Neural Network* (HM-RNN), tirée de [Chu+16]. La structure hiérarchique entre les mots d'une phrase est détectée automatiquement.

4.3.7 Découverte d'options discrètes

Nous utilisons ici la version discrète du modèle décrite en 4.2.5. Nous utilisons l'environnement 3×3-rooms, avec un nombre d'options possibles $K = 9$. Nous pouvons voir un exemple de trajectoires générées sur la figure 4.7. Nous voyons que la politique apprise parvient bien au but, mais l'apprentissage est plus difficile du fait des options discrètes. L'agent apprend à changer d'options beaucoup plus souvent car elles sont moins expressives, et, du fait de la difficulté d'apprentissage (les algorithmes en apprentissage profond ont en général plus de difficultés à apprendre des décisions discrètes), l'agent change parfois d'options alors que ce ne serait pas nécessaire, par exemple pour la politique verte "aller vers la droite". Apprendre des options représentées par des vecteurs latents continus parait donc plus intéressant.

4.4 Autres travaux pertinents

L'architecture la plus proche de BHNN est le Hierarchical Multiscale Recurrent Neural Network [Chu+16], qui découvre des structures hiérarchiques dans des séquences (les expériences sont effectuées sur de la modélisation de langage au niveau des caractères et sur de la génération de séquences écrites à la main). On peut voir une illustration de l'architecture proposée en figure 4.8. Les différentes couches du réseau récurrent sont séparées par un détecteur binaire appris grâce à un estimateur *straight-through* [Ben+13].

Dès 1996, [Han+96] s'intéresse au coût de la récupération d'observations pour un robot dans un environnement incertain. L'agent doit donc apprendre à alterner entre l'observation et la prise de décision pour réduire le coût tout en gardant une bonne idée de l'état de l'environnement. Le concept d'effort cognitif utilisé dans notre modèle a été succintement introduit dans [BP15a] (mais dans le cas d'options discrètes), et les options intrinsèques (c'est-à-dire les options définies par des vecteurs latents) ont été utilisées dans [Gre+16] (l'utlisation de vecteurs pour représenter le but a été d'abord utilisé dans [Sch+15]). L'utilisation de cet effort cognitif permet de définir clairement comment apparaît la hiérarchie.

La *configuration aveugle* du modèle BHNN, décrite dans la section 4.3.3, est très proche de l'utilisation de macro-actions (stochastiques), utilisée par exemple dans [Hau+98 ; Mni+16c] : la politique associe à un état donné une séquence d'actions élémentaires. Le framework FiGAR [Sha+17] auquel on compare notre modèle en est un cas simplifié qui permet à l'agent de répéter la même action plusieurs fois étant donné une observation. Il permet d'augmenter n'importe quel algorithme d'apprentissage par renforcement qui utilise une politique explicite, en lui ajoutant la capacité de prédire pour combien de pas de temps l'action choisie doit être répétée. Ce choix se fait en fonction de l'état courant de l'environnement. Pour déterminer le nombre de répétition d'une action, FiGAR met à jour une seconde politique (qui renvoit le nombre de répétitions à effectuer).

Le problème de ces politiques qui mêlent *open-loop* et *closed-loop* est qu'elles sont trop limitées dans des environnements complexes et/ou stochastiques si elles n'utilisent pas assez souvent les observations de l'environnement.

Il est également possible de séparer l'espace des états en plusieurs parties. Dans [Hee+16], un contrôleur haut-niveau a accès à toutes les informations, alors qu'un contrôleur bas-niveau a uniquement accès aux informations proprioceptives. [Flo+16] factorise également l'espace d'état en deux composants sur un ensembles de tâches. L'observation de l'agent possède une partie commune entre toutes les tâches (qui peut représenter les informations sur la dynamique du robot) alors qu'une autre est spécifique à certaines tâches. L'agent apprend d'abord à résoudre des tâches simples en utilisant l'observation commune entre les tâches, chaque tâche correspondant à une compétence. Une nouvelle tâche plus complexe peut ensuite utiliser ces compétences : l'agent choisit grâce à l'observation complète quelle compétence

il souhaite utliser, puis utilise l'observation commune pour décider d'une action grâce à un réseau de neurones stochastiques (chaque compétence est encodée par un vecteur binaire).

L'architecture générale de BHNN, qui utilise deux observations plus ou moins informatives, ressemble davantage à ces modèles où le contrôleur bas-niveau a accès à moins d'informations que le contrôleur haut-niveau, mais peut tout de même échanger avec l'environnement. Cependant, ces modèles n'apprennent jusqu'à présent pas *quand* ils doivent utiliser le contrôleur haut-niveau.

4.5 Conclusion et pistes

Nous avons proposé dans ce chapitre un modèle hiérarchique pour l'apprentissage par renforcement, où l'agent apprend quand est-ce qu'il doit acquérir une information plus complète mais plus coûteuse. L'idée est en lien avec le concept d'habitudes en neurosciences, et la volonté d'avoir un effort cognitif moindre. Le modèle est appris grâce à un apprentissage budgétisé : l'acquisition de l'information additionnelle, et donc le calcul d'une nouvelle option, engendre un coût qui s'ajoute à la récompense fournie par l'environnement. La politique apprise est alors un compromis entre efficacité et effort cognitif.

Les résultats expérimentaux montrent qu'il est possible de réduire cet effort cognitif (i.e. l'acquisition et l'utilisation des observations) sans que les performances se dégradent trop. Nous avons également montré l'intérêt d'utiliser plusieurs niveaux d'observations et d'utiliser en permanence une observation bas-niveau, plutôt que d'observer soit tout, soit rien : il est ainsi possible d'utiliser moins souvent l'observation haut-niveau tout en conservant un coût cognitif peu élevé. Enfin, l'observation des options trouvées permet de montrer leur pertinence.

Ce travail ouvre différentes perspectives. La première serait d'étudier si BHNN peut être utilisé dans des problèmes d'apprentissages par renforcement multi-tâches : en effet, l'environnement $k \times k$-rooms peut être vu comme un problème multi-tâches, étant donné que la position du but est choisie aléatoirement à chaque épisode. Une autre possibilité serait d'étudier des tâches où l'agent peut acquérir différentes observations (et pas uniquement

deux comme étudié ici), par exemple si un robot a plusieurs capteurs et doit choisir lesquels utiliser. Un problème intéressant mais difficile serait l'interaction avec un autre modèle plus coûteux, comme le suggère les dernières expériences de la section 4.3.6 : l'agent pourrait décider quand utiliser un algorithme model-based (à temps d'exécution plus lent) en utilisant le reste du temps un algorithme model-free. Le soucis de cette configuration est l'apprentissage conjoint des deux modèles, et rejoint les problématiques d'apprentissage multi-agents.

Enfin, il est possible de considérer que l'observation haut-niveau, plus coûteuse, est une instruction qui provient d'un humain qu'on ne souhaite pas solliciter à chaque pas de temps. Cette idée mène au chapitre suivant.

5 Interaction en langage naturel

Notre troisième contribution concerne l'interaction en langage naturel. En effet, dans le cadre d'une interaction d'un agent artificiel avec un humain, l'utilisation du langage naturel serait souhaitable car plus facile à appréhender pour un humain non expert. Nous allons d'abord présenter l'environnement où nous effectuerons nos expériences afin de bien placer le cadre du problème. Nous reproduisons ensuite un modèle d'*acteur* (un agent qui suit des instructions en langage naturel), puis un modèle de *locuteur* (un agent qui génère des instructions). Afin de pallier au manque de données fréquent en interaction, nous testons ensuite un apprentissage joint des deux modèles précédents. Enfin, nous présenterons un essai d'architecture hiérarchique, utile lorsque l'agent suit plusieurs instructions, ou génère plusieurs instructions, à la suite.

5.1 Introduction

Le langage naturel permet de transmettre des informations complexes et variées, mais, si l'énorme avantage est qu'un humain le connaît déjà et n'aura besoin que d'une adaptation minime afin d'échanger avec un agent artificiel, le suivi d'instructions en langage naturel n'est pas encore résolu. Les instructions peuvent en effet être ambiguës, contenir des dépendances complexes (éventuellement avec l'environnement), et une même instruction peut être exprimée de manières différentes.

Dans le chapitre précédent, section 4.3.6, nous utilisions l'architecture BHNN pour représenter la politique d'un agent qui se déplace dans un labyrinthe et doit choisir quand demander une instruction afin de savoir où aller. L'instruction consistait uniquement en l'action optimale, donnée par un oracle. Nous souhaitons cette fois utiliser des instructions plus complexes. Nous étudions ainsi dans ce chapitre l'interaction en langage naturel avec un agent artificiel. L'agent évolue au sein d'un environnement, et les instructions dé-

pendent souvent de l'environnement : ainsi, l'agent doit relier l'instruction à ses observations de l'environnement afin d'agir.

De nombreux travaux en apprentissage supervisé existent déjà sur la compréhension d'instructions en langage naturel [Mei+16]. Quelques travaux, toujours essentiellement en apprentissage supervisé, existent également sur la génération d'instructions [Dan+17], mais dans un cas comme dans l'autre, le problème principal est souvent le manque de données. Certains travaux pallient à ce problème en générant automatiquement des instructions [Yu+17], mais les instructions sont alors très simples et le langage généré par l'agent est forcément limité. La récupération de données instructionnelles est malheureusement coûteuse, et nous cherchons dans ce chapitre des solutions à ce problème.

La solution que nous proposons est d'utiliser un apprentissage joint locuteur/acteur, inspiré de l'apprentissage dual en traduction [He+16a]. En traduction automatique, on dispose généralement d'un jeu de données qui donne des correspondances entre les langues A et B, mais également de corpus importants dans chacune des langues. L'apprentissage dual permet de tirer parti de ces corpus mono-lingual sans traduction, afin d'améliorer la qualité du modèle de traduction.

Dans le cas qui nous intéresse d'interaction en langage naturel, nous proposons d'apprendre à la fois un **acteur** qui agit dans l'environnement en suivant une instruction en langage naturel (équivalent à une traduction A$\rightarrow$B), et un **locuteur** qui génère une instruction pour décrire une tâche (équivalent à une traduction B$\rightarrow$A). L'interaction entre les deux agents locuteur/acteur peut permettre d'améliorer les résultats de l'un et/ou de l'autre sans nécessiter de corpus supplémentaire, car nous pouvons dans beaucoup de cas générer facilement de nouvelles trajectoires - l'équivalent d'un corpus mono-lingual dans la langue B.

La figure 5.1 décrit l'architecture globale. Les modèles de locuteur et d'acteur sont appris dans un premier temps en apprentissage supervisé grâce à un jeu de données. Dans un second temps, l'acteur permet d'obtenir un retour sur la qualité des instructions générées par le locuteur : le locuteur génère des instructions à partir d'une trajectoire aléatoire, l'acteur suit ces instructions, et les paramètres du locuteur peuvent être améliorés grâce à des techniques d'apprentissage par renforcement selon que l'acteur ait atteint son but ou pas.

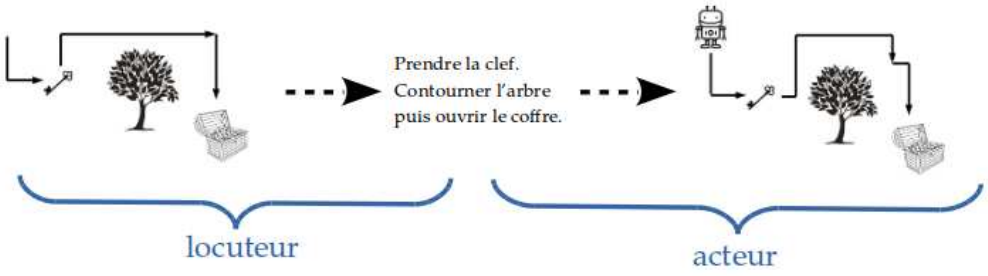


Figure 5.1: Illustration du modèle locuteur/acteur. Le **locuteur** génère une instruction à partir d'une trajectoire ; l'**acteur** choisit quelle action effectuer d'après une instruction reçue (éventuellement générée par le locuteur, comme sur la figure) en interaction avec l'environnement.

