Apprentissage par renforcement développemental

Matthieu Zimmer

▶ **To cite this version:**

Matthieu Zimmer. Apprentissage par renforcement développemental. Intelligence artificielle [cs.AI]. Université de Lorraine, 2018. Français. NNT : 2018LORR0008 . tel-01735202

HAL Id: tel-01735202
https://theses.hal.science/tel-01735202

Submitted on 15 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

École doctorale IAEM Lorraine

Apprentissage par renforcement développemental

THÈSE

présentée et soutenue publiquement le 15 janvier 2018

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Matthieu Zimmer

Composition du jury

Rapporteurs :	Olivier Sigaud Olivier Pietquin	Professeur, UPMC - Paris 6, ISIR - INRIA Professeur, Université Lille 1, Google DeepMind
Examinateurs :	Isabelle Debled-Rennesson Céline Teulière	Professeur, Université de Lorraine, LORIA MCF, Université Blaise Pascal, Institut Pascal
Directeur de thèse :	Alain Dutech	CR HDR, INRIA Grand Est, LORIA
Encadrant de thèse :	Yann Boniface	MCF, Université de Lorraine, LORIA

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

Je tiens à remercier mes deux encadrants de thèse Alain Dutech et Yann Boniface qui m'ont guidé avec cordialité et bienveillance durant ces trois dernières années. Avec Nicolas Rougier, ce sont les premiers à m'avoir accordé leur confiance en me proposant un stage de recherche en licence, bien qu'à ce moment, je n'avais que peu de connaissances en intelligence artificielle. Ils m'ont ainsi ouvert les portes de la recherche, me permettant d'être accepté dans d'autres stages et d'aboutir à cette thèse.

De même, je souhaite remercier les différents membres constituant le jury de thèse pour avoir accepté de donner de leur temps pour évaluer mes travaux. En particulier, mes deux rapporteurs Olivier Sigaud et Olivier Pietquin pour leurs nombreux retours sur le manuscrit.

Je remercie également mon référent Hervé Frezza-Buet et Bruno Scherrer pour les réunions qui m'ont aidé à clarifier certaines notions concernant l'apprentissage par renforcement.

Je remercie mes différents collègues pour tous leurs partages et leur aide : Nassim, Quan, Xuan-Son, Vassilis, Omar, Thomas, Iñaki, Mihai, Arsène, Mohammed, Théo, Nicolas, Achille, Abdallah, Manel, et tous les autres.

Je remercie plusieurs permanents que ce soit pour leurs conseils ou leur encadrement antérieur qui m'a permis de construire cette thèse : Stéphane Doncieux, Paul Weng, Paolo Viappiani, Nicolas Rougier, Oliver Buffer, Francis Colis, Vincent Thomas, Jean-Philippe Mangeot, Bernard Girau, Lucas Nussbaum et François Charpillet.

Je remercie aussi les nombreux développeurs des différents outils utilisés au cours de cette thèse (Annexe B page 141), plus particulièrement l'équipe de la plateforme Grid5000 que j'ai utilisée intensivement, sans oublier les équipes INRIA gérant la forge et l'intégration continue.

Pour finir, je remercie ma compagne Lan, pour l'équilibre qu'elle apporte dans ma vie et son soutien constant jusqu'à son aide sur plusieurs figures de ce document.

À mes parents pour leur dévouement,
à mes sœurs pour leur indulgence,
à mon frère pour sa ténacité,
à M. René Lapeyre, professeur de mathématiques,

et à tout lecteur de cette thèse.

Table des matières

Notations **viii**

Introduction **1**

1 Motivations . . . 1
2 Problématique et exemple d'environnement . . . 3
3 Contributions . . . 6
4 Plan de thèse . . . 6

Chapitre 1 Apprentissage automatique **9**

1.1 Types de représentations de fonctions . . . 10
1.1.1 Propriétés des modèles . . . 10
1.1.2 Quelques modèles classiques . . . 12
1.1.3 Réseaux de neurones . . . 13
1.2 Optimisation des paramètres . . . 15
1.2.1 Descente de gradient . . . 16
1.2.2 Méthodes de second ordre . . . 19
1.2.3 Optimisation *black-box* . . . 19
1.2.4 Application aux réseaux de neurones . . . 20
1.3 Limitations . . . 20
1.3.1 Surapprentissage ou généralisation . . . 20
1.3.2 Biais ou variance . . . 23
1.4 Conclusion . . . 24

Chapitre 2 Apprentissage par renforcement **25**

2.1 Formalisation . . . 25
2.1.1 Politique et critère d'optimisation . . . 27
2.1.2 Observabilité partielle . . . 29
2.1.3 Méthodes de résolution . . . 29
2.1.4 Classification des différents agents apprenants . . . 30

2.2 Fonctions de valeur - *critic-only* 31
2.2.1 Différentes fonctions de valeurs 31
2.2.2 Programmation dynamique 32
2.2.3 Évaluation par Monte-Carlo 33
2.2.4 Apprentissage par *Temporal Difference* : TD(0) 34
2.2.5 Le compromis TD(λ) 35
2.2.6 Évaluation de politique avec des données *off-policy* 37
2.2.7 Généralisation et approximation 40
2.2.8 Conclusion 46
2.3 Policy Search - *Actor-only* 47
2.3.1 Politiques paramétrées 48
2.3.2 Espace d'exploration 49
2.3.3 *Policy-gradient* 50
2.4 *Actor-Critic* 52
2.4.1 Gradient d'une politique stochastique avec fonction de valeur 52
2.4.2 Gradient d'une politique déterministe avec fonction de valeur 55
2.4.3 Méthode alternative 56
2.4.4 Conclusion 56
2.5 Limitations 57
2.5.1 Exploration ou exploitation 57
2.5.2 Efficacité en données ou mise à l'échelle 58
2.6 Conclusion 60

Chapitre 3 Environnements continus **63**
3.1 Cartpole 63
3.2 Acrobot 64
3.3 Half-Cheetah 65
3.4 Humanoïde 67
3.5 Conclusion 69

Chapitre 4 Architectures acteur-critique neuronales on-policy **71**
4.1 État de l'art des algorithmes *online semi-gradient* 71
4.1.1 SAC 72
4.1.2 DAC 72
4.1.3 CACLA 72
4.1.4 Bilan des mises à jour 72
4.1.5 Validation expérimentale 73

4.2 Limitation de CMA-ES 75
4.3 Neural Fitted Actor-Critic 76
4.4 Première validation expérimentale de NFAC 78
4.5 NFAC(λ) avec traces d'éligibilité 79
4.6 Analyses de NFAC 81
4.6.1 Mise à jour de l'acteur par CACLA 81
4.6.2 Ordre des mises à jour de NFAC 82
4.6.3 Comparaisons supplémentaires 83
4.7 Conclusion 84

Chapitre 5 Architectures acteur-critique neuronales off-policy **87**
5.1 Le framework Neural Fitted Actor-Critic 87
5.2 One step avec Q et le gradient d'une politique déterministe 89
5.2.1 DDPG 89
5.2.2 Off-policy NFAC(0)-∂Q 90
5.2.3 Validation expérimentale 91
5.3 One step avec Q et CACLA 93
5.4 Pondération des transitions off-policy 94
5.5 Multi-step avec la fonction de valeur d'état V 96
5.5.1 Tree-Backup généralisé avec V 96
5.5.2 Validation expérimentale 96
5.6 Multi-step avec la fonction de valeur Q 98
5.6.1 Off-policy NFAC(λ)-∂Q 98
5.6.2 Validation expérimentale 98
5.7 Comparaison finale 100
5.8 Conclusion 101

Chapitre 6 Exploration guidée de l'espace sensorimoteur **103**
6.1 Robotique développementale 104
6.2 Augmentation de l'espace sensorimoteur de l'agent 104
6.2.1 Définition 104
6.2.2 Définition alternative 105
6.2.3 Problématique 106
6.2.4 Changement d'espace de recherche 106
6.3 Couche neuronale développementale 106
6.3.1 Emplacement 107
6.3.2 Action neutre 108

6.3.3 Changement d'espace de recherche 108
6.3.4 Limitations 108
6.3.5 Recherche connexe 109
6.4 Démonstration de faisabilité des couches développementales 109
6.4.1 Recherche dans une grille 110
6.4.2 Environnements 110
6.4.3 Validation expérimentale 111
6.4.4 Discussion 115
6.5 Conclusion 115

Bilan **117**

Conclusion **121**

Perspectives **125**

Annexes **129**

Annexe A Métaparamètres **129**
A.1 Métaparamètres fixés 129
A.2 Algorithmes acteur-critique online on-policy 130
A.3 Limitations de CMAES 131
A.4 CACLA et NFAC sur Cartpole et Acrobot 131
A.5 NFAC(λ) sur Half-Cheetah 132
A.6 Algorithmes *on-policy* sur Half-Cheetah 132
A.7 NFAC(0) $- \partial Q$ 133
A.8 NFAC(0) $- \delta Q$ 136
A.9 Comparaison finale 136
A.10 Couches développementales 137

Annexe B Outils informatiques **141**

Bibliographie **143**

Table des figures **153**

Liste des tableaux **155**

Table des définitions et des théorèmes **156**

Table des algorithmes **157**

Notations

Variables

α	taux d'apprentissage
$\boldsymbol{\mathcal{I}}$	matrice d'information de Fisher
$\boldsymbol{H}$	matrice hessienne
$\mathcal{D}$	*replay buffer*
$\mathcal{F}$	ensemble des fonctions représentables par un modèle
$\mathcal{H}$	fonction de Heaviside
$\mathcal{N}$	loi normale
$\mathcal{R}$	terme de régularisation
μ	moyenne
ϕ	fonctions de base, fonctions primitives et fonctions d'activations
Ψ	un modèle de représentation
σ	variance
Θ	l'espace des paramètres θ
θ	les paramètres à apprendre
ζ	décalage pour éviter une division par 0
J	critère à optimiser
w	les paramètres du critique

Général

$(x_i)_{\{1,\dots,k\}} \sim y$	un échantillon de k éléments est tiré de la distribution de y
$(x_i)_{\{1,\dots,k\}}$	échantillon de k élements $x_1, x_2, \dots, x_k$
$[x]_+$	fonction qui vaut x lorsque $x > 0$ sinon 0
$[x]_{i,j}$	$i^{\text{ème}}$ ligne et $j^{\text{ème}}$ colonne de la matrice x
$\Delta\hat{x}$	direction de modification
$\hat{x}$	estimation de x
$\nabla_\theta x$ ou $\frac{\partial x}{\partial \theta}$	gradient de x par rapport à θ

$\underset{x}{\mathbb{1}}$	fonction qui vaut 1 si x est vrai et 0 sinon
$\lvert x \rvert$	nombre d'éléments dans l'ensemble x (cardinal)
$\lvert\lvert x \rvert\rvert_2$	norme L2 du vecteur x
$x \approx y$	x est environ égal à y (approximation)
$x \leftarrow y$	x prend la valeur y (mise à jour)
$x \ll y$	x est beaucoup plus petit que y
$x \sim y$	la variable aléatoire x a la distribution de y

MDP

δ	erreur temporelle
γ	facteur d'actualisation
$\mathcal{A}$	l'ensemble des actions
$\mathcal{S}$	l'ensemble des états
$\mathcal{S}^*$	l'ensemble des états absorbants
π	une politique
τ	une trajectoire
A	la fonction avantage
L	longeur d'un épisode (nombre d'étapes)
R	la fonction de récompense
T	la fonction de transition
T_0	distribution ou densité de probabilité sur l'état initial

Introduction

> Les hommes se trompent quand ils se croient libres ; cette opinion consiste en cela seul qu'ils sont conscients de leurs actions et ignorants des causes par lesquelles ils sont déterminés.
>
> Baruch Spinoza

1 Motivations

En posant les bases de l'intelligence artificielle, Turing (1950) émit l'idée de concevoir des agents capables d'apprendre comme des enfants plutôt que de tenter de reproduire directement l'intelligence d'un adulte. Ce concept est mis de côté au détriment des approches symboliques de haut niveau, c'est dans les années 90 que Brooks (1991) insistera sur l'importance de l'incarnation, de l'interaction avec l'environnement et de l'apprentissage pour développer des agents adaptatifs efficacement. En opposition à l'idée que le corps n'est qu'une interface de la pensée, où seraient manipulés les symboles, la théorie de l'*embodiment* postule qu'un corps est nécessaire à l'établissement de l'intelligence et ne peut-être dissocié de la pensée. De ces courants émerge la *robotique développementale* dans les années 2000 (Asada *et al.*, 2009), où les chercheurs entreprennent de doter les robots d'algorithmes d'apprentissage sophistiqués, sans fournir de représentations prédéfinies et avec le moins de connaissances spécifiques données *a priori*.

Cette thèse s'inscrit conjointement dans ces différents cadres en ajoutant l'hypothèse que le but de cet agent agnostique est de maximiser un signal de récompense : il apprend par renforcement. Son corps se situe dans un environnement riche et continu, il ne manipule pas de symboles discrets, il ne dispose ainsi pas d'un ensemble dénombrable d'actions ou d'états préconçus. Les modèles utilisés par l'agent sont non linéaires puisque l'agent doit construire ses propres représentations à travers ses nombreuses interactions avec l'environnement. Tandis que beaucoup d'algorithmes d'apprentissage par renforcement sont discrets ou reposent sur des schémas d'approximation linéaires, la question principalement abordée est de savoir comment apprendre à long terme par renforcement en espace continu d'états et d'actions avec des schémas d'approximation non linéaires et le moins de connaissances spécifiques possible. Pour aborder ce problème difficile, une dernière hypothèse est formulée : le corps de l'agent et la difficulté du problème qu'il résout croissent avec le temps afin de permettre une exploration partiellement guidée de l'espace de recherche.

Mettre en œuvre un tel apprentissage par renforcement *développemental*, qui peut continuellement construire sur ce qu'il a appris, est essentiel pour résoudre des problèmes où l'environnement

est inconnu et complexe, par exemple pour rendre l'exploration spatiale plus autonome, pour demain développer des robots domestiques qui s'adaptent à différents domiciles, pour rendre la voiture autonome moins dépendante de la cartographie, etc.

Apprendre à partir de récompenses, en se basant sur la théorie de l'apprentissage par renforcement, permet de bénéficier des nombreuses recherches de ce domaine et de ses fondements théoriques bien établis. Définir une fonction de récompense permet de définir un but global à l'agent, sans forcément lui fournir la solution pour y parvenir. Contrairement à *l'apprentissage supervisé*, les fonctions de récompenses sont peu informatives. Au regard de la diversité des comportements possibles, apprendre à partir de récompenses est peu contraignant tout en garantissant une évaluation des différentes politiques. Enfin, le paradigme de l'apprentissage par renforcement, basé sur une séparation agent-environnement en interaction, se marie bien avec celui de la robotique *développementale*.

Résoudre des problèmes en environnement continu a plusieurs avantages, le premier étant qu'on se rapproche de problèmes réalistes, le second étant qu'on s'inscrit plus dans une approche *développementale*, car apprendre les concepts physiques de base est primordial pour espérer développer un agent ayant des connaissances du niveau de l'humain (Guerin, 2011). Il semble évident qu'apprendre ces concepts physiques est plus naturel sur des domaines continus. Enfin, apprendre en environnements continus permet de ne pas insuffler de symboles discrets à l'agent : il ne se repose pas sur des ensembles discrets et dénombrables d'actions ou d'états qui limiteraient ses capacités de perception et d'interaction.

Les approximateurs non linéaires, plus particulièrement les réseaux de neurones, disposent de plusieurs propriétés qui nous intéressent. Ils permettent justement de faire des approximations sur les domaines continus. Avec le développement de *l'apprentissage profond*, les réseaux de neurones ont montré leurs bonnes capacités de généralisation, de *mise à l'échelle* et leur large *pouvoir de représentation*. Avec un grand nombre de données, ils permettent justement d'éviter de définir *a priori* des *fonctions de base* qui sont ensuite combinées de manière linéaire, méthode répandue jusqu'à maintenant en robotique (Kober *et al.*, 2013), mais d'apprendre les leurs par l'expérience : l'agent dispose ainsi de ses propres représentations qu'il aura construites.

Ne pas insuffler trop de connaissances prédéfinies à l'agent permet de ne pas restreindre les représentations et les solutions qu'il produit, maintenant ainsi sa faculté à nous surprendre par ses solutions et le rendant plus adaptatif aux situations inconnues durant la phase de développement (Doncieux, 2016). En définissant beaucoup de connaissances *a priori*, nous limitons la façon dont l'agent résout ses problèmes à une manière humaine de faire, qui peut ne pas être optimale vis-à-vis de l'espace sensorimoteur de l'agent. Ainsi, se reposer sur peu de connaissances *a priori* en laissant l'agent construire ses propres représentations, c'est déjà s'inscrire dans une approche *développementale* (Guerin, 2011; Sigaud et Droniou, 2016). Pour cette raison, nous considérons déjà le cadre de *l'apprentissage par renforcement profond en environnements continus* où l'agent contrôle son corps, comme faisant partie de *l'apprentissage par renforcement développemental* : deux chapitres de contributions y sont consacrés.

Toutes les contraintes fixées jusqu'à maintenant, à savoir, apprendre par renforcement, dans des environnements continus, avec des approximateurs non linéaires, vont dans le même sens : augmenter l'espace dans lequel l'agent recherche une politique, donc l'éventail des comportements possibles de l'agent. Or, si ne partir de rien pour tout apprendre est possible, il faudrait un temps considérable pour résoudre des problèmes où l'espace de recherche est large. Pour contraster avec

cette augmentation de la liberté de l'agent, afin de l'aider à construire des solutions dans un temps raisonnable, nous explorons l'hypothèse que le laisser contrôler la complexité de la tâche qu'il résout sera bénéfique à son apprentissage (Oudeyer et Kaplan, 2008). Pour cela, nous nous intéresserons à une définition plus restrictive de *l'apprentissage par renforcement développemental* où la dimension et la complexité de l'espace sensorimoteur de l'agent augmentent pendant qu'il est en train d'apprendre. Une conséquence positive de cette augmentation progressive est que l'espace de recherche peut s'en trouver également plus réduit au début de l'apprentissage.

2 Problématique et exemple d'environnement

Dans cette thèse, en cohérence avec l'approche développementale, l'agent n'a pas connaissance du modèle de l'environnement, ainsi il n'a pas connaissance des conséquences de ses actions ni des états qu'il doit atteindre : il doit apprendre à travers ses interactions avec cet environnement qui lui est totalement inconnu.

L'apprentissage est réalisé par *épisodes*. Au début de chaque *épisode*, l'agent est placé dans une situation initiale, puis commence à interagir avec l'environnement pour maximiser un signal de récompense. L'*épisode* prend fin lorsque l'agent atteint un *état absorbant* ou si le temps limite est dépassé. Ce mode d'apprentissage n'est pas très réaliste, mais il s'agit du plus répandu dans la littérature, car il permet de procéder à des comparaisons efficaces entre plusieurs algorithmes d'apprentissage.

Durant la phase d'exécution dans l'environnement, le comportement d'un agent est entièrement régi par une fonction mathématique, appelée *politique*. Puisque nous nous situons dans des domaines continus, nous avons besoin d'utiliser un approximateur. Cette estimation de la *politique* est habituellement une combinaison linéaire de fonctions de base qui doivent être soigneusement prédéfinies par un expert. Leur nature et leur nombre limitent l'expressivité de la fonction apprise. Cela entrave la capacité de l'agent à développer ses propres représentations. Pour ces raisons, et celles décrites précédemment dans nos motivations, nous avons choisi d'utiliser les réseaux de neurones comme approximateur. Tout ce que l'agent perçoit arrive en entrée de la *politique* : un neurone de la couche d'entrée est dédié à chaque dimension sensorielle. De même, dans la couche de sortie, un neurone est dédié à chaque degré de liberté. À intervalles réguliers, le réseau propose des actions en fonction de l'état de l'environnement. Ainsi, trouver un bon comportement consiste à trouver un bon ensemble de paramètres prenant des valeurs continues, en l'occurrence les poids des réseaux de neurones.

Nous avons choisi de suivre une approche *model-free*, ce qui signifie que l'agent ne va pas chercher à construire directement un modèle de l'environnement. Si l'approche *model-based* s'est déjà montrée efficace (Deisenroth et Rasmussen, 2011), elle ne permet pas toujours d'apporter facilement une amélioration lorsqu'elle est combinée aux réseaux de neurones (Gu *et al.*, 2016b). Par ailleurs, il paraît difficile qu'un agent puisse prédire avec exactitude l'implication de tous les évènements possibles dans un environnement complexe.

Par conséquent, la problématique principale de cette thèse est de trouver des politiques paramétriques en environnements continus (actions et états), en temps discrétisé, sans modélisation de l'environnement (*model-free*), en apprenant des approximateurs non linéaires, tels que les réseaux de neurones, avec des méthodes ayant une complexité linéaire en nombre de paramètres pour ne pas diminuer la capacité de *mise à l'échelle* des réseaux de neurones.

Par exemple, l'un des problèmes que nous cherchons à résoudre est illustré dans la figure 1 page suivante. Dans ce problème, l'agent contrôle un «demi-guépard» dans un plan en deux

dimensions. L'objectif de l'agent est d'avancer le plus vite possible vers l'avant en interaction avec un sol immobile dans un plan en étant soumis à des contraintes physiques réalistes. Pour ce faire, il doit définir les couples à appliquer sur chacune de ses 6 articulations à partir de 18 dimensions sensorielles. Pour ce problème, avec un réseau de neurones composé de deux couches cachées de 50 et 25 unités, la dimensionnalité de l'espace de recherche est d'environ 2500 paramètres à apprendre. Ce premier réseau de neurones représentant la politique fait partie de l'*acteur*.

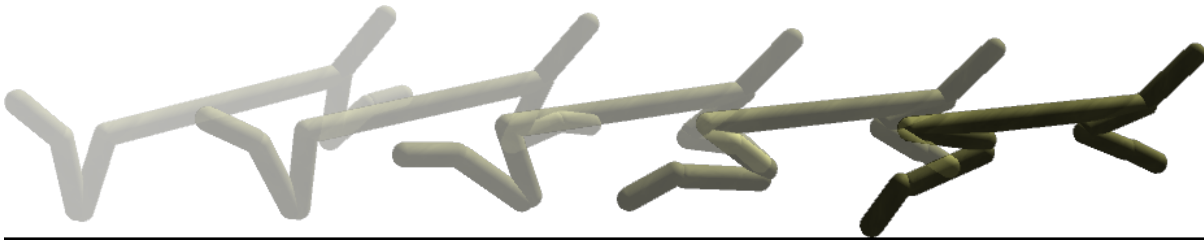


FIGURE 1 – Un des problèmes que nous souhaitons résoudre : un «demi-guépard» qui apprend à courir en interaction avec un sol. Les corps transparents représentent les instants antérieurs.

À ces paramètres, il faut ajouter ceux du *critique* qui sont d'environ du même ordre, ce qui fait un espace de recherche total d'environ 5000 paramètres. En effet, tout au long de cette thèse, nous nous appuyons sur l'architecture *acteur-critique* comme algorithme d'apprentissage principal de l'agent (Konda et Tsitsiklis, 1999). Nous avons déjà décrit le rôle de *l'acteur* qui est de choisir une action en fonction de l'état, tandis que le *critique* permet d'évaluer la qualité de différentes situations, servant à améliorer la politique. Dans le *critique*, tout ce que l'agent perçoit arrive en entrée du réseau : un neurone de la couche d'entrée est dédié à chaque dimension sensorielle. Selon le type de *critique*, il est également possible d'ajouter les dimensions motrices en entrée (la sortie de la politique). En couche de sortie, le *critique* dispose d'un seul neurone, qui représente la qualité estimée de l'entrée fournie.

Pour souligner la difficulté à trouver une bonne politique, la figure 2 page ci-contre montre les couples appliqués sur les articulations du corps de l'agent par une politique apprise avec nos algorithmes. Remarquons qu'il s'agit d'une simplification illustrative, car chacun de ces 6 degrés de liberté dépend en réalité de 18 dimensions d'entrées, mais pas directement du temps. L'agent doit découvrir de lui-même ces structures temporelles et synchroniser ses membres pour pouvoir courir.

On peut remarquer que, dans ces structures, peu de points ont exactement la même amplitude qui est presque chaque fois différente. Cela atteste de l'utilité de ne pas définir d'actions discrètes prédéfinies.

L'*efficacité en données* correspond au nombre de données dont un algorithme d'apprentissage a besoin pour atteindre une performance satisfaisante. Un algorithme *efficace en données* aura besoin de moins de données, mais passera plus de temps sur chacune d'elles. Cette mesure est importante lorsqu'il est très coûteux de produire des données, par exemple en robotique. *A contrario* si les données sont très simples à produire, comme dans le domaine des *big data*, il est improductif de passer un temps excessif à tirer toutes les informations possibles d'un échantillon.

En début de thèse, peu d'algorithmes *acteur-critique* utilisant des réseaux de neurones dans des environnements continus existaient, la plupart étaient *online* et peu *efficaces en données*, car ils n'utilisaient qu'une seule fois les interactions générées. Aussi, les contributions de cette thèse

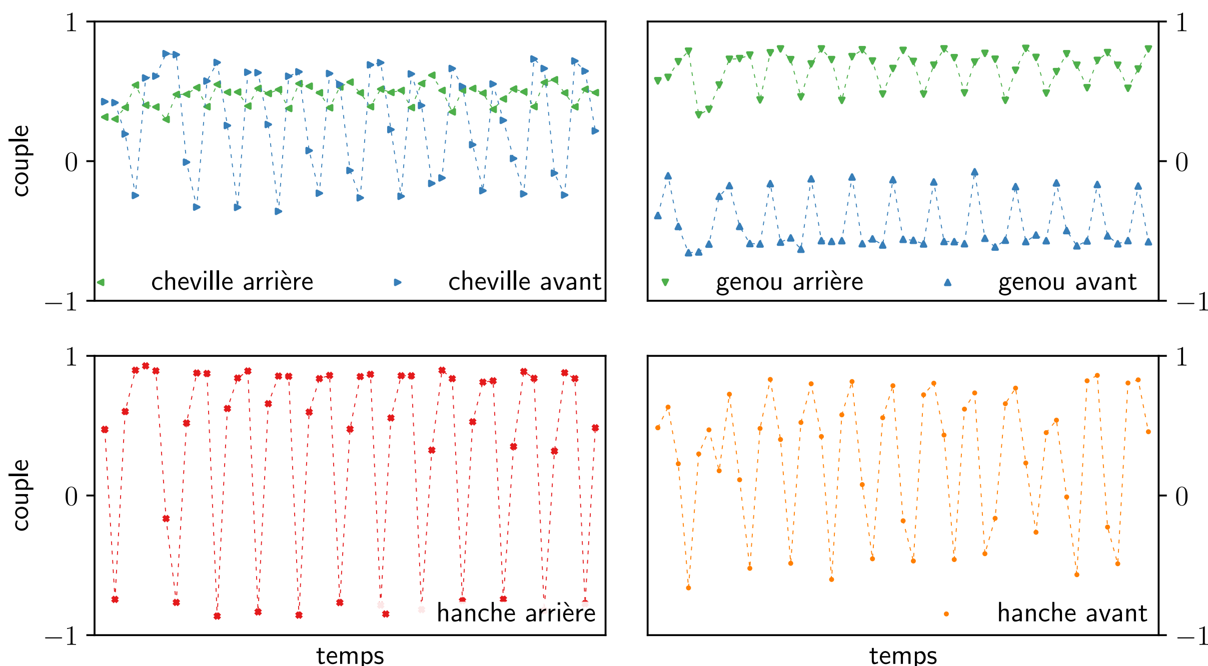


FIGURE 2 – Une politique satisfaisante apprise par nos algorithmes.

peuvent être divisées en deux parties, la première partie tente de pallier le manque d'*efficacité en données* des algorithmes *acteur-critique* et la seconde explore une façon d'augmenter progressivement l'espace sensorimoteur de l'agent. Chacune de ces parties s'inscrit dans *l'apprentissage par renforcement développemental*.

Au cours de la thèse, parallèlement à nos recherches, de nombreuses contributions ont été réalisées dans le domaine de *l'apprentissage par renforcement profond*. En particulier, le calcul et l'utilisation du gradient du coût d'une politique déterministe (Silver *et al.*, 2014), qu'on croyait jusqu'alors dépendants du modèle de l'environnement, combinés aux succès de *l'apprentissage par renforcement profond* dans des environnements discrets (Mnih *et al.*, 2015), a permis l'émergence d'algorithmes efficaces en environnements continus (Lillicrap *et al.*, 2015; Mnih *et al.*, 2016; Wang *et al.*, 2016). Plusieurs propositions ont été faites pour améliorer la précision du *critique* (Munos *et al.*, 2016; Gu *et al.*, 2016a, 2017; Mahmood *et al.*, 2017; Wang *et al.*, 2015) et d'autres pour stabiliser *l'acteur* (Schulman *et al.*, 2015a,b). De même, concernant l'augmentation de l'espace sensorimoteur, des propositions ont été soumises sur la façon de ne pas oublier les apprentissages précédents dans les réseaux de neurones (Fernando *et al.*, 2017; Kirkpatrick *et al.*, 2017; Rusu *et al.*, 2016).