L'utilisation simultanée des deux agents permet donc d'évaluer la qualité des instructions générées sans nécessiter l'intervention d'un être humain.

5.2 Contributions et plan

Dans un premier temps, nous décrirons l'environnement que nous utilisons pour tester nos architectures. Nous exposerons ensuite l'architecture de l'acteur tirée de [Mei+16] et nos expériences pour reproduire les résultats de l'article. Nous présenterons ensuite nos contributions : le modèle de locuteur (qui suit la même architecture que l'acteur) et les expériences associées, l'apprentissage joint entre le locuteur et l'acteur, et enfin une architecture hiérarchique afin de travailler sur des séquences d'instructions. Nous finirons par une bibliographie exposant les articles liés.

5.3 Un exemple d'environnement

Nous souhaitons étudier la compréhension et la génération d'instructions d'un agent en interaction avec un environnement. Nous avons donc besoin d'un environnement où l'agent peut effectuer des tâches différentes, et où ces tâches peuvent être décrites en langage naturel ; mais nous avons également besoin d'un jeu de données avec des exemples d'instructions qui décrivent une tâche.

Figure 5.2: Image tirée de [Mac+06]. Exemple de la vue d'un humain à la première personne dans l'un des environnements.

Un tel jeu de données est celui introduit dans [Mac+06]. Nous allons l'utiliser pour toutes nos expériences. [1]

Nous allons dans cette partie détailler l'environnement utilisé et la manière dont les observations sont encodées. L'environnement consiste en trois labyrinthes virtuels, où les sols et les murs ont des couleurs différentes, et où des objets peuvent être posés sur le sol (voir figures 5.2 et 5.3). Des trajectoires ont été tirées entre deux points du labyrinthe, et des humains testeurs ont donné des instructions en langage naturel qui permettent d'aller du point de départ au point d'arrivée. Le jeu de données contient 703 instructions constituées de plusieurs phrases, qui une fois divisées en phrases donnent 3234 instructions plus simples. Les instructions contiennent parfois des erreurs grammaticales et d'orthographe.

Le jeu de données contient également des trajectoires effectuées par plusieurs humains qui ont suivi les instructions afin de les tester. Uniquement 68.4% de ces trajectoires atteignent le but (normalement) décrit par l'instruction : les instructions sont donc peu claires, et lorsqu'elles sont coupées en phrases, 354 des phrases (sur 3234) n'ont été suivi correctement par aucun des testeurs. On peut donc supposer que ces instructions en particulier sont incorrectes car jamais suivies correctement.

1. Ces derniers mois d'autres jeux de données semblables sont sortis, qui pourraient également être utilisés ; voir section 5.9

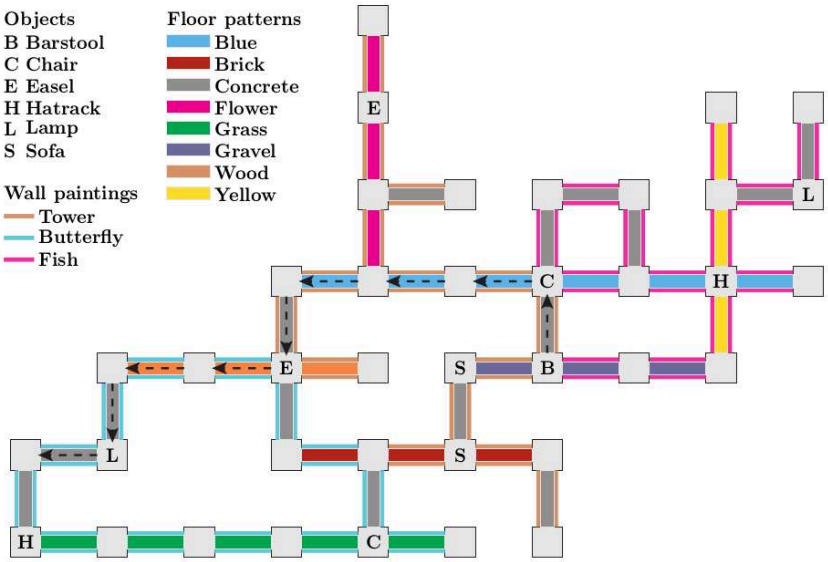


Figure 5.3: Image tirée de [Mei+16]. L'un des labyrinthes utilisé comme environnement. Les couleurs des sols et des murs varient, et les lettres désignent des objets posés aux intersections. Un exemple de trajectoire associée à un paragraphe instructionnel est montré par les flèches pointillées.

nombre d'instructions (paragraphes)	703
nombre d'instructions (phrases)	3234
nombre d'instructions (phrases) qu'au moins un testeur a réussi à suivre	2880

Table 5.1: détails du jeu de données

On peut voir dans le tableau 5.1 les détails sur le jeu de données. La figure 5.4 montre l'histogramme du nombre d'actions par trajectoire, et 5.5 l'histogramme du nombre de mots par phrases. Nous remarquons que beaucoup de trajectoires sont très courtes (319 des trajectoires ne contiennent qu'une action ! et 1517 en contiennent deux, ce qui fait que plus de la moitié du jeu de données est constitué de trajectoires (phrases) d'une ou deux actions...) Les tailles de phrases sont un peu plus équilibrées, même si beaucoup d'entre elles sont tout de même constituées d'uniquement deux mots.

Voici des exemples d'instructions du jeu de données :

— move forward past the chair to the hatrack

— turn left and move straight to the end of the green octagon alley

— turn left

— go to blue

— then pink area

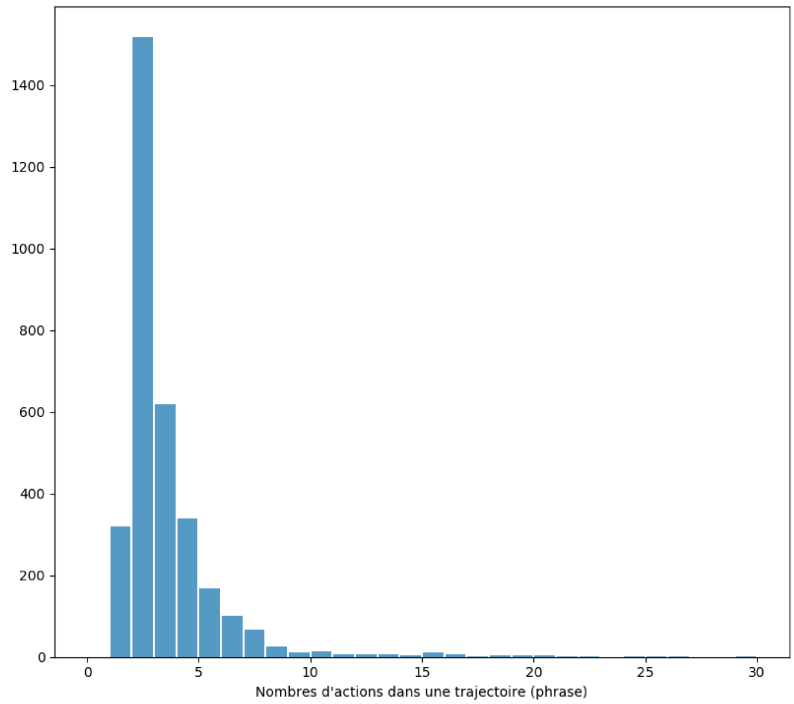


Figure 5.4: Histogramme du nombre d'actions par trajectoire.

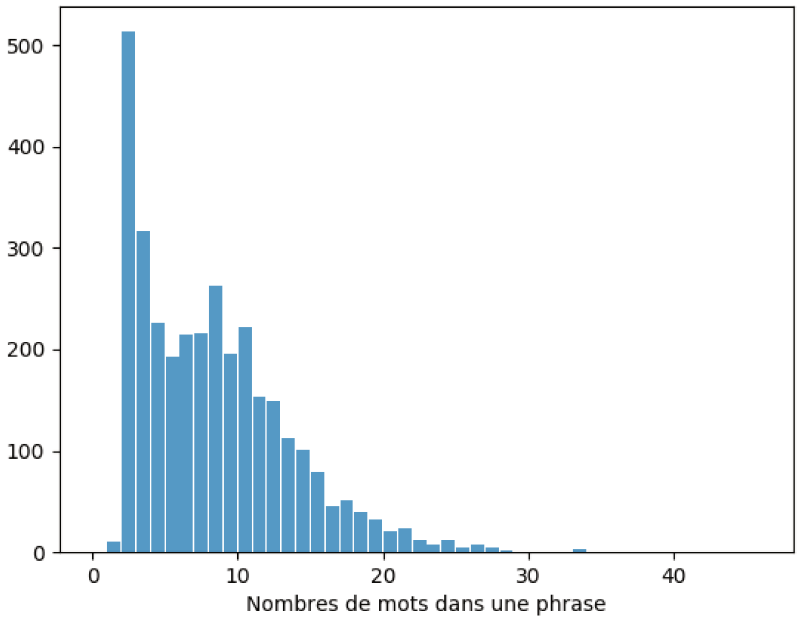


Figure 5.5: Histogramme du nombre de mots par phrase.

— position 7 is in the middle of the grey floored hall with a coat rack or unkownword

— go forward one segment

Nous disposons de trois cartes différentes. Dans les expériences, nous utilisons une validation croisée 3-fold : les données de deux cartes sont utilisées en entraînement, et la troisième carte est utilisée en test. Nous alternons ensuite la carte utilisée en test et nous faisons la moyenne de tous les résultats. Dans toutes les expériences, sauf mention contraire, les paragraphes sont coupés en phrases (ainsi que les trajectoires associées), et les données utilisées sont les phrases et non les paragraphes complets.

Observations de l'environnement : Les données sont encodées comme dans [Mei+16] : l'observation o_t est un vecteur binaire. Il s'agit de la concaténation de vecteurs sacs-de-mots qui contiennent l'information comme quoi un objet, un motif au sol, une peinture au mur, est présent ou pas, pour chaque direction (gauche, devant, droite) . Il contient également l'information comme quoi il y a un mur juste à gauche, devant, à droite de l'agent. Dans le cas de l'acteur, la dernière action effectuée est également ajoutée à l'observation, comme souvent fait en apprentissage par renforcement.

5.4 Modèle locuteur/acteur et notations

Notre but est d'obtenir un agent qui apprend à suivre des instructions en langage naturel (l'**acteur**) ainsi qu'un agent qui apprend à générer des instructions en langage naturel (le **locuteur**). Chaque modèle est d'abord appris séparément en apprentissage supervisé, grâce au jeu de données dont on dispose. Les deux modèles seront ensuite utilisés conjointement. La figure 5.1 illustre l'utilisation jointe des deux modèles.

On note une instruction en langage naturel $(w_1, ...w_n)$, chaque w_i étant un mot de la phrase, représenté par un vecteur *one-hot* (c'est-à-dire un vecteur constitué de zéros et d'un "un" à l'indice qui correspond au mot).

Au temps t, l'action effectuée par l'agent est notée a_t, et son observation de l'environnement est o_t. Une trajectoire est définie par les observations et les actions entre l'état initial $t = 1$ et l'état final $t = T$: $(o_1, a_1, ...a_{T-1}, o_T, a_T)$ (avec a_T l'action "stop").

Les modèles utilisés pour l'acteur et le locuteur sont tirés de [Mei+16], où le modèle est uniquement utilisé pour du suivi d'instructions. Il s'agit d'une architecture encodeur-aligneur-decodeur, couramment utilisée en traduction automatique [Bah+14].

L'acteur A prend en entrée une instruction $(w_1, ...w_n)$, puis, à chaque pas de temps t, reçoit l'observation de l'environnement o_t et génère une action a_t. Le locuteur L prend en entrée la trajectoire $(o_1, a_1, ...o_T, a_T)$ et génère une instruction $(w_1, ...w_n)$ afin que l'instruction donne toutes les informations nécessaires pour aller du point de départ au point d'arrivée.

Encodage des mots : Les mots sont encodés sous forme de vecteur one-hot puis une représentation des mots est apprise conjointement au modèle. On ne corrige pas les erreurs d'orthographes et grammaticales des instructions, mais les mots utilisés moins de quatre fois dans tout le jeu de données sont remplacé par un token "unknown". Le vocabulaire contient en tout 271 mots, c'est pourquoi nous n'avons pas utilisé des librairies avec des représentations de mots déjà apprises : nous pouvons travailler avec des vecteurs de tailles bien inférieures.

5.5 Apprentissage supervisé de l'acteur

Le but de l'acteur est de suivre une instruction en langage naturel selon son environnement. L'agent doit relier son observation de l'environnement à l'instruction : ainsi, l'instruction "Prendre la première à gauche" ne donnera pas la même séquence d'actions selon où se trouve le premier chemin allant vers la gauche. Les paramètres du modèle d'acteur sont appris en apprentissage supervisé, grâce au jeu de données décrit précédemment en section 5.3.

Le modèle utilisé par l'acteur est une adaptation directe de [Mei+16]. Il s'agit d'une architecture encodeur-aligneur-decodeur, que nous allons détailler dans la suite. Nous présenterons ensuite les résultats obtenus : nous cherchons à reproduire les résultats obtenus des articles précédents, mais également à mieux comprendre ces résultats en testant différents capteurs.

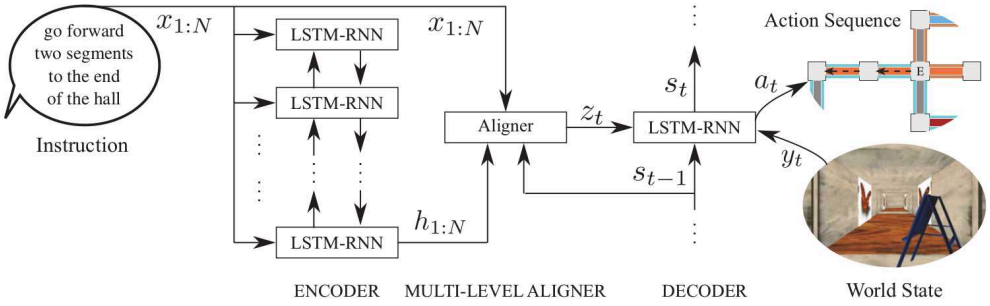


Figure 5.6: Image tirée de [Mei+16]. Architecture de l'acteur.

5.5.1 Architecture

L'agent génère à chaque pas de temps t une action a_t ; la trajectoire se termine quand l'action STOP est émise. Sommairement, l'architecture est constituée d'un encodeur-aligneur-décodeur :

— l'encodeur utilise l'instruction en langage naturel $(w_1, ...w_n)$ et calcule sa représentation $(h_1, ...h_n)$ grâce à un réseau récurrent bi-directionel.

— l'aligneur récupère les mots $(w_1, ...w_n)$, la représentation de l'instruction donnée par l'encodeur $(h_1, ...h_n)$ et l'état courant du décodeur s_{t-1} (qui représente le dernier état de l'agent), et calcule un vecteur z_t qui encode les instructions pour un temps t donné comme une somme modérée des vecteurs w_j et h_j.

— le décodeur est un réseau récurrent neuronal d'état caché s_t qui prend en entrée le vecteur z_t de l'aligneur et l'état du monde y_t, et renvoie la distribution de probabilité de la prochaine action.

L'architecture est détaillée en figure 5.6. Plus précisémment :

Encodeur

— **Entrée :** $(w_1, ...w_n)$ une instruction de n mots ; chaque w_i est un vecteur binaire qui représente le mot i de la phrase

— **Sortie :** $(h_1, ...h_n)$ l'encodage des n mots grâce à un Bi-LSTM

Aligneur

— **Entrée au temps t :** l'instruction $(w_1, ...w_n)$, la sortie de l'encodeur $(h_1, ...h_n)$, l'état précédent du décodeur s_{t-1}

— **Sortie au temps t :** z_t le vecteur de contexte qui représente l'instruction sous forme d'une somme pondérée des vecteurs des mots $(w_1, ...w_n)$ et de leur encodage $(h_1, ...h_n)$:

$$z_t = \sum_i \alpha_{ti} \begin{pmatrix} x_i \\ h_i \end{pmatrix} \tag{5.1}$$

avec les poids α_{ti} :

$$\alpha_{ti} = exp(\beta_{ti}) / \sum_j exp(\beta_{tj}) \tag{5.2}$$

$$\beta_{tj} = lin(concat(s_{t-1}, x_j, h_j)) \tag{5.3}$$

où lin représente un réseau neuronal à une couche (i.e. perceptron)

Décodeur

— **Entrée au temps t :** z_t la sortie de l'aligneur, l'état précédent du décodeur s_{t-1}, l'observation de l'environnement par l'agent o_t

— **Sortie au temps t :** s_t le nouvel état du décodeur après une cellule LSTM qui prend en entrée la concaténation de z_t et o_t, et les probabilités de chaque action p_a :

$$a_t = softmax(f(concat(z_t, o_t, s_t))) \tag{5.4}$$

où f est un réseau de neurones linéaire à deux couches.

5.5.2 Apprentissage

Les données utilisées pour l'apprentissage sont des triplets contenant une instruction, et la trajectoire associée avec d'une part la séquence d'actions utilisées et d'autre part les observations récupérées à chaque pas de temps : une donnée est donc constituée de *(instructions, observations, actions)*.

L'apprentissage est supervisé : l'agent cherche à reproduire la trajectoire associée à l'instruction. Le modèle est donc entraîné afin de prédire la séquence d'actions $(a_1, ...a_T)$ étant donné une instruction $(w_1, ...w_n)$ et une séquence d'observations $(o_1, ...o_T)$ de l'ensemble d'entraînement.

Nous utilisons un coût **negative log-likehood** pour l'action à chaque pas de temps t :

$$L_t = -\log P(a_t^*|o_t, w_1, ...w_n) \tag{5.5}$$

où a_t^* est l'action utilisée au pas de temps t dans les données d'entraînement.

Le modèle étant différentiable, les paramètres peuvent être appris par back-propagation.

5.5.3 Expériences

Métrique d'évaluation : L'évaluation de l'acteur se fait sur la résolution ou non de la tâche. La tâche est résolue si l'agent arrive à la bonne position finale décrite par l'instruction, dans la bonne orientation. Le résultat donné est la moyenne de la précision sur toutes les données de tests.

Description des expériences : L'architecture de notre acteur est autant que possible identique à celle de [Mei+16]. La différence la plus importante est que nous n'utilisons pas d'algorithme de recherche en faisceau (*beam search*) qui dans [Mei+16] maintient une liste des 10 meilleures hypothèses lors de la phase de test pour sélectionner la plus probable. De plus, afin de réduire le bruit, 10 modèles initialisés aléatoirement y sont utilisés, et l'action choisie utilise la moyenne des probabilités de ces 10 modèles. Dans nos expériences, nous n'utilisons aucune des deux méthodes. Nous les avons testées sur le modèle d'acteur mais la recherche en faisceau n'a pas amélioré les résultats et l'utilisation de 10 modèles ralenti énormément les tests, pour une amélioration minime des résultats.

Nous testons également quelques variantes concernant les observations et les données utilisées.

De base, l'observation de l'environnement concerne, comme on l'a vu en 5.3, les couleurs du couloir, les motifs des murs, les objets visibles devant, à droite et à gauche de l'agent, ainsi que les murs adjacents. Nous pouvons aussi tester le modèle sans aucune observation (o_t contient alors uniquement l'action a_t), ou avec une observation qui contient uniquement les informations sur les

murs adjacents, afin de mieux comprendre à quel point le modèle utilise les informations de l'environnement.

Enfin, étant donné que les testeurs humains n'ont jamais atteint le but pour 11% des instructions, nous testons également les modèles sur les données notées "*instructions correctes*", qui contiennent uniquement les instructions où au moins un des humains testeurs (pas forcément tous) ont atteint le but en les suivant. Cela nous permet ainsi d'éliminer les instructions trop incorrectes qui risquent de bruiter les résultats.

Résultats : Nous pouvons voir les résultats dans le tableau 5.2. Le modèle "basique" est l'architecture utilisée dans [Mei+16], sans la recherche en faisceau ni l'utilisation de la moyenne de 10 modèles différents pour choisir l'action en test. Les résultats lorsque nous reproduisons ce modèle (ligne 4) sont de 0.65, ce qui reste légèrement inférieur à ceux donnés dans l'article (0.70). L'utilisation de 10 modèles n'améliore que très peu les résultats, avec des résultats de 0.66 (dernière ligne) tout en ralentissant de manière conséquente les expériences. La recherche en faisceau testée n'a pas du tout amélioré les résultats. Plusieurs explications sont possibles :

— de légères différences d'architecture

— un beam-search spécifique utilisé dans [Mei+16], en particulier la fonction d'évaluation de la qualité de la trajectoire, car nous en avons testé plusieurs sans résultat

Afin de mieux comprendre comment le modèle utilise les observations o_t, nous testons d'autres observations de l'environnement :

— uniquement les informations sur la présence ou non de murs adjacents, les résultats sont alors de 0.65

— les informations sur la présence ou non de murs adjacents et la dernière action utilisée, les résultats sont également de 0.65

— aucune observation de l'environnement, les résultats sont alors de 0.58

On remarque dans un premier temps que supprimer toutes les informations sur les couleurs des murs, des sols, et sur les objets présents, donne les mêmes résultats que l'observation complète. En utilisant uniquement les informations sur la présence ou pas de murs adjacents (et donc sur s'il peut avancer dans telle direction ou pas), l'agent obtient des performances équivalentes. L'acteur n'a donc probablement pas appris à relier les instructions du style "*follow this hall until it's with the yellow tiled hall*" à la couleur du sol ou des murs.

détails du modèle	précision
baseline : humains (pour les paragraphes)	0.684
MBW	0.70
Fried Andreas Klein	0.68
basique	0.65
obs = murs visibles	0.65
obs = murs visibles + dernière action	0.65
pas d'obs	0.58
basique, données = instructions correctes	0.68
basique, moyenne de 10 modèles	0.66

Table 5.2: Résultat pour l'acteur (apprentissage supervisé). MBW est le modèle de [Mei+16] que nous reproduisons ; FAK est le modèle de [Fri+17] qui reproduit MBW.

L'agent peut même n'utiliser aucune information provenant de l'environnement (il suit donc l'instruction "en aveugle") tout en obtenant des résultats convenables. Cela s'explique par la présence importante d'instructions simples (du style "tourner à droite") qui ne sont reliées qu'à un faible nombre d'actions (entre une et trois).

L'agent apprend donc facilement à suivre les instructions simples, sans avoir besoin de les relier à des informations complexes de l'environnement ; mais peine dès qu'il doit suivre des instructions plus détaillées.

Comme certaines instructions suivies par des testeurs humains paraissent incorrectes, nous pouvons nous demander à quel point elles influent sur l'apprentissage. Nous testons donc l'acteur sur les données où au moins un des humains qui ont suivi les instructions ont atteint le but, afin d'éliminer les instructions trop incorrectes. Dans ce cas, le modèle donne un score de 0.68 au lieu de 0.65 : l'influence de ces instructions incorrectes est donc minime.

5.5.4 Conclusion

Notre reproduction de l'architecture donne des résultats proches de ceux de la littérature, bien que légèrement inférieurs. Il est plus problématique de se rendre compte que le modèle n'utilise que très peu les informations pertinentes de l'environnement : les objets, les couleurs... ne sont finalement pas utilisés par le modèle. Les données utilisées sont malheureusement mal équilibrées : nous y trouvons beaucoup de données simples (nombre de mots

et d'actions faibles) et peu de données plus complexes, ce qui ne permet pas un apprentissage correct des paramètres des réseaux de neurones pour suivre les instructions complexes.