Pour rédiger cette thèse, nous avons pris en compte les nouvelles contributions importantes proposées par d'autres chercheurs. Nous avons également tenté de comparer nos algorithmes au maximum avec ceux de l'état de l'art. Néanmoins, ce domaine de recherche étant très dynamique durant la période de la thèse, nous n'avons pas pu effectuer de comparaisons avec les algorithmes les plus récents proposés après 2016.

3 Contributions

Pour pallier le manque *d'efficacité en données* des méthodes existantes en début de thèse, nous avons proposé un nouvel algorithme *acteur-critique*, *Neural Fitted Actor-Critic* (Zimmer *et al.*, 2016a,d), capable d'être utilisé avec des réseaux de neurones, combinant les succès de deux algorithmes précédents (Van Hasselt et Wiering, 2007; Riedmiller, 2005). Notre algorithme est *on-policy offline model-free* : cela signifie qu'il n'utilise pas de transitions produites par d'autres politiques que celle exécutée et que ses mises à jour sont réalisées à la fin d'un *épisode*. Pour augmenter son *efficacité en données*, il utilise un *replay buffer* (Lin, 1992) avec des méthodes *fitted value iteration* (Gordon, 1995). Au cours de la thèse, nous avons montré expérimentalement qu'en utilisant les avancées récentes de *l'apprentissage par renforcement profond*, notre algorithme était plus efficace que d'autres développés parallèlement (Lillicrap *et al.*, 2015; Mnih *et al.*, 2016).

De par nos expériences au cours de cette thèse, nous avons mis en exergue le dilemme entre *efficacité en données* et *mise à l'échelle* particulièrement présent dans *l'apprentissage par renforcement profond*. Nous avons détaillé les différents composants et les choix influençant ce dilemme afin d'en trouver un compromis.

Parce que *l'apprentissage par renforcement profond* en domaine continu manque d'environnements ouverts pour effectuer de solides comparaisons (Islam *et al.*, 2017; Duan *et al.*, 2016), nous avons proposé plusieurs environnements ouverts et gratuits utilisant le moteur physique ODE (Smith, 2005).

Pour prendre en compte des transitions générées depuis des politiques antérieures, afin de réaliser un apprentissage *off-policy*, nous avons proposé d'étendre notre premier algorithme *Neural Fitted Actor-Critic* (Zimmer *et al.*, 2016c,b) pour le rendre encore plus *efficace en données*. Nous avons vérifié expérimentalement qu'avec l'apprentissage *off-policy*, il pouvait être *efficace en données* en trouvant rapidement de bonnes solutions, mais qu'il perdait de sa stabilité. Pour choisir un compromis entre *efficacité en données* et *mise à l'échelle*, il est possible de choisir le nombre de trajectoires gardées en mémoire dans ce nouvel algorithme.

Enfin, nous avons proposé une manière d'effectuer une augmentation progressive de l'espace sensorimoteur dans un cadre *d'apprentissage par renforcement profond* en environnement continu. Pour éviter d'avoir à modifier les environnements ou les algorithmes d'apprentissage, notre solution se base seulement sur des modifications internes aux réseaux de neurones. Il s'agit d'une *couche développementale* qui contrôle quelles dimensions de l'espace sensorimoteur sont perceptibles et contrôlables. À travers plusieurs validations expérimentales sur différents algorithmes d'apprentissage, nous avons remarqué que les *couches développementales*, et donc l'augmentation progressive de l'espace sensorimoteur de l'agent, permettaient, dans plusieurs cas, d'accélérer l'apprentissage et également d'augmenter la qualité de la solution trouvée asymptotiquement. Ces derniers travaux n'ont pas encore été publiés.

4 Plan de thèse

Au cours du chapitre 1, nous détaillerons plusieurs méthodes provenant de l'apprentissage supervisé et de l'apprentissage profond. L'apprentissage supervisé sera utilisé pour apprendre les fonctions de valeurs présentes en apprentissage par renforcement et mettre à jour les poids de la *politique*. Nous évoquerons plusieurs types de représentations de fonctions, en nous attardant en particulier sur les réseaux de neurones et la *descente de gradient* utilisés tout au long de cette

thèse. Nous expliquerons le dilemme *biais-variance* sous-jacent à tout algorithme d'apprentissage utilisant des données.

Dans le chapitre 2, nous décrirons les fondements théoriques de l'apprentissage par renforcement, les façons d'apprendre des fonctions de valeurs, l'apprentissage par différence temporelle avec des traces d'éligibilité, l'apprentissage *on-policy* et *off-policy*, et les façons d'utiliser des approximateurs. Dans un premier temps, nous nous focaliserons sur les approches *critic-only* pour apprendre un critique, puis sur les approches *actor-only* qui définissent la notion de politique paramétrée, pour enfin combiner les deux dans les techniques *acteur-critique*.

Pendant le chapitre 3, nous exposerons les différents environnements continus que nous utiliserons au cours de nos validations expérimentales. Ils sont au nombre de quatre dont plusieurs ayant un espace sensorimoteur conséquent.

Durant le chapitre 4, nous proposerons notre premier algorithme *acteur-critique* utilisant des réseaux de neurones pour résoudre des problèmes en environnements continus avec une bonne *efficacité en données* et peu de métaparamètres à régler. Nous verrons comment les traces d'éligibilité peuvent l'étendre afin de contrôler le dilemme *biais-variance*. Nous montrerons expérimentalement que son apprentissage est plus efficace que plusieurs autres algorithmes de l'état de l'art.

Lors du chapitre 5, nous analyserons comment notre algorithme pourrait profiter de l'apprentissage *off-policy* afin d'augmenter encore son *efficacité en données*. À cette fin, nous modifierons la fonction de valeur apprise et la mise à jour de l'acteur. Nous procéderons à plusieurs expériences pour voir quelle méthode est la plus efficace entre les méthodes *off-policy one-step* qui utilisent une seule transition et les méthodes *off-policy multi-step* qui combinent à la fois estimation *off-policy* et traces d'éligibilité.

Finalement, au fil du chapitre 6, disposant d'algorithmes d'apprentissage par renforcement robustes, nous donnerons une définition à l'augmentation progressive de l'espace sensorimoteur de l'agent en nous appuyant sur celle des *processus décisionnels de Markov*. Nous explorerons comment cette augmentation peut être réalisée sans modifications de l'environnement ou de l'algorithme d'apprentissage. Nous suggérerons de travailler uniquement sur les représentations de l'agent, plus précisément sur les entrées et sorties des réseaux de neurones.

Nous terminerons par un bilan de ce que nous avons réalisé, une conclusion de ce que nous en déduisons, et les perceptives que nous imaginons pour de futures recherches.

Chapitre 1

Apprentissage automatique

Ce chapitre sert principalement à introduire des techniques d'apprentissage automatique que nous utiliserons en apprentissage par renforcement dans le chapitre suivant ; en particulier, la régression par descente de gradient avec des réseaux de neurones.

L'apprentissage automatique consiste à modéliser une fonction Ψ à partir de données. On parle de l'apprentissage d'un *modèle* mathématique : une description d'un système en utilisant une formulation mathématique. Selon les données disponibles et l'objectif, il existe trois grandes classes de méthodes d'apprentissage d'un modèle $\Psi : X \to Y$ (Figure 1.1 page suivante).

Apprentissage supervisé L'apprentissage *supervisé* (Bishop, 2006; Hastie *et al.*, 2009) repose sur l'utilisation de données étiquetées : pour un échantillon d'entrée de taille n, noté $(x_i)_{\{1,\dots,n\}} \in X^n$, chaque entrée x_i est étiquetée par une sortie correspondante $y_i \in Y$. On parle d'une *base de données* $(x_i, y_i)_{\{1,\dots,n\}} \in X^n \times Y^n$. Le modèle Ψ tente d'associer correctement chaque entrée x_i à sa sortie y_i.

Apprentissage non supervisé L'apprentissage *non supervisé* (Barlow, 1989) ne repose que sur des données sans étiquettes $(x_i)_{\{1,\dots,n\}} \in X^n$. La sortie n'est pas connue *a priori*. L'association à une sortie est réalisée à travers la ressemblance avec d'autres entrées : on parle de *clustering*. Cette classe d'apprentissage n'a pas été utilisée au cours de cette thèse, nous ne la détaillerons donc pas plus.

Apprentissage par renforcement Enfin, l'apprentissage *par renforcement* (Sutton et Barto, 1998) repose sur l'utilisation de données indirectement étiquetées par des récompenses. Cet étiquetage est moins informatif qu'en apprentissage supervisé. Là où il existe un oracle du type $\text{oracle}(x_i) = y_i$ pour étiqueter les données *a priori* en apprentissage supervisé, dans l'apprentissage par renforcement cet oracle n'est capable que de quantifier une association (x_i, y_i) par un scalaire : $\text{oracle}(x_i, y_i) = \text{récompense}$. De plus, les données (et leur ordre) présentées à l'oracle ne sont pas maîtrisées entièrement par l'utilisateur. L'hypothèse d'avoir des données i.i.d [1] n'est pas valable et il n'existe pas de base de données *a priori*. Or, cette hypothèse est largement utilisée dans l'analyse formelle de nombreux algorithmes d'apprentissage automatique. Le chapitre 2 est consacré à l'apprentissage par renforcement avec des explications plus poussées.

L'apprentissage supervisé d'une fonction $\Psi : X \to Y$ peut encore se diviser en deux catégories selon le type de l'ensemble Y. Lorsque Y est un ensemble dénombrable, on parle de problème de

1. indépendantes et identiquement distribuées

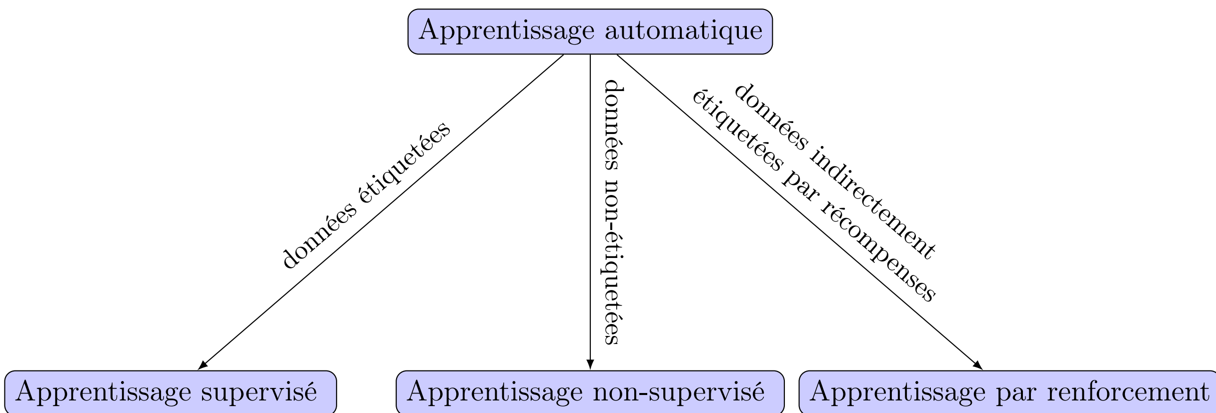


FIGURE 1.1 – Les trois grandes classes d'apprentissage automatique.

classification. Sinon, on parle de problème de *régression*. En classification, on essaie de catégoriser les entrées dans les bonnes classes. En régression, on estime une relation entre entrée et sortie. Dans la suite de ce chapitre, nous nous intéresserons plus particulièrement à la régression qui est très utilisée en apprentissage par renforcement pour l'approximation de fonction.

1.1 Types de représentations de fonctions

Nous allons maintenant présenter les formes mathématiques et les propriétés des modèles que nous utiliserons au cours de la thèse et ceux utilisés en général pour faciliter la compréhension globale. Dans toute cette partie, X et Y représentent l'ensemble d'entrées et l'ensemble de sorties d'un modèle $\Psi : X \rightarrow Y$.

1.1.1 Propriétés des modèles

Les formes de modèles peuvent être très diverses, aussi nous allons exposer quelques caractéristiques permettant de les distinguer les unes des autres.

Discret ou continu Les modèles discrets s'appliquent généralement sur des ensembles dénombrables. Ils utilisent l'hypothèse que X et Y sont des ensembles discrets. Les modèles continus ne font pas cette hypothèse et se servent d'ensembles non dénombrables en général. La figure 1.2 page ci-contre illustre cette différence.

Paramétrique ou non paramétrique Les modèles paramétriques, que nous noterons Ψ_θ, où θ représente l'ensemble des paramètres utilisés par le modèle, reposent sur l'interaction mathématique entre les paramètres θ et les entrées $x_i \in X$ pour calculer les sorties $y_i \in Y$. Ainsi, apprendre un modèle paramétrique, c'est chercher un ensemble de paramètres θ. Au fur et à mesure que l'on présente des données au modèle, il modifie ses paramètres. La forme de la fonction d'approximation est limitée à une classe de fonctions paramétrées fixe, on parle aussi de complexité du modèle. Un modèle non paramétrique ne repose pas sur la mise à jour de paramètres, mais sur l'ensemble des données précédemment présentées. Par exemple, en combinant les données, la sortie peut être estimée grâce à une notion de distance. Il s'agit du principe utilisé dans la méthode non paramétrique des k plus proches voisins (Dudani, 1976). L'intérêt

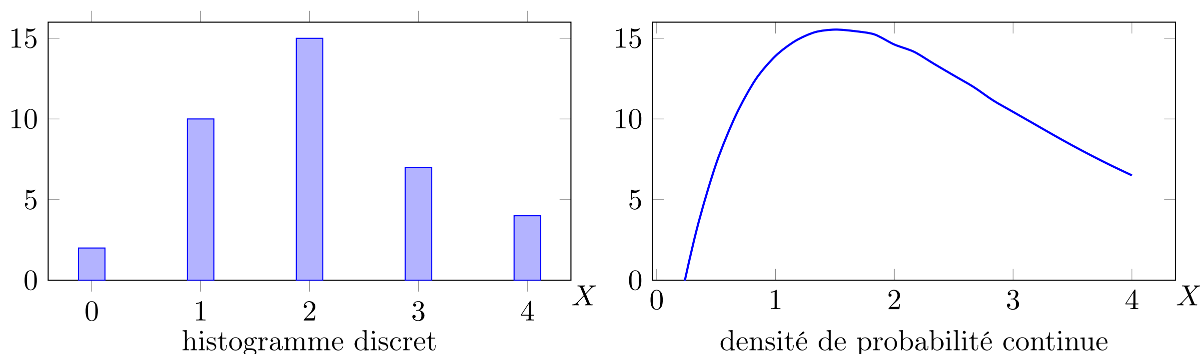


FIGURE 1.2 – Exemple de différence entre discret et continu. Pour estimer la fréquence des évènements sur un domaine X, à gauche on compte avec un histogramme (cas discret), à droite on estime une densité de probabilité (cas continu).

des modèles non paramétriques étant que la complexité du modèle n'a pas besoin d'être fixée *a priori*. Par contre, le temps de calcul augmente avec le nombre de données, contrairement aux modèles paramétriques.

Linéaire ou non linéaire Si toutes les opérations effectuées dans un modèle sont linéaires [2], on parle de modèle linéaire. Dans le cas contraire, il s'agit d'un modèle non linéaire. L'avantage des modèles non linéaires réside dans leur expressivité, ils sont capables de décrire davantage de systèmes ayant une complexité supérieure. On dit qu'ils ont un plus grand *pouvoir de représentation* (Figure 1.3 page suivante). À l'inverse, les modèles linéaires ont un *pouvoir de représentation* moindre, mais permettent une meilleure étude mathématique fournissant plusieurs garanties formelles (convergence, erreur d'approximation bornée, etc.).

Déterministe ou stochastique Un modèle déterministe associe toujours la même sortie à une entrée, contrairement à un modèle stochastique qui est probabiliste. Si Y est discret, la sortie intermédiaire (avant le tirage aléatoire) d'un modèle stochastique représente la distribution de probabilité sur Y, sinon lorsque Y est continu, la sortie intermédiaire correspond à une densité de probabilité sur Y.

Statique ou dynamique Un modèle statique ne prend pas en compte le temps pour calculer sa sortie, il reste identique à mesure que le temps passe, contrairement au modèle dynamique. On parle aussi de processus *stationnaire*. En présentant la même entrée à un modèle statique à deux instants différents, sa sortie (ou sa loi de probabilité pour un modèle stochastique) sera la même, ce qui ne sera pas forcément le cas dans un modèle dynamique. Par exemple, les réseaux de neurones récurrents sont des modèles dynamiques (Hochreiter et Schmidhuber, 1997).

Nous nous intéresserons principalement aux modèles continus, paramétriques et non linéaires que sont les réseaux de neurones statiques et déterministes ; pour les raisons suivantes :

— nous souhaitons résoudre des problèmes dans des environnements continus,

2. opérations respectant l'additivité et l'homogénéité

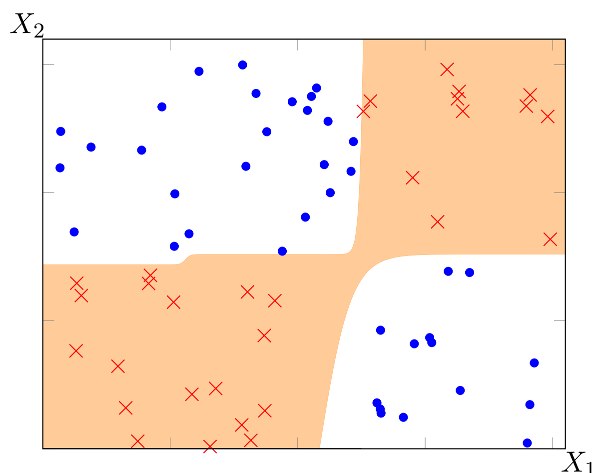


FIGURE 1.3 – Exemple d'un problème non discriminable par un modèle linéaire. La fonction XOR, fonction logique «ou exclusive», ne peut être discriminée par une simple droite. Dans cet exemple, $X = (X_1 \times X_2)$ est composé de deux dimensions et l'échantillon a été discriminé par la fonction XOR(X_1, X_2) résultant des croix et des ronds. Il est impossible de trouver une droite, et donc un modèle linéaire, où tous les ronds seraient d'un côté et toutes les croix seraient de l'autre. La surface orange représente une frontière déterminant l'appartenance à l'ensemble des ronds selon un modèle non linéaire : ce modèle dispose d'un plus grand *pouvoir de représentation* qu'un modèle linéaire.

— les réseaux de neurones, qui sont des modèles paramétriques, disposent d'une bonne capacité de mise à l'échelle (LeCun *et al.*, 2012), caractéristique importante pour traiter les environnements en hautes dimensions,
— ils disposent d'un meilleur *pouvoir de représentation* que les modèles linéaires,
— les réseaux de neurones, en raison de leur non-linéarité, permettent à un agent d'acquérir ses propres représentations sans s'appuyer sur des fonctions de base définies *a priori* pour suivre une approche développementale (Asada *et al.*, 2009).

Nous allons d'abord présenter quelques modèles plus simples, avant d'expliquer les réseaux de neurones, pour faciliter la compréhension globale, comprendre les différences mathématiques entre les modèles, et définir des modèles utilisés par certains algorithmes d'apprentissage par renforcement abordés au chapitre 2.

1.1.2 Quelques modèles classiques

Les modèles que nous allons présenter dans cette section supposent que $Y = \mathbb{R}$. La généralisation à un ensemble de sorties $\mathbb{R}^n$ est triviale et ne sera pas abordée. Nous décrivons ici des modèles utilisés en apprentissage par renforcement ; il en existe de nombreux autres.

Modèles tabulaires Un modèle tabulaire est un modèle *exact* : il ne fait pas d'approximation et ne bénéficie pas de généralisation. Il est discret sur X et paramétrique. On peut le stocker numériquement sous forme la forme d'un tableau.

Définition 1.1 (Modèle tabulaire)**.** *Soit $\Psi_\theta : X \to \mathbb{R}$ un modèle tabulaire, pour tout $x_i \in X$:*

$$\Psi_\theta(x_i) = \theta_i,$$

et le nombre de paramètres $|\theta| = |X|$.

Modèles linéaires Un modèle linéaire est un modèle paramétrique continu, potentiellement discret sur X. Il est défini par un vecteur de paramètres.

Définition 1.2 (Modèle linéaire)**.** *Soit $\Psi_\theta : \mathbb{R}^n \to \mathbb{R}$ un modèle linéaire, pour tout $x \in \mathbb{R}^n$:*

$$\Psi_\theta(x) = \theta x = \sum_{i=1}^{n} \theta_i x_i,$$

et le nombre de paramètres $|\theta| = n$.

Modèles linéaires avec fonctions de base fixées Un modèle linéaire avec des fonctions de base fixées est un modèle paramétrique, potentiellement discret sur X. La sortie globale peut être non linéaire, néanmoins les opérations avec θ restent linéaires. Ces modèles sont très utilisés, mais nécessitent de définir un ensemble de fonctions de base *a priori*.

Définition 1.3 (Modèle linéaire avec fonctions de base fixées)**.** *Soit $\Psi_{\theta,\phi} : \mathbb{R}^n \to \mathbb{R}$ un modèle linéaire avec m fonctions de base fixées $\phi_i : \mathbb{R}^n \to \mathbb{R}$, pour tout $x \in \mathbb{R}^n$:*

$$\Psi_{\theta,\phi}(x) = \sum_{i=1}^{m} \theta_i \phi_i(x),$$

et le nombre de paramètres $|\theta| = m$.

Il est possible de considérer X discret, dans ce cas on prendra $m \ll |X|$ et des fonctions de base appropriées $\phi_i : X \to \mathbb{R}$. La construction de fonctions de base fut l'objet de nombreuses recherches. Celles principalement utilisées en apprentissage par renforcement étant le *coarse coding*, le *tile coding*, les fonctions de base radiales et les méthodes de *feature selection* (Albus, 1971; Hinton, 1984; Broomhead et Lowe, 1988; Blum et Langley, 1997).

1.1.3 Réseaux de neurones

Nous allons maintenant aborder le modèle qui nous intéresse dans cette thèse : les réseaux de neurones. Il s'agit de modèles paramétriques continus. Contrairement aux modèles linéaires avec fonctions de base fixées, il n'y a plus besoin de fournir des fonctions de base. On définira tout de même des opérations de base, aussi appelées fonctions primitives ou fonctions d'activations, qui serviront à construire automatiquement l'équivalent des fonctions de base du réseau de neurones. Par ailleurs, elles ne seront justement pas fixées, mais devront s'améliorer : leurs paramètres sont intégrés à θ. Plutôt que de fournir les fonctions de base, on fournit une structure et des opérateurs élémentaires.

Il existe de nombreuses structures de réseaux de neurones. Nous avons choisi de travailler avec des *perceptrons multicouches* pour des raisons de simplicité et pour leurs bonnes performances

(Hornik *et al.*, 1989), en théorie ils peuvent approximer n'importe quelle fonction avec un nombre de neurones suffisant. Le réseau est organisé en couches où chaque neurone est connecté à l'ensemble des neurones de la prochaine couche. Cette organisation ne fait pas d'hypothèses *a priori* sur les données d'entrée, ce qui est intéressant. Le réseau est composé d'unités simples, les *neurones formels* (Figure 1.4). Le *neurone formel* est un modèle paramétrique, continu et statique.

Définition 1.4 (Neurone formel)**.** *Soit* $\Psi_{\theta,\phi} : \mathbb{R}^n \to \mathbb{R}$ *un neurone formel déterministe avec une fonction d'activation* $\phi : \mathbb{R} \to \mathbb{R}$*, pour tout* $x \in \mathbb{R}^n$ *:*

$$\Psi_{\theta,\phi}(x) = \phi\Big(\theta_0 + \sum_{i=1}^{n} \theta_i x_i\Big),$$

et le nombre de paramètres $|\theta| = n + 1$*.*

Remarquons la ressemblance avec un modèle linéaire, la principale différence étant la fonction d'activation ϕ à fournir. Elle est à différencier des fonctions de base qui travaillent directement sur l'espace X. Ici, la fonction d'activation est une fonction primitive, une opération mathématique de $\mathbb{R} \to \mathbb{R}$. Le *perceptron* (Rosenblatt, 1958; Widrow et Hoff, 1960) est cas particulier de *neurone formel* linéaire où ϕ est la fonction de Heaviside. La non-linéarité du modèle provient potentiellement de la fonction d'activation ϕ. θ_0 est appelé biais.

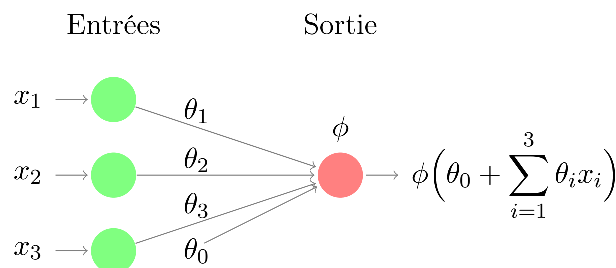


Figure 1.4 – Un *neurone formel* avec $X = \mathbb{R}^3$.

En combinant plusieurs neurones sur plusieurs couches, on obtient un *perceptron multicouche* (Figure 1.5 page suivante). Il s'agit d'un modèle non linéaire à la fois en sortie et en paramètre.

Définition 1.5 (Perceptron multicouche)**.** *Soit* $\Psi_{\theta,\phi} : \mathbb{R}^n \to \mathbb{R}$ *un perceptron multicouche avec* m *fonctions d'activation* $\phi_i : \mathbb{R} \to \mathbb{R}$ *fixées par couche, et* m *couches cachées constituées de* $(h_k)_{\{1,\ldots,m\}} \in \mathbb{N}^m$ *neurones chacune, alors pour tout* $x \in \mathbb{R}^n$ *:*

$$\Psi_{\theta,\phi}(x) = \Psi_{\theta,\phi}(x, 1, m+1),$$

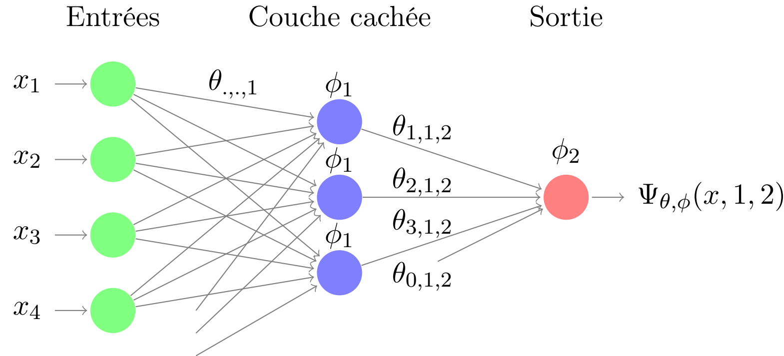
avec $\Psi_{\theta,\phi}(x, j, k)$ *défini récursivement comme étant :*

$$\Psi_{\theta,\phi}(x, j, k) = \phi_k\Big(\theta_{0,j,k} + \sum_{i=1}^{h_{k-1}} \theta_{i,j,k}\Psi_{\theta,\phi}(x, j, k-1)\Big),$$

$$\Psi_{\theta,\phi}(x, j, 1) = \phi_1\Big(\theta_{0,j,1} + \sum_{i=1}^{n} \theta_{i,j,1}x_i\Big),$$

où $k \in \{1, \ldots, m+1\}$ *indice les couches,* $j \in \{1, \ldots, h_k\}$ *indice les neurones des couches et* i *indice les poids d'un perceptron. Le nombre de paramètres* $|\theta| = (n+1) \cdot h_1 + \sum_{i=2}^{m}(h_{i-1} + 1) \cdot h_i + h_m + 1$.

Pour éviter de calculer plusieurs fois l'activation d'un même neurone (dans cette formule par récurrence), il est habituel de faire un calcul couche par couche en partant de la couche d'entrée et en gardant en mémoire chaque activation pour la couche suivante.



Dans cet exemple, la fonction calculée est la suivante :

$$\Psi_{\theta,\phi}(x) = \phi_2\left(\theta_{0,1,2} + \sum_{j=1}^{3}\left(\theta_{j,1,2}\phi_1\Big(\theta_{0,j,1} + \sum_{i=1}^{4}\theta_{i,j,1}x_i\Big)\right)\right).$$

FIGURE 1.5 – Un perceptron multicouche avec $X = \mathbb{R}^4$ et une couche cachée de 3 neurones.

1.2 Optimisation des paramètres

Nous allons maintenant nous intéresser aux manières d'apprendre les paramètres des précédents modèles. Pour cela, il faut définir en premier lieu la qualité d'un modèle. Pour mesurer la qualité, on utilise un critère ; on parle aussi d'*objectif*. L'un des critères les plus utilisés en

régression se nomme MSE [3] ou erreur quadratique moyenne.

Définition 1.6 (Mean Square Error)**.** *Soit un modèle* $\Psi : X \to Y$ *et une base de données étiquetées* $(x_i, y_i)_{\{1,\ldots,n\}} \in X^n \times Y^n$*, le critère empirique MSE définit la qualité du modèle par :*

$$MSE(\Psi) = \frac{1}{n}\sum_{i=1}^{n} ||\Psi(x_i) - y_i||_2^2,$$

où $||.||_2$ *représente la norme L2 (aussi appelée distance euclidienne).*

Plus la MSE est petite, meilleur est le modèle. Pour un modèle paramétrique Ψ_θ, les paramètres optimaux θ^*_{MSE} sont donc :

$$\theta^*_{\text{MSE}} = \arg\min_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^{n} ||\Psi_\theta(x_i) - y_i||_2^2.$$

Pour résoudre ce problème d'optimisation, il existe de nombreuses méthodes ayant chacune des hypothèses et garanties différentes. La table 1.1 n'est pas exhaustive, elle décrit les méthodes les plus couramment utilisées dans ce domaine.