5.6 Apprentissage supervisé du locuteur

Le locuteur résout la tâche inverse de l'acteur : il génère une instruction pour indiquer comment arriver à un but. Nous simplifions la tâche en fournissant directement au locuteur une trajectoire possible du point de départ au point d'arrivée : le locuteur n'a donc pas besoin de trouver comment arriver au but. L'instruction générée doit contenir toutes les informations nécessaire afin de suivre cette trajectoire.

Nous allons dans un premier temps décrire l'architecture utilisée et l'apprentissage, puis décrire nos expériences.

5.6.1 Architecture

Le locuteur utilise presque la même architecture que l'acteur, avec de légères différences :

— il prend en entrée la représentation d'une trajectoire dans l'environnement : une séquence $(o_1, a_1, ...o_T, a_T)$ contenant pour chaque pas de temps t l'observation du monde o_t et l'action a_t effectuée

— il génère une séquence de mots $(w_1, ...w_n)$

— à l'itération i, le décodeur utilise le mot précédent w_{i-1} (à la place de o_{t-1} pour l'acteur)

Comme l'acteur, le locuteur génère une phrase qui se finit quand le mot STOP est émis.

5.6.2 Apprentissage

Les paramètres du locuteur sont appris avec des techniques d'apprentissage supervisé de la même manière que ceux de l'acteur. Les données d'entraînement sont les mêmes, de la forme *(instructions, observations, actions)*.

Le modèle doit prédire l'instruction $(w_1, ...w_n)$ étant donné une trajectoire $(o_1, a_1, ...o_T, a_T)$. On utilise un coût **negative log-likehood** :

$$L = -\log P(w_1^*, ...w_n^* | o_1, a_1, ...o_T, a_T) \tag{5.6}$$

où $(w_1^*, ...w_n^*)$ est l'instruction des données d'entraînement.

Le modèle étant différentiable, les paramètres peuvent être appris par back-propagation.

5.6.3 Expériences

Métrique d'évaluation : On utilise un score bleu pour évaluer le locuteur : on évalue l'instruction générée par l'agent par rapport à l'instruction donnée par un humain dans l'ensemble de test. Le résultat dans les tableaux est la moyenne de ces scores bleus pour toutes les données de test.

On teste ici le modèle de locuteur appris seul, en apprentissage supervisé. Les résultats sont disponibles dans le tableau 5.3.

Comme dans le cas de l'acteur, le modèle est testé avec les observations complètes mais également avec les observations qui contiennent uniquement l'information sur les murs adjacents, ou pas d'observation du tout. Le modèle a par contre toujours accès aux actions de la trajectoires. Ces deux derniers cas donnent un score bleu plus élevé (0.4) que dans le cas d'une observation complète (0.385). Le modèle, encore une fois, ne parvient pas à relier les informations disponibles dans l'observation, aux mots qui doivent être générés. Pire, le locuteur a même plus de difficultés à générer des instructions avec une observation plus complète.

En 5.5, nous avons un exemple de phrases générées (sur l'ensemble de test), comparées aux phrases générées par un être humain. On voit que les instructions simples peuvent être reproduites, mais pas les plus complexes.

5.6.4 Conclusion

Le problème du modèle de locuteur est le même que celui du modèle d'acteur : il apprend à générer les instructions simples qui ne dépendent pas

détails du modèle	score bleu
basique	0.385
obs = murs visibles	0.4
pas d'obs	0.4
basique, données = instructions correctes	0.39

Table 5.3: résultat pour le locuteur (apprentissage supervisé)

des informations de l'environnement. Les instructions plus complexes, plus longues ou dont les points d'intérêt dépendent de l'environnement, ne sont pas générées correctement à cause de la petite taille du corpus. Les réseaux de neurones nécessitent davantage de données pour obtenir des résultats corrects.

5.7 Apprentissage joint du locuteur et de l'acteur

Nous cherchons ensuite à améliorer les résultats en apprenant conjointement les modèles de l'acteur et du locuteur.

En effet, le problème principal pour apprendre à suivre ou à générer des instructions en langage naturel est d'avoir un jeu de donnée assez conséquent. Récupérer les instructions en langage naturel est coûteux pour les utiliser en apprentissage supervisé, mais plus encore pour les techniques d'apprentissage par renforcement où l'interaction doit être constante.

L'idée est donc d'apprendre conjointement un acteur et un locuteur : chaque modèle sera d'abord appris en apprentissage supervisé comme dans les parties précédentes, puis sera amélioré grâce à l'autre modèle, qui permettra d'obtenir un retour sur la qualité de la trajectoire ou de l'instruction générée sans devoir faire appel à un être humain.

Pour ajouter des données artificiellement, il est difficile de générer des instructions complexes et proches de celles d'un être humain. Nous pouvons par contre générer des trajectoires dans notre environnement. Il suffit de choisir un point de départ et un point d'arrivée aléatoires, puis d'utiliser un algorithme (de Dijkstra par exemple) pour trouver le chemin le plus court entre les deux points. Il serait également possible de générer une trajectoire

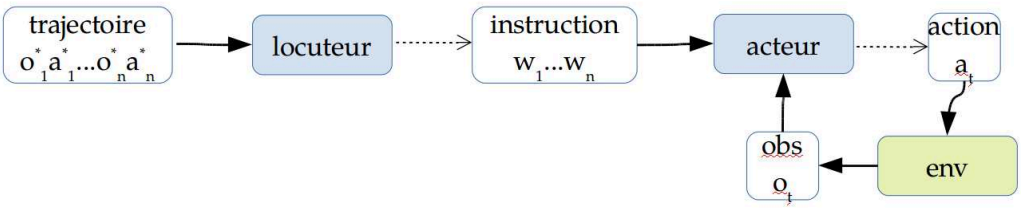


Figure 5.7: Illustration du modèle locuteur/acteur. Le locuteur génère une instruction à partir d'une trajectoire ; l'acteur choisit quelles actions effectuer dans l'environnement selon l'instruction qu'il reçoit.

aléatoire qui suit un chemin possible. Nous pouvons observer sur la figure 5.7 la structure jointe du locuteur et de l'acteur : le locuteur pré-appris génère une instruction en langage naturel à partir de cette trajectoire, et l'acteur utilise cette instruction pour retrouver le but. Le fait que l'acteur parvienne ou non au but permet d'obtenir un retour sur la qualité des décisions.

L'apprentissage se déroule donc en plusieurs étapes :

— les modèles d'acteur et de locuteur sont appris grâce à un jeu de données instructions↔trajectoires comme dans les sections 5.5 et 5.6.1

— des tâches sont générées aléatoirement dans l'environnement, et des trajectoires sont trouvées pour résoudre ces tâches

— le locuteur génère des instructions selon ces trajectoires

— l'acteur suit ces instructions jusqu'à arriver à ce qu'il pense être le but

— le locuteur et/ou l'acteur sont améliorés grâce aux résultats

5.7.1 Apprentissage

Une fois que l'acteur a suivi l'instruction générée par le locuteur, nous souhaitons améliorer les modèles grâce au résultat de ses actions. Nous étudions dans cette partie les différentes méthodes d'apprentissage possibles.

L'apprentissage de l'acteur peut se faire en supervisé (on compare la trajectoire obtenue avec la trajectoire qui était désirée) ou en renforcement (est-ce que l'agent arrive au but ? ou toute autre récompense fournie par l'environnement). Les techniques sont donc identiques à l'apprentissage seul de l'acteur, et nous pouvons espérer que l'ajout de nouvelles trajectoires améliore les résultats. Malheureusement, nos expériences n'ont jamais permis d'améliorer

les résultats de l'acteur, sans doute parce que les instructions supplémentaires données par le modèle de locuteur n'étaient pas assez correctes (et que les bonnes instructions sont celles plus simples, que l'acteur arrive déjà à suivre).

Le locuteur ne peut par contre être appris que par des algorithmes d'apprentissage par renforcement à cause de la génération stochastique de l'instruction, qui empêche la backpropagation habituelle.

Apprentissage par renforcement du locuteur

Pour évaluer la qualité de l'instruction générée par le locuteur, nous utilisons une récompense fournie par l'acteur après qu'il ait suivi cette instruction. La récompense peut être un coût entre les actions choisies par l'acteur et les actions de la trajectoire utilisée par le locuteur, ou la récompense fournie par l'environnement. Nous avons testé les deux possibilités ; dans le cas de la récompense fournie par l'environnement, elle est $R = 1$ si l'acteur a atteint son but, $R = 0$ sinon.

Ainsi, si l'on reprend les étapes de l'apprentissage joint :

— des tâches sont générées aléatoirement dans l'environnement, et des trajectoires sont trouvées pour résoudre ces tâches
— le locuteur génère des instructions selon ces trajectoires
— l'acteur (connu) suit ces instructions jusqu'à arriver à ce qu'il pense être le but
— le locuteur reçoit la récompense de l'acteur

Nous utilisons ensuite cette récompense pour apprendre le modèle du locuteur avec un algorithme d'apprentissage par renforcement. Il est possible d'utiliser n'importe quel algorithme adapté aux réseaux neuronaux récurrents. Dans notre cas, nous utilisons REINFORCE (voir section 2.3.1).

Si nous utilisons uniquement la récompense fournie par l'acteur afin de modifier le locuteur, les instructions générées par le locuteur risquent cependant de ne plus être comprises par un humain et de trop s'éloigner des instructions du jeu de données. C'est pourquoi en pratique, on continue également

détails du modèle	λ	score bleu	précision de l'acteur
baseline (locuteur non amélioré)	-	0.37	0.56
récompense=coût	0	0.39	0.59
récompense=coût	0.25	0.40	0.64
récompense=coût	0.5	0.37	0.66
récompense=env	0	0.39	0.59
récompense=env	0.25	0.395	0.62
récompense=env	0.5	0.395	0.66
récompense=env	0.75	0.39	0.67

Table 5.4: Amélioration du locuteur avec l'apprentissage dual. Le score de l'acteur est calculé lorsqu'il suit l'instruction générée par le locuteur. "récompense=coût" signifie que la récompense est le coût entre les actions choisies par l'acteur et les actions de la trajectoire décrite par le locuteur ; "récompense=env" est la récompense venant de l'environnement (1 si le but est atteint, 0 sinon)

l'apprentissage supervisé sur le jeu de données. Le paramètre λ permet de contrôler le compromis entre l'apprentissage supervisé et l'apprentissage par renforcement. Quand $\lambda = 0$, l'apprentissage est uniquement supervisé ; plus λ s'approche de 1 et plus l'apprentissage par renforcement (et donc le retour de l'acteur) est pris en compte.

5.7.2 Expériences

Métrique d'évaluation : Comme lors de l'apprentissage supervisé du locuteur, nous utilisons le score bleu pour évaluer la qualité des instructions générées sur le jeu de données. Lors de l'utilisation jointe d'un acteur et d'un locuteur, le locuteur peut également être évalué sur la capacité de l'acteur à atteindre le but lorsqu'il suit l'instruction générée. Le score est alors la précision de l'acteur lorsqu'il suit les instructions générées par le locuteur sur 200 trajectoires aléatoires.

Résultats : On peut voir dans le tableau 5.4 les résultats de l'apprentissage dual. Deux récompenses ont été utilisées : le coût entre la trajectoire correcte et celle suivie par l'acteur, et la récompense de l'environnement.

Dans les deux cas, nous observons que le score bleu ne varie pas beaucoup, et est légèrement amélioré par rapport à la baseline, même lorsque seul l'apprentissage supervisé est utilisé ($\lambda = 0$). Nous pouvons supposer supposer que reprendre l'apprentissage permet d'améliorer légèrement le score.

Plus intéressant, la précision de l'acteur augmente lorsque la valeur de λ augmente et donc que le retour de l'acteur prend plus d'importance. Si l'apprentissage par renforcement ne permet donc pas d'améliorer le score bleu, il améliore bien la compréhension par l'acteur de l'instruction générée. Les résultats sont légèrement meilleurs lorsqu'on utilise la récompense fournie par l'environnement, sans que les résultats soient très concluants.

On peut voir en tableau 5.5 des exemples d'instructions avant et après l'apprentissage joint. L'apprentissage joint permet parfois d'améliorer légèrement les résultats, mais les résultats sont encore loin de l'être humain, faute à un nombre suffisant de données.

Instruction réelle	générée (apprentissage supervisé du locuteur)	générée (apprentissage joint du locuteur)
go forward once	the side alley to your right should be blue	walk forward once
p 4 is at the end of the alley with the green octagon floor	go forward the segments the	walk forward twice
turn left	turn left	turn left
then pink area	go the the the the the the the the the the the	go the the path the wall turn right

Table 5.5: Exemple d'instructions (réelles, et générées par le locuteur)

5.7.3 Conclusion

Dans cette section, nous avons cherché à améliorer la génération d'instructions grâce à un apprentissage dual du locuteur et de l'acteur. Si les instructions sont effectivement plus facilement compréhensibles par l'acteur après cette amélioration, le score bleu des instructions n'est malheureusement pas amélioré et les phrases générées, une fois examinées, montrent que les améliorations ne sont pas notables. L'un des problèmes est que les modèles d'acteurs et de locuteurs ont tout deux appris (relativement) correctement sur les instructions et les trajectoires simples, mais qu'aucun des deux n'a de résultats convenables sur les données plus complexes. Ainsi, si l'acteur ne sait pas suivre une phrase complexe, le locuteur ne pourra recevoir aucun bon retour sur la phrase qu'il génère et ne pourra donc pas améliorer ces générations. Il faudrait également qu'un humain évalue la qualité des instructions, car elle n'est pas forcément directement liée au score bleu.

Une solution pourrait être d'améliorer progressivement l'acteur *et* le locuteur grâce à l'apprentissage dual. Le premier soucis est que l'amélioration de l'acteur par l'apprentissage dual n'a pas fonctionné, sans doute car le locuteur ne fournit des instructions convenables que pour les trajectoires simples - que l'acteur sait déjà suivre. Le second problème est que dans ce jeu de données, les instructions simples ("tourner à droite", "avancer de deux cases") sont très différentes des instructions plus difficiles à comprendre ("aller jusqu'au canapé"), et que comprendre les instructions simples ne permet pas de comprendre les instructions qui utilisent davantage l'environnement.

5.8 Extension : architecture hiérarchique

Le jeu de données que nous utilisons (voir section 5.3) associe au départ des instructions longues, sous forme de paragraphes, à des trajectoires. Pour simplifier l'apprentissage, les paragraphes ont été séparé en phrases (associées aux sous-trajectoires correspondantes), et ce sont ces phrases qui ont été utilisées dans les sections précédentes.

Nous souhaitons ici expérimenter directement sur les paragraphes. Dans un second temps, le but serait d'obtenir une architecture en lien avec le *Budgeted Hierarchical Neural Network* introduit dans le chapitre précédent

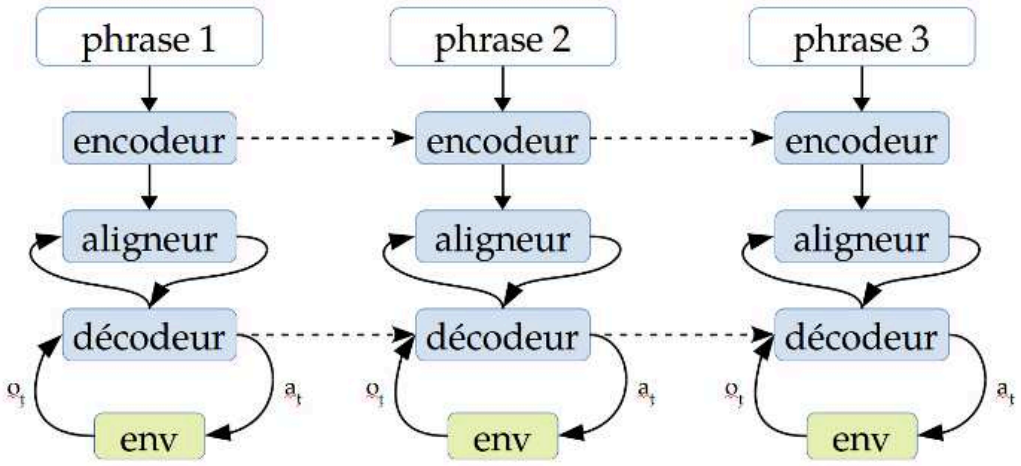


Figure 5.8: Illustration de l'architecture hiérarchique. Les flèches en pointillés montrent les dépendances possibles de mémoire, entre l'encodeur et/ou le décodeur, entre les différentes phrases d'une même instruction. L'acteur passe à la phrase suivante lorsque l'action "STOP" est émise.

(4), avec l'acteur qui peut demander des instructions quand il en a besoin uniquement (l'instruction correspond alors à l'observation haut-niveau du modèle BHNN).

Dans ce but, nous allons tester plusieurs architectures pour que l'acteur parvienne à suivre des paragraphes d'instructions (constitués de plusieurs phrases).

Architecture basique : Nous considérons simplement le modèle présenté en section 5.5 : l'acteur passe à l'instruction suivante lorsque l'action "STOP" est émise, et l'agent n'a pas la mémoire de l'instruction précédente, ni de ses actions effectuées.

Ajout de mémoire dans l'architecture basique : Nous souhaitons cette fois que l'agent ait une mémoire du passé. Dans la même architecture que précedemment, une récurrence est ajoutée au niveau de l'encodeur et/ou du decodeur en initialisant leur état initial par le dernier état (de l'encodeur ou du décodeur) de la phrase précédente du paragraphe. Une illustration des dépendances est visible en figure 5.8.

détails du modèle	accuracy
baseline : humains (pour les paragraphes)	0.684
MBW	0.261
Fried Andreas Klein	0.248

Table 5.6: Baselines pour l'acteur sur les paragraphes. MBW est le modèle de [Mei+16] que nous reproduisons ; FAK est le modèle de [Fri+17] qui reproduit MBW.

5.8.1 Apprentissage

L'apprentissage se fait de la même manière que pour l'acteur (voir section 5.5.2) par apprentissage supervisé. Il est également possible d'utiliser des techniques d'apprentissage par renforcement en plus des techniques supervisées, mais dans nos tests elles n'ont pas permis d'améliorer les résultats.

5.8.2 Expériences

Les résultats des testeurs humains et des modèles des articles antérieurs sont visibles dans la table 5.6. Nous pouvons déjà remarquer que les modèles ont des scores bien inférieurs (environ 25% de réussites) aux scores sur les phrases (qui étaient de presque 70%). En effet, les erreurs se cumulent et rendent l'arrivée au but final bien plus difficile. Nous remarquons également que les humains n'atteignent un score "que" de 68%, signifiant que beaucoup d'instructions sont peu claires.

Afin de mieux comparer les différentes architectures possibles sur les paragraphes, nous utilisons dans nos expériences les données dites "correctes", qui contiennent uniquement les instructions qu'au moins un des testeurs humains est parvenu à suivre correctement. Nous testons les différentes architectures hiérarchiques possibles, avec une conservation ou pas de la mémoire entre les différentes phrases d'une même instruction, pour l'encodeur et/ou le décodeur. Les résultats sont visibles dans la table 5.7. Si on compare les deux premières lignes aux deux dernières, nous remarquons que la mémoire sur l'encodeur n'est pas utile (perte de 1% de précision dans les deux cas), par contre la mémoire sur le décodeur permet d'améliorer les résultats en conservant une mémoire des observations passées (augmentation de 3% de précision dans les deux cas). Dans le cas des données utilisées, ces résultats paraissent normaux : les phrases des instructions sont indépendantes entre

mémoire sur l'encodeur	mémoire sur le décodeur	précision
F	F	0.28
F	T	0.31
T	F	0.27
T	T	0.30

Table 5.7: Résultats pour les différentes architectures d'acteur sur les paragraphes. Utilisation des données "correctes".

elles et ne nécessitent pas la phrase précédente pour être comprise (et la mémoire sur l'encodeur n'a donc pas de raison d'être utile). Par contre, la mémoire sur le décodeur permet à l'agent de garder la mémoire de son environnement et de ses actions passées, qui peuvent influer sur ses actions futures. Les différences de précision sont cependant faibles car l'agent peut explorer à nouveau autour de lui s'il a besoin de plus d'informations et le temps mis à résoudre la tâche n'est ici pas pénalisé.

5.8.3 Conclusion

Dans le cas de notre jeu de données, l'architecture hiérarchique avec mémoire sur l'encodeur ou le décodeur n'améliore que peu les résultats. Conserver une mémoire au niveau de l'encodeur permet d'augmenter légèrement la précision, mais il faudrait faire les mêmes expériences dans un environnement où les actions choisies dépendent davantage du passé pour que les résultats soient plus concluants.

5.9 Bibliographie

Nous allons présenter ici les travaux déjà existants sur la traduction duale (qui même s'il s'agit de traduction, ressemble fortement au problème acteur-locuteur), sur le suivi puis la génération d'instructions, et enfin sur l'apprentissage joint de suivi et génération d'instructions en interaction avec un environnement.

5.9.1 Traduction duale

Le modèle joint acteur/locuteur de la section 5.7 est fortement inspiré de travaux en traduction automatique. En effet, en traduction, on dispose généralement d'un jeu de données duales peu important (des traductions entre les langages A et B), alors que les données monolinguales, non supervisées, pour chaque langage A et B séparément, sont disponibles en quantités très importantes. Cette remarque a incité les travaux de [Sen+15] sur ce qui est appelée *back-translation* : il s'agit de construire des données d'entraînement additionnelles en utilisant les données monolingual. Une phrase du langage A, dont nous ne connaissons pas la traduction en langage B, sera traduite par le modèle de traduction A $\rightarrow$ B. Elle sera ensuite ajoutée aux données utilisées pour apprendre le modèle B $\rightarrow$ A. L'article [He+16a] utilise la même idée (toujours pour de la traduction automatique) en utilisant des techniques d'apprentissage par renforcement pour améliorer les modèles. On suppose qu'on a deux modèles de traductions, peu performants, A $\rightarrow$ B et B $\rightarrow$ A. On dispose également de deux modèles (bien entraînés) L_A et L_B qui renvoient la confiance qu'ils ont qu'une phrase soit correcte dans le langage A et dans le langage B (ils peuvent être entraînés sur les données monolinguales, nombreuses, des langages). Ces confiances sur la qualité du langage généré par les modèles de traduction sont utilisées comme récompenses afin de les améliorer. [Che+16] utilise également des données monolingual afin d'améliorer les résultats de traduction. Cette fois, un auto-encodeur est utilisé afin de passer du langage A, au langage B, au langage A (et respectivement B $\rightarrow$ A $\rightarrow$ B). Les données monolinguales peuvent être utilisées pour l'apprentissage (l'encodeur-decodeur doit être capable de retrouver la même phrase), ainsi que les données traduites (l'encodeur et le décodeur doivent apprendre à traduire les données A $\rightarrow$ B et B $\rightarrow$ A).

5.9.2 Suivi d'instructions

Comme dans notre cas, beaucoup de modèles sont évalués dans des labyrinthes où les lieux peuvent être reconnaissables par des objets, des couleurs... et où l'agent doit suivre une instruction pour atteindre un but. [Mac+06] introduit l'environnement et le jeu de données que nous utilisons, réutilisé par [Mei+16] pour tester le modèle encodeur-aligneur-décodeur que nous avons utilisé et décrit dans la partie 5.5. [VJ10] introduit un jeu de données pour la navigation sur des cartes (type carte au trésor avec des images à

différents endroits), mais les instructions sont au départ orales, et la navigation plus difficile à simuler, ce qui rend le jeu de données plus difficile à utiliser. Le corpus n'est également constitué que de 128 dialogues, trop peu pour qu'il soit utilisable en apprentissage profond. [AK15] décrit un modèle pour interpréter des instructions dans des environnements interactifs (lecture de carte, navigation dans un labyrinthe, résolution de puzzle) en utilisant d'abord un parseur sémantique pour interpréter le langage.