Lorsqu'une solution analytique n'existe pas, il convient de passer aux méthodes approchées. Par exemple avec des méthodes itératives : en partant d'une solution aléatoire θ_0, ces méthodes observent la qualité de la solution et, de manière itérative, en calculent une prochaine, θ_1, souhaitée meilleure. La *descente de gradient* (Cauchy, 1847) en fait partie, elle utilise le calcul du gradient $\nabla_\theta \text{MSE}(\Psi_\theta)$ sur des modèles paramétriques. L'optimisation *black-box* (Jones *et al.*, 1998) n'utilise pas l'information du gradient, mais seulement la valeur du critère (Section 1.2.3 page 19). Ces méthodes sont donc facilement applicables sans condition sur le modèle. Néanmoins, elles peuvent se révéler plus lentes à trouver une solution par rapport aux méthodes qui reposent sur le gradient, car elles sont moins informées (sauf dans les cas où le gradient est inutile).

	Modèles	
Méthodes	linéaire ou fonctions de base fixées	quelconque (non linéaire)
solution analytique	existe	n'existe pas
descente de gradient	optimum	optimum local si gradient calculable
méthode de Newton	optimum	optimum local si Hessien calculable
optimisation *black-box*	optimum local sans calcul de gradient	

Table 1.1 – Récapitulatif des méthodes d'optimisation et leurs hypothèses. Un gradient calculable pour un critère J signifie que $\nabla_\theta J(\Psi_\theta)$ existe et est disponible.

1.2.1 Descente de gradient

Nous allons passer plus de temps à détailler la descente de gradient, car c'est cette méthode que nous utiliserons le plus au long de cette thèse et celle aussi à l'origine de l'optimisation du

3. Pour *Mean Square Error*

calcul pour les *perceptrons multicouches*. La raison de son utilisation étant sa capacité de mise à l'échelle. En effet, le calcul de la prochaine solution est linéaire : il se fait en $O(n)$ opérations où n est le nombre de paramètres. Par rapport aux méthodes d'optimisation *black-box*, la descente de gradient est plus informée, la rendant plus efficace lorsque n est grand et que le gradient est pertinent. Néanmoins, comme toutes méthodes dépendantes du gradient, elles sont sujettes à rester bloquées dans un optimal local et sont lentes lorsque des plateaux surviennent dans le gradient.

Algorithme 1.1 (Descente de gradient)**.** *Étant donné un modèle paramétrique $\Psi_\theta : X \rightarrow Y$ avec un critère $J(\Psi_\theta)$ continu et dérivable à minimiser, la descente de gradient met à jour θ par :*

$$\theta_{t+1} \leftarrow \theta_t - \alpha \frac{\partial J(\Psi_{\theta_t})}{\partial \theta_t},$$

où α est un taux d'apprentissage.

Au départ, θ_0 est initialisé aléatoirement. Le taux d'apprentissage définit la vitesse de déplacement des paramètres. Il peut changer à chaque itération. Un déplacement trop rapide peut entraîner une divergence des paramètres et un déplacement trop lent nécessitera plus d'itérations. Selon le taux d'apprentissage choisi, la convergence vers un minimum local est garantie. Dans cette thèse, nous utiliserons également une seconde notation plus compacte :

$$\Delta\theta_t = -\frac{\partial J(\Psi_{\theta_t})}{\partial \theta_t},$$

où $\Delta\theta_t$ représente la direction de modification à prendre par θ_t indépendamment du taux d'apprentissage. Selon qu'on cherche à minimiser ou à maximiser le critère, on parle de *descente* ou de *montée* de gradient, d'où la présence du signe moins dans la formule précédente puisqu'il s'agit d'une *descente* de gradient.

La difficulté réelle de ces méthodes réside dans la bonne estimation du gradient. Le nombre de données, utilisées pour calculer J, détermine la version de descente de gradient utilisée. La *descente de gradient stochastique* se sert d'un sous-ensemble tiré aléatoirement parmi les échantillons disponibles. On utilise également le terme de *mini-batch*, ou encore *online* lorsque qu'un seul échantillon est utilisé. Ainsi, le calcul du critère J, et donc de son gradient, est moins coûteux en temps, car il est seulement calculé sur un sous-ensemble. La version *stochastique* permet potentiellement de sortir des optima locaux. Tandis que la version *batch*, qui utilise toutes les données disponibles, calcule une meilleure approximation du gradient.

Suppression du taux d'apprentissage Définir le taux d'apprentissage est un des inconvénients principaux des descentes de gradient. Pour pallier ce problème, il existe des algorithmes permettant de ne pas définir le taux d'apprentissage *a priori* (Igel et Hüsken, 2000; Riedmiller et Braun, 1992). Le taux d'apprentissage existe bel et bien, mais le concepteur n'a pas besoin de le définir : il est défini par l'algorithme. L'algorithme Rprop[4] permet cela (Algorithme 1.2 page suivante), de plus, au lieu d'avoir un seul taux d'apprentissage scalaire, α devient un vecteur définissant un taux d'apprentissage par paramètre, accélérant ainsi l'apprentissage (LeCun *et al.*, 2012). Le principe de Rprop est de regarder si le signe du gradient a changé par rapport au précédent, s'il n'a pas changé, le taux d'apprentissage du paramètre augmente ; dans le cas contraire, il diminue. Pour avoir une certaine stabilité, il est ainsi préférable d'utiliser une partie importante de l'échantillon disponible.

4. Pour *resilient backpropagation*

Algorithme 1.2 (Resilient backpropagation iRPROP-)

Données : Un modèle paramétrique Ψ_θ, un critère J à minimiser, $(\Delta_{\max}, \Delta_{\min}) \in \mathbb{R}_+^2$ la variation maximale et minimale sur une itération, $(\Delta^+, \Delta^-) \in \mathbb{R}_+^2$ l'augmentation et la diminution de la variation sur une itération, et $t \in \mathbb{N}$ l'itération

pour $\theta_i \in \Theta$ **faire**

$z \leftarrow \nabla_{\theta_i} J^{(t-1)} \cdot \nabla_{\theta_i} J^{(t)}$

$$\Delta\theta_i{}^{(t)} \leftarrow \begin{cases} \min\big(\Delta\theta_i{}^{(t-1)}\Delta^+, \Delta_{\max}\big) & \text{si } z < 0 \\ \max\big(\Delta\theta_i{}^{(t-1)}\Delta^-, \Delta_{\min}\big) & \text{si } z > 0 \end{cases}$$

si $z < 0$ **alors**

$\nabla_{\theta_i} J^{(t)} \leftarrow 0$

fin

$\Delta\theta_i{}^{(t)} \leftarrow -\Delta\theta_i{}^{(t)} \text{ signe}(\nabla_{\theta_i} J^{(t)})$

$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} + \Delta\theta_i{}^{(t)}$

fin

Taux d'apprentissage adaptatif Il existe de nombreuses améliorations possibles à la *descente de gradient stochastique* classique, nous allons en détailler une qui est utilisée dans l'état de l'art et au cours de cette thèse. Il s'agit d'ADAM [5], l'algorithme 1.3 estime la moyenne et la variance du gradient pour chaque paramètre de façon géométriquement décroissante par rapport au temps (Kingma et Ba, 2015). On parle aussi de *momentum* qui ajoute de l'inertie à la mise à jour. Grâce à ces estimations, la mise à jour des paramètres est plus lisse et la variance réduite. Néanmoins, dans cet algorithme, il faut définir un taux d'apprentissage global α. Cet algorithme est utilisé en *deep learning* (Goodfellow *et al.*, 2016) et en renforcement (Lillicrap *et al.*, 2015).

Algorithme 1.3 (Adaptive Moment Estimation Optimizer)

Données : Un modèle paramétrique Ψ_θ, un critère J à minimiser, $\alpha \in \mathbb{R}$ le taux d'apprentissage, $(\beta_1, \beta_2) \in \mathbb{R}_+^2$ taux d'apprentissage pour l'estimation exponentiellement décroissante de la moyenne et de la variance, $\zeta \in \mathbb{R}$ décalage pour éviter une division par 0 et $t \in \mathbb{N}$ l'itération

pour $\theta_i \in \Theta$ **faire**

$\mu_i^{(t)} \leftarrow \beta_1 \mu_i^{(t-1)} + (1-\beta_1) \nabla_{\theta_i} J^{(t)}$

$\sigma_i^{(t)} \leftarrow \beta_2 \sigma_i^{(t-1)} + (1-\beta_2) \nabla_{\theta_i} J^{(t)2}$

$$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} - \alpha \frac{\sqrt{1-\beta_2}}{1-\beta_1} \frac{\mu_i^{(t)}}{\sqrt{\sigma_i^{(t)}} + \zeta}$$

fin

5. Pour *Adaptive Moment Estimation Optimizer*

1.2.2 Méthodes de second ordre

Les méthodes de Newton sont plus informées que la *descente de gradient* : elles utilisent la dérivée seconde. Elles ont donc tendance à converger vers une solution avec un nombre plus réduit d'itérations.

Algorithme 1.4 (Méthode de Newton pour l'optimisation de paramètres)**.** *Étant donné un modèle paramétrique $\Psi_\theta : X \to Y$ avec un critère $J(\Psi_\theta)$ continu et deux fois dérivables à minimiser, la méthode de Newton met à jour θ par :*

$$\theta_{t+1} \leftarrow \theta_t - \frac{\nabla_{\theta_t} J(\Psi_{\theta_t})}{\boldsymbol{H}_{\theta_t} J(\Psi_{\theta_t})},$$

où $\boldsymbol{H}_{\theta_t}$ est la matrice hessienne.

Néanmoins, le calcul d'une itération est plus coûteux à cause de la matrice hessienne qui nécessite $O(n^2)$ opérations en nombre de paramètres. Ces méthodes sont utilisées par les algorithmes *natural gradient* (Kakade, 2002; Peters et Schaal, 2006) que nous détaillerons à la section 2.3.3 page 51.

1.2.3 Optimisation *black-box*

L'optimisation *black-box* (Jones *et al.*, 1998) et les métaheuristiques sont généralement des algorithmes stochastiques itératifs qui ne nécessitent que le calcul d'un critère J. Nous nous intéressons ici à l'algorithme CMA-ES[6] : c'est une méthode appartenant à la catégorie des *stratégies d'évolution* au sein des *algorithmes évolutionnistes* (Hansen et Ostermeier, 2001). CMA-ES est l'une des meilleures métaheuristiques pour les problèmes continus (Auger et Hansen, 2005; Stulp et Sigaud, 2013). Pour cette raison, nous l'avons utilisée à titre de comparaison au cours de cette thèse. Nous ne détaillerons que brièvement son fonctionnement, car nos recherches ne se sont pas concentrées sur cette méthode, en raison de sa plus faible capacité de mise à l'échelle.

Comme tout *algorithme évolutionniste*, CMA-ES maintient à jour une population de solutions. À partir d'un ensemble de *parents*, à chaque *génération*, un ensemble d'*enfants* est produit à partir des meilleurs parents. Au sein d'une génération, la fonction de coût $J(\Psi_\theta)$ est calculée plusieurs fois : autant de fois que la taille de la population. En supposant que $\theta \in \mathbb{R}^n$, l'équation qui permet de produire m enfants pour la prochaine génération, aussi appelée *opérateur de mutation*, est la suivante :

$$(\theta_i)^{t+1}_{\{1,\ldots,m\}} \sim \mu^t + \alpha^t \mathcal{N}(0, C^t),$$

où $\mathcal{N} : \mathbb{R} \times \mathbb{R}^+_* \to \mathbb{R}$ est la loi normale, $\mu^t \in \mathbb{R}^n$ est un vecteur représentant la meilleure solution trouvée jusqu'à présent, $\alpha^t \in \mathbb{R}_+$ est un taux d'apprentissage, $C^t \in \mathbb{R}^{n \times n}$ est la matrice de covariance de la loi normale, déterminant la forme de la distribution, et t la génération. La façon dont μ^t, α^t et C^t sont mis à jour ne sera pas détaillée ici, voir Hansen et Ostermeier (2001) pour plus d'informations. Remarquons que le coût de cet algorithme est en $O(n^2)$, en nombre de paramètres du modèle, compte tenu du calcul de la matrice de covariance, d'où sa difficulté de mise à l'échelle.

6. Pour *Covariance Matrix Adaptation Evolution Strategy*

1.2.4 Application aux réseaux de neurones

L'application aux réseaux de neurones de la descente de gradient donne naissance à l'algorithme de rétropropagation du gradient (LeCun, 1985). Il permet de calculer le gradient de l'erreur pour chaque neurone, de la dernière couche vers la première. Plus un paramètre contribue à engendrer une erreur importante, plus il se verra modifié. Pour un critère J, l'algorithme 1.5 cherche à définir le terme $\nabla_\theta J(\Psi_\theta)$ pour mettre à jour θ. Pour les paramètres appartenant à la couche de sortie la solution est triviale, mais pour les poids intermédiaires il faut utiliser le théorème de dérivation des fonctions composées.

Algorithme 1.5 (Rétropropagation du gradient)

Données : Un perceptron multicouche $\Psi_{\theta,\sigma} : X \to Y$ avec σ différenciables, un critère $J : Y \to \mathbb{R}$ à minimiser, $\alpha \in \mathbb{R}$ le taux d'apprentissage et un échantillon $(x_i)_{\{1,..,n\}} \in X^n$ utilisé par J
Passe avant : Calculer les sorties $\Psi_{\theta,\sigma}(x_i)$ de l'entrée vers la sortie (couche par couche)
Évaluer : Calculer le critère J à partir $\Psi_{\theta,\sigma}(x)$
Passe arrière : Calculer $\nabla_{\theta_t} J(\Psi_{\theta_t})$ de la sortie vers l'entrée (couche par couche)
Mettre à jour : $\theta_{t+1} \leftarrow \theta_t - \alpha \nabla_{\theta_t} J(\Psi_{\theta_t})$

Pour appliquer l'algorithme CMA-ES aux réseaux de neurones, il suffit de lui faire modifier la matrice des poids du réseau en la transformant en vecteur. Aucune de ces approches n'a d'influence sur la structure du réseau.

1.3 Limitations

Il existe plusieurs limitations à l'utilisation de modèles pour approcher une fonction. Nous allons maintenant aborder les plus importantes.

1.3.1 Surapprentissage ou généralisation

Lors de l'optimisation de paramètres, il est possible qu'un phénomène de *surapprentissage* survienne (Sarle, 1996) : il s'agit d'une mauvaise généralisation sur des données qui n'ont pas été présentées au modèle (Figure 1.6 page ci-contre). La source du *surapprentissage* peut provenir essentiellement de deux origines : une trop grande liberté laissée au modèle (trop de neurones, de couches, de fonctions de base, etc.) ou d'un temps d'apprentissage trop long sur certaines données. On dit d'un modèle qui a surappris qu'il a réussi à apprendre le bruit présent dans l'échantillon et non la relation sous-jacente entre l'entrée et la sortie.

Validation croisée Pour détecter ce phénomène de *surapprentissage*, il est possible d'utiliser la *validation croisée* (Kohavi, 1995). Dans sa version la plus simple, il s'agit de découper l'échantillon, sur lequel J est calculé, en deux sous-ensembles : *l'ensemble d'apprentissage* et *l'ensemble de test*. Les échantillons appartenant à *l'ensemble d'apprentissage* seront utilisés pour mettre à jour les paramètres du modèle. Tandis que ceux appartenant à *l'ensemble de test* ne serviront qu'à évaluer le modèle sans le modifier (Figure 1.7 page 22).

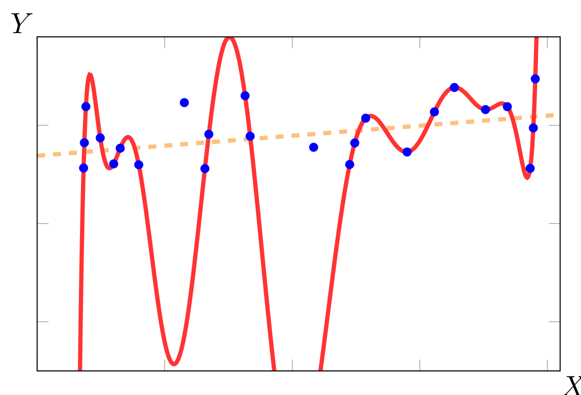


FIGURE 1.6 – Surapprentissage ou généralisation. Exemple de deux modèles $\Psi : \mathbb{R} \to \mathbb{R}$ qui doivent apprendre les points bleus via régression. Le modèle orange (en pointillé) ne fait pas de surapprentissage et dispose d'une bonne capacité de généralisation, tandis que le modèle rouge est très bon sur les données d'entrées (surapprentissage) mais incapable de généraliser.

Régularisation La régularisation permet de modifier le critère à optimiser de façon à privilégier les modèles avec une complexité plus simple. La justification de cette pratique provient du principe du *rasoir d'Ockham*. En pratique, elle permet de limiter de manière efficace le surapprentissage (Girosi *et al.*, 1995). Mathématiquement, on ajoute un terme au critère J à optimiser :

$$J(\Psi_\theta) + \beta \mathcal{R}(\Psi_\theta),$$

où $\mathcal{R}$ est le terme de régularisation, et β un méta paramètre permettant d'équilibrer les deux termes. Le terme de régularisation peut prendre de nombreuses formes (Bishop, 2006) :

- Par exemple, celle d'une distance L1 sur les paramètres : $\mathcal{R}(\Psi_\theta) = ||\theta||_1$. De cette manière, on privilégie les ensembles de poids épars (avec plusieurs paramètres nuls).
- On peut aussi utiliser une distance L2 : $\mathcal{R}(\Psi_\theta) = ||\theta||_2$. Cette distance empêche les paramètres de devenir trop grands. Dans les réseaux de neurones, la régularisation L2 peut être mise en œuvre par *weight decay* : il suffit de faire décroître tous les paramètres à chaque itération.

Nous allons également détailler deux formes de régularisation développées spécifiquement pour les réseaux de neurones. Nous avons utilisé la première au cours de cette thèse et introduisons la seconde pour faciliter la compréhension de l'une de nos contributions. Les deux sont mises en place par l'insertion de nouvelles couches à l'intérieur du réseau, où chaque neurone n'est connecté qu'à un seul neurone de la prochaine couche (Figure 1.8 page suivante). Ces couches s'insèrent après la fonction d'activation d'une couche ou directement après la couche d'entrée. Nous allons utiliser les notations suivantes pour décrire les deux approches : v_i représente la sortie après régularisation, z_i l'entrée de la couche précédente et ϑ_i les paramètres internes utilisés par ces méthodes. Nous distinguons ϑ de θ (les paramètres des perceptrons présents dans le réseau de neurones) car ces paramètres sont appris ou fournis indépendamment de la méthode d'optimisation utilisée. Par exemple, les paramètres θ peuvent être appris par rétropropagation (Algorithme 1.5 page précédente) mais pas ϑ. Remarquons qu'en utilisant de telle couche le modèle n'est plus forcément stationnaire.

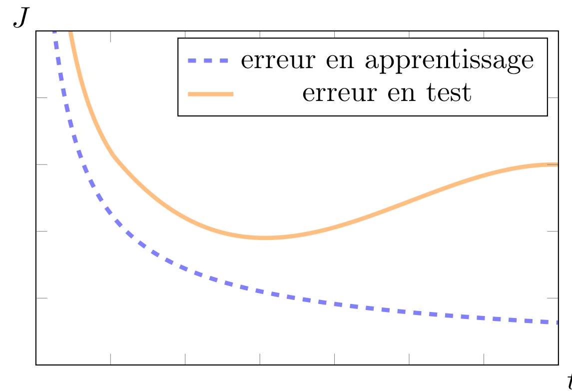


FIGURE 1.7 – Exemple d'un modèle sujet au surapprentissage. Grâce à l'erreur calculée sur l'ensemble de test, il est possible de stopper l'apprentissage au moment le plus propice ; ce qui ne serait pas possible en connaissant uniquement l'erreur en apprentissage qui diminue continuellement jusqu'à convergence.

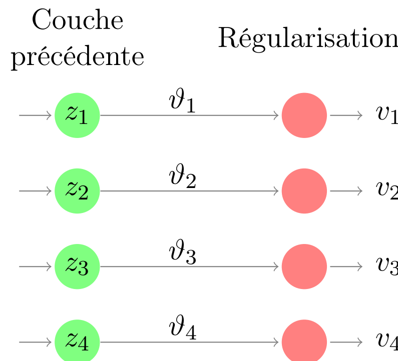


FIGURE 1.8 – Couche de régularisation : *batch normalization* ou *dropout*.

Batch Normalization La *batch normalization* est une technique de régularisation (Ioffe et Szegedy, 2015), elle permet un apprentissage plus rapide grâce à sa stabilité. Cela permet également d'utiliser des taux d'apprentissages plus importants. Sa stabilité provient de l'utilisation des variables centrées réduites (d'où le terme de normalisation). Pour un *mini-batch* (sous-ensemble de l'échantillon total), nous notons z_i^k pour l'activation du $i^{\text{ème}}$ neurone de la couche précédente à l'échantillon k. La sortie de la couche de *batch normalization* se calcule de la façon suivante :

$$\begin{aligned}
\mu_i &\leftarrow \frac{1}{K}\sum_k z_i^k, \\
\sigma_i &\leftarrow \frac{1}{K}\sum_k (z_i^k - \mu_i), \\
v_i^k &\leftarrow \vartheta_{i,1}\frac{z_i^k - \mu_i}{\sqrt{\sigma_i + \zeta}} + \vartheta_{i,2},
\end{aligned}$$

où ζ est un méta paramètre pour éviter les divisions par 0. Le nombre de paramètres par neurone est donc de 2. Remarquons la similarité avec l'algorithme 1.3 page 18. La façon dont les

paramètres ϑ sont appris n'est décrite ici, mais se trouve dans les travaux de Ioffe et Szegedy (2015).

Dropout L'idée de la régularisation par *dropout* (Srivastava *et al.*, 2014) consiste à désactiver certains neurones dans le réseau. Plutôt que de se reposer sur les connexions voisines, *dropout* permet à chaque neurone d'apprendre une information pertinente. L'activation stochastique d'une unité se calcule simplement par :

$$\begin{aligned} \mathbb{P}(v_i = z_i) &= \vartheta_i, \\ \mathbb{P}(v_i = 0) \ &= 1 - \vartheta_i. \end{aligned}$$

Dans cette technique le paramètre ϑ est généralement global et unique (partagé par tous les neurones), il s'agit d'un métaparamètre. On peut néanmoins en définir un par couche pour désactiver moins d'unités dans les premières couches du réseau.

1.3.2 Biais ou variance

La seconde limitation, bien connue en statistique, est celle du dilemme biais-variance (Fisher, 1925). On dit d'un estimateur $\hat{x}$ de x qu'il est non biaisé si $\mathbb{E}[\hat{x}] = x$. La variance caractérise la dispersion des valeurs par rapport à la moyenne. En effet, un estimateur non biaisé reflète bien la moyenne, mais les échantillons qu'il produit peuvent être plus ou moins dispersés autour de cette moyenne. Aussi, on peut préférer choisir un autre estimateur admettant une faible erreur sur la moyenne en ayant moins de variance.

En apprentissage supervisé, ce dilemme est aussi présent (Geman *et al.*, 1992), il est lié au problème du surapprentissage lors du choix du modèle. Lorsqu'on tente d'optimiser un critère, l'erreur provient de deux facteurs : le biais et la variance (Figure 1.9). Le biais résulte d'un modèle contenant des hypothèses erronées, par exemple, si l'on essayait d'apprendre la forme d'un cercle avec un modèle linéaire. On parle alors de *sous-apprentissage*. La variance résulte des variations présentes dans l'échantillon d'apprentissage. Lorsqu'elle est élevée dans un modèle, cela signifie qu'il a appris le bruit aléatoire présent dans l'échantillon et ainsi qu'il a surappris (Figure 1.6 page 21).

1.4 Conclusion

Au cours de ce chapitre, nous avons abordé les différences entre l'apprentissage supervisé et l'apprentissage par renforcement. Nous avons détaillé plusieurs modèles de représentations de fonctions avec leurs caractéristiques et différentes façons de les apprendre par régression. En particulier, nous avons justifié pour quelles raisons nous choisissons les réseaux de neurones comme modèles de représentation de fonctions dans cette thèse :
- la possibilité de traiter des données continues,
- la mise à l'échelle,
- le pouvoir de représentation,
- la capacité de construire des représentations avec peu de connaissances humaines requises.

Nous avons introduit le dilemme *biais-variance* qui nous suivra et nous guidera dans tous les algorithmes présentés au cours de cette thèse. Nous nous servirons de ces modèles dans le prochain chapitre dédié à l'apprentissage par renforcement. Nous utiliserons la rétropropagation, CMA-ES, et la *batch normalization* pour optimiser nos réseaux de neurones durant nos validations expérimentales.

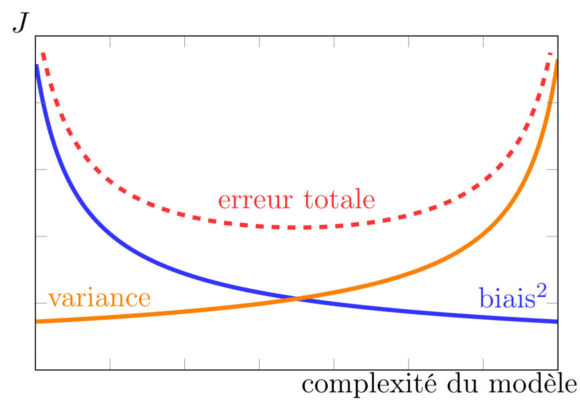


FIGURE 1.9 – Illustration du compromis biais-variance en fonction de la complexité du modèle. Plus un modèle est complexe, moins son biais est élevé, mais plus sa variance est élevée (surapprentissage). À l'inverse, lorsqu'un modèle est trop simple, il peut souffrir de sous-apprentissage. Le modèle à la complexité idéale minimise les deux erreurs.

Chapitre 2

Apprentissage par renforcement

Dans ce chapitre, nous allons expliquer le cadre de l'apprentissage par renforcement ; en particulier, la façon dont il se découpe en deux sous-problèmes : l'évaluation d'une stratégie et son amélioration. Nous commencerons par un cadre restreint avec beaucoup d'hypothèses (environnement discret et connu) pour aller vers le général (environnement continu et inconnu).

2.1 Formalisation

L'apprentissage par renforcement, au sens général, est un cadre formel qui modélise des problèmes décisionnels séquentiels. Au sein de ce cadre, un agent [7] apprend à prendre des décisions optimales en interagissant avec l'environnement (Narendra et Thathachar, 1974; Bush et Mosteller, 1955). Lorsqu'il effectue une action, l'état du système change et l'agent reçoit une valeur scalaire, appelée récompense, qui encode les informations sur la qualité de la transition (voir Figure 2.1).

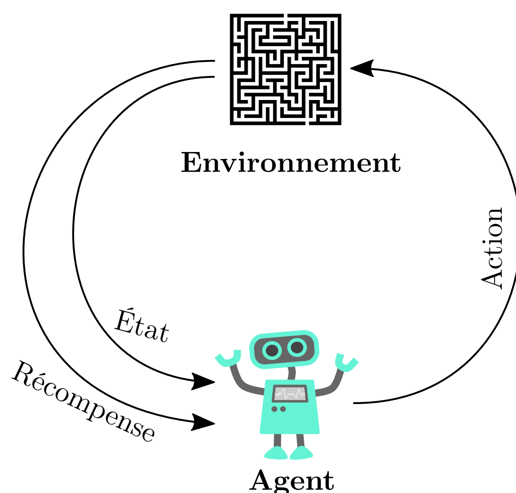


FIGURE 2.1 – Illustration du cadre général de l'apprentissage par renforcement. Adapté depuis Wikipédia (2017b).

7. entité capable de percevoir son environnement grâce à des capteurs et d'agir sur celui-ci à travers des effecteurs afin de réaliser des buts

L'une des premières hypothèses faites sur l'environnement dans cette thèse est celle de Markov.

Définition 2.1 (Séquence markovienne)**.** *Un état s_t de l'environnement au temps t est markovien si et seulement si :*

$$\mathbb{P}(s_{t+1}|s_0, s_1, ..., s_t) = \mathbb{P}(s_{t+1}|s_t).$$

Autrement dit, l'ajout de l'historique n'apporte pas d'information : les conditions de l'état suivant ne dépendent que de l'état courant. Cette hypothèse n'est pas obligatoire dans le cadre général de l'apprentissage par renforcement. Néanmoins, elle permet de se reposer sur un paradigme théorique solide et bien étudié : le formalisme des processus décisionnels de Markov (MDP [8]) (Puterman, 1994).

Définition 2.2 (Processus décisionnel de Markov)**.** *Un MDP à états et actions discrets, stationnaire, avec des récompenses déterministes et un espace d'action indépendant de l'état, est formellement défini comme étant un n-uplet $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ où :*

- $\mathcal{S}$ *est l'ensemble des états de cardinal $|\mathcal{S}|$,*
- $\mathcal{A}$ *est l'ensemble des actions de cardinal $|\mathcal{A}|$,*
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0;1]$, $s, a, s' \mapsto \mathbb{P}(s'|s,a)$ *est la fonction de transition stochastique qui représente la probabilité d'atteindre l'état s' à partir de l'état s en exécutant l'action a,*
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, $s, a, s' \mapsto R(s,a,s')$ *est la fonction de récompense.*

Dans le cas où les états et les actions appartiendraient à un ensemble discret, les MDP peuvent être représentés par un graphe où les nœuds sont des états et les actions des arcs pour naviguer entre les états comme sur la Figure 2.2 page ci-contre.

Il convient de différencier l'apprentissage par renforcement général [9], qui définit un cadre où un agent apprend à partir d'un signal de renforcement, de l'apprentissage par renforcement classique (RL [10]) définit au sens de Sutton et Barto (1998) où ils font l'hypothèse de Markov et définissent un MDP où l'agent n'a pas accès à la fonction de transition T et la fonction de récompense R (voir Figure 2.3 page 28). Dans la suite du document, lorsque nous utiliserons le terme d'apprentissage par renforcement ou RL, nous ferons référence à la définition de Sutton.

Dans cette thèse, nous nous intéressons plus particulièrement à une généralisation des MDP aux espaces continus d'état et d'action.

8. Pour *Markov Decision Process*
9. décrit dans le premier paragraphe et sur la Figure 2.1 page précédente
10. Pour *Reinforcement Learning*

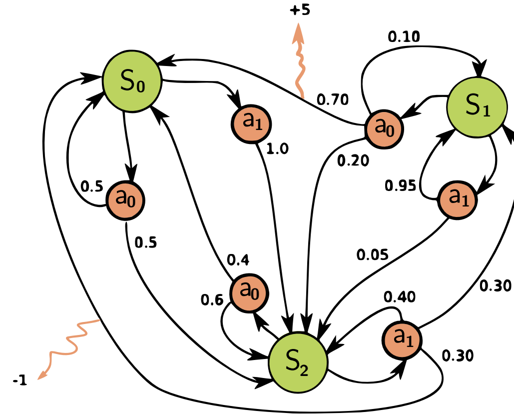


Figure 2.2 – Illustration d'un MDP à espace discret. Ce MDP à espace discret contient 3 états (S_0, S_1, S_3) représentés par des cercles verts et 2 actions (a_0, a_1) en orange. La fonction de récompense est non nulle sur deux transitions (flèches jaunes). Lorsqu'il est dans un état, l'agent décide de l'action à entreprendre pour en atteindre un autre. Reproduction depuis Wikipédia (2017d).

Définition 2.3 (Processus décisionnel de Markov continu en espace)**.** *Un MDP à états et actions continus, stationnaire, avec des récompenses déterministes et un espace d'action indépendant de l'état, est formellement défini comme étant un n-uplet* $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ *où :*

— $\mathcal{S} = \mathbb{R}^{d_\mathcal{S}}$ *est l'espace des états de dimension* $d_\mathcal{S}$*,*

— $\mathcal{A} = \mathbb{R}^{d_\mathcal{A}}$ *est l'espace des actions de dimension* $d_\mathcal{A}$*,*

— $T: \begin{array}{l} \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^+ \\ s, a, s' \mapsto T(s'|s,a) \end{array}$ *est la fonction de transition stochastique qui représente la densité de probabilité d'atteindre l'état* s' *à partir de l'état* s *en exécutant l'action* a*,*

— $R: \begin{array}{l} \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R} \\ s, a, s' \mapsto R(s,a,s') \end{array}$ *est la fonction de récompense.*

Par définition, $\int_\mathcal{S} T(s'|s,a)ds' = 1$. Dans le cas continu, la figure 2.2 contiendrait une infinité d'arcs vers une infinité de nœuds.

2.1.1 Politique et critère d'optimisation

Résoudre un MDP, c'est trouver une stratégie optimisant un critère donné qui sera adoptée par l'agent. Une *politique* est définie comme étant une fonction qui permet de choisir une action dans un état donné. Pour les politiques stochastiques, nous utiliserons la notation $\pi(a|s)$.

Définition 2.4 (Politique stochastique)**.** *Une politique stochastique* π *associe une densité de probabilité sur l'espace des actions* $\mathcal{A}$ *à un état* $s \in \mathcal{S}$ *:*

$$\pi: \begin{array}{l} \mathcal{S} \times \mathcal{A} \to \mathbb{R}^+ \\ s, a \mapsto \pi(a|s) \end{array} \qquad \textit{avec} \int_\mathcal{A} \pi(a|s)da = 1.$$

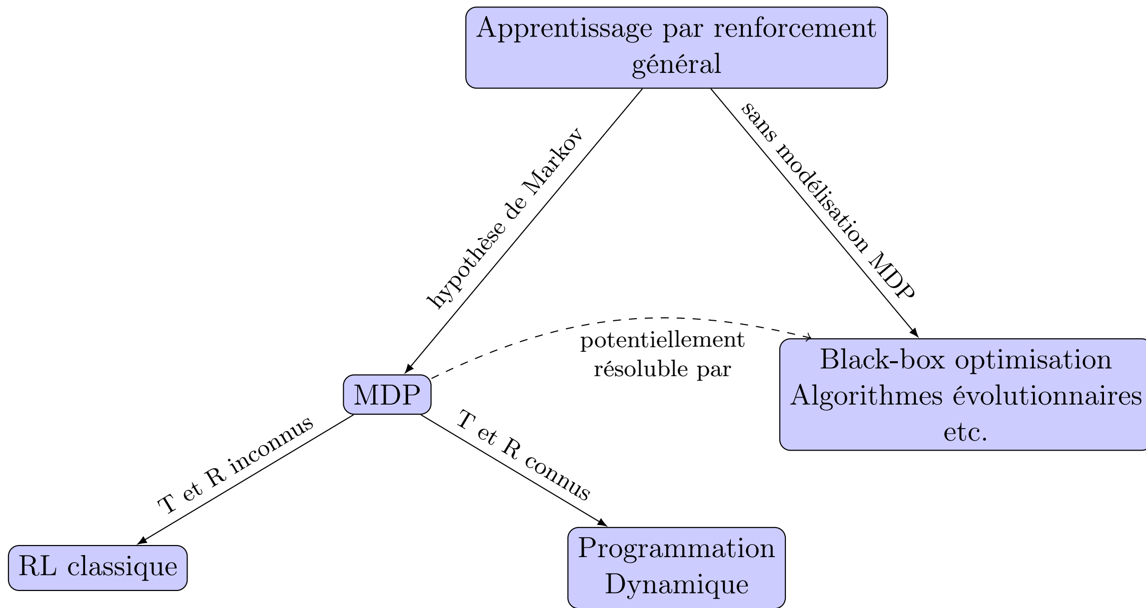


FIGURE 2.3 – Différence entre l'apprentissage par renforcement, au sens général, avec l'apprentissage par renforcement classique de Sutton.

Pour les politiques déterministes, nous utiliserons la notation $\pi(s)$. Dans ce cas, l'action choisie est toujours la même pour un état s.

Définition 2.5 (Politique déterministe)**.** *Une politique déterministe π associe un état $s \in \mathcal{S}$ à une action $a \in \mathcal{A}$:*

$$\begin{aligned} \pi :\ & \mathcal{S} \to \mathcal{A} \quad . \\ & s \mapsto \pi(s) \end{aligned}$$

Pour pouvoir définir un ordre total [11] sur l'ensemble des politiques afin de les comparer, il est nécessaire d'utiliser un critère. Nous ne nous intéressons qu'au seul critère *γ-pondéré* qui est l'espérance de la somme des récompenses décomptées par γ (aussi appelé *critère actualisé*).

Définition 2.6 (Critère γ-pondéré)**.** *Étant donné un MDP à espaces continus et une politique π, le critère γ-pondéré permet de quantifier la politique π par un scalaire :*

$$J(\pi) = \int_{\mathcal{S}} T_0(s_0)\ \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \ \Big|\ \pi, s_0\Big] ds_0,$$

où $0 \leq \gamma \leq 1$ représente l'importance des récompenses futures et T_0 la densité de probabilité définie sur l'état initial.

On utilisera le terme *étape* (*step*) pour désigner une transition à l'instant t et *épisode* (*episode*) pour désigner un ensemble d'étapes. Il est possible d'avoir $\gamma = 1$ dans le cas *épisodique* où il existe un (ou plusieurs) état absorbant qui terminent l'épisode. Les intérêts du critère *γ-pondéré* sont multiples : il converge lorsque le temps tend à l'infini, simplifiant son étude mathématique, et il est possible de choisir un compromis entre récompenses immédiates et futures.

11. relation binaire respectant la transitivité, l'anti-symétrie et la totalité

Définition 2.7 (Ordre total sur espace des politiques)**.** *On dit qu'une politique π_a domine une politique π_b, selon le critère J si et seulement si :* $J(\pi_a) \geq J(\pi_b) \equiv \pi_a \geq \pi_b$.

Ainsi, l'objectif principal est de trouver une politique qui maximise le critère 2.6 page ci-contre. En général, il n'y a pas d'unicité de la politique optimale. Une politique optimale n'est dominée strictement par aucune autre.

Définition 2.8 (Politique optimale)**.** *Une politique optimale π^* selon le critère J satisfait la contrainte suivante :*

$$\pi^* \in \arg\max_{\pi} J(\pi).$$

Nous nous restreignons ici au cas des *politiques stationnaires*, ce qui signifie qu'on ne cherche pas une politique qui change au cours d'un *épisode*.

Définition 2.9 (Politique stationnaire)**.** *Une politique π est dite stationnaire ou markovienne si elle ne dépend pas du temps :*

$$\forall t \in \mathbb{N}, \pi_t = \pi.$$

Néanmoins, cela ne nous interdit pas de changer la politique durant la *phase d'apprentissage* au sein d'un même *épisode* (ces méthodes sont dites *online* ou *incremental*). Scherrer et Lesner (2012) se sont intéressés à la recherche de politiques non stationnaires au sein d'un MDP stationnaire. Enfin, l'exécution des politiques se fera à des temps discrets, et à intervalle constant défini *a priori*. Doya (2000) a proposé plusieurs méthodes pour aborder les MDP en temps continu.

2.1.2 Observabilité partielle

Dans plusieurs environnements, il arrive que l'agent n'ait pas accès à l'état complet de celui-ci. Dans ce cas, on parle de processus décisionnels de Markov partiellement observables (POMDP [12]). Étant donné que nous utiliserons des simulateurs physiques pour les environnements durant notre validation expérimentale, toutes les variables internes de ce simulateur ne seront pas accessibles. L'agent n'en perçoit qu'une représentation partielle : la plupart du temps, les angles et les vitesses angulaires des articulations. Dans certains environnements simples, ces informations suffisent à définir un vrai MDP. Tandis que dans les environnements plus compliqués avec plus de dimensions, il est difficile de déterminer si c'est toujours le cas. Bien que la résolution des POMDP ne soit pas une problématique essentielle de cette thèse, nous allons néanmoins définir nos politiques sur l'espace des observations. Même si l'hypothèse de Markov n'est alors plus forcément vérifiée, des observations similaires pouvant être obtenues dans des états différents, nous traiterons ces problèmes comme si c'était des MDP. En toute rigueur, il faudrait définir nos politiques sur un espace d'information suffisant (par exemple la suite des observations), mais cela nous entraînerait vers des complications algorithmiques bien au-delà de la thématique de cette thèse.

2.1.3 Méthodes de résolution

Le problème d'optimisation étant posé, il existe différentes méthodes de résolution selon les hypothèses faites au départ. La programmation dynamique (Bellman, 1956) s'intéresse à

12. Pour *Partially Observable Markov Decision Process*

la résolution de MDP lorsque les modèles de l'environnement T et R sont connus et discrets. Dans cette thèse, comme en RL classique (Sutton et Barto, 1998; Kaelbling *et al.*, 1996), nous faisons l'hypothèse que les fonctions T et R ne sont pas connues. L'agent apprend à travers son interaction avec l'environnement en échantillonnant ces fonctions en des points précis.

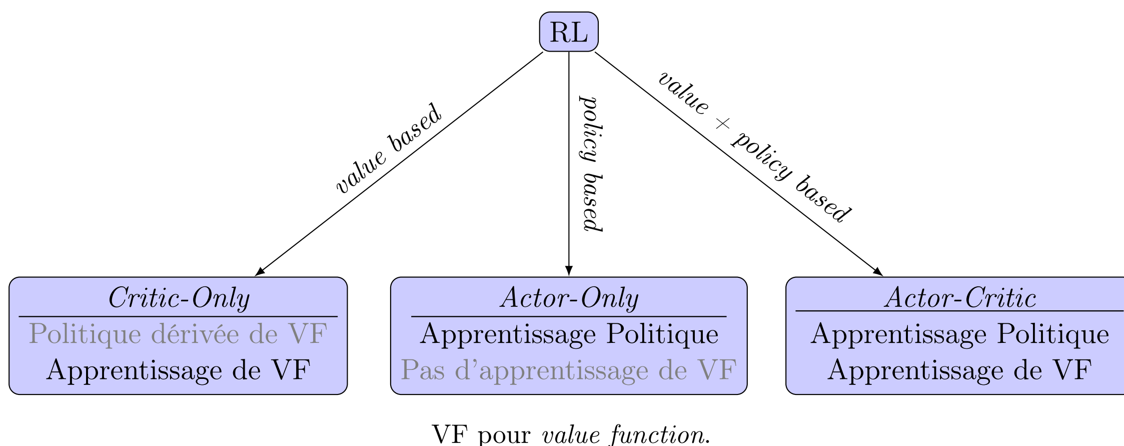
Dans ce cas, il y a deux grandes approches de résolution : celles basées sur des modèles (*model-based*) et celles sans modèles (*model-free*). Les méthodes *model-based* cherchent à apprendre le modèle de l'environnement (les fonctions T et R) puis à utiliser, entre autres, la programmation dynamique sur ces approximations. Les méthodes *model-free* cherchent une politique optimale sans représenter explicitement le modèle de l'environnement.

2.1.4 Classification des différents agents apprenants

Dans cette thèse, nous allons adopter un point de vue découpant les différentes approches en fonction de ce qu'elles cherchent à apprendre (voir Figure 2.4). L'*acteur* est le composant de l'agent qui agit sur l'environnement : il est défini par la politique. Le *critique* sert à critiquer les actions faites par l'agent ; il s'appuie sur les fonctions de valeurs (Section 2.2 page ci-contre) ou VF [13].

Les méthodes *actor-only* cherchent à apprendre directement une politique meilleure que la précédente (potentiellement en estimant des fonctions de valeur mais sans les apprendre), tandis que les méthodes *critic-only* apprennent des fonctions de valeurs pour seulement ensuite en déduire des politiques. Enfin, les méthodes *acteur-critique* apprennent à la fois les fonctions de valeur et les politiques. Ces différentes méthodes ont chacune leurs avantages et inconvénients que nous éliciterons (Section 2.6 page 60) après les avoir détaillées.

Un autre axe de découpage est possible selon les hypothèses faites sur l'agent : en particulier s'il connaît ou découvre le modèle de l'environnement. Même si nous avions préalablement fait l'hypothèse que l'agent ne connaissait pas l'environnement, nous allons tout de même expliquer les méthodes et résultats importants de la programmation dynamique, car ils servent à la compréhension globale des méthodes *model-free*. De même, nous avons fait l'hypothèse de MDP continu, néanmoins, pour faciliter la compréhension de l'exposé, nous expliquerons aussi au préalable quelques méthodes de résolution discrètes.



VF pour *value function.*

FIGURE 2.4 – Découpage des méthodes en apprentissage par renforcement.

13. Pour *Value Function*

2.2 Fonctions de valeur - *critic-only*

Nous allons maintenant définir les différentes fonctions de valeur possibles et expliquer les principales méthodes pour les apprendre. Cette section se consacrera au problème de l'*évaluation d'une politique*. Nous commencerons par des hypothèses fortes (environnement connu et discret) que nous relâcherons peu à peu. La première hypothèse supprimée sera celle de la connaissance de l'environnement, puis celle que les transitions sont générées par la politique à évaluer (données *on-policy*), pour enfin voir les méthodes d'approximations nécessaires pour le cas continu.

2.2.1 Différentes fonctions de valeurs

Une des façons de résoudre des MDP est de réussir à associer une valeur à un état s qui permettrait de quantifier l'intérêt d'être dans cet état. Cette quantification dépend du critère à optimiser, pour le critère γ-*pondéré*, il s'agit de l'espérance du cumul des récompenses futures (décomptées par γ) atteignables à partir de cet état s en suivant une politique π fixée.

Définition 2.10 (Fonction de valeur d'état)**.** *Étant donnée une politique π, pour tout état $s \in \mathcal{S}$, la fonction de valeur d'état V^π est définie telle que :*

$$V^\pi(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| \pi, s_t = s \right].$$

Cette définition provient directement du critère $J(\pi)$ (Définition 2.6 page 28) défini cette fois sur l'ensemble des états. La seconde fonction de valeur utilisée, notée Q^π, n'évalue plus seulement la pertinence d'un état, mais d'un couple état-action.

Définition 2.11 (Fonction action-valeur)**.** *Étant donnée une politique π, pour tout couple $(s, a) \in \mathcal{S} \times \mathcal{A}$, la fonction action-valeur $Q^\pi(s, a)$ est définie telle que :*

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| \pi, s_t = s, a_t = a \right].$$

Le lien direct entre les deux fonctions de valeur est le suivant : $V^\pi(s) = Q^\pi(s, \pi(s))$. Remarquons qu'en connaissant l'environnement, choisir la prochaine action dans un environnement discret peut se faire en évaluant tous les états atteignables par V^π de manière gloutonne et en choisissant l'action qui maximise l'espérance de la valeur.

Définition 2.12 (Politique gloutonne par rapport à V^π)**.** *Une politique est gloutonne par rapport à une fonction de valeur d'état V^π si :*

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} T(s'|s, a) \Big[R(s, a, s') + \gamma V^\pi(s') \Big],$$

ou, dans le cas de $\mathcal{S}$ continu :

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \int_{s' \in \mathcal{S}} T(s'|s, a) \Big[R(s, a, s') + \gamma V^\pi(s') \Big] ds'.$$

La fonction Q permet de s'affranchir de la connaissance de T et R pour choisir une action. Il suffit d'évaluer toutes les actions dans un état précis pour choisir la meilleure.

Définition 2.13 (Politique gloutonne par rapport à Q^π). *Une politique est gloutonne par rapport à une fonction action-valeur Q^π si :*

$$\pi(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s,a).$$

Cette formule permet d'entrevoir que le passage aux actions continues ne sera pas trivial à cause de l'opérateur max qui énumère tout l'espace $\mathcal{A}$.

Remarquons que notre critère d'optimisation 2.8 page 29 peut être réécrit :

$$\pi^* = \arg\max_\pi J(\pi) = \arg\max_\pi \int_{\mathcal{S}} T_0(s_0) V^\pi(s_0) ds_0.$$

Cela permet de définir une fonction de valeur optimale à partir d'une politique optimale.

Définition 2.14 (Fonction de valeur optimale). *Pour tout état $s \in \mathcal{S}$ et toute action $a \in \mathcal{A}$, les fonctions de valeurs optimales sont définies telles que :*

$$\begin{aligned} V^*(s) = V^{\pi^*}(s) \quad &= \max_\pi V^\pi(s). \\ Q^*(s,a) = Q^{\pi^*}(s,a) \quad &= \max_\pi Q^\pi(s,a). \end{aligned}$$

La fonction de valeur optimale associe la plus grande valeur attribuable à un état ou un couple état-action sur l'espace des politiques. Cette propriété n'est pas triviale et provient de l'hypothèse markovienne (Puterman, 1994). La figure 2.5 donne un exemple de deux fonctions de valeur dont l'une est optimale.

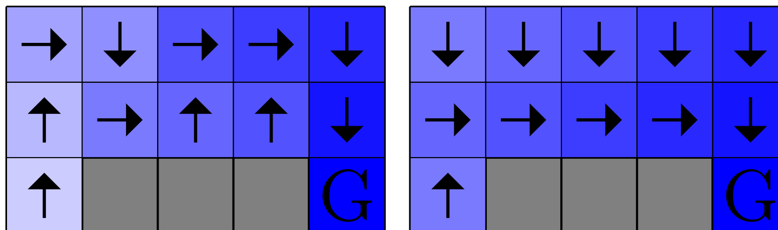


FIGURE 2.5 – Exemple de deux fonctions de valeurs V^π et V^*. Exemple de deux politiques déterministes dans un environnement en grille où la récompense est -1 à chaque pas de temps. L'agent dispose de quatre actions $\rightarrow, \leftarrow, \uparrow, \downarrow$ pour se déplacer et son état est composé de sa position cartésienne dans la grille. L'agent doit atteindre l'emplacement G qui est un état absorbant. Les trois cases grises en bas sont des obstacles où l'agent ne peut aller. À gauche, il s'agit d'une politique π quelconque (différente de π^*). À droite, un exemple de π^* est représenté. Plus l'intensité de couleur est forte, meilleure est la fonction de valeur en cette position.

2.2.2 Programmation dynamique

Un des premiers résultats importants pour évaluer les fonctions de valeur provient de la programmation dynamique (Bellman, 1956). Les équations de Bellman expriment le fait que

les fonctions de valeurs peuvent être décomposées en deux parties, grâce à l'hypothèse markovienne, la récompense immédiate et la valeur décomptée de l'état successeur. Ce résultat, d'abord découvert sur des MDP à espaces discrets, peut s'étendre aux MDP continus.

Théorème 2.1 (Équation de Bellman)**.** *Pour tout $s(,a) \in \mathcal{S}(\times\mathcal{A})$, si π est une politique déterministe :*

$$V^\pi(s) = \int_{\mathcal{S}} T(s'|s,\pi(s)) \Big[R(s,\pi(s),s') + \gamma V^\pi(s') \Big] ds',$$

$$Q^\pi(s,a) = \int_{\mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma Q^\pi(s',\pi(s')) \Big] ds',$$

si π est stochastique :

$$V^\pi(s) = \int_{\mathcal{A}} \pi(a|s) \int_{\mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma V^\pi(s') \Big] ds' da,$$

$$Q^\pi(s,a) = \int_{\mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma \int_{\mathcal{A}} \pi(a'|s') Q^\pi(s',a') da' \Big] ds'.$$

Ce théorème est important, car il permet de définir les deux fonctions de valeurs en fonction de leurs successeurs directs grâce à l'hypothèse de Markov. Il va donner lieu à un premier algorithme de programmation dynamique : *Value Iteration*. Nous utilisons la notation $\hat{V}^\pi$ pour désigner l'estimation actuelle de la fonction de valeur V^π. Dans le cas discret, cet algorithme peut être résumé par cette équation :

$$\hat{V}_k(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma \hat{V}_{k-1}(s') \Big],$$

où k représente le nombre d'itérations déjà effectué sur la formule. Cette mise à jour est exécutée plusieurs fois (indice k) pour raffiner la précision de l'estimation de la fonction de valeur jusqu'à convergence vers V^*. Il est ensuite simple de choisir une action en suivant la politique gloutonne (Définition 2.12 page 31) par rapport à $\hat{V}_k$:

$$\hat{\pi}(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma \hat{V}_k(s') \Big] \approx \pi^*(s).$$

Value-Iteration permet de résoudre efficacement des MDP à espaces discrets où le modèle (T et R) est connu. Une extension aux états continus est possible en passant par une approximation de la fonction de valeur. La représentation n'est alors plus tabulaire (Section 1.1.2 page 12). L'algorithme s'appelle *Smooth Value Iteration* (Boyan et Moore, 1995).

2.2.3 Évaluation par Monte-Carlo

Nous restons dans le sous-problème de l'évaluation d'une politique, par une fonction de valeur, en retirant maintenant l'hypothèse que le modèle soit connu, car il s'agit du problème qui nous intéresse dans cette thèse (*model-free*). Dans ce cas, il est possible d'échantillonner la fonction de transition T et la fonction de récompense R. On parle de méthode de Monte-Carlo pour approcher une valeur en utilisant des procédés aléatoires. Pour faire cet échantillonnage, on se donne une politique π qui sera jouée dans l'environnement, en mémorisant l'expérience de l'agent :

$s_0, r_1, s_1, r_2, s_2, ..., r_L, s_L$ qui termine à l'étape L. La séquence $s_0, s_1, ..., s_L \in \mathcal{S}^{L+1}$ représente les états par lesquels l'agent est passé, et $r_1, r_2, ..., r_L \in \mathbb{R}^L$ la séquence de récompenses. Ce processus est répété sur plusieurs *épisodes*, on écrit alors r_t^k, s_t^k et L^k pour le $k^{ième}$ épisode. Ainsi, il est possible d'approximer $\hat{V}^\pi$ dans chaque état s_t où l'agent est passé :

$$\hat{V}^\pi(s) \leftarrow \frac{1}{N(s)} \sum_{k=1}^{K} \sum_{t=i}^{L^k} \gamma^{t-i} r_{t+1}^k, \tag{2.1}$$

où $N(s)$ est un compteur représentant le nombre de trajectoires où l'état s est présent et i le plus petit indice tel que $s_i^k = s$. Il s'agit de moyenner les retours obtenus après la première visite dans un état jusqu'à la fin de la trajectoire : cet algorithme est appelé *First Visit Monte-Carlo Evaluation*. Plus l'état est rencontré, meilleure sera son estimation par la fonction de valeur :

$$\lim_{K \to \infty} \hat{V}^\pi(s) = \lim_{N(s) \to \infty} \hat{V}^\pi(s) = V^\pi(s).$$

Ainsi, pour améliorer $\hat{V}^\pi(s)$ dans les états s générés par π, il suffit d'augmenter le nombre de trajectoires jouées. En général, les méthodes de Monte-Carlo sont des estimateurs non biaisés, car elles utilisent directement l'ensemble des échantillons provenant de la vérité terrain, sans reposer sur une estimation (Singh et Sutton, 1996). Par contre, elles souffrent d'une grande variance. L'agrégation de la stochasticité présente dans T (et la politique si elle est stochastique), sur l'ensemble d'un épisode, conduit l'estimation $\hat{V}$ à être très bruitée.

Les méthodes purement de Monte-Carlo reposent sur l'hypothèse que l'épisode doit se terminer avant de pouvoir calculer la formule 2.1.

Définition 2.15 (États absorbants)**.** *Étant donné un MDP $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$, on appelle ensemble des états absorbants $\mathcal{S}^* \subset \mathcal{S}$ le sous-ensemble des états desquels l'agent ne peut plus sortir :*

$$\forall t \in \mathbb{N}, s_t \in \mathcal{S}^* \implies \mathbb{P}(s_{t+1} = s_t) = 1.$$

Le terme *état but* est aussi utilisé dans la littérature. Lorsque l'agent atteint l'un de ces états, l'épisode se termine. On parle alors de situations *épisodiques*. Remarquons que $\mathcal{S}^*$ peut être potentiellement vide, auquel cas il sera impossible d'utiliser ces méthodes sans utiliser une approximation introduisant un biais.

2.2.4 Apprentissage par *Temporal Difference* : TD(0)

L'idée des méthodes TD [14] (Sutton, 1988) est de remarquer qu'il n'y a pas besoin d'aller jusqu'à la fin de l'épisode pour estimer $\hat{V}^\pi(s)$ en combinant les méthodes de Monte-Carlo avec les équations de Bellman (Théorème 2.1 page précédente). Le *bootstrapping* (Efron, 1979) permet de réduire la variance d'un estimateur, en mettant à jour une estimation $\hat{x}_t$ à partir d'une estimation précédente $\hat{x}_{t-1}$ à l'aide de nouveaux échantillons. L'estimation *en ligne* de la variance d'un échantillon de données sans stocker chaque élément (Chan *et al.*, 1983; Welford, 1962) est un exemple bien connu de *bootstrapping*. La même idée est utilisée par Sutton (1988) pour mettre à jour V^π.

14. Pour *Temporal Difference*

Algorithme 2.1 (TD(0))**.** *Étant donnés un MDP à espaces discrets $\langle \mathcal{S}, \mathcal{A}, T, R\rangle$, une politique π, et un échantillon $(s_t, r_{t+1}, s_{t+1}) \in \mathcal{S} \times \mathbb{R} \times \mathcal{S}$ généré en suivant π dans le MDP, TD(0) estime $V^\pi(s_t)$ par :*

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha\Big[r_{t+1} + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)\Big],$$

où α est équivalent à un taux d'apprentissage, il représente la vitesse à laquelle l'estimation doit changer.