D'autres articles [Nar+15 ; He+16b] cherchent à résoudre des jeux textuels (où l'environnement est décrit par un texte, et le joueur doit écrire au clavier entre un et trois mots afin d'effectuer une action). Cependant, les actions possibles dans un état donné sont généralement très limitées, ce qui facilite l'apprentissage.

Les tâches de *questions answering* [Kum+16] concernent également la compréhension (et la génération limitée) de langage. Les tâches de *visual question answering* [Wu+17] sont plus proches de celles qui nous intéressent, avec une image qui décrit l'environnement, mais habituellement l'environnement n'est pas interactif. Dans [Gor+18], cependant, les auteurs introduisent la tâche *Interactive Question Answering* (IQA) avec un nouveau jeu de données. IQA pose une question à l'agent, du genre "*Are there any apples in the fridge ?*" et un point de départ, et l'agent doit naviguer dans l'environnement et intéragir avec lui (par exemple en ouvrant le frigo) afin de trouver la réponse, avant de répondre. Le modèle proposé, HIMN (*Hierarchical Interactive Memory Network*) est une architecture avec des réseaux de neurones utilisant une mémoire de ce qu'a vu l'agent et plusieurs niveaux d'abstractions temporelles afin d'explorer l'environnement, de récupérer les informations nécessaires, et pour répondre à la question.

Afin de palier au manque de données, le modèle de [DL16] couple des techniques d'apprentissage profond sequence-to-sequence à l'apprentissage symbolique afin d'obtenir des formes logiques (les actions) à partir d'une instruction et des connaissances sur l'environnement. D'autres créent artificiellement des données : dans [Yu+17], les instructions ("*please go to the apple*", "*What is the red object ?*") sont générées par un ensemble de règles selon les caractéristiques de l'environnement. L'architecture utilisée par l'agent est un ensemble de plusieurs modules : modules de langage, de perception, d'action et de reconnaissance. L'apprentissage se fait ici purement en renforcement (sans apprentissage supervisé sur un jeu de données) mais se fait

sur des tâches de plus en plus complexes (avec toujours des instructions générées). D'autres modèles, surtout en robotique [Hem+15], utilisent un pré-processing du langage afin d'annoter les phrases en langage naturel et d'en extraire plus facilement les informations pertinentes.

Finalement, ces dernières années, plusieurs nouveaux jeux de données concernant l'interprétation d'instructions dans un environnement interactif ont été publiés. [Her+17] utilise l'environnement en trois dimensions de [Bea+16] et construit des instructions artificielles pour qu'un agent apprenne ensuite à les suivre, mais les instructions restent très simples ("*pick the X*" où *X* désigne le nom d'un objet). Il n'est donc pas généralisable à des tâches plus complexes. [And+18] présente le simulateur Matterport3D qui reproduit un bâtiment et ses différents étages. Un agent peut se déplacer dans ce bâtiment, et récupère comme observation une image (photo) de l'environnement. Un jeu de données d'instructions associées à des trajectoires a été collecté. L'agent doit apprendre à suivre une instruction depuis son point de départ afin d'atteindre le but (décrit par l'instruction).

5.9.3 Génération d'instruction

Lorsqu'on dispose d'un jeu de données donnant des exemples d'instructions, une possibilité afin d'apprendre à les générer (à partir du but ou de la trajectoire) est d'utiliser des techniques d'apprentissage par renforcement inverse (inverse RL). Ces techniques, étudiées dans [Dan+17], consistent à apprendre la fonction de récompense associée aux trajectoires expertes. L'architecture permet, à partir d'une trajectoire, de générer une commande intermédiaire en langage formel, puis la transforme en langage naturel grâce à un modèle typique de traduction de type encodeur-aligneur-décodeur. Cette méthode nécessite qu'il soit possible d'obtenir les commandes en langages formels pour les trajectoires des données utilisées en entraînement, mais il s'agit d'un des seuls travaux générant des instructions (plus complexes que quelques mots) à partir de trajectoires.

5.9.4 Suivi *et* génération d'instructions

D'autres modèles que le notre s'intéressent à l'apprentissage simultané de modèles pour le suivi et la génération d'instructions.

Dans [Yam+18], le framework proposé transforme les actions d'un robot en leur description linguistique et inversement. Le modèle consiste en deux auto-encodeur récurrents, et un coût supplémentaire est utilisé pour inciter le rapprochement, dans l'espace latent de représentation, de la représentation d'une action et de sa description associée. Dans un premier temps, les expériences sont effectuées sur un robot réel, mais sur des descriptions très simples (des combinaisons de push/pull/slide + red/green/yellow + slow/fast). Des expériences supplémentaires sont effectuées avec le jeu de données *KIT motion-language dataset* [Pla+16] où des mouvements humains sont reliés à leur description. Cependant, le jeu de données ne contient pas d'informations visuelles.

[Fri+17] utilise le même modèle que nous au départ pour la génération et le suivi d'instructions (un encodeur-aligneur-decodeur avec des réseaux de neurones profonds) puis étend l'architecture afin de construire un acteur et un locuteur *pragmatiques* : le locuteur génère l'instruction que l'acteur a le plus de chance de suivre correctement, et l'acteur suit la trajectoire pour laquelle le locuteur a le plus de chance de générer la même instruction. L'acteur pragmatique permet d'améliorer les résultats de quelques pourcents, mais c'est surtout les résultats du locuteur qui sont intéressants : des humains ont suivi les instructions générées et parviennent à suivre plus facilement celles du locuteur pragmatique que celles du locuteur basique, pourtant les scores bleu du premier sont inférieurs. C'est une preuve que le score bleu n'est pas suffisant pour évaluer la qualité d'une instruction, et qu'il faut trouver des méthodes alternatives si on ne souhaite pas faire appel à des testeurs humains.

Une extension du modèle précédent est présentée dans [Fri+18], paru alors que nous travaillions sur notre modèle acteur-locuteur. Le locuteur permet d'ajouter des données en générant des instructions à partir de trajectoires, et l'acteur apprend ensuite en utilisant à la fois les données d'entraînement et ces données supplémentaires, avant d'apprendre plus finement sur les données originales d'entraînement uniquement. Les expériences sont effectuées sur le nouveau jeu de donnée *Room-to-Room vision-and-language navigation* [And+18]. Ces expériences laissent penser que le jeu de données influence énormément les résultats : sur ces données, [Fri+18] rapporte une amélioration de l'acteur s'il utilise les données augmentées par le locuteur, alors que des expériences identiques que nous avons effectuées plus tôt sur nos données n'ont pas permis d'améliorations.

Enfin, des travaux ont été fait sur la communication entre plusieurs agents, avec des algorithmes du gradient de la politique [SF+16] (pour des jeux ou des tâches coopératifs), une version multi-agent de DQN [Foe+16] (pour résoudre une énigme ou des jeux sur MNIST), ou, plus proches de nos tâches, pour du dialogue visuel coopératif [Laz+16 ; Das+17] (un agent a accès à une photo, et doit la faire deviner à un second agent en dialoguant). Dans la plupart des cas (en dehors de [Das+17]), les communications sont simples et limitées à quelques mots de vocabulaire.

5.10 Conclusion et pistes

Dans le cas du locuteur, le score bleu ne donne pas une très bonne évaluation des instructions générées : il faudrait qu'un humain évalue si les instructions sont correctes, comme dans [Fri+17], mais l'évaluation est alors très couteuse. Une autre possibilité serait d'utiliser un acteur (qui sait déjà suivre des instructions émanant d'humains) afin de vérifier à moindre coût que les instructions générées sont correctes. Le problème est de déjà avoir ce modèle d'acteur : dans notre cas, l'acteur et le locuteur ont tendance à avoir de bons résultats sur les mêmes données (les instructions en peu de mots, de style "*turn right*") et des difficultés sur les phrases plus longues. C'est sans doute à cause de ce problème que l'apprentissage joint de l'acteur et du locuteur, présenté dans la section 5.7, ne donne que peu d'améliorations. Concernant l'architecture hiérarchique, l'ajout de mémoire au niveau du décodeur pour l'acteur améliore légèrement les résultats, mais l'intérêt ou non de mémoire dépend essentiellement de la tâche à résoudre.

Pour étendre ces expériences, la première chose à faire serait d'utiliser un jeu de données plus équilibré et plus conséquent. Plusieurs environnements possibles sont sortis en 2018, avec parfois des ensembles de données avec instructions et trajectoires associées. Le problème est que l'observation de l'agent est souvent plus complexe (une image, avec parfois des informations complémentaires) et que la complexité de la tâche grandit encore. Dans d'autres articles, des solutions ont été tentées en générant automatiquement des instructions, mais apprendre un modèle de locuteur sur des instructions déjà générées perd de l'intérêt.

6 Conclusion et pistes

Nous nous sommes intéressés dans cette thèse à la notion de *budget* afin de réduire la complexité de différents problèmes.

Dans le chapitre 3, nous avons d'abord abordé le problème de classification avec un grand nombre de classes. Afin de réduire la complexité en calcul des méthodes de classification, nous avons présenté un arbre de décision où la structure de l'arbre et les classifieurs sont appris en une seule étape, grâce à des techniques d'apprentissage par renforcement ; puis nous avons exposé une architecture qui permet d'associer les données à un code binaire, lui même associé à une classe, où les associations des codes avec les données et avec les classes sont faites en une seule étape. Il serait encore possible d'améliorer le modèle RECOC afin d'obtenir de meilleurs résultats, car des résultats équivalents à l'état de l'art sont obtenus avec des codes de grandes tailles : en améliorant la manière dont les codes sont appris, il devrait être possible d'obtenir les mêmes résultats avec des codes plus petits, et donc une complexité en calculs moindre.

Le chapitre 4 traite de l'apprentissage par renforcement hiérarchique, et de la division d'une tâche complexe en plusieurs tâches plus simples. Ici, le budget est l'observation par l'agent de son environnement, et l'utilisation de cette observation ; à rapprocher de l'effort cognitif décrit en neuroscience. Grâce à un apprentissage budgétisé, l'agent apprend à observer l'environnement (l'équivalent de commencer une nouvelle tâche, ou nouvelle *option*) uniquement quand c'est nécessaire, tout en conservant une bonne performance. Cette architecture pourrait être étendue dans plusieurs cadres : en apprentissage par renforcement, l'utilisation jointe de deux modèles (l'un coûteux en temps mais parfois plus précis, et l'autre plus rapide mais parfois inexact) ; l'utilisation par un agent de nombreux capteurs et le choix de les utiliser ou non ; enfin, on peut considérer que l'observation est une *instruction* que l'agent peut demander quand il en a besoin - mais uniquement quand c'est nécessaire, à cause du coût de l'interaction.

Nous abordons dans le chapitre 5 les échanges d'instructions en langage naturel : nous étudions d'une part un agent *acteur* qui apprend à suivre des

instructions, et d'autre part un agent *locuteur* qui apprend à générer des instructions. Nous utilisons pour cela un jeu de données constitué d'instructions en langage naturel associées à des trajectoires dans un labyrinthe, introduit dans [Mac+06]. Nous nous apercevons cependant que les modèles utilisés et les données sont limités : en effet, si les instructions simples ("*turn right*") sont comprises et générées correctement, il n'en est pas de même pour les instructions qui nécessitent une compréhension plus poussée de l'instruction en lien avec l'environnement. D'autres modèles existent qu'il faudrait tester, mais les expériences ne pourront pas être concluantes sans un jeu de données approprié. De nouveaux environnements, associés à des données instructionnelles, ont été publiés récemment, mais nécessitent un traitement poussé des images et du texte avant de pouvoir se concentrer sur le lien entre instructions et environnement.

L'apprentissage joint du locuteur et de l'acteur que nous avons tenté, du fait des données peu adaptées, n'a été que peu concluant. Il serait possible de refaire ces expériences dans un autre environnement.