Remarquons que si $\alpha = 1$, la formule devient $\hat{V}^\pi(s_t) \leftarrow r_{t+1} + \gamma\hat{V}\pi(s_{t+1})$ qui est la version échantillonnée du théorème de Bellman (Théorème 2.1 page 33), nous appellerons également ce terme la *cible* du *bootstrapping*. Cet algorithme est appelé TD(0) car il n'utilise qu'une seule transition pour faire sa mise à jour. Il s'agit d'une estimation biaisée de V^π, mais avec une variance réduite (propriété due au *bootstrapping*). L'erreur temporelle δ_t est définie telle que $\delta_t = r_{t+1} + \gamma\hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$. Nous nous en servirons dans de nombreux algorithmes dans la suite. La mise à jour TD(0) peut-être réécrite avec cette notation plus compacte :

$$\Delta\hat{V}^\pi(s_t) = \underbrace{r_{t+1} + \gamma\hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)}_{\delta_t}.$$

Cette notation facilite la lecture en retirant le taux d'apprentissage. Nous nous autorisons à utiliser les deux notations au cours de cette thèse en fonction de la largeur des formules. $\Delta\hat{V}^\pi(s_t)$ représente la cible vers laquelle l'estimation doit se déplacer indépendamment du taux d'apprentissage :

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha\Delta\hat{V}^\pi(s_t).$$

Dans le cas particulier où l'épisode se termine en $s_{t+1} \in \mathcal{S}^*$, la formule est simplifiée en $\Delta\hat{V}^\pi(s_t) \leftarrow r_{t+1}$. Les méthodes de Monte-Carlo et TD(0) peuvent être facilement étendues à l'apprentissage de Q.

2.2.5 Le compromis TD(λ)

En partant du constat que les méthodes de Monte-Carlo souffrent d'une trop grande variance en utilisant toute la trajectoire pour estimer V^π, tandis que TD(0) souffre d'un biais élevé en utilisant seulement une transition, TD(λ) cherche un compromis entre les deux. Nous nous attarderons plus en détails sur cette méthode, car elle sera utilisée dans certains des algorithmes que nous proposerons dans cette thèse. Plus formellement, pour évaluer $\hat{V}^\pi(s_t)$:

- TD(0) utilise le retour d'une seule transition (*1-step return*) : $r_{t+1} + \gamma\hat{V}^\pi(s_{t+1})$,
- l'évaluation de Monte-Carlo utilise tous les retours : $\sum_{i=t}^{L} \gamma^{i-t} r_{i+1}$.

On définit alors le retour de n transitions (*n-step return*) par :

$$R_n(s_t) = \underbrace{\left(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_{i+1}\right)}_{\text{n transitions}} + \underbrace{\gamma^n \hat{V}^\pi(s_{t+n})}_{\text{estimation}}.$$

La partie n transitions est peu biaisée avec une variance élevée, tandis que la partie estimation est source de biais, mais réduit la variance. Lorsque $n + t > L$, L étant la taille de la trajectoire, alors le *n-step return* devient simplement l'évaluation de Monte-Carlo. La figure 2.6 page suivante illustre les différents types de retours. Pour définir la règle d'apprentissage de TD(λ), plutôt que

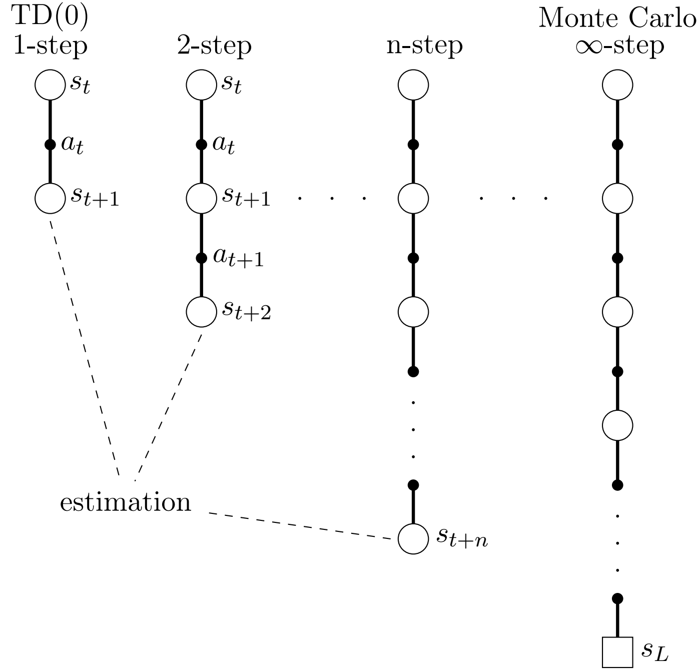


FIGURE 2.6 – Illustration du *n-step return* comparé à TD(0) et Monte-Carlo. Les cercles vides représentent les états échantillonnés $s_t \in \mathcal{S}$ depuis l'environnement en suivant une politique fixée. Les cercles noirs représentent les actions $a_t \in \mathcal{A}$. Une seule transition est utilisée pour la mise à jour de TD(0). Tandis que les méthodes de Monte-Carlo utilisent l'ensemble de la trajectoire avec $s_L \in \mathcal{S}^*$. Le *n-step return* est un compromis entre les deux, il utilise n transitions avec une estimation pour l'état s_{t+n}.

de choisir un nombre précis de transitions, l'ensemble des différents *n-step return* est utilisé, en pondérant exponentiellement les plus longs par λ. Si $s_L \in \mathcal{S}^*$, les λ-*return* sont définis tels que :

$$R_\lambda(s_t) = (1-\lambda) \sum_{n=1}^{L-t-1} \Big(\lambda^{n-1} R_n(s_t) \Big) + \lambda^{L-t-1} \underbrace{\sum_{i=t}^{L} \gamma^{i-t} r_{i+1}}_{Monte-Carlo\ return} .$$

Dans le cas où la trajectoire est terminée en L mais pourrait continuer ($s_L \notin \mathcal{S}^*$) :

$$R_\lambda(s_t) = (1-\lambda) \sum_{n=1}^{L-t-1} \Big(\lambda^{n-1} R_n(s_t) \Big) + \lambda^{L-t-1} \Big(\sum_{i=t}^{L} \gamma^{i-t} r_{i+1} + \gamma^{L-t+1} \hat{V}^\pi(s_L) \Big).$$

Remarquons que les coefficients λ somment bien à 1 :

$$(1-\lambda) \sum_{n=1}^{L-t-1} \lambda^{n-1} + \lambda^{L-t-1} = (1-\lambda) \frac{1-\lambda^{L-t-1}}{1-\lambda} + \lambda^{L-t-1} = 1.$$

La règle de mise à jour de TD(λ) utilise le λ-*return* $R_\lambda(s_t)$ comme cible :

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha \Big[R_\lambda(s_t) - \hat{V}^\pi(s_t) \Big].$$

Les λ-*returns* peuvent se réécrire plus simplement comme étant la somme pondérée des erreurs temporelles par λ et γ.

Algorithme 2.2 (TD(λ)). *Étant donnés un MDP* $\langle \mathcal{S}, \mathcal{A}, T, R\rangle$*, une politique* π *et une trajectoire échantillonnée* $(s_t, r_{t+1}, s_{t+1})_{\{0,\ldots,L-1\}} \in \mathcal{S}^L \times \mathbb{R}^L \times \mathcal{S}^L$ *générée en suivant* π *dans le MDP, TD(*λ*) estime* V^π *en chaque état de la trajectoire par :*

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha \left[\sum_{n=t}^{\infty} (\gamma\lambda)^{n-t} \underbrace{\left(r_{n+1} + \gamma \hat{V}^\pi(s_{n+1}) - \hat{V}^\pi(s_n) \right)}_{\delta_n} \right].$$

TD(λ) sert de compromis entre le biais de l'estimateur et la variance des données (Kearns et Singh, 2000). Plus λ est proche de 1, plus la variance est élevée, au contraire, plus il est proche de 0, plus le biais est élevé.

On dit que TD(λ) utilise des *traces d'éligibilité*, elles représentent le crédit total qui devrait être attribué à une transition pour toute erreur ultérieure dans l'évaluation. Nous avons présenté ici la version *forward view* des *traces d'éligibilité*, car elle s'applique facilement dans un contexte *offline* qui nous intéressera dans nos contributions. Il existe des versions *online* à condition de se limiter à des représentations tabulaires ou des approximateurs linéaires (Van Seijen *et al.*, 2016; Van Hasselt *et al.*, 2014). Enfin, remarquons que l'application de TD(λ), dans un contexte *offline*, a un coût de calcul supplémentaire à TD(0) très négligeable. En effet, dans les deux algorithmes, on calcule le même nombre d'erreurs temporelles qui est l'opération la plus coûteuse.

Il est possible d'utiliser les erreurs temporelles pour apprendre la fonction de valeur Q plutôt que V. Dans ce cas, le *n-step return* en $s_t \in \mathcal{S}$ s'écrit : $\left(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_{i+1}\right) + \gamma^n Q^\pi(s_{t+n}, a_{t+n})$, avec $a_{t+n} \sim \pi(\cdot, s_{t+n})$. On utilise la notation $x \sim y$ pour dire que la variable aléatoire x a la distribution de y. En utilisant la même pondération par λ que dans TD(λ), cela donne l'algorithme Sarsa(λ) (Rummery et Niranjan, 1994).

Algorithme 2.3 (Sarsa(λ)). *Étant donnés un MDP* $\langle \mathcal{S}, \mathcal{A}, T, R\rangle$*, une politique* π *et une trajectoire échantillonnée* $(s_t, a_t, r_{t+1}, s_{t+1})_{\{0,\ldots,L-1\}} \in \mathcal{S}^L \times \mathcal{A}^L \times \mathbb{R}^L \times \mathcal{S}^L$ *générée en suivant* π *dans le MDP, Sarsa(*λ*) estime* Q^π *en chaque couple état-action de la trajectoire par :*

$$\hat{Q}^\pi(s_t, a_t) \leftarrow \hat{Q}^\pi(s_t, a_t) + \alpha \left[\sum_{n=t}^{\infty} (\gamma\lambda)^{n-t} \underbrace{\left(r_{n+1} + \gamma \hat{Q}^\pi(s_{n+1}, a_{n+1}) - \hat{Q}^\pi(s_n, a_n) \right)}_{\delta_n} \right].$$

2.2.6 Évaluation de politique avec des données *off-policy*

Jusqu'à maintenant, nous avons seulement répertorié des méthodes *model-free on-policy* (en dehors de la programmation dynamique) qui apprennent à partir de données générées depuis la politique π qu'on cherche à évaluer. Néanmoins, être capable de retirer l'hypothèse *on-policy* peut conduire à une meilleure efficacité en données. Le problème de l'évaluation de politiques *off-policy* consiste à évaluer V^π ou Q^π à partir de données générées par une autre politique μ. Ce problème a donné lieu à de nombreuses recherches récentes lorsque l'on cherche à l'utiliser avec les

traces d'éligibilité (Mahmood *et al.*, 2017; Sutton *et al.*, 2016; Hachiya *et al.*, 2009; Harutyunyan *et al.*, 2016; Munos *et al.*, 2016).

Importance Sampling

L'une des solutions classiques à l'estimation de fonctions de valeur avec des données *off-policy* est de passer par l'utilisation de l'*importance sampling* (Rubinstein et Kroese, 2016). L'idée générale étant de pondérer une transition, générée par μ, par sa vraisemblance dans π. Supposons un échantillon i.i.d. [15] $x_1, ..., x_n \in X^n$ tiré par la fonction de densité de probabilité h_0, nous voulons calculer l'espérance d'une fonction $g(x)$ si le tirage avait été fait depuis une autre fonction de densité de probabilité h_1. Nous cherchons donc à approximer $\mathbb{E}[g(x)|x \sim h_1(\cdot)]$ à partir de $x_1, ..., x_n \in X^n$.

Appliqué à l'apprentissage par renforcement, l'échantillon $(x_i)_{\{1,...,n\}}$ représente les transitions *off-policy* $(s_t, a_t)_{\{1,...,n\}}$, générées par une politique μ (équivalent à la densité de probabilité h_0), par lesquelles on cherche à estimer la fonction de valeur de la politique π (équivalent à la densité de probabilité h_1).

$$\begin{aligned}
\mathbb{E}\Big[g(x)\Big|x \sim h_1(\cdot)\Big] &= \int_X h_1(x)g(x)dx \\
&= \int_X h_0(x)\frac{h_1(x)}{h_0(x)}g(x)dx \\
&= \mathbb{E}\Big[\frac{h_1(x)}{h_0(x)}g(x)\Big|x \sim h_0(\cdot)\Big] \\
&= \lim_{k\to\infty}\frac{1}{k}\sum_{i=1}^{k}\frac{h_1(x_i)}{h_0(x_i)}g(x_i) \\
&\approx \frac{1}{n}\sum_{i=1}^{n}\frac{h_1(x_i)}{h_0(x_i)}g(x_i).
\end{aligned} \tag{2.2}$$

Pour le problème de l'évaluation *off-policy*, on obtient ainsi le ratio $\frac{\pi(a|s)}{\mu(a|s)}$ à la place de $\frac{h_1(x_i)}{h_0(x_i)}$. Nous cherchons maintenant à calculer $Q^\pi(s,a)$ en utilisant l'évaluation de *Sarsa* (Algorithme 2.3 page précédente) en utilisant l'*importance sampling* sur un échantillon (s_t, a_t) généré par μ.

Rappelons la définition de $Q^\pi(s,a)$:

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle|\pi, s_t = s, a_t = a\right].$$

En appliquant l'*importance sampling* :

$$Q^\pi(s,a) = \mathbb{E}\left[\Big(\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\Big)\prod_{k=t+1}^{\infty}\frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}\middle|\mu, s_t = s, a_t = a\right].$$

Precup (2000) montre qu'on peut en déduire la règle d'apprentissage suivante :

$$\Delta\hat{Q}^\pi(s_t,a_t) = \sum_{n=t}^{\infty}(\gamma\lambda)^{n-t}\Big(\prod_{k=t+1}^{t+n}\frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}\Big)\Big(r_{n+1}+\gamma\frac{\pi(a_{n+1}|s_{n+1})}{\mu(a_{n+1}|s_{n+1})}\hat{Q}^\pi(s_{n+1},a_{n+1})-\hat{Q}^\pi(s_n,a_n)\Big).$$

15. indépendant et identiquement distribués

L'utilisation de l'*importance sampling* a plusieurs inconvénients. Il repose sur l'hypothèse de données i.i.d. ce qui n'est pas le cas en RL : les transitions sont dépendantes les unes des autres. Il fait l'hypothèse que si $\pi(a|s) > 0$ alors $\mu(a|s) > 0$ ce qui oblige à n'utiliser que certaines formes de politiques. Enfin, c'est un estimateur non biaisé en valeur, mais avec une grande variance. En pratique, le produit de ratio $\prod_{k=t+1}^{t+n} \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}$ explose souvent, notamment lorsque $\mu(a_k, s_k)$ est très proche de 0. Rappelons que chaque donnée a_k a été générée par μ, ainsi $\mu(a_k|s_k) > 0$, ce qui n'empêche pas $\mu(a_k|s_k)$ d'être très proche de 0. Dans ce cas, la fraction $\frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}$ devient très grande et, si cela arrive sur plusieurs actions de la trajectoire, le produit $\prod_{k=t+1}^{t+n} \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)}$ explose d'autant plus ; d'où la grande variance de l'*importance sampling*.

Tree-Backup

Pour pallier les limites de l'*importance sampling*, il est possible de se servir de la définition même de Q. Rappelons que $Q^\pi(s, a)$ est l'estimation de la qualité d'effectuer l'action a dans l'état s et puis de suivre π. Il n'y a pas de condition sur la façon dont s et a sont générés dans la définition de $Q^\pi(s, a)$. Cette assertion deviendra néanmoins problématique lors de l'utilisation d'approximateurs (Section 2.2.7 page suivante). La mise à jour de *Sarsa(0)* peut ainsi facilement s'étendre à des données *off-policy*, il n'y a pas de problème à ce que les couples (s_t, a_t) soient générées par une politique μ, à condition que $a_{t+1} \sim \pi(\cdot, s_{t+1})$:

$$\hat{Q}^\pi(s_t, a_t) \leftarrow \hat{Q}^\pi(s_t, a_t) + \alpha\Big(r_{t+1} + \gamma\hat{Q}^\pi(s_{t+1}, a_{t+1}) - \hat{Q}^\pi(s_t, a_t)\Big).$$

Il est possible de réduire le biais de cet estimateur lors de l'utilisation de politiques stochastiques en utilisant l'espérance des retours Q sur un échantillon d'actions générées par la politique π. Cette mise à jour est utilisée par *Expected Sarsa(0)* :

$$\Delta\hat{Q}^\pi(s_t, a_t) = r_{t+1} + \gamma\mathbb{E}\Big[\hat{Q}^\pi(s_{t+1}, a)\Big|a \sim \pi(\cdot|s_{t+1})\Big] - \hat{Q}^\pi(s_t, a_t). \tag{2.3}$$

La généralisation de *Expected Sarsa(0)* aux λ-*returns* a été proposée par Precup (2000), dans l'algorithme *Tree-Backup*(λ) :

$$\Delta\hat{Q}^\pi(s_t, a_t) = \sum_{n=t}^{\infty}(\gamma\lambda)^{n-t} \prod_{k=t+1}^{t+n} \pi(a_k|s_k)\Big(r_{n+1} + \gamma\mathbb{E}\Big[\hat{Q}^\pi(s_{n+1}, a)\Big|a \sim \pi(\cdot|s_{n+1})\Big] - \hat{Q}^\pi(s_n, a_n)\Big).$$

Pour unifier plusieurs algorithmes provenant de *Tree-Backup* avec la même notation, cette formule est réécrite en déplaçant λ à l'intérieur du produit et en généralisant la définition des poids du produit.

Algorithme 2.4 (Tree-Backup(λ) généralisé)**.** *Étant donnés un MDP* $\langle\mathcal{S}, \mathcal{A}, T, R\rangle$*, deux politiques* π *et* μ*, et une trajectoire échantillonnée* $(s_t, a_t, r_{t+1}, s_{t+1})_{\{0,\dots,L-1\}} \in \mathcal{S}^L \times \mathcal{A}^L \times \mathbb{R}^L \times \mathcal{S}^L$ *générée en suivant* μ *dans le MDP, Tree-Backup(*λ*) généralisé estime* Q^π *en chaque couple état-action de la trajectoire par :*

$$\Delta\hat{Q}^\pi(s_t, a_t) = \sum_{n=t}^{\infty}\gamma^{n-t} \prod_{k=t+1}^{t+n} c_k\Big(r_{n+1} + \gamma\mathbb{E}\Big[\hat{Q}^\pi(s_{n+1}, a)\Big|a \sim \pi(\cdot|s_{n+1})\Big] - \hat{Q}^\pi(s_n, a_n)\Big),$$

avec $c_k = \lambda\pi(a_k|s_k)$ *pour la version Tree-Backup(*λ*) classique.*

En remarquant que *Tree-Backup*(λ) n'est pas efficace dans le cas où π et μ sont proches, ce que nous allons détailler, Harutyunyan *et al.* (2016) proposent de réduire c_k à λ, dans l'algorithme $Q^\pi(\lambda)$ *with off-policy corrections*, en retirant la pondération par la probabilité de la politique. En effet, les traces d'éligibilité de données *on-policy* sont coupées rapidement par le produit $\prod_{k=t+1}^{t+n} \pi(a_k|s_k)$ dans *Tree-Backup*(λ) à cause de l'exploration effectuée en pratique lorsque l'on récupère un échantillon de transition (voir Section 2.5.1 page 57). *Tree-Backup*(λ) est strict dans le choix de ses coupures, ce qui lui permet de converger sans condition sur π et μ. À l'inverse, $Q^\pi(\lambda)$ *with off-policy corrections* est plus souple et plus efficace, dans le sens où il se sert de plus de données. Par contre, il ne converge qu'à condition que μ et π soient assez proches.

Enfin, dans l'optique de profiter des avantages de ces différentes approches, en trouvant un compromis, Munos *et al.* (2016) proposent de réintroduire de l'*importance sampling* dans *Retrace(λ)*. Pour éviter l'explosion précédemment décrite de l'*importance sampling*, *Retrace(λ)* tronque son terme lorsque c'est nécessaire : $c_k = \lambda \min(1, \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)})$. De cette façon, lorsque le terme d'*importance sampling* explose, ce qui signifie que π et μ sont proches, l'algorithme est réduit à $Q^\pi(\lambda)$ *with off-policy corrections* qui est efficace dans ce cas précis. À l'inverse, lorsque μ et π sont trop différents, les traces sont coupées. Les avantages et inconvénients des différentes méthodes sont résumés dans la table 2.1.

Algorithmes	c_k	Variance de l'estimation	Convergence	Utilise tous les retours
Importance Sampling	$\frac{\pi(a_k\|s_k)}{\mu(a_k\|s_k)}$	forte	pour tout π et μ	oui
Tree-backup(λ)	$\lambda\pi(a_k\|s_k)$	faible	pour tout π et μ	non
$Q^\pi(\lambda)$ *off-policy*	λ	faible	π proche de μ	oui
Retrace(λ)	$\lambda \min\Big(1, \frac{\pi(a_k\|s_k)}{\mu(a_k\|s_k)}\Big)$	faible	pour tout π et μ	oui

Table 2.1 – Récapitulatif des avantages et inconvénients des variantes de *Tree-Backup(λ)*. *Retrace(λ)* est un compromis entre les 3 autres méthodes *off-policy*. Adapté depuis Munos *et al.* (2016).

2.2.7 Généralisation et approximation

Nous allons maintenant relâcher l'hypothèse d'espace discret $\mathcal{S}$ pour voir les méthodes *model-free* avec approximation dans le cas où $\mathcal{S}$ est continu. Il s'agit du cas qui nous intéresse réellement dans cette thèse. Nous allons voir que la plupart des méthodes précédemment expliquées sont applicables, sauf que dans certains cas on perdra la propriété de convergence et que dans tous les cas la fonction de valeur réelle sera hors d'atteinte (Figure 2.7 page suivante).

Lorsque l'espace des états $\mathcal{S}$ est continu (ou discret, mais $|\mathcal{S}|$ très grand), il n'est plus possible d'utiliser une représentation tabulaire et exacte pour calculer les fonctions de valeurs. La représentation tabulaire souffre de la *malédiction de la dimension* (Bellman, 1957). La première limite provient de la complexité en mémoire pour stocker l'ensemble de ces informations dans un tableau. La seconde limite vient du fait que l'espace, que les fonctions de valeurs devraient

couvrir, augmente tellement vite lorsque le nombre de dimensions augmente, que les échantillons disponibles deviennent *épars* dans cet espace et se retrouvent isolés.

Pour pallier ces limitations, une possibilité est d'abandonner la représentation exacte (tabulaire) de la fonction valeur et de passer par des approximations. Nous nous intéressons particulièrement aux représentations paramétrées. Dans ce cas, on ne cherche plus à modifier l'ensemble des états, potentiellement très grand ou infini, mais un ensemble restreint et fini de paramètres θ (tel que $|\theta| \ll |\mathcal{S}|$ pour le cas discret).

Définition 2.16 (Fonction de valeur paramétrée)**.** *Étant donnés une politique π et un modèle de représentation Ψ, les fonctions de valeurs paramétrées par un ensemble de n paramètres $\theta \in \mathbb{R}^n$ sont définies telles que :*

$$V^\pi_\Psi : \begin{array}{l} \mathcal{S} \times \mathbb{R}^n \to \mathbb{R} \\ s, \theta \quad \mapsto V^\pi_\Psi(s,\theta) \approx V^\pi(s) \end{array}, \; et \quad Q^\pi_\Psi : \begin{array}{l} \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \to \mathbb{R}^+ \\ s, a, \theta \quad \mapsto Q^\pi_\Psi(a|s,\theta) \approx Q^\pi(s,a) \end{array}.$$

Le modèle Ψ définit la façon dont les états (et actions pour Q) et les paramètres θ interagissent pour donner l'approximation (Section 1.1 page 10). Par exemple, pour Ψ linéaire, $V^\pi_\Psi(s,\theta) = \sum_{i=1}^{d_\mathcal{S}} s_i\theta_i \approx V^\pi(s)$. Dans la suite du document, nous utiliserons une notation simplifiée : V^π_θ et Q^π_θ en remplacement de $V^\pi_{\Psi_\theta}$ et $Q^\pi_{\Psi_\theta}$. Lorsque le modèle est fixé (tâche laissée au concepteur du système), il reste à trouver les paramètres θ qui minimisent l'erreur d'approximation entre la fonction de valeur réelle et les fonctions de valeurs paramétrées. On notera ces paramètres θ^* qu'on appellera paramètres optimaux. Remarquons que la notion d'optimalité des

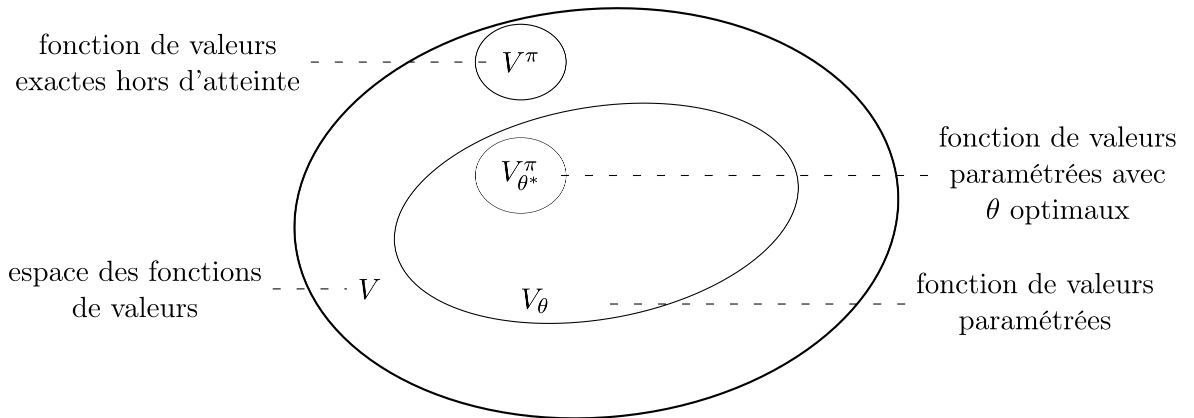


Figure 2.7 – Paysage des fonctions de valeurs d'état paramétrées pour un modèle Ψ donné.

paramètres est relative à la politique, ainsi trouver $V^\pi_{\theta^*}$ permet d'avoir une mesure de V^π et non de la fonction de valeur optimale V^*. L'optimalité des paramètres n'est pas facile à définir. Une tentative consiste à définir θ^* en adaptant le critère MSE [16] de l'apprentissage supervisé.

16. Pour *Mean Square Error*

Définition 2.17 (Mean Square Value Error)**.** *Le critère MSVE définit les paramètres optimaux d'une fonction de valeur approximée V^π_θ par un modèle Ψ_θ comme étant ceux minimisant la distance entre la vraie fonction de valeur V^π et sa projection dans l'espace des fonctions représentables par Ψ_θ :*

$$\theta^*_{MSVE} = \operatorname*{arg\,min}_{\theta\in\Theta} \int_{\mathcal{S}} \eta^\pi(s)\Big(V^\pi(s) - V^\pi_\theta(s)\Big)^2 ds.$$

La définition de $\eta^\pi(s)$ est sujette à débat. En général, $\eta^\pi(s)$ représente la fraction du temps que l'agent passe dans l'état s selon π. On peut également soutenir que $\eta^\pi(s)$ devrait prendre en compte le facteur γ et avantager les états présents au début des trajectoires. Nous ne donnerons pas de définition formelle à $\eta^\pi(s)$ car dans le cas *model-free* de telles mesures sont inaccessibles. Nous supposons ainsi que les états apparaissent avec la même distribution et utilisons le critère sur les seuls échantillons tirés. Cela implique que notre définition des paramètres optimaux est déjà biaisée, nous les noterons $\hat{\theta}^*_{\text{MSVE}}$. Soit un échantillon $(s_t, r_t) \in \mathcal{S}^L \times \mathbb{R}^L$ généré par une politique π, d'après nos hypothèses l'adaptation du critère MSVE (Définition 2.17) s'écrit :

$$\theta^*_{\text{MSVE}} \approx \hat{\theta}^*_{\text{MSVE}} = \operatorname*{arg\,min}_{\theta\in\Theta} \sum_{t=1}^{L} \Big(V^\pi(s_t) - V^\pi_\theta(s_t)\Big)^2. \tag{2.4}$$

En pratique, il est nécessaire de disposer d'une estimation de V^π, l'une des solutions étant d'utiliser des méthodes de *Monte-Carlo* pour faire cette estimation sans *bootstrapping*. Nous n'allons pas détailler cette solution, car nous avons choisi une approche différente dans cette thèse. Nous voulons garder le *bootstrapping* pour des questions d'efficacité en données. V^π étant inconnu, il est possible d'utiliser le théorème de Bellman (Théorème 2.1 page 33) à partir de l'estimation actuelle : $V^\pi(s) \approx \int_{\mathcal{A}} \pi(a|s) \int_{\mathcal{S}} T(s'|s,a)\big[R(s,a,s') + \gamma V^\pi_\theta(s')\big] ds' da$. Il s'agit du critère MSBE [17], mais il nécessite la connaissance du modèle de l'environnement. On utilise alors sa version échantillonnée MSTDE [18].

Définition 2.18 (Mean Square Temporal Difference Error)**.** *Le critère MSTDE définit les paramètres optimaux d'une fonction de valeur approximée V^π_θ par un modèle Ψ_θ comme étant ceux minimisant les résidus de Bellman :*

$$\theta^*_{MSTDE} = \operatorname*{arg\,min}_{\theta\in\Theta} \int_{\mathcal{S}} \eta^\pi(s)\left(\mathbb{E}\Big[r_{t+1} + \gamma V^\pi_\theta(s_{t+1})\Big|\pi\Big] - V^\pi_\theta(s)\right)^2 ds.$$

En utilisant une notation simplifiée et la même hypothèse que précédemment sur $\eta^\pi(s)$, le critère devient :

$$\theta^*_{\text{MSTDE}} \approx \hat{\theta}^*_{\text{MSTDE}} = \operatorname*{arg\,min}_{\theta\in\Theta} \sum_{t=1}^{L} \left(\mathbb{E}\Big[r_{t+1} + \gamma V^\pi_\theta(s_{t+1})\Big|\pi\Big] - V^\pi_\theta(s_t)\right)^2. \tag{2.5}$$

Il reste un dernier critère utilisé dans la communauté : MSPBE [19]. En remarquant que les cibles de MSBE ($\int_{\mathcal{A}} \pi(a|s) \int_{\mathcal{S}} T(s'|s,a)\big[R(s,a,s') + \gamma V^\pi_\theta(s')\big] ds' da$) sur l'ensemble des états ne sont pas

17. Pour *Mean Square Bellman Error*
18. Pour *Mean Square Temporal Difference Error*
19. Pour *Mean Square Projected Bellman Error*

forcément représentables dans l'espace d'approximation de Ψ_θ, MSPBE utilise un opérateur de projection pour trouver les cibles correspondantes représentables sur l'espace d'approximation. Ce critère a été la source de nombreuses recherches récentes (Sutton *et al.*, 2016; Scherrer, 2010; Sutton *et al.*, 2009). Pour les réseaux de neurones, l'opérateur de projection est plus complexe, plutôt que de faire une projection sur un plan comme avec les approximateurs linéaires, il faut le faire sur une variété non linéaire (Bhatnagar *et al.*, 2009). Pour cette raison, nous ne retiendrons pas ce critère. Chaque critère a des avantages et inconvénients décrits comme dans la figure 2.8. Dans les chapitres suivants, nous utiliserons essentiellement le critère MSTDE.

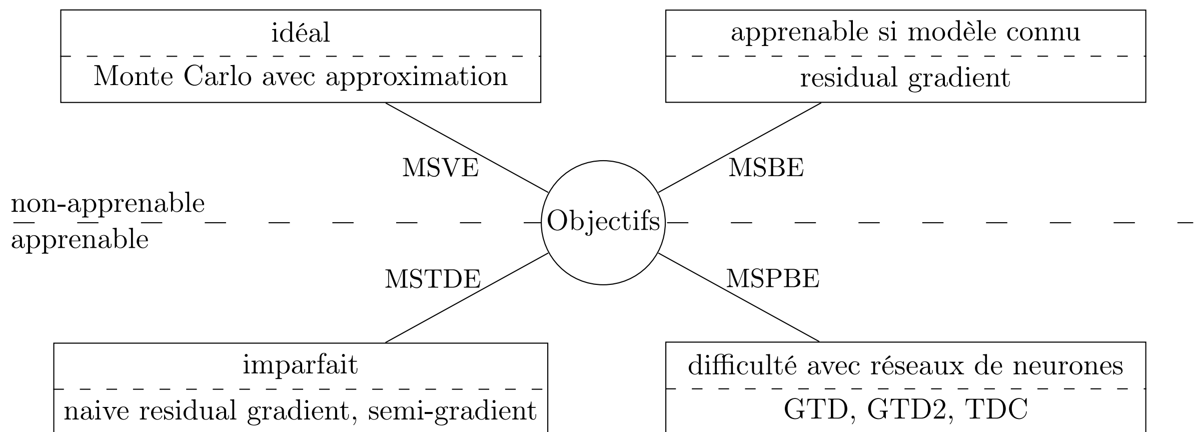


FIGURE 2.8 – Les différents critères qualifiants l'optimalité des paramètres d'une fonction de valeur approximée. Les parties inférieures des rectangles présentent des algorithmes utilisant le critère en question. Nous allons en détailler quelques-uns dans la suite. MSVE est le critère idéal, mais il faudrait connaître V^π pour l'apprendre. Il serait possible de remplacer V^π par une estimation de Monte-Carlo, mais on perdrait alors le *bootstrapping*. MSBE se sert du théorème de Bellman, néanmoins Sutton et Barto (2017) montrent que ce critère n'est en général pas apprenable si le modèle n'est pas connu. MSTDE se sert des différences temporelles. Sur un problème déterministe, MSBE et MSTDE sont équivalents. En pénalisant les erreurs temporelles, MSTDE réalise plus un lissage temporel qu'une prédiction précise : l'ordre des solutions qu'il propose n'est en général pas le même que MSVE. Enfin, MSPBE projette MSBE dans l'espace de représentation avec un second ensemble de paramètres à apprendre, le rendant apprenable.

Le passage aux fonctions de valeurs paramétrées ne permettra en général pas de trouver une politique optimale, mais seulement de s'en approcher. Nous allons maintenant voir les principales méthodes existantes pour apprendre ces paramètres.

Les méthodes de gradient sur MSTDE

L'une des grandes classes de méthodes pour apprendre θ repose sur le calcul du gradient du critère (Section 2.5 page ci-contre). Il en résulte deux approches : les méthodes *true gradient descent* (Baird III et Moore, 1999) et les méthodes *semi-gradient* (Sutton, 2015) qui ne sont pas

de vraies descentes de gradient.

$$\frac{\partial \text{MSTDE}}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{t=1}^{L} \left(\mathbb{E}\left[r_{t+1} + \gamma V_\theta^\pi(s_{t+1}) \middle| \pi \right] - V_\theta^\pi(s_t) \right)^2$$
$$= \sum_{t=1}^{L} 2 \left(\mathbb{E}\left[r_{t+1} + \gamma V_\theta^\pi(s_{t+1}) \middle| \pi \right] - V_\theta^\pi(s_t) \right) . \frac{\partial}{\partial \theta} \left(\mathbb{E}\left[\gamma V_\theta^\pi(s_{t+1}) \middle| \pi \right] - V_\theta^\pi(s_t) \right).$$

Dans la littérature, cette équation donne lieu à des algorithmes appelés *residual gradient* qui sont de vraies descentes de gradient. On peut remarquer qu'ils nécessitent un double échantillonnage de s_{t+1}, ce qui est une hypothèse forte faite sur la manière dont le temps se déroule dans l'environnement car il faudrait pouvoir revenir en arrière (ou avoir un environnement déterministe). Nous nous intéresserons plutôt à une variante biaisée relâchant cette hypothèse. Les méthodes *semi-gradient* (SG) font l'hypothèse que la cible $r_{t+1} + \gamma V_\theta^\pi(s_{t+1})$ ne dépend pas des paramètres θ.

$$\frac{\partial \text{MSTDE}}{\partial \theta} \underset{\text{SG}}{\approx} \sum_{t=1}^{L} -2 \left(r_{t+1} + \gamma \hat{V}_\theta^\pi(s_{t+1}) - V_\theta^\pi(s_t) \right) \frac{\partial V_\theta^\pi(s_t)}{\partial \theta}.$$

Cette hypothèse introduit un biais, car la cible dépend des paramètres, donc $\frac{\partial V_\theta^\pi(s_{t+1})}{\partial \theta} \neq 0$ en général, sauf que sans cette hypothèse les méthodes de descente de gradient stochastiques ne pourraient pas s'appliquer.

Algorithme 2.5 (Semi-gradient TD(0)). *Étant donnés un MDP $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ avec $\mathcal{S}$ continu, un modèle Ψ_θ de représentation de fonction de valeur, une politique π, et un échantillon $(s_t, r_{t+1}, s_{t+1}) \in \mathcal{S} \times \mathbb{R} \times \mathcal{S}$ généré en suivant π dans le MDP, Semi-gradient TD(0) estime $V^\pi(s_t)$ par $V_\theta^\pi(s_t)$ en apprenant θ tel que :*

$$\theta_{k+1} \leftarrow \theta_k + \alpha \left[r_{t+1} + \gamma V_{\theta_k}^\pi(s_{t+1}) - V_{\theta_k}^\pi(s_t) \right] \frac{\partial V_{\theta_k(s_t)}^\pi}{\partial \theta_k},$$

où α est équivalent à un taux d'apprentissage, il représente la vitesse à laquelle l'estimation doit changer.

On peut étendre cette idée facilement aux algorithmes précédemment détaillés telle que TD(λ) et Sarsa(λ). Ces méthodes sont simples et convergent lorsque le modèle choisi est linéaire (Sutton, 1988), d'où leur popularité (Mnih *et al.*, 2015; Van Hasselt et Wiering, 2007).

Lorsqu'on passe aux cas *off-policy*, c'est bien plus compliqué. En effet, rappelons que, par exemple, l'utilisation de Sarsa(0) avec des données *off-policy* repose sur la définition même de $Q^\pi(s, a)$ et l'hypothèse que la façon dont s et a sont générés n'est pas importante (voir Section 2.2.6 page 39). Dans le cas où Q est paramétrée ce n'est plus vrai, la façon dont s et a sont générés devient importante, car la répétition de couples état-action déplacera les paramètres θ du modèle de Q lors d'un apprentissage par descente de gradient. Le problème de l'apprentissage *off-policy* peut se découper en deux parties : calculer la cible de la mise à jour d'apprentissage et utiliser une bonne distribution des mises à jour. Avec une représentation exacte, seul le premier sous-problème apparaît (Section 2.2.6 page 37) mais lors de l'utilisation d'approximateurs le second doit être également considéré. Sutton (2015) parle alors de «*deadly triad*» pour les problèmes d'évaluation de politique mélangeant l'utilisation de *bootstraping*, de fonction de valeurs paramétrées et d'échantillon *off-policy*. Ces trois parties sont pourtant nécessaires pour

être plus efficaces en données. Il s'agit d'un problème restant encore ouvert en particulier avec l'utilisation d'approximateurs non linéaires. Dans le cas des approximateurs linéaires, plusieurs solutions existent grâce à MSPBE (Sutton *et al.*, 2016; Van Hasselt *et al.*, 2014).

Méthodes *least-squares*

Nous allons maintenant parler d'une dernière famille de méthodes nommée *least-squares*, elles nous intéressent, car elles sont plus efficaces en données (Boyan, 1998; Bradtke *et al.*, 1996). Plutôt que de calculer des gradients sur les différents critères précédents, ces méthodes calculent directement la solution analytique d'un problème de moindres carrés. De cette façon, il n'y a plus besoin de définir un taux d'apprentissage. Par contre, la complexité passe à $O(n^2)$ en nombre de paramètres. Par exemple, LSTD [20] qui minimise MSTDE trouvera directement la solution de l'équation 2.5 page 42 :

$$\arg\min_{\theta \in \Theta} \sum_{t=1}^{L} \left(\mathbb{E}\Big[r_{t+1} + \gamma V_\theta^\pi(s_{t+1}) \Big| \pi \Big] - V_\theta^\pi(s_t) \right)^2 .$$

Les méthodes *least-squares* utilisent l'ensemble des données disponibles pour minimiser un des critères précédents, contrairement, par exemple, à *semi-gradient TD(0)* (Algorithme 2.5) qui n'en utilise qu'une à la fois. C'est pour cette raison que les méthodes *least-squares* sont plus *efficaces en données*.

Ces méthodes ne pourraient pas s'appliquer directement sur un réseau de neurones, car la solution n'est plus calculable analytiquement. Dans ce cas, une version itérative réintroduisant un taux d'apprentissage et un calcul de gradient est possible, de manière similaire à Geramifard *et al.* (2006); Gordon (1995). On parle de *Fitted Q Iteration* (Ernst *et al.*, 2005). Ainsi, la complexité n'est plus aussi élevée que $O(n^2)$, mais il n'y a plus de garantie d'avoir une solution optimale. La seule différence avec *semi-gradient TD(0)* est alors le nombre de données utilisées pour calculer le gradient.

NFQ [21] (Riedmiller, 2005) est un cas particulier de LSTD itératif. Un réseau de neurones est utilisé pour approximer la fonction Q^* en minimisant MSTDE :

$$Q^*_{\theta_{k+1}} = \arg\min_{Q} \sum_{t=1}^{L} \Big[\big(r_{t+1} + \gamma \max_{a' \in A} Q^*_{\theta_k}(s_{t+1}, a') \big) - Q(s_t, a_t) \Big]^2 .$$

Nous nous intéresserons plutôt à son algorithme d'évaluation de politique décrit ci-dessous.

Algorithme 2.6 (Neural Fitted Q-Iteration)**.** *Étant donné un MDP* $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ *avec* $\mathcal{S}$ *continu, un modèle* Ψ_θ *de représentation de fonction de valeur, une politique* π*, et une trajectoire* $(s_t, a_t, r_{t+1}, s_{t+1})_{\{1,\dots,L\}} \in \mathcal{S}^L \times \mathcal{A}^L \times \mathbb{R}^L \times \mathcal{S}^L$ *générée en suivant* π *dans le MDP, NFQ estime* $Q^\pi(s_t, a_t)$ *par* $Q^\pi_\theta(s_t, a_t)$ *en apprenant* θ *par descente de gradient :*

$$\theta_{k+1} = \theta_k + \alpha \sum_{t=1}^{L} \Big[\big(r_{t+1} + \gamma Q^\pi_{\theta_k}(s_{t+1}, \pi(s_{t+1})) \big) - Q^\pi_{\theta_k}(s_t, a_t) \Big] \frac{\partial Q^\pi_{\theta_k}(s_t, a_t)}{\partial \theta_k} .$$

20. Pour *Least-Squares Temporal Difference*
21. Pour *Neural Fitted Q-Iteration*

Une version légèrement modifiée (Mnih *et al.*, 2015), avec des données *off-policy* et des *target networks*, a été utilisée pour apprendre à résoudre plusieurs jeux Atari (Bellemare *et al.*, 2013). Pour gagner en stabilité, les *target networks* modifient la cible $r_{t+1} + \gamma Q^{\pi}_{\theta_k}(s_{t+1}, \pi(s_{t+1}))$ avec un second jeu de paramètres θ'_k. La cible à apprendre $r_{t+1} + \gamma Q^{\pi}_{\theta'_k}(s_{t+1}, \pi(s_{t+1}))$ devient ainsi indépendante des paramètres actuels. Les paramètres θ'_k se déplacent en suivant les valeurs de θ_k sur une échelle de temps plus lente : $\theta'_k \leftarrow \theta'_k + \alpha' \theta_k$.

2.2.8 Conclusion

Maintenant que nous avons vu l'ensemble des méthodes qui nous intéresse pour évaluer une politique (Table 2.2), nous allons nous resituer dans le problème du RL.

Hypothèse sur $\mathcal{S}$	Hypothèse sur T et R	Remarques	Algorithmes
discret	connu		*Value Iteration*
	inconnu		Monte-Carlo, TD(0)
		traces d'éligibilité	TD(λ)
		données off-policy	Importance sampling, Tree-backup
continu	inconnu		Semi-gradient, LSTD(λ), NFQ

Table 2.2 – Récapitulatif des méthodes d'évaluation de politiques avec leurs hypothèses.

Tous les algorithmes d'apprentissage par renforcement peuvent être décrits dans un cadre appelé *Generalized Policy Iteration* (voir Figure 2.9 page suivante). L'idée générale étant de faire interagir l'évaluation de la politique et son amélioration. En itérant, un certain nombre de fois, entre ces deux sous-problèmes, on tente de converger vers une politique ayant une fonction de valeur optimale. Il y a donc deux problèmes distincts :

— l'évaluation d'une politique : étant donnée une politique π, comment évaluer sa fonction de valeur V^{π} ou Q^{π} ?

— l'amélioration d'une politique : étant données une politique π_t et sa fonction de valeur V^{π_t} ou Q^{π_t}, comment définir π_{t+1} telle que $\pi_{t+1} \geq \pi_t$ (Définition 2.7 page 29) ?

Cette distinction est particulièrement visible dans les architectures *acteur-critique*. En *critic-only*, l'évaluation est réalisée par les méthodes présentées précédemment et l'amélioration de la politique est réalisée par l'opérateur max. Tandis qu'en *actor-only*, l'évaluation de la politique se fait à travers le coût J ou potentiellement en estimant une fonction de valeurs mais sans l'apprendre explicitement : lors de l'amélioration de la politique, l'estimation de la fonction de valeur est oubliée.

On peut remarquer que l'évaluation et l'amélioration sont en compétition. En effet, l'amélioration de la politique augmente l'erreur dans l'évaluation puisque la politique change, tandis qu'un meilleur affinage de l'évaluation permet de trouver où l'amélioration de la politique peut être réalisé. Dans certains cas, cette compétition peut être problématique et empêcher la convergence, notamment lors de l'utilisation de fonctions d'approximations.

Maintenant que le problème de l'évaluation d'une politique et ses solutions ont été développés, nous allons nous intéresser au problème de l'amélioration de la politique.

Contrôle en *Critic-only* Jusqu'à présent, nous avons uniquement montré des méthodes de *policy evaluation*. Le *control*, qui sert à approximer des politiques optimales, n'est pas évident

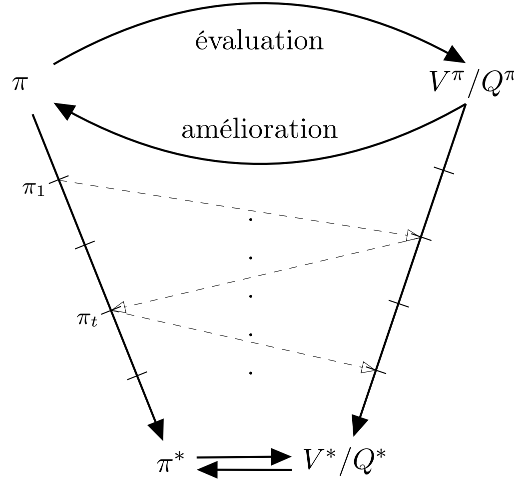


FIGURE 2.9 – Illustration du cadre *Generalized Policy Iteration*

dans le cadre des MDP continus avec les méthodes *critic-only* car il repose sur l'utilisation d'un opérateur max pour être glouton par rapport à ces fonctions de valeurs (Définitions 2.12 et 2.13). Rappelons qu'une politique gloutonne est définie par l'une de ces deux équations :

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \int_{s' \in \mathcal{S}} T(s'|s,a) \Big[R(s,a,s') + \gamma V^\pi(s') \Big] ds',$$
$$\pi(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s,a,s').$$

L'opérateur max sur l'espace des actions coûte cher à calculer lorsque l'espace des actions est continu. Pour éviter d'avoir à énumérer toutes les possibilités d'un ensemble infini, il faudrait passer par des approximations, par exemple avec les méthodes présentées section 1.2, sans pouvoir garantir des solutions exactes dans le cas général. Le calcul de la solution de l'opérateur max devant se faire à chaque étape d'amélioration de la politique, les méthodes *critic-only* ne sont pas utilisables en pratique sur des problèmes réalistes. Citons, tout de même, l'algorithme très connu de *control* qu'est le Q-Learning (Watkins et Dayan, 1992). Il utilise une estimation de Q plutôt que V pour éviter d'avoir à connaître le modèle de transition T pour choisir une action. L'une des solutions pour traiter les espaces continus en action avec des méthodes *critic-only*, qui n'est pas sans introduire de biais, consiste à définir un ensemble discret d'actions continues (Pazis et Lagoudakis, 2009; Zimmer et Doncieux, 2017).

2.3 Policy Search - *Actor-only*

Nous passons maintenant aux agents du type *actor-only* (classification présentée Section 2.1.4 page 30). Rappelons qu'il s'agit d'agents cherchant à apprendre directement une politique optimale pour résoudre un MDP, sans passer par l'utilisation d'une fonction de valeur. L'un des avantages par rapport à la dérivation de politiques à partir de fonctions de valeurs, c'est que la politique peut être une fonction plus simple à approximer et que les espaces en actions continues seront plus simples à traiter. Ces méthodes peuvent également être vues à travers le spectre de *Generalized Policy Iteration* (Section 2.2.8 page ci-contre) sauf que l'étape de *policy evaluation* se fait sans apprentissage de fonction de valeur via le coût J. Le problème principalement abordé

dans cette section est donc celui de l'amélioration de la politique (*policy improvement*). Seules les méthodes *model-free* avec actions continues seront traitées, car ce sont celles qui nous intéressent dans cette thèse. Pour une vue plus globale, Deisenroth *et al.* (2013) ont également étudié les approches *model-based* et Peters *et al.* (2010) les actions discrètes.

2.3.1 Politiques paramétrées

Nous rappelons le problème de l'amélioration d'une politique. Étant données une politique π_k à l'épisode k et une trajectoire Monte-Carlo $(s_0, r_1, s_1, ..., r_L, s_L)$ de taille L générée par l'environnement en utilisant π_k, comment définir $\pi_{k+1} \geq \pi_k$? En l'état, le problème est mal posé, car nous nous intéressons à un espace continu non dénombrable. Il va donc falloir passer par des approximateurs (Section 1.1 page 10).

Définition 2.19 (Politique paramétrée stochastique). *Une politique stochastique π_Ψ, paramétrée par un ensemble de n paramètres $\theta \in \mathbb{R}^n$ et un modèle Ψ_θ, associe un état $s \in \mathcal{S}$ et les paramètres θ à une densité de probabilité sur l'espace des actions $\mathcal{A}$:*

$$\begin{aligned} \pi_\Psi :\ & \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \to \mathbb{R}^+ \qquad \text{avec} \int_{\mathcal{A}} \pi_\Psi(a|s,\theta)da = 1. \\ & s, a, \theta \quad \mapsto \pi_\Psi(a|s,\theta) \end{aligned}$$

Pour les politiques stochastiques, les paramètres θ permettent de contrôler l'exploration en un état s.

Définition 2.20 (Politique paramétrée déterministe). *Une politique déterministe π_Ψ, paramétrée par un ensemble de n paramètres $\theta \in \mathbb{R}^n$, associe un état $s \in \mathcal{S}$ et les paramètres θ à une action $a \in \mathcal{A}$:*

$$\begin{aligned} \pi_\Psi :\ & \mathcal{S} \times \mathbb{R}^n \to \mathcal{A} \quad . \\ & s, \theta \mapsto \pi_\Psi(s,\theta) \end{aligned}$$

Nous écrirons π_θ, et potentiellement π par abus de notation lorsqu'il est sous-entendu que π est une politique paramétrée π_{Ψ_θ}. L'ensemble des paramètres ne suffit pas à décrire complètement une politique paramétrée, il reste à définir le *modèle* de la politique Ψ_θ qui détermine la façon dont s et θ interagissent. Ainsi, on ne cherche plus à trouver une politique optimale en tout point de l'espace, mais à trouver les meilleurs paramètres pour un *modèle* Ψ donné (Figure 2.10 page ci-contre). Le choix du modèle exact est laissé au concepteur. Nous nous intéressons aux algorithmes qui ne font pas d'hypothèse de linéarité sur le modèle pour pouvoir les utiliser avec des réseaux de neurones.

Le problème de l'amélioration de la politique se réécrit $\pi_{\theta_{k+1}} \geq \pi_{\theta_k}$. Trouver une politique paramétrée optimale c'est trouver le meilleur ensemble de poids θ pour un modèle donné.

Définition 2.21 (Politique paramétrée optimale). *Étant donné un modèle de politique paramétrée Ψ_θ, les paramètres $\theta \in \Theta = \mathbb{R}^n$ sont optimaux, pour le modèle donné, s'ils satisfont la contrainte suivante :*

$$\theta^* = \underset{\theta \in \Theta}{\arg\max}\ J\big(\pi_\theta\big).$$

Là encore, il n'y a en général pas unicité de la solution.

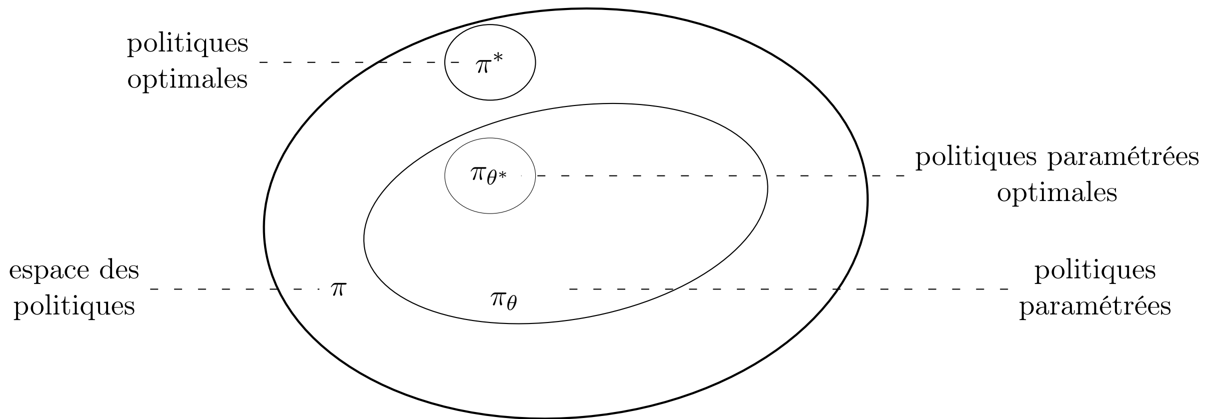


FIGURE 2.10 – Paysage des politiques paramétrées pour un modèle Ψ_θ donné.

2.3.2 Espace d'exploration

Le problème étant maintenant défini correctement, nous allons aborder les différentes méthodes existantes. Elles peuvent être décrites au sein de l'algorithme général *Model-free Policy Search* (Algorithme 2.7).

Algorithme 2.7 (Model-free Policy Search)

Données : Politique paramétrisée π_{θ_0} par θ_0 initialisé aléatoirement
répéter
  Explorer : Générer m trajectoires à partir de π_{θ_k}
  Évaluer : Calculer le coût J sur les m trajectoires
  Mettre à jour : Calculer θ_{k+1} sachant les coûts J et les m trajectoires
jusqu'à $\theta_k \approx \theta_{k+1}$;

Contrairement aux approches *critic-only*, il est souvent nécessaire de jouer plusieurs fois une politique dans l'environnement pour avoir une bonne estimation du coût J. Cela réduit l'efficacité en données de ces méthodes. Pour déterminer une direction de modification des paramètres θ, permettant l'amélioration du coût J, l'étape d'exploration est cruciale. Une section y est consacrée pour plus de détails (Section 2.5.1 page 57). À l'étape d'exploration, ce n'est pas exactement la politique π_k qui est jouée, mais sa version exploratoire.

Il existe deux grandes approches d'exploration qui se font à des échelles différentes : dans l'espace des actions et dans l'espace des paramètres. L'exploration dans l'espace des actions se fait à l'échelle de temps des *étapes* tandis que l'exploration dans l'espace des paramètres est réalisée, en général, dans l'échelle de temps des *épisodes*. Rappelons qu'un *épisode* est une trajectoire constituée de plusieurs *étapes*.

Explorer dans l'espace d'actions, c'est modifier l'action proposée par la politique. Par exemple, si la politique propose l'action $u_t \in \mathcal{A}$ dans l'état $s_t \in \mathcal{S}$ au temps t, noté $u_t \sim \pi_\theta(s_t)$, ce n'est pas u_t qui sera jouée dans l'environnement, mais $a_t \sim g_t(\cdot|u_t)$ où g_t est une densité de probabilité définie sur l'espace des actions : $g : \mathcal{A} \times \mathcal{A} \to \mathbb{R}^+$, appelée fonction d'exploration.

Explorer dans l'espace des paramètres, c'est modifier les paramètres θ de la politique directe-

ment. En général, ils sont fixés durant tout un épisode. On parle aussi de *direct policy search*. Par exemple, il est possible d'utiliser une méthode d'optimisation *black-box* (Section 1.2.3 page 19) pour explorer l'espace des paramètres et optimiser J.

2.3.3 *Policy-gradient*

L'une des grandes classes d'algorithmes *actor-only* repose sur le gradient de la politique. Le but étant de trouver la direction vers laquelle déplacer les poids θ pour maximiser le critère J (2.6 page 28). Rappelons que, pour π_θ, ce critère est défini ainsi :

$$J(\pi_\theta) = \int_{\mathcal{S}} T_0(s_0)\ \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t\ \Big|\ \pi_\theta, s_0\Big] ds_0.$$

Le but étant de trouver le gradient de ce critère par rapport à θ sans qu'il dépende de l'un des gradients de l'environnement : ∇R ou ∇T (ces fonctions sont supposées inconnues). J peut être décomposé par la densité de probabilité d'une trajectoire, et la somme des récompenses décomptées de cette trajectoire.