Finalement, notre but était de faire le lien entre le suivi d'instructions et l'architecture hiérarchique du BHNN, afin d'avoir un acteur qui puisse *demander* des instructions à un locuteur quand il en a besoin (ou de la même manière, un locuteur qui puisse donner des instructions/conseils à un acteur pour l'aider dans sa tâche). Cette idée rejoint le travail du chapitre 4 sur la découverte de hiérarchie, avec un agent qui apprend à utiliser une observation coûteuse (qui serait ici une instruction) le moins souvent possible. Avant de travailler sur ce problème, il est cependant nécessaire d'avoir des modèles fonctionnels pour l'acteur et le locuteur afin que la génération et la compréhension d'instructions soient possibles. Nous pourrions également aborder cette idée sur des problèmes plus simples, où l'acteur et le locuteur échangent dans un langage plus simple que le langage naturel.

Bibliographie

[AK15] Jacob ANDREAS et Dan KLEIN. „Alignment-based compositional semantics for instruction following“. In : *arXiv preprint arXiv :1508.06491* (2015) (cf. p. 97).

[And+18] Peter ANDERSON, Qi WU, Damien TENEY et al. „Vision-and-language navigation : Interpreting visually-grounded navigation instructions in real environments“. In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, p. 3674–3683 (cf. p. 98, 99).

[Bah+14] Dzmitry BAHDANAU, Kyunghyun CHO et Yoshua BENGIO. „Neural machine translation by jointly learning to align and translate“. In : *arXiv preprint arXiv :1409.0473* (2014) (cf. p. 78).

[BB96] Kristin P BENNETT et J BLUE. „Optimal decision trees“. In : *Rensselaer Polytechnic Institute Math Report* 214 (1996) (cf. p. 45).

[BB99] J. BAXTER et P. BARTLETT. *Direct Gradient-Based Reinforcement Learning*. 1999 (cf. p. 26, 30).

[Bea+16] Charles BEATTIE, Joel Z. LEIBO, Denis TEPLYASHIN et al. „DeepMind Lab“. In : *CoRR* abs/1612.03801 (2016). arXiv : 1612.03801 (cf. p. 98).

[Ben+] Samy BENGIO, J WESTON et D. GRANGIER. „Label embedding trees for large multi-class tasks“. In : *Adv. Neural Inf. Process. Syst.* 1 (), p. 163–171 (cf. p. 45).

[Ben+10] Samy BENGIO, Jason WESTON et David GRANGIER. „Label embedding trees for large multi-class tasks“. In : *Advances in Neural Information Processing Systems*. 2010, p. 163–171 (cf. p. 26).

[Ben+13] Yoshua BENGIO, Nicholas LÉONARD et Aaron COURVILLE. „Estimating or propagating gradients through stochastic neurons for conditional computation“. In : *arXiv preprint arXiv :1308.3432* (2013) (cf. p. 36, 66).

[Bot+09] Matthew BOTVINICK, Yael NIV et Andrew C. BARTO. „Hierarchically Organized Behavior and Its Neural Foundations : A Reinforcement-Learning Perspective“. In : *cognition* 113.3 (2009) (cf. p. 17).

[BP15a] Pierre-Luc BACON et Doina PRECUP. „Learning with options : Just deliberate and relax“. In : (2015) (cf. p. 67).

[BP15b] Pierre-Luc BACON et Doina PRECUP. „The option-critic architecture“. In : *NIPS Deep Reinforcement Learning Workshop*. 2015 (cf. p. 19, 21).

[Bro+16] Greg BROCKMAN, Vicki CHEUNG, Ludwig PETTERSSON et al. *OpenAI Gym*. 2016. eprint : arXiv:1606.01540 (cf. p. 56).

[Bui+02] Hung Hai BUI, Svetha VENKATESH et Geoff WEST. „Policy recognition in the abstract hidden markov model“. In : *Journal of Artificial Intelligence Research* 17 (2002), p. 451–499 (cf. p. 22).

[Che+16] Yong CHENG, Wei XU, Zhongjun HE et al. „Semi-supervised learning for neural machine translation“. In : *arXiv preprint arXiv :1606.04596* (2016) (cf. p. 96).

[Cho+14] Kyunghyun CHO, Bart VAN MERRIËNBOER, Caglar GULCEHRE et al. „Learning phrase representations using RNN encoder-decoder for statistical machine translation“. In : *arXiv preprint arXiv :1406.1078* (2014) (cf. p. 51, 55).

[Chu+16] Junyoung CHUNG, Sungjin AHN et Yoshua BENGIO. „Hierarchical Multiscale Recurrent Neural Networks“. In : *arXiv preprint arXiv :1609.01704* (2016) (cf. p. 66).

[Cis+] M CISSÉ, T ARTIÈRES et P GALLINARI. „Learning compact class codes for fast inference in large multi class classification“. In : *Eur. Conf. Mach. Learn.* () (cf. p. 46).

[Cis+11] M CISSÉ, T ARTIERES et Patrick GALLINARI. „Learning efficient error correcting output codes for large hierarchical multi-class problems“. In : *Workshop on Large-Scale Hierarchical Classification ECML/PKDD*. T. 11. 2011, p. 37–49 (cf. p. 26).

[CL14a] Anna CHOROMANSKA et John LANGFORD. „Logarithmic time online multiclass prediction“. In : *arXiv preprint arXiv :1406.1822* (2014) (cf. p. 42).

[CL14b] Anna CHOROMANSKA et John LANGFORD. „Logarithmic Time Online Multiclass prediction“. In : (2014), p. 1–13. arXiv : arXiv:1406.1822v1 (cf. p. 45).

[Con+16] Gabriella CONTARDO, Ludovic DENOYER et Thierry ARTIÈRES. „Recurrent Neural Networks for Adaptive Feature Acquisition“. In : *International Conference on Neural Information Processing*. Springer International Publishing. 2016, p. 591–599 (cf. p. 53).

[DA+12] Gabriel DULAC-ARNOLD, Ludovic DENOYER, Philippe PREUX et Patrick GALLINARI. „Sequential approaches for learning datum-wise sparse representations“. In : *Machine Learning* 89.1-2 (2012), p. 87–122 (cf. p. 53).

[DA14] Gabriel DULAC-ARNOLD. „A General Sequential Model for Constrained Classification“. Thèse de doct. Paris 6, 2014 (cf. p. 23).

[Dan+16] Christian DANIEL, Herke van HOOF, Jan PETERS et Gerhard NEUMANN. „Probabilistic Inference for Determining Options in Reinforcement Learning“. In : (2016) (cf. p. 19).

[Dan+17] Andrea F DANIELE, Mohit BANSAL et Matthew R WALTER. „Navigational instruction generation as inverse reinforcement learning with neural machine translation“. In : *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI*. IEEE. 2017, p. 109–118 (cf. p. 72, 98).

[Das+17] Abhishek DAS, Satwik KOTTUR, José MF MOURA, Stefan LEE et Dhruv BATRA. „Learning cooperative visual dialog agents with deep reinforcement learning“. In : *Proceedings of the IEEE International Conference on Computer Vision*. 2017, p. 2951–2960 (cf. p. 100).

[DB] TG DIETTERICH et Ghulum BAKIRI. „Solving multiclass learning problems via error-correcting output codes“. In : *arXiv Prepr. cs/9501101* () (cf. p. 46).

[DB12] Amir DEZFOULI et Bernard W BALLEINE. „Habits, action sequences and reinforcement learning“. In : *European Journal of Neuroscience* 35.7 (2012), p. 1036–1051 (cf. p. 47, 49).

[DB13] Amir DEZFOULI et Bernard W BALLEINE. „Actions, action sequences and habits : evidence that goal-directed and habitual action control are hierarchically organized“. In : *PLoS Comput Biol* 9.12 (2013), e1003364 (cf. p. 47).

[DB95] Thomas G. DIETTERICH et Ghulum BAKIRI. „Solving multiclass learning problems via error-correcting output codes“. In : *Journal of artificial intelligence research* (1995), p. 263–286 (cf. p. 26).

[DG14] Ludovic DENOYER et Patrick GALLINARI. „Deep Sequential Neural Network“. In : *arXiv preprint arXiv :1410.0510 - Workshop Deep Learning NIPS 2014* (2014) (cf. p. 30).

[DH93] Peter DAYAN et Geoffrey E HINTON. „Feudal reinforcement learning“. In : *Advances in neural information processing systems*. Morgan Kaufmann Publishers. 1993, p. 271–271 (cf. p. 18).

[Die98] Thomas G DIETTERICH. „The MAXQ Method for Hierarchical Reinforcement Learning.“ In : *ICML*. Citeseer. 1998, p. 118–126 (cf. p. 18).

[DL16] Li DONG et Mirella LAPATA. „Language to logical form with neural attention“. In : *arXiv preprint arXiv :1601.01280* (2016) (cf. p. 97).

[Flo+16] Carlos FLORENSA, Yan DUAN et Pieter ABBEEL. „Stochastic Neural Networks for Hierarchical Reinforcement Learning“. In : *ICLR* (2016) (cf. p. 22, 67).

[Foe+16] Jakob FOERSTER, Ioannis Alexandros ASSAEL, Nando de FREITAS et Shimon WHITESON. „Learning to communicate with deep multi-agent reinforcement learning“. In : *Advances in Neural Information Processing Systems*. 2016, p. 2137–2145 (cf. p. 100).

[Fri+17] Daniel FRIED, Jacob ANDREAS et Dan KLEIN. „Unified pragmatic models for generating and following instructions“. In : *arXiv preprint arXiv :1711.04987* (2017) (cf. p. 83, 94, 99, 100).

[Fri+18] Daniel FRIED, Ronghang HU, Volkan CIRIK et al. „Speaker-follower models for vision-and-language navigation“. In : *Advances in Neural Information Processing Systems*. 2018, p. 3318–3329 (cf. p. 99).

[Gal+13] Mikel GALAR, Alberto FERNÁNDEZ, Edurne BARRENECHEA, Humberto BUSTINCE et Francisco HERRERA. „Dynamic classifier selection for one-vs-one strategy : avoiding non-competent classifiers“. In : *Pattern Recognition* 46.12 (2013), p. 3412–3424 (cf. p. 42).

[Ger+17] Thomas GERALD, Nicolas BASKIOTIS et Ludovic DENOYER. „Binary Stochastic Representations for Large Multi-class Classification“. In : *International Conference on Neural Information Processing*. Springer. 2017, p. 155–165 (cf. p. 33, 35, 38, 42).

[GK11] Tianshi GAO et Daphne KOLLER. „Active classification based on value of classifier“. In : *Advances in Neural Information Processing Systems*. 2011, p. 1062–1070 (cf. p. 23).

[Gor+18] Daniel GORDON, Aniruddha KEMBHAVI, Mohammad RASTEGARI et al. „Iqa : Visual question answering in interactive environments“. In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, p. 4089–4098 (cf. p. 97).

[Gra+13] Alex GRAVES, Abdel-rahman MOHAMED et Geoffrey HINTON. „Speech recognition with deep recurrent neural networks“. In : *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE. 2013, p. 6645–6649 (cf. p. 13).

[Gre+16] Karol GREGOR, Danilo Jimenez REZENDE et Daan WIERSTRA. „Variational Intrinsic Control“. In : *arXiv preprint arXiv :1611.07507* (2016) (cf. p. 49, 52, 67).

[Han+96] Eric A HANSEN, Andrew G BARTO et Shlomo ZILBERSTEIN. „Reinforcement learning for mixed open-loop and closed-loop control“. In : *NIPS*. 1996, p. 1026–1032 (cf. p. 22, 67).

[Hau+98] Milos HAUSKRECHT, Nicolas MEULEAU, Leslie Pack KAELBLING, Thomas DEAN et Craig BOUTILIER. „Hierarchical solution of Markov decision processes using macro-actions“. In : *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 1998, p. 220–229 (cf. p. 22, 67).