Définition 2.22 (Densité probabilité d'une trajectoire)**.** *Soit τ une trajectoire état-action de taille L (potentiellement infinie), telle que $\tau = (s_t, a_t)_{\{0,\dots,L-1\}}$ ainsi $\tau \in (\mathcal{S} \times \mathcal{A})^L$. On définit la densité de probabilité de cette trajectoire, selon la politique stochastique π, telle que :*

$$d^\pi(\tau) = T_0(s_0) \prod_{t=0}^{L} \pi(a_t|s_t) T(s_{t+1}|s_t, a_t).$$

Dans le cas où π est déterministe :

$$d^\pi(\tau) = T_0(s_0) \prod_{t=0}^{L} T(s_{t+1}|s_t, \pi(s_t)).$$

On remarque qu'il s'agit bien d'une densité de probabilité : π et T étant elles-mêmes des densités de probabilités, $d^\pi(\tau)$ est toujours positive et $\int_\tau d^\pi(\tau) d\tau = \int_{s_0} T_0(s_0) \int_{a_0} \pi(a_0|s_0) \int_{s_1} T(s_1|s_0, a_0) ... = 1$. Formellement, la somme des récompenses décomptées d'une trajectoire τ s'écrit :

$$R_\gamma(\tau) = \sum_{t=0}^{L} \gamma^t R(s_t, a_t, s_{t+1}).$$

La décomposition de J peut ainsi se réécrire :

$$J(\pi_\theta) = \int_\tau d^{\pi_\theta}(\tau) R_\gamma(\tau) d\tau.$$

Cette première réécriture permet de supprimer la dépendance des récompenses aux paramètres θ. Il convient maintenant d'en calculer la dérivée.

$$\begin{aligned} \frac{\partial J(\pi_\theta)}{\partial \theta} &= \int_\tau R_\gamma(\tau) \frac{\partial d^{\pi_\theta}(\tau)}{\partial \theta} d\tau && (2.6) \\ &= \int_\tau d^{\pi_\theta} R_\gamma(\tau) \frac{\partial \log d^{\pi_\theta}(\tau)}{\partial \theta} d\tau \end{aligned}$$

L'insertion du log permet de retirer de la multiplication les termes non dépendants de θ.

Algorithme 2.8 (REINFORCE)**.** *Étant données une politique paramétrée stochastique π_θ et une trajectoire $\tau \in (\mathcal{S}\times\mathcal{A})^L$, une estimation non biaisée du gradient de la politique (Williams, 1992) est :*

$$\frac{\partial J(\pi_\theta)}{\partial \theta} = \mathbb{E}\Big[R_\gamma(\tau) \sum_{t=0}^{L} \frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\Big| \tau \sim \pi_\theta\Big].$$

Ce résultat est intéressant, car le gradient ne dépend pas de l'environnement. Néanmoins, comme toutes les méthodes de Monte-Carlo, il souffre d'une grande variance. Une possibilité pour réduire cette variance est de recourir à la soustraction d'une *baseline* optimale (Peters *et al.*, 2010). Nous détaillerons les intérêts d'une *baseline* dans le chapitre dédié aux méthodes *acteur-critique* (Section 2.4.1 page 53). Enfin, remarquons qu'il reste le calcul de $\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}$ qui dépend du choix du modèle pour la politique π_θ.

Une seconde information importante, provenant de l'équation 2.6 page ci-contre, est que, dans le cas où π est déterministe, le gradient de la politique n'existe pas :

$$\frac{\partial J(\pi_\theta)}{\partial \theta} = \mathbb{E}\Big[R_\gamma(\tau) \sum_{t=0}^{L} \frac{1}{T(s_{t+1}|s_t,a_t)} \frac{\partial T(s_{t+1}|s_t,a_t)}{\partial a_t}\Big|_{a_t=\pi_\theta(s_t)} \frac{\partial \pi_\theta(s_t)}{\partial \theta}\Big| \tau \sim \pi_\theta\Big].$$

On dit que le gradient de la politique n'existe pas au sens où il n'existe pas sans connaître le modèle de l'environnement à cause de $\frac{\partial T(s_{t+1}|s_t,a_t)}{\partial a_t}$. Ce résultat explique qu'il y ait eu moins de recherche sur les politiques déterministes à l'aide de *policy-gradient*. Contrairement aux méthodes *actor-only*, nous verrons que dans le cadre *acteur-critique* ce gradient existe.

Natural policy gradient Pour améliorer la montée de gradient, en dehors de l'ajout d'une *baseline*, il est possible de restreindre la mise à jour de la politique d'une façon plus efficace qu'avec un simple taux d'apprentissage. Ce type de mise à jour est appelé *natural policy gradient* (Kakade, 2002; Peters et Schaal, 2006). Elle utilise la matrice d'information de Fisher :

$$[\boldsymbol{\mathcal{I}}(\theta)]_{i,j} = \frac{\partial \log \pi_\theta(s,a)}{\partial \theta_i} \frac{\partial \log \pi_\theta(s,a)}{\partial \theta_j},$$

de dimensions $n \times n$ en nombre de paramètres θ, à ne pas confondre avec la matrice Hessienne,

$$[\boldsymbol{H}(\theta)]_{i,j} = \frac{\partial^2 \pi_\theta(s,a)}{\partial \theta_i \partial \theta_j},$$

qui a les mêmes dimensions, mais calcule les dérivées secondes. La matrice d'information de Fisher tend à converger vers la matrice Hessienne, en cela on peut faire le parallèle avec les méthodes de Newton (Section 1.2.2 page 19). La mise à jour des poids utilise la même règle en remplaçant la matrice Hessienne par celle de Fisher. L'information de la dérivée seconde n'est ainsi plus nécessaire. Par contre, le coût reste toujours de $O(n^2)$ en nombre de paramètres. Cette mise à jour a plusieurs propriétés : elle rend $\Delta\theta$ invariable aux paramètres θ et l'effet plateau est moins sévère qu'avec les gradients classiques (Amari, 1998).

Trust Region Policy Optimisation Dans le même ordre d'idée, Schulman *et al.* (2015a) proposent de restreindre la modification des paramètres en limitant la divergence de Kullback-Leibler entre la nouvelle et l'ancienne politique. Ce faisant, ils sont capables d'apprendre des politiques stochastiques, représentées par des réseaux de neurones profonds, plus efficacement que les méthodes black-box grâce à leur algorithme TRPO [22] (Duan *et al.*, 2016). TRPO peut également profiter d'un critique.

Comme nous l'avons expliqué au cours de cette section, la difficulté principale des méthodes *actor-only* provient de la forte variance présente dans l'estimation du coût J. Pour réduire cette variance, les architectures *acteur-critique* introduisent un biais à travers l'utilisation d'un critique.

2.4 *Actor-Critic*

Maintenant que nous avons étudié les différentes façons d'apprendre une fonction de valeur et les façons d'améliorer une politique, nous allons pouvoir décrire les architectures *acteur-critique* qui combinent les deux méthodes. Nous aborderons essentiellement les méthodes traitant des espaces continus en omettant donc celles qui reposent sur un acteur discret.

2.4.1 Gradient d'une politique stochastique avec fonction de valeur

Nous savons comment apprendre une fonction de valeur d'une politique (paramétrée ou non), mais nous n'avons pas encore abordé l'amélioration une politique paramétrée à partir d'une fonction de valeur. Les politiques gloutonnes (Définitions 2.13 page 32 et 2.12 page 31) pouvant ne pas être représentables dans l'espace d'approximation et être trop coûteuses à calculer (Section 2.2.8 page 46). Rappelons que le critère J à optimiser pour une politique paramétrée π_θ est le suivant (Définitions 2.6 page 28 et 2.21 page 48) :

$$\theta^* = \arg\max_{\theta \in \Theta} J\big(\pi_\theta\big) = \arg\max_{\theta \in \Theta} \int_{\mathcal{S}} T_0(s_0)\ \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \ \Big|\ \pi_\theta, s_0\Big] \mathrm{d}s_0.$$

En définissant $d_\gamma^\pi(s)$, la densité de probabilité (décomptée par γ) d'être dans l'état s en suivant la politique π, le critère d'optimisation peut être réécrit par :

$$\theta^* = \arg\max_{\theta \in \Theta} \int_{\mathcal{S}} d_\gamma^{\pi_\theta}(s) \int_{\mathcal{A}} \pi_\theta(a|s) R(s,a) \mathrm{d}a \mathrm{d}s.$$

Définition 2.23 (Densité de probabilité d'un état)**.** *Soit π une politique stochastique, on définit la densité de probabilité d'un état $s \in \mathcal{S}$, décomptée par γ, par :*

$$d_\gamma^\pi(s) = \int_{\mathcal{S}} T_0(s_0) \sum_{t=1}^{\infty} \gamma^{t-1} d^\pi(s|s_0,t) \mathrm{d}s_0,$$

où $d^\pi(s|s_0,t)$ est la densité de probabilité d'être dans l'état s au temps t en partant de l'état s_0 tout en suivant la politique π. À cause du décompte par γ, $d_\gamma^\pi(s)$ est une densité non conforme, car son intégrale ne somme pas à 1.

Attention à ne pas confondre $d^\pi(\tau)$, la densité de probabilité d'une trajectoire (Définition 2.22 page 50), avec $d_\gamma^\pi(s)$ qui est définie sur l'espace des états. Le théorème du gradient sur les

22. Pour *Trust Region Policy Optimisation*

politiques (Sutton *et al.*, 1999a) montre que la dérivée du critère d'optimisation s'écrit à partir de la fonction de valeur Q. Cela donne lieu à une nouvelle classe d'algorithmes où on apprend à la fois la politique π et sa fonction de valeur Q^π. Initialement proposé sur les MDP à actions discrètes, le résultat s'étend au cas des MDP continus.

Théorème 2.2 (Policy gradient avec fonction de valeur)**.** *Pour tout MDP, le gradient du coût d'une politique π_θ paramétrée, stochastique et différentiable s'écrit :*

$$\begin{aligned}\frac{\partial J(\pi_\theta)}{\partial \theta} &= \int_{\mathcal{S}} d_\gamma^\pi(s) \int_{\mathcal{A}} \frac{\partial \pi_\theta(a|s)}{\partial \theta} Q^\pi(s,a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} Q^\pi(s,a) \middle| s \sim d_\gamma^\pi, a \sim \pi_\theta(\cdot|s) \right],\end{aligned}$$

où d_γ^π est la densité de probabilité d'être dans un état s en suivant la politique π_θ.

La seconde forme sert pour la version échantillonnée puisqu'en pratique le calcul des intégrales serait trop coûteux, d'autant que $d_\gamma^\pi(s)$ ne peut qu'être approchée lorsque le modèle de l'environnement est inconnu. Remarquons la différence avec REINFORCE (Algorithme 2.8 page 51) qui définit son gradient sur des trajectoires Monte-Carlo alors que cette approche utilise l'espace des états. La force de ce théorème est de ne pas dépendre des dérivées des modèles de l'environnement ∇T et ∇R, tout comme REINFORCE.

L'introduction du critique par le biais de la fonction Q^π permet de réduire la variance de l'estimation du gradient, car on ne s'appuie plus sur des méthodes purement de Monte-Carlo. Pour continuer dans cette réduction, il est possible de retirer une « ligne de base » (*baseline*) qui ne dépend que de l'état : il s'agit d'une fonction de référence. En remarquant que l'ajout de la *baseline* $f(s)$ ne modifie pas l'espérance :

$$\int_{\mathcal{S}} d_\gamma^\pi(s) \int_{\mathcal{A}} \frac{\partial \pi_\theta(a|s)}{\partial \theta} f(s) \mathrm{d}a\mathrm{d}s = 0.$$

On peut redéfinir le gradient du coût d'une politique stochastique :

$$\frac{\partial J}{\partial \theta} = \mathbb{E}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \Big(Q^\pi(s,a) - f(s) \Big) \middle| s \sim d^\pi, a \sim \pi_\theta(\cdot|s) \right].$$

On choisit $f(s)$ de façon à réduire $\mathrm{Var}[\nabla_\theta J]$. En général, l'utilisation de $V^\pi(s)$ est suffisante. Cela fait apparaître la fonction avantage $A^\pi(s,a) = Q^\pi(s,a) - V(s)^\pi$.

$$= \mathbb{E}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} A^\pi(s,a) \middle| s \sim d^\pi, a \sim \pi_\theta(\cdot|s) \right].$$

Sachant que la fonction avantage ne satisfait aucune équation de Bellman (Geist et Pietquin, 2010), son apprentissage doit forcément être décomposé. L'estimation de la fonction avantage peut se faire via plusieurs moyens : à partir des erreurs TD, ce qui correspond aux premières approches classiques *acteur-critique* (Barto *et al.*, 1983), ou à partir des erreurs TD(λ) pour avoir un compromis entre biais et variance (Section 2.2.5 page 35). L'utilisation des erreurs TD(λ) est aussi appelée GAE [23] (Schulman *et al.*, 2015b). Certains travaux estiment la fonction avantage

23. Pour *Generalized Advantage Estimation*

à l'aide d'un seul réseau de neurones en contraignant certaines parties du réseau à apprendre V, A et Q (Gu *et al.*, 2016b; Wang *et al.*, 2015).

En dehors de l'aspect asynchrone et de la façon dont le critique est appris, Mnih *et al.* (2016) proposent d'estimer $Q^\pi(s_t, a_t)$ par $\sum_i^{t-1} \gamma^i r_i + \gamma^t V^\pi(s_{t+1})$ dans leur algorithme *on-policy* A3C[24]. L'estimation du gradient nécessite alors d'obtenir une trajectoire entière et se fait de manière *offline*.

Algorithme 2.9 (Asynchronous Advantage Actor-Critic)**.** *Étant donnés une politique paramétrée π_θ, un échantillon $(s_i, r_i, a_i)_{\{0,\dots,L-1\}} \in \mathcal{S}^L \times \mathbb{R}^L \times \mathcal{A}^L$ généré en suivant π_θ et V^π la fonction de valeur d'état de la politique π, A3C estime le gradient du critère J d'une politique stochastique par :*

$$\widehat{\frac{\partial J}{\partial \theta}} = \sum_{t=0}^{L-1} \left(\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} \Big(\sum_{i=0}^{t-1} \gamma^i r_i + \gamma^t V^\pi(s_{t+1}) - V^\pi(s_t) \Big) \right).$$

Estimation *off-policy* De la même manière que pour l'apprentissage du critique (Section 2.2.6 page 37), il est possible de prendre en compte des données générées par une politique μ différente de celle à améliorer π_θ durant l'estimation de $\Delta_\theta J$. À cette fin, Degris *et al.* (2012) proposent d'utiliser le ratio d'*importance sampling* introduit en section 2.2.6 page 37 :

$$\widehat{\frac{\partial J}{\partial \theta}} = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\mu(a|s)} \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} A^\pi(s,a) \middle| s \sim d^\mu, a \sim \mu(\cdot|s) \right].$$

Comme vu précédemment, en pratique, l'utilisation de ce ratio pose des problèmes lorsqu'il explose. Wang *et al.* (2016) proposent donc de se servir de *Retrace*(λ) avec l'estimation suivante dans leur algorithme ACER[25] :

$$\widehat{\frac{\partial J}{\partial \theta}} = \mathbb{E}\Bigg[\min\Big(c, \frac{\pi_\theta(a|s)}{\mu(a|s)}\Big) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} A^\pi(s,a) + \\ \mathbb{E}\left[\left[\frac{\frac{\pi_\theta(a'|s)}{\mu(a'|s)} - c}{\frac{\pi_\theta(a'|s)}{\mu(a'|s)}} \right]_+ \frac{\partial \log \pi_\theta(a'|s)}{\partial \theta} A^\pi(s,a') \middle| a' \sim \pi(\cdot|s) \right] \Bigg| s \sim d^\mu, a \sim \mu(\cdot|s) \Bigg],$$

où c est un méta paramètre et $[x]_+$ vaut x lorsque $x > 0$ sinon 0. Le premier terme correspond à la trajectoire qui a été jouée dans l'environnement, il est donc possible d'utiliser *Retrace*(λ) pour estimer $A^\pi(s,a)$ en estimant V à l'aide de Q : $V^\pi(s) = \mathbb{E}[Q^\pi(s,a)|a \sim \pi(\cdot|s)]$. Le second appel à la fonction avantage $A^\pi(s,a')$ ne peut cependant pas être estimé par *Retrace* car la trajectoire est inconnue. L'algorithme ACER ne peut se résumer à l'équation précédente, il s'agit une agrégation de plusieurs techniques récentes : TRPO (Section 2.3.3 page 52), des réseaux neuronaux représentant à la fois la politique, les fonctions de valeur Q et V et la fonction avantage (Wang *et al.*, 2015), le tout avec des mises à jour asynchrones.

24. Pour *Asynchronous Advantage Actor-Critic*
25. Pour *Actor-Critic with Experience Replay*

2.4.2 Gradient d'une politique déterministe avec fonction de valeur

Contrairement au cas *actor-only*, il est possible de définir le gradient du coût d'une politique déterministe grâce à la fonction Q (Silver *et al.*, 2014). Ce résultat fut précédemment exploité dans l'algorithme *Action Dependent Heuristic Dynamic Programming* (ADHDP) par Werbos (1977); Prokhorov et Wunsch (1997).

Théorème 2.3 (Policy gradient déterministe avec fonction de valeur)**.** *Pour tout MDP, le gradient du coût d'une politique π_θ paramétrée, déterministe et différentiable s'écrit :*

$$\begin{aligned}\frac{\partial J}{\partial \theta} =& \int_{\mathcal{S}} d^\pi_\gamma(s) \frac{\partial Q^\pi(s,a)}{\partial a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \mathrm{d}s \\ =& \mathbb{E}\left[\frac{\partial Q^\pi(s,a)}{\partial a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi_\gamma \right],\end{aligned}$$

où d^π_γ est la densité de probabilité d'être dans un état s en suivant la politique π_θ.

De plus, il s'étend facilement au cas *off-policy one-step* de par son utilisation de Q. L'algorithme DDPG [26] se base sur ce résultat pour apprendre des politiques déterministes avec des réseaux de neurones (Lillicrap *et al.*, 2015). Contrairement au gradient du coût d'une politique stochastique où l'on peut soustraire une *baseline*, ce n'est plus aussi simple ici. Ainsi, l'utilisation de la fonction avantage A est possible, mais son implication sur la variance ne sera plus si importante.

Retour aux politiques stochastiques Gu *et al.* (2016a) proposent de se servir du gradient du coût d'une politique déterministe pour améliorer l'estimation du gradient du coût d'une politique stochastique. En utilisant une variable de contrôle et l'extension de Taylor de premier ordre, toute fonction $f(s,a)$ peut être approximée de la façon suivante : $f(s,a) \approx \widehat{f(s,a)} = f(s,a') + \Delta_a f(s,a)(a-a')$. Ainsi, dans leur algorithme *Q-prop*, ils proposent d'estimer le gradient du coût d'une politique de la façon suivante :

$$\widehat{\frac{\partial J}{\partial \theta}} = \mathbb{E}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \hat{A}^\pi_\omega(s,a) + \mathbb{E}\left[\omega(s) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \frac{\partial Q^\pi(s,a')}{\partial a'} \Big| a' \sim \pi(\cdot|s) \right] \Big| s \sim d^\pi, a \sim \pi(\cdot|s) \right],$$

où $\omega(s)$ est une fonction de pondération modulant l'importance de la variable de contrôle. Ils proposent d'estimer Q^π avec des données *off-policy*, par exemple avec l'algorithme *Retrace*(λ) et d'estimer la fonction avantage A en mélangeant des informations *on-policy* et *off-policy* :

$$\hat{A}^\pi_\omega(s,a) = \hat{A}^\pi(s,a) - \omega(s)(a-a') \frac{\partial Q^\pi(s,a')}{\partial a'}\Big|_{a'=\mathbb{E}[\pi(\cdot|s)]},$$

où $\hat{A}^\pi(s,a)$ est estimée avec des données *on-policy*, par exemple avec GAE (Section 2.4.1 page 53).

Ainsi, l'idée principale de *Q-prop* est de combiner à la fois l'information provenant de GAE, qui est *on-policy* non biaisée mais avec forte variance, avec un estimateur biaisé *off-policy* de la fonction de valeur.

26. Pour *Deep Determinist Policy Gradient*

2.4.3 Méthode alternative

Il existe d'autres formes de méthodes *acteur-critique* et nous allons en présenter deux maintenant. Nous nous sommes appuyés sur ces deux méthodes pour développer plusieurs de nos contributions.

Cacla CACLA[27] est un algorithme *online on-policy* qui apprend une politique paramétrée π_θ et sa fonction de valeur V^π par des réseaux de neurones (Van Hasselt et Wiering, 2007). La fonction de valeur est apprise par TD(0). L'innovation de cette méthode provient de la mise à jour de son acteur, qui utilise la sortie de la politique d'exploration μ (Section 2.5.1 page ci-contre) :

$$\theta_{t+1} \leftarrow \theta_t + \alpha \underset{\hat{\delta}_\mu(s_t,s_{t+1},a')>0}{\mathbb{1}} \Big(a' - \pi_\theta(s_t)\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}\Big|_{a' \sim \mu(\cdot|s_t)},$$

où $\hat{\delta}_\mu(s_t, s_{t+1}, a') = R(s_t, a') + \gamma \hat{V}^\mu(s_{t+1}) - \hat{V}^\mu(s)$ et $\underset{x}{\mathbb{1}}$ est la fonction qui vaut 1 si x est vrai et 0 sinon. On qualifie cet algorithme de *on-policy* car il utilise l'algorithme TD(0) et l'erreur temporelle δ.

Fitted Actor-critic Dans la lignée de l'algorithme *Neural Fitted Q-Iteration* et des méthodes *least-square* (Section 2.2.7 page 45), il est possible de s'appuyer sur l'algorithme *Least-Squares Policy Iteration* (Lagoudakis et Parr, 2003; Cheng *et al.*, 2011) pour utiliser aux mieux les données disponibles. On cherche d'abord à trouver une bonne estimation de la fonction de valeur à l'aide de LSTD, puis on améliore la politique et on répète ces deux opérations. Par exemple, Antos *et al.* (2008); Melo et Lopes (2008) ont proposé des algorithmes *acteur-critique* plus efficaces en données. Les algorithmes *fitted actor-critic* calculent une série de fonctions de valeur et de politiques sur un échantillon de données $\mathcal{D} = (s_t, a_t, s_{t+1}, r_{t+1})_{\{1,\dots,L\}}$ fixé, récupéré en suivant la politique μ :

$$\begin{aligned}
Q_{k+1} &= \underset{Q \in \mathcal{F}_c}{\operatorname{argmin}} \sum_{(s_t,a_t,r_{t+1},s_{t+1}) \in \mathcal{D}} \omega(s_t, a_t) \Big[Q(s_t, a_t) - \Big(r_{t+1} + \gamma Q_k(s_{t+1}, \pi_k(s_{t+1}))\Big)\Big]^2, \\
\pi_{k+1} &= \underset{\pi \in \mathcal{F}_a}{\operatorname{argmax}} \sum_{s_t \in \mathcal{D}} Q_{k+1}\big(s_t, \pi_k(s_t)\big),
\end{aligned}$$

où $\omega(s_t, a_t)$ est une fonction à définir pour influencer le poids d'une transition. Par exemple, $\omega(s_t, a_t) = \frac{1}{\mu(a_t|s_t)}$ permet de ne pas donner trop de poids aux transitions qui sont plus nombreuses dans l'échantillon.

2.4.4 Conclusion

Dans un premier temps, nous avons abordé les façons d'apprendre une fonction de valeur avec les méthodes *critic-only* sur des espaces continus d'états. Puis, pour gérer les espaces d'actions continus, nous avons examiné les méthodes *actor-only*, notamment la notion de politique paramétrée. Enfin, pour être plus *efficaces en données*, nous avons vu qu'il existait plusieurs manières d'améliorer un acteur à partir de l'évaluation faite par un critique avec les méthodes *acteur-critique*.

Nous allons maintenant aborder différentes limitations inhérentes à toute méthode d'apprentissage par renforcement.

27. Pour *Continuous Actor-Critic Learning Agent*

2.5 Limitations

En plus du compromis *biais-variance*, que nous avons exploré jusqu'à maintenant, il existe d'autres dilemmes bien étudiés en apprentissage par renforcement : celui de l'*exploration* contre l'*exploitation* et celui de l'efficacité en données contre la mise à l'échelle. Ces limitations sont présentes pour toutes les méthodes utilisées (*acteur-critique*, *critic-only* et *actor-only*).

2.5.1 Exploration ou exploitation

Le dilemme *exploration-exploitation* consiste à choisir entre prendre la meilleure décision étant donné les informations courantes (*exploitation*) ou collecter plus d'informations (*exploration*). Trop exploiter mène à des politiques non optimales, tandis que trop explorer ralentit inutilement l'apprentissage. En effet, l'utilisation d'une politique gloutonne (Définitions 2.13 page 32 et 2.12 page 31) ne permet pas de garantir que l'agent puisse découvrir suffisamment l'espace des états pour trouver la politique optimale (Figure 2.11). L'utilisation de politiques gloutonnes est associée à l'*exploitation* de l'information courante.

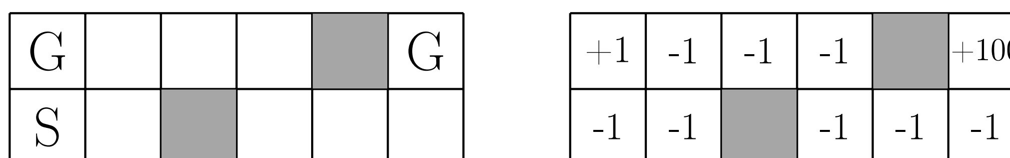


FIGURE 2.11 – Un environnement où sans exploration la politique trouvée ne sera jamais optimale lorsque γ est proche de 1. L'agent dispose de quatre actions $\rightarrow, \leftarrow, \uparrow, \downarrow$ pour se déplacer et son état est composé de sa position cartésienne dans la grille. L'agent part de la position S et doit atteindre un emplacement G (état absorbant). La fonction de récompense est représentée sur la grille de droite. Les cases grises sont des obstacles où l'agent ne peut pas aller. Sans exploration, la politique gloutonne converge vers la stratégie qui consiste à monter d'une case et atteindre un but ; or la politique optimale consiste à aller chercher la récompense de la case en haut à droite.

Les politiques gloutonnes suivent les actions qu'elles pensent meilleures jusqu'à présent. Mais, en général, suivre ces actions ne permet pas de trouver une politique optimale. Pour cette raison, l'exploration introduit une stochasticité dans le processus décisionnel. En fonction du type de politiques finales que l'on souhaite obtenir, déterministe ou stochastique, les stratégies ne sont pas les mêmes. Dans les deux cas, il faut distinguer les *phases d'apprentissage* et *phases de test* (Section 1.3.1 page 20). Il faut également distinguer si l'exploration se produit dans l'espace des paramètres ou l'espace des états (Section 2.3.2 page 49). Dans tous les cas, en phase de test, la politique doit être du même type que celle que l'on souhaite finalement obtenir. En phase

	Politiques d'exploration	
Espace d'exploration	Politiques déterministes	Politiques stochastiques
Espace d'action	stochastique	stochastique
Espace des paramètres	déterministe	stochastique

TABLE 2.3 – Les politiques d'exploration à adopter en fonction des politiques à apprendre et de l'espace d'exploration.

d'apprentissage la politique d'exploration, aussi appelée *behavior policy*, n'est pas toujours stochastique (Table 2.3 page précédente). Dans le cas de l'apprentissage d'une politique déterministe avec une exploration de l'espace d'actions, la politique d'exploration doit être stochastique au risque de produire un problème similaire à la figure 2.11 page précédente. Lorsqu'on explore l'espace des paramètres pour apprendre une politique déterministe, il n'est pas nécessaire d'utiliser une politique stochastique pour explorer, car l'exploration a déjà eu lieu préalablement dans l'espace des paramètres. On remarque ainsi que l'apprentissage *off-policy* est particulièrement important lorsque la politique d'exploration n'est pas la même que la politique qu'on cherche à apprendre.

Les stratégies d'exploration Il existe de nombreuses façons d'explorer l'espace d'états d'un environnement (March, 1991; Auer, 2002; Bellemare *et al.*, 2016) : exploration informée, non informée, hypothèse optimiste, recherche d'états peu rencontrés, etc. Nous avons choisi d'utiliser les stratégies non informées et naïves, car ce sont les plus simples à mettre en place et elles respectent notre hypothèse d'un apprentissage développemental peu informé. La plus connue d'entre elles est ϵ-greedy, elle consiste à choisir une action aléatoire dans l'espace des actions avec une probabilité ϵ. L'exploration gaussienne est la seconde stratégie sur laquelle nous nous sommes appuyés.

Définition 2.24 (Politique d'exploration gaussienne)**.** *Soit π_θ une politique d'exploration gaussienne, paramétrée par un modèle Ψ_θ, si on cherche à apprendre une politique stochastique alors le modèle modélise également l'exploration :*

$$\pi_\theta(a|s) \sim \frac{1}{[\Psi_\theta(s)]_2\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{a-[\Psi_\theta(s)]_1}{[\Psi_\theta(s)]_2}\right)^2}.$$

Dans le cas où on cherche à apprendre la politique déterministe $\pi_\theta(s) = \Psi_\theta(s)$, alors la politique d'exploration gaussienne est la suivante :

$$\pi_\theta(a|s) \sim \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{a-\Psi_\theta(s)}{\sigma}\right)^2},$$

où σ est un méta paramètre réglant le degré d'exploration.

Nous avons utilisé une variante de la politique d'exploration gaussienne en utilisant une loi gaussienne tronquée. L'idée est la même, seule la formule change légèrement pour forcer le tirage de la Gaussienne à appartenir à un intervalle donné.