[He+12] He HE, Jason EISNER et Hal DAUME. „Imitation learning by coaching“. In : *Advances in Neural Information Processing Systems*. 2012, p. 3149–3157 (cf. p. 23).

[He+15] Kaiming HE, Xiangyu ZHANG, Shaoqing REN et Jian SUN. „Deep Residual Learning for Image Recognition“. In : *arXiv :1512.03385* (2015) (cf. p. 42).

[He+16a] Di HE, Yingce XIA, Tao QIN et al. „Dual learning for machine translation“. In : *Advances in Neural Information Processing Systems*. 2016, p. 820–828 (cf. p. 72, 96).

[He+16b] Ji HE, Jianshu CHEN, Xiaodong HE et al. „Deep reinforcement learning with an action space defined by natural language“. In : (2016) (cf. p. 97).

[Hee+16] Nicolas HEESS, Greg WAYNE, Yuval TASSA et al. „Learning and Transfer of Modulated Locomotor Controllers“. In : *arXiv preprint arXiv :1610.05182* (2016) (cf. p. 22, 67).

[Hem+15] Sachithra HEMACHANDRA, Felix DUVALLET, Thomas M HOWARD et al. „Learning models for following natural language directions in unknown environments“. In : *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2015, p. 5608–5615 (cf. p. 98).

[Her+17] Karl Moritz HERMANN, Felix HILL, Simon GREEN et al. „Grounded language learning in a simulated 3d world“. In : *arXiv preprint arXiv :1706.06551* (2017) (cf. p. 98).

[JC07] Shihao JI et Lawrence CARIN. „Cost-sensitive feature acquisition and classification“. In : *Pattern Recognition* 40.5 (2007), p. 1474–1485 (cf. p. 23).

[JJ94] Michael I. JORDAN et Robert A. JACOBS. „Hierarchical mixtures of experts and the EM algorithm“. In : *Neural computation* 6.2 (1994), p. 181–214 (cf. p. 45).

[KB14a] Diederik KINGMA et Jimmy BA. „Adam : A method for stochastic optimization“. In : *arXiv preprint arXiv :1412.6980* (2014) (cf. p. 43, 55).

[KB14b] Wouter KOOL et Matthew BOTVINICK. „A labor/leisure tradeoff in cognitive control.“ In : *Journal of Experimental Psychology : General* 143.1 (2014), p. 131 (cf. p. 52).

[Ker+11] Mehdi KERAMATI, Amir DEZFOULI et Payam PIRAY. „Speed/accuracy trade-off between the habitual and the goal-directed processes“. In : *PLoS Comput Biol* 7.5 (2011), e1002055 (cf. p. 47, 49).

[Kri+12] Alex KRIZHEVSKY, Ilya SUTSKEVER et Geoffrey E HINTON. „Imagenet classification with deep convolutional neural networks“. In : *Advances in neural information processing systems*. 2012, p. 1097–1105 (cf. p. 13).

[Kul+16] Tejas D KULKARNI, Karthik R NARASIMHAN, Ardavan SAEEDI et Joshua B TENENBAUM. „Hierarchical deep reinforcement learning : Integrating temporal abstraction and intrinsic motivation“. In : *arXiv preprint arXiv :1604.06057* (2016) (cf. p. 19, 20).

[Kum+16] Ankit KUMAR, Ozan IRSOY, Peter ONDRUSKA et al. „Ask me anything : Dynamic memory networks for natural language processing“. In : *International Conference on Machine Learning*. 2016, p. 1378–1387 (cf. p. 97).

[Laz+16] Angeliki LAZARIDOU, Alexander PEYSAKHOVICH et Marco BARONI. „Multi-agent cooperation and the emergence of (natural) language“. In : *arXiv preprint arXiv :1612.07182* (2016) (cf. p. 100).

[LB05] John LANGFORD et Alina BEYGELZIMER. „Sensitive error correcting output codes“. In : *Learning Theory*. Springer, 2005, p. 158–172 (cf. p. 26).

[Liu+13a] Baoyuan LIU, Fereshteh SADEGHI, Marshall TAPPEN, Ohad SHAMIR et Ce LIU. „Probabilistic label trees for efficient large scale image classification“. In : *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, p. 843–850 (cf. p. 26).

[Liu+13b] Baoyuan LIU, Fereshteh SADEGHI, Marshall TAPPEN, Ohad SHAMIR et Ce LIU. „Probabilistic label trees for efficient large scale image classification“. In : *Comput. Vis. Pattern Recognit.* (2013), p. 843–850 (cf. p. 45).

[Mac+06] Matt MACMAHON, Brian STANKIEWICZ et Benjamin KUIPERS. „Walk the talk : Connecting language, knowledge, and action in route instructions“. In : *Def* 2.6 (2006), p. 4 (cf. p. 74, 96, 102).

[Mei+16] Hongyuan MEI, Mohit BANSAL et Matthew R WALTER. „Listen, Attend, and Walk : Neural Mapping of Navigational Instructions to Action Sequences.“ In : *AAAI*. T. 1. 2016, p. 2 (cf. p. 72, 73, 75, 77–79, 81–83, 94, 96).

[MK15] Paul MINEIRO et Nikos KARAMPATZIAKIS. „Fast Label Embeddings via Ramdomized Linear Algebra“. In : *arXiv preprint arXiv :1412.6547* (2015) (cf. p. 42).

[Mni+15] Volodymyr MNIH, Koray KAVUKCUOGLU, David SILVER et al. „Human-level control through deep reinforcement learning“. In : *Nature* 518.7540 (2015), p. 529–533 (cf. p. 13).

[Mni+16a] Volodymyr MNIH, Adria Puigdomenech BADIA, Mehdi MIRZA et al. „Asynchronous methods for deep reinforcement learning“. In : *International conference on machine learning*. 2016, p. 1928–1937 (cf. p. 16).

[Mni+16b] Volodymyr MNIH, Adria Puigdomenech BADIA, Mehdi MIRZA et al. „Asynchronous methods for deep reinforcement learning“. In : *International Conference on Machine Learning*. 2016, p. 1928–1937 (cf. p. 57).

[Mni+16c] Volodymyr MNIH, John AGAPIOU, Simon OSINDERO et al. „Strategic Attentive Writer for Learning Macro-Actions“. In : *arXiv preprint arXiv :1606.04695* (2016) (cf. p. 22, 67).

[Nar+15] Karthik NARASIMHAN, Tejas KULKARNI et Regina BARZILAY. „Language understanding for text-based games using deep reinforcement learning“. In : *arXiv preprint arXiv :1506.08941* (2015) (cf. p. 97).

[Nor+14] M. NOROUZI, A. PUNJANI et D. J. FLEET. „Fast Exact Search in Hamming Space With Multi-Index Hashing“. In : *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.6 (2014), p. 1107–1119 (cf. p. 36).

[Nor+15] Mohammad NOROUZI, Maxwell COLLINS, Matthew A JOHNSON, David J FLEET et Pushmeet KOHLI. „Efficient non-greedy optimization of decision trees“. In : *Advances in Neural Information Processing Systems*. 2015, p. 1720–1728 (cf. p. 45).

[Pan+02] Bo PANG, Lillian LEE et Shivakumar VAITHYANATHAN. „Thumbs up ? : sentiment classification using machine learning techniques“. In : *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics. 2002, p. 79–86 (cf. p. 25).

[Par+15] Ioannis PARTALAS, Aris KOSMOPOULOS, Nicolas BASKIOTIS et al. „Large Scale Hierarchical Text Classification Challenge : A Benchmark for Large-Scale Text Classification“. In : *arXiv :1503.08581v1* (2015) (cf. p. 42).

[Pla+16] Matthias PLAPPERT, Christian MANDERY et Tamim ASFOUR. „The KIT motion-language dataset“. In : *Big data* 4.4 (2016), p. 236–252 (cf. p. 99).

[PR98] Ronald PARR et Stuart RUSSELL. „Reinforcement learning with hierarchies of machines“. In : *Advances in neural information processing systems* (1998), p. 1043–1049 (cf. p. 18).

[Rüc+13] Thomas RÜCKSTIESS, Christian OSENDORFER et Patrick van der SMAGT. „Minimizing data consumption with sequential online feature selection“. In : *International Journal of Machine Learning and Cybernetics* 4.3 (2013), p. 235–243 (cf. p. 23).

[Sch] RE SCHAPIRE. „Using output codes to boost multiclass learning problems“. In : *ICML* 1 (), p. 1–9 (cf. p. 46).

[Sch+15] Tom SCHAUL, Daniel HORGAN, Karol GREGOR et David SILVER. „Universal value function approximators“. In : *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, p. 1312–1320 (cf. p. 67).

[Sch97] Robert E SCHAPIRE. „Using output codes to boost multiclass learning problems“. In : *ICML*. T. 97. 1997, p. 313–321 (cf. p. 26).

[Sen+15] Rico SENNRICH, Barry HADDOW et Alexandra BIRCH. „Improving neural machine translation models with monolingual data“. In : *arXiv preprint arXiv :1511.06709* (2015) (cf. p. 96).

[SF+16] Sainbayar SUKHBAATAR, Rob FERGUS et al. „Learning multiagent communication with backpropagation“. In : *Advances in Neural Information Processing Systems*. 2016, p. 2244–2252 (cf. p. 100).

[Sha+17] Sahil SHARMA, Aravind S LAKSHMINARAYANAN et Balaraman RAVINDRAN. „Learning to repeat : Fine grained action repetition for deep reinforcement learning“. In : *arXiv preprint arXiv :1702.06054* (2017) (cf. p. 22, 58, 67).

[Sut+99] Richard S SUTTON, Doina PRECUP et Satinder SINGH. „Between MDPs and semi-MDPs : A framework for temporal abstraction in reinforcement learning“. In : *Artificial intelligence* 112.1 (1999), p. 181–211 (cf. p. 18, 57).

[VJ10] Adam VOGEL et Dan JURAFSKY. „Learning to follow navigational directions“. In : *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2010, p. 806–814 (cf. p. 96).

[Wan+18] Jingdong WANG, Ting ZHANG, Nicu SEBE, Heng Tao SHEN et al. „A survey on learning to hash“. In : *IEEE transactions on pattern analysis and machine intelligence* 40.4 (2018), p. 769–790 (cf. p. 46).

[Wes+] Jason WESTON, A MAKADIA et Hector YEE. „Label partitioning for sublinear ranking“. In : *Int. Conf. Mach. Learn.* () (cf. p. 45).

[Wes+13] Jason WESTON, Ameesh MAKADIA et Hector YEE. „Label partitioning for sublinear ranking“. In : *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. 2013, p. 181–189 (cf. p. 26).

[Wie+10] Daan WIERSTRA, Alexander FÖRSTER, Jan PETERS et Jürgen SCHMIDHUBER. „Recurrent policy gradients“. In : *Logic Journal of the IGPL* 18.5 (2010), p. 620–634 (cf. p. 15).

[Wil92] Ronald J WILLIAMS. „Simple statistical gradient-following algorithms for connectionist reinforcement learning“. In : *Machine learning* 8.3-4 (1992), p. 229–256 (cf. p. 13, 15).

[Wu+17] Qi WU, Damien TENEY, Peng WANG et al. „Visual question answering : A survey of methods and datasets“. In : *Computer Vision and Image Understanding* 163 (2017), p. 21–40 (cf. p. 97).

[Yam+18] Tatsuro YAMADA, Hiroyuki MATSUNAGA et Tetsuya OGATA. „Paired Recurrent Autoencoders for Bidirectional Translation Between Robot Actions and Linguistic Descriptions“. In : *IEEE Robotics and Automation Letters* 3.4 (2018), p. 3441–3448 (cf. p. 99).

[Yu+17] Haonan YU, Haichao ZHANG et Wei XU. „A deep compositional framework for human-like language acquisition in virtual environment“. In : *arXiv preprint arXiv :1703.09831* (2017) (cf. p. 72, 97).

[ZX13] Bin ZHAO et Eric P XING. „Sparse output coding for large-scale visual recognition“. In : *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, p. 3350–3357 (cf. p. 26).

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.