2.5.2 Efficacité en données ou mise à l'échelle

La dernière limitation, dont la réflexion s'est nourrie au cours de nos travaux, est celle de l'efficacité en données contre la capacité de mise à l'échelle (*scalability*) en apprentissage par renforcement. Cette limitation n'est pas nouvelle en apprentissage supervisé. Plus un modèle dispose de paramètres, et plus il aura besoin de données pour être correctement appris (Halevy *et al.*, 2009; LeCun *et al.*, 2012). Ainsi, pour un temps de calcul limité, il existe une concurrence entre deux processus en apprentissage par renforcement : le temps accordé à l'environnement pour générer des données et le temps accordé à l'agent pour exploiter ces données. Par exemple, en

robotique, la production de données coûte très cher, on préférera alors un algorithme plus efficace en données qui les exploitera au maximum. Or, le faible nombre de données est problématique pour une mise à l'échelle avec de nombreux degrés de liberté à cause de la *malédiction de la dimension*. À l'inverse, pour certains environnements virtuels où la génération de données est très rapide, on préfèrera un algorithme qui ne s'attardera pas trop sur chacune des données et qui sera capable de gérer son flux. Notre définition de la mise à l'échelle englobe ainsi deux phénomènes : celui de l'augmentation des données disponibles et celui de l'augmentation des dimensions de chacune des données.

Nous allons récapituler les différentes propriétés ayant une influence sur l'efficacité en données et la mise à l'échelle pour étayer notre proposition.

Les approximateurs Le choix de l'approximateur de la politique et de la fonction de valeur (Section 1.1 page 10) a une influence sur le nombre de données requis pour disposer d'une bonne estimation. Les modèles qui ont le moins besoin de données sont ceux qui ont un nombre de degrés de liberté le plus réduit, réduisant ainsi leur espace de recherche. Par exemple, des modèles linéaires avec des fonctions de base bien choisies permettront de diminuer le temps de recherche. À l'inverse, avec des réseaux de neurones, il sera plus facile de traiter des espaces très grands nécessitant un nombre important de fonctions de base.

Les fonctions de valeurs (et donc les méthodes *critic based*) permettent d'évaluer une situation, qui n'a jamais été rencontrée au préalable, grâce la généralisation des approximateurs de fonctions. Elles augmentent ainsi l'efficacité en données tout en augmentant les calculs nécessaires pour leur apprentissage.

Model-based Lorsque le coût de génération des données est très important, il peut être intéressant de modéliser l'environnement afin de générer soi-même des données. Deisenroth et Rasmussen (2011) ont proposé un algorithme *model-based* efficace en données : *Probabilistic Inference for Learning Control* (PILCO). Coulom (2002) s'est intéressé aux MDP en environnement continu avec temps continu et modélisation de l'environnement. Cependant, lorsque la dimensionnalité des données augmente, PILCO n'est plus utilisable (Wahlström *et al.*, 2015) et l'utilisation des approches *model-based* n'augmentent pas toujours la qualité de la politique avec des réseaux de neurones (Gu *et al.*, 2016a).

Replay buffer* et Apprentissage *off-policy Le *replay buffer* consiste à garder en mémoire des transitions pour les rejouer (Lin, 1992). Cette méthode permet d'augmenter l'efficacité en données tout en augmentant les calculs nécessaires lorsque les données repassent dans l'apprentissage. Si les données du *replay buffer* sont *off-policy* et si l'algorithme est capable de les traiter adéquatement (Section 2.2.6 page 37), cela permet également de tirer davantage d'informations des expériences précédentes. Cette technique peut être vue comme un compromis entre les approches *model-based* et les approches *model-free* (Vanseijen et Sutton, 2015).

Méthodes de second ordre La descente de gradient peut être plus stable à l'aide des méthodes de Newton en calculant la matrice Hessienne ou à l'aide des gradients naturels en calculant la matrice d'information de Fisher (Section 1.2.2 page 19 et 2.3.3 page 51). L'algorithme Natural Actor Critic (Peters et Schaal, 2008) utilise ce principe pour apprendre son critique et son acteur. L'inconvénient principal de ces méthodes est leur complexité en $O(n^2)$ en nombre de paramètres. Le nombre moyen de paramètres dans nos réseaux est d'environ 3000 paramètres, ce qui produit des matrices d'environ 10 millions d'éléments à calculer. La descente de gradient est essentielle

pour garder une bonne capacité de mise à l'échelle des réseaux de neurones, néanmoins elle est légèrement moins efficace en données que les méthodes de second ordre.

Méthodes *least-squares* Les méthodes *least-squares* tentent de trouver une des meilleures solutions vis-à-vis des données actuellement collectées, contrairement aux autres approches classiques qui cherchent à améliorer légèrement la précision des fonctions de valeur. L'approche *least-squares* est ainsi plus efficace en données, au détriment du coût nécessaire pour trouver l'une des meilleures solutions. Certaines d'entre elles sont calculées en $O(n^2)$ et d'autres sont linéaires en nombre de paramètres (Section 2.2.7 page 45). Elles sont, en général, exécutées de manière hors-ligne (*offline*).

Conclusion Le choix des réseaux de neurones peut être remis en question lors d'une recherche plus poussée de l'efficacité en données. Le plus simple étant d'avoir un nombre de paramètres restreint pour appliquer des méthodes de second ordre et nécessiter moins de données. Les modèles avec fonctions de base permettent, par ailleurs, d'introduire de la connaissance *a priori* qui peut encore améliorer l'efficacité en données.

Une partie de cette thèse est dédiée à la problématique de l'efficacité en données, mais il faut alors préciser sous quelles conditions pour la mise à l'échelle. Les obligations minimales que nous nous fixons sont celles de l'utilisation de réseaux de neurones et d'une complexité linéaire en nombre de paramètres pour l'apprentissage. Rappelons que nous tenons à utiliser les réseaux de neurones, car ils ont la capacité de construire des représentations avec peu de connaissances humaines requises ; ce qui nous place dans un cadre développemental.

2.6 Conclusion

Après avoir présenté la problématique de l'apprentissage par renforcement, nous avons passé en revue les différentes méthodes pour apprendre une politique et l'améliorer. Nous pouvons ainsi récapituler les différentes solutions possibles pour traiter des MDP continus (Figure 2.12 page ci-contre).

Les méthodes *actor-only* permettent de traiter des espaces continus grâce à une représentation de la politique. Néanmoins, la variance de l'estimation de la performance d'une politique est très forte. Les méthodes *critic-only* permettent d'introduire du biais pour réduire cette variance à travers la représentation des fonctions de valeurs. Néanmoins, ces méthodes sont peu efficaces pour gérer des espaces continus. Enfin, les architectures *acteur-critique* tentent de s'approprier les avantages des deux méthodes précédentes : un critique pour réduire la variance et un acteur pour gérer les espaces continus.

Formellement, cette thèse s'intéresse à trouver des politiques dans des MDP avec environnements continus (actions et états), en temps discrétisé, sans modélisation de l'environnement (*model-free*), en utilisant des modèles non linéaires, tels que les réseaux de neurones, pour apprendre les fonctions de valeur et les politiques avec des méthodes ayant une complexité linéaire en nombre de paramètres.

Nos contributions utiliseront essentiellement des architectures *acteur-critique*, en optimisant le critère MSTDE, en combinaison avec des approches *fitted* pour permettre une meilleure *efficacité en données*.

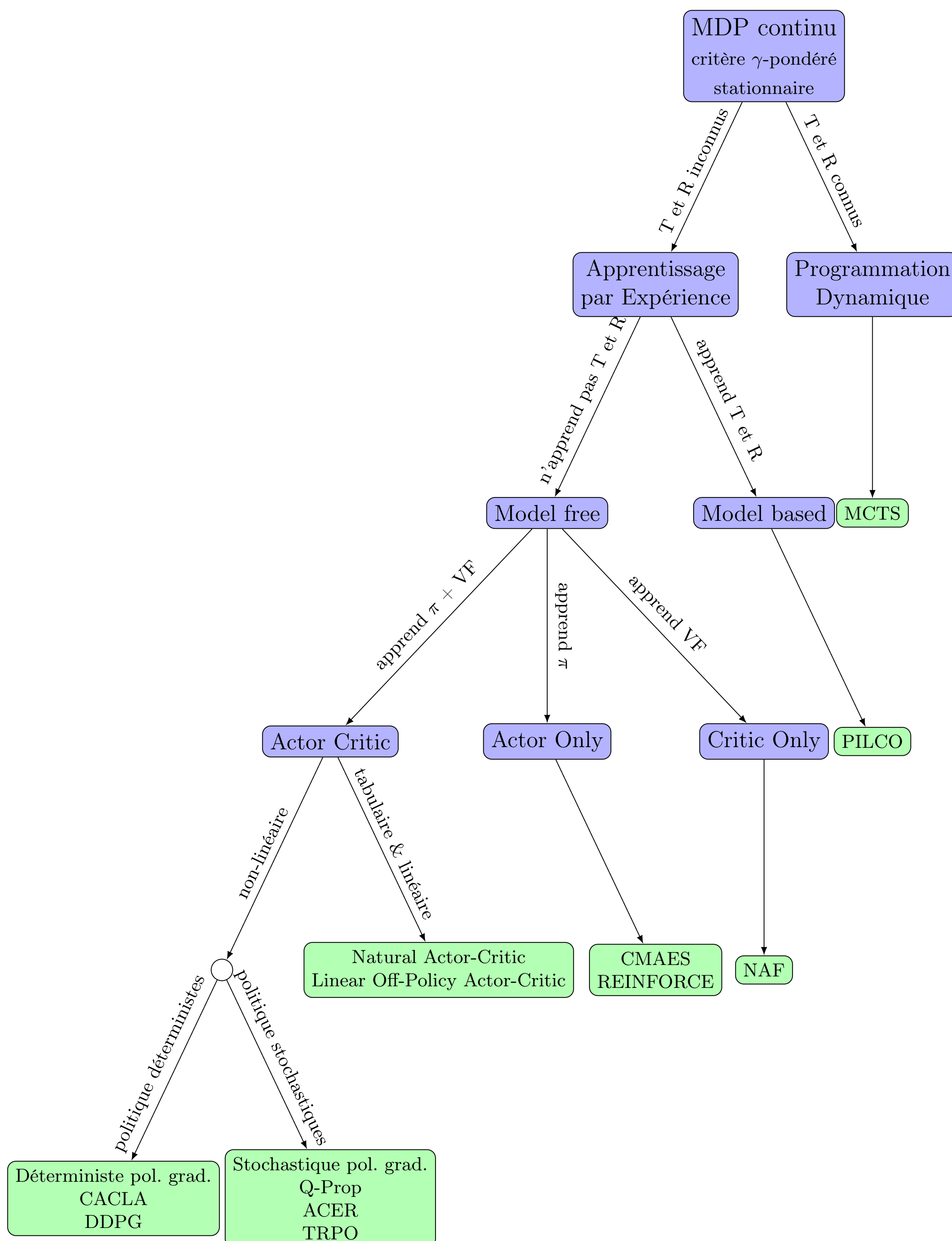


FIGURE 2.12 – État de l'art sur la résolution des MDP continus. Le chemin le plus à gauche liste les hypothèses faites dans cette thèse. Les algorithmes qui n'ont pas été introduits sont les suivants : *Monte-Carlo Tree Search* (MCTS) et *Normalized Advantage Function* (NAF).

Chapitre 3

Environnements continus

Dans ce chapitre, nous allons décrire les environnements sur lesquels nous avons validé expérimentalement les différents algorithmes que nous proposons. Nous nous sommes appuyés sur quatre environnements continus en état et en action, dont deux en hautes dimensions. En nous inspirant du framework OpenAI Gym (Brockman *et al.*, 2016) et du moteur Mujoco (Todorov *et al.*, 2012), payant et non libre, nous avons développé les quatre environnements avec un moteur physique libre : *Open Dynamics Engine* (ODE) (Smith, 2005). Le code source de ces environnements est disponible à l'adresse suivante : `https://github.com/matthieu637/ddrl`.

Dans ce chapitre, et uniquement dans celui-ci, l'utilisation du symbole π ne représente pas une politique, mais bien la constante d'Archimède. La constante gravitationnelle est la même pour tous les environnements : $9.81 \text{ m} \cdot \text{s}^{-2}$.

3.1 Cartpole

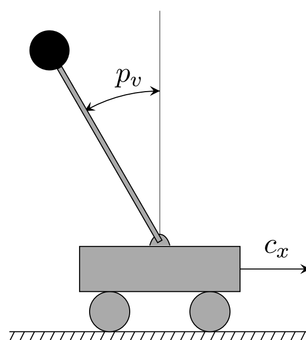


FIGURE 3.1 – Illustration du problème du Cartpole. Adapté de Wikipédia (2017a).

Cartpole (Riedmiller *et al.*, 2007), aussi appelé le problème du pendule inversé, est constitué de deux articulations (Figure 3.1). La première articulation, à laquelle aucun couple ne peut être appliqué, est un point de pivot entre le chariot et le pendule. La seconde articulation contrôle le mouvement horizontal du chariot. L'objectif de l'agent est de garder le pendule en équilibre en déplaçant le chariot horizontalement.

L'espace des états $\mathcal{S}$ est constitué de quatre informations : l'abscisse du chariot dans le plan avec sa vitesse et l'angle du pendule avec sa vitesse $(c_x, \dot{c_x}, p_\nu, \dot{p_\nu}) = s \in \mathcal{S}$. L'ensemble des états

absorbants $\mathcal{S}^*$ est défini tel que :

$$\mathcal{S}^* = \left\{ s \in \mathcal{S} \;\middle|\; |c_x| \geq 2.4 \cup |p_\nu| \geq \frac{\pi}{6} \right\}.$$

Ils correspondent aux états où l'agent a échoué, parce que le pendule a perdu son équilibre ou que le chariot s'est trop éloigné.

La fonction de récompense est définie telle que :

$$R(s,a) = \begin{cases} 0 & \text{si } |c_x| \leq 0.05 \cap |p_\nu| \leq \frac{\pi}{60}, \\ -2(500 - L) & \text{si } s \in \mathcal{S}^*, \\ -1 & \text{sinon.} \end{cases}$$

Rappelons que L définit la taille de la trajectoire. Cette fonction de récompense permet de hiérarchiser 3 types de comportements : les plus mauvais où le pendule tombe, ceux qui maintiennent le pendule jusqu'au bout, mais sont instables et enfin ceux qui le maintiennent avec stabilité.

La taille maximale d'une trajectoire est de 500 pas de temps (10s de temps simulé) avec $\gamma = 0.99$. L'utilisation de l'information L dans la fonction de récompense, alors qu'elle n'est pas présente directement dans l'état, rend l'environnement partiellement observable, mais nous considérerons, tout de même, le problème comme s'il s'agissait d'un MDP (Section 2.1.2 page 29).

L'état initial est aléatoire avec l'angle du pendule tiré uniformément sur $p_\nu \in [-\frac{\pi}{18}; \frac{\pi}{18}]$, de même que la position du chariot $c_x \in [-0.5 \text{ m}; 0.5 \text{ m}]$. La densité du chariot et du pendule vaut $1062 \text{ kg} \cdot \text{m}^{-3}$. L'action proposée par l'agent est mise à l'échelle linéairement sur $[-10 \text{ N}; 10 \text{ N}]$. Le chariot pèse 1 kg et le pendule 0.1 kg. Le chariot est un cube dont les arrêtes ont la taille de $\frac{1}{1062}^{\frac{1}{3}} \approx 0.098$ m. Le pendule est un parallélépipède rectangle de hauteur 1 m et ses largeurs valent $\sqrt{\frac{0.1}{1062}} \approx 0.009$ m. La fréquence de décision et de simulation est de 50Hz. Il n'y a ni friction, ni collision, ni inertie.

3.2 Acrobot

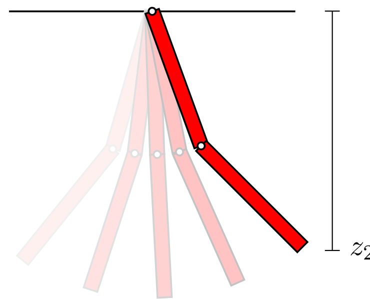


FIGURE 3.2 – Illustration de l'environnement Acrobot. Adapté de Wikipédia (2017c).

L'acrobot (Spong, 1995) est un bras de deux articulations sous-actionnées (Figure 3.2). La première articulation ne peut pas exercer de couple contrairement à la seconde. L'objectif de l'agent est de trouver un enchaînement qui lui permettra de s'élever à la verticale.

L'espace des états $\mathcal{S}$ est constitué de quatre informations : les angles des deux articulations et leurs vitesses. En supposant que la hauteur de l'extrémité de la seconde articulation est calculable

par la fonction z_2, l'ensemble des états absorbants $\mathcal{S}^*$ est défini tel que :

$$\mathcal{S}^* = \left\{ s \in \mathcal{S} \,\middle|\, \frac{z_2(s)}{\max\limits_{s' \in \mathcal{S}} z_2(s')} > 0.99 \right\}.$$

Ils correspondent aux états buts où l'agent a réussi à s'élever à la verticale.

La fonction de récompense est définie telle que :

$$R(s,a) = \begin{cases} 1 & \text{si } s \in \mathcal{S}^*, \\ \max\limits_{t \in \{1,\dots,L\}} \frac{z_2(s_t)}{\max_{s' \in \mathcal{S}} z_2(s')} & \text{si } L = 1500, \\ 0 & \text{sinon.} \end{cases}$$

Si l'agent arrive à se dresser, il reçoit une récompense de 1, sinon il faut atteindre la fin de l'épisode et il recevra une récompense en fonction de la hauteur maximale qu'il a pu atteindre durant l'épisode, mais ne dépassant jamais la valeur 1. On dit que cette fonction de récompense est *hors-ligne*, car elle informe l'agent de sa performance seulement lorsque celui-ci a atteint un état absorbant ou que la simulation se termine.

L'utilisation de L et de l'historique $(s_t)_{\{1,\dots,L\}}$ dans la fonction de récompense, alors qu'ils ne sont pas présents dans l'état, rend la fonction de récompense partiellement observable, mais nous considérons, tout de même, le problème comme étant un MDP (Section 2.1.2 page 29).

Le nombre maximal de pas de temps de simulation est de 1500 (15s en temps simulé) et la taille maximale d'une trajectoire est de 75 transitions avec $\gamma = 0.9$. La fréquence de simulation est de 100Hz et celle de décision 5Hz : l'agent ne perçoit l'état que tous les 10 pas de temps et applique la même action pendant ces 10 pas de temps.

La position initiale est toujours la même (verticale basse). La densité des membres vaut 1062 kg $\cdot$ m^{-3} et leur poids est de 1kg chacun. L'action proposée par l'agent est mise à l'échelle linéairement sur $[-1.5 \text{ N}; 1.5 \text{ N}]$. Chaque membre est un parallélépipède rectangle de hauteur 1 m, ses largeurs valent $\sqrt{\frac{1}{1062}} \approx 0.030$ m. L'inertie sur chaque membre est de 1 kg $\cdot$ m^2. Il n'y a ni friction ni collision.

3.3 Half-Cheetah

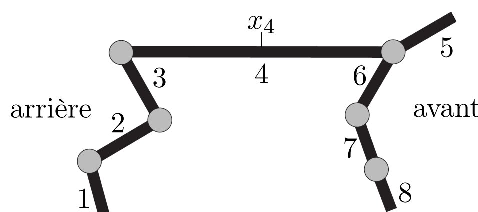


FIGURE 3.3 – Illustration de l'environnement Half-Cheetah. Reproduction de Wawrzyński (2007).

L'environnement *Half-Cheetah* représente un guépard simplifié dans un plan en deux dimensions (Wawrzyński, 2007). Il est composé de 8 membres et 6 articulations (Figure 3.3). L'objectif de l'agent est d'avancer le plus vite possible vers l'avant en interaction avec un sol immobile. L'espace d'actions a une taille de 6 dimensions contrôlant les 6 articulations.

L'espace des états $\mathcal{S}$ est constitué de 18 informations : les angles des 6 articulations avec leurs vitesses angulaires (6×2), l'abscisse et l'altitude du centre de la colonne vertébrale avec leurs vitesses du centre (2×2) et l'angle d'inclinaison de la colonne vertébrale par rapport au sol avec sa vitesse angulaire (1×2). En supposant que la collision de la tête avec le sol soit détectable par la fonction c_5, l'ensemble des états absorbants $\mathcal{S}^*$ est défini tel que :

$$\mathcal{S}^* = \left\{ s \in \mathcal{S} \;\middle|\; c_5(s) \text{ est vrai} \right\}.$$

Ces états correspondent aux échecs survenant lorsque le corps de l'agent est renversé.

En notant $\dot{x}_4$ la vitesse de l'abscisse du centre de la colonne vertébrale, la fonction de récompense est définie telle que :

$$R(s,a) = \begin{cases} -1000 & \text{si } s \in \mathcal{S}^*, \\ \frac{\dot{x}_4}{0.01 \times 5} - 0.1 \times 6 \cdot ||a||_2^2 & \text{sinon.} \end{cases}$$

Cette fonction de récompense est informative et en-ligne : elle favorise les mouvements de l'agent qui utilisent peu de force. Si l'agent tombe sur la tête, la simulation s'arrête et il perçoit une pénalité.

Le nombre maximal de pas de temps de simulation est de 1000 (10s de temps simulé) et la taille maximale d'une trajectoire est de 200 transitions avec $\gamma = 0.99$. La fréquence de simulation est de 100Hz et celle de décision 20Hz : l'agent ne perçoit que l'état tous les 5 pas de temps, et il applique toujours la même action durant ces 5 pas de temps.

Le corps de l'agent commence toujours dans la même position (légèrement surélevé par rapport au sol). La densité des membres vaut 660.9 $\text{kg} \cdot \text{m}^{-3}$ et le poids total est de 14kg. Les poids et les dimensions de chaque membre sont décrits dans la table 3.1. Chaque membre est une capsule : un cylindre où une demi-sphère de même rayon est ajoutée à chaque extrémité. Le rayon de tous les membres est de 0.046 m. L'action proposée par l'agent est mise à l'échelle linéairement en

	Poids	Longueur	Numéro
Patte gauche	1.095 kg	0.188 m	1
Tibia gauche	1.587 kg	0.3 m	2
Fémur gauche	1.543 kg	0.29 m	3
Colonne vertébrale	4.662 kg	1 m	4
Tête	1.587 kg	0.3 m	5
Fémur droit	1.438 kg	0.266 m	6
Tibia droit	1.200 kg	0.212 m	7
Patte droite	0.884 kg	0.14 m	8

Table 3.1 – Poids et dimensions de Half-Cheetah.

fonction de l'articulation (Table 3.2 page suivante). L'inertie de chaque membre est calculée par le moteur physique ODE à partir de la densité, du rayon et de la longueur des capsules (voir la fonction «dMassSetCapsule» de ODE pour plus de détails). À cela s'ajoute l'inertie du rotor, valant ici 0.1 $\text{kg} \cdot \text{m}^2$ qui est ajouté à la diagonale de la matrice d'inertie de chaque membre. Chaque membre peut entrer en collision avec le sol, mais ils ne peuvent se percuter entre eux. Les coefficients de friction valent : 0.4 pour le frottement de glissement (agissant selon les deux axes du plan tangent) et 0.1 pour le frottement de roulement (agissant autour des deux axes du plan tangent). Lors d'une collision, le coefficient CFM (mélange de forces de contraintes) vaut 0.001 : cela permet de simuler un sol très légèrement mou pour stabiliser la simulation en absorbant les forces de collisions.

	Domaine	Moment	Numéros
Cheville gauche	$[-0.4; 0.785]$	$[-60\text{ N}; 60\text{ N}]$	1–2
Genou gauche	$[-0.785; 0.785]$	$[-90\text{ N}; 90\text{ N}]$	2–3
Hanche gauche	$[-0.52; 1.05]$	$[-120\text{ N}; 120\text{ N}]$	3–4
Hanche droite	$[-1; 0.7]$	$[-90\text{ N}; 90\text{ N}]$	5–6
Genou droit	$[-1.2; 0.87]$	$[-60\text{ N}; 60\text{ N}]$	6–7
Cheville droite	$[-0.5; 0.5]$	$[-30\text{ N}; 30\text{ N}]$	7–8

Table 3.2 – Articulations de Half-Cheetah.

3.4 Humanoïde

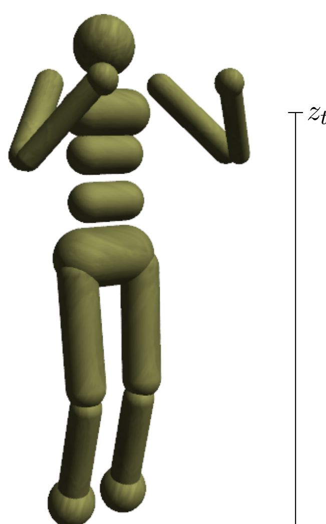


Figure 3.4 – Illustration de l'environnement Humanoïde.

L'environnement Humanoïde (Tassa *et al.*, 2012) représente un squelette humain simplifié dans un monde en 3 dimensions (Figure 3.4). Il est composé de 17 membres avec 10 articulations. Il n'est pas capable de directement mouvoir sa tête ni ses pieds. L'objectif de l'agent est de courir le plus vite possible vers l'avant. L'espace d'actions est composé de 17 dimensions contrôlant les 10 articulations : certaines articulations sont contrôlées par plusieurs dimensions. Par exemple, pour contrôler une épaule, deux dimensions sont nécessaires pour les 2 axes.

L'espace des états $\mathcal{S}$ est constitué de 45 informations :

- la hauteur du centre du thorax,
- la rotation du thorax (quaternion de 4 dimensions),
- les angles des 17 dimensions des 10 articulations,
- la vitesse linéaire du thorax (3 dimensions),
- la vitesse angulaire du thorax (3 dimensions),
- et les vitesses angulaires des 17 dimensions des 10 articulations.

En supposant que la hauteur du centre du thorax soit notée z_t, l'ensemble des états absorbants

$\mathcal{S}^*$ est défini tel que :

$$\mathcal{S}^* = \Big\{ s \in \mathcal{S} \Big| z_t < 0.8 \Big\}.$$

Ces états absorbants correspondent aux échecs où le corps de l'agent perd l'équilibre et tombe trop bas.

En notant $\dot{x}_c$ la vitesse du centre de masse de l'humanoïde, la fonction de récompense est définie telle que :

$$R(s,a) = \begin{cases} 0 & \text{si } s \in \mathcal{S}^*, \\ 3 + 5 \cdot \frac{\dot{x}_c}{0.003} - 0.05 \times 17 \cdot ||a||_2^2 & \text{sinon.} \end{cases}$$

Cette fonction de récompense est informative et en-ligne : elle favorise les mouvements vers l'avant de l'agent qui utilisent peu de force. Si l'agent chute, la simulation s'arrête. Il ne faut pas trop pénaliser la chute, car il est préférable d'avoir un agent qui avance puis chute, plutôt qu'un agent qui ne chute pas, mais n'avance pas.

Le nombre maximal de pas de temps de simulation est de 5000 (15s en temps simulé) et la taille maximale d'une trajectoire est de 1000 transitions avec $\gamma = 0.99$. La fréquence de simulation est de 333 Hz et celle de décision 66 Hz : l'agent ne perçoit que le dernier état sur 5 pas de temps, mais continue d'appliquer la même action sur les états non perçus.

Le corps de l'agent commence toujours dans la même position (légèrement surélevé par rapport au sol). La densité des membres vaut 943 kg $\cdot$ m^{-3} et le poids total est de 39.645kg. Les poids et dimensions de chaque membre sont décrits dans la table 3.3. Rappelons qu'une capsule est un cylindre où une demi-sphère de même rayon est ajoutée à chaque extrémité. L'inertie de

	Solide	Poids	Rayon	Longueur	Inertie du rotor
Pieds	Sphère	1.767 kg	0.075 m		
Tibias	Capsule	2.632 kg	0.049 m	0.3 m	0.006
Fémurs	Capsule	4.525 kg	0.06 m	$\sqrt{0.01^2 + 0.34^2}$ m	0.01
Bassin	Capsule	5.852 kg	0.09 m	0.14 m	0.02
Abdomen	Capsule	2.035 kg	0.06 m	0.12 m	0.02
Thorax inférieur	Capsule	2.113 kg	0.06 m	0.12 m	
Thorax	Capsule	2.852 kg	0.07 m	0.14 m	
Arrière-bras	Capsule	1.594 kg	0.04 m	$\sqrt{3 \cdot 0.16^2}$ m	0.0068
Avant-bras	Capsule	0.877 kg	0.031 m	$\sqrt{3 \cdot 0.16^2}$ m	0.0028
Mains	Sphère	0.244 kg	0.04 m		
Tête	Sphère	3.355 kg	0.09 m		

Table 3.3 – Poids et dimensions de l'environnement Humanoïde.

chaque membre est calculée par le moteur physique ODE à partir de la densité, du rayon et de la longueur. À cela s'ajoute l'inertie du rotor (Table 3.3), qui est ajouté à la diagonale de la matrice d'inertie de chaque membre. L'action proposée par l'agent est mise à l'échelle linéairement en fonction de l'articulation (Table 3.4 page suivante). Chaque membre peut entrer en collision avec le sol et ils peuvent se percuter entre eux. Lorsque deux membres se percutent, il n'y a pas de friction. Les coefficients de friction avec le sol valent : 1 pour le frottement de glissement (agissant selon les deux axes du plan tangent) et 0.1 pour le frottement de roulement (agissant autour des deux axes du plan tangent). Lors d'une collision, le coefficient CFM (mélange de forces de contraintes) vaut 0.001 : cela permet de simuler un sol et un corps légèrement mou

	Degrés de liberté	Domaine	Moment
Genoux	1 (z)	$[-\frac{8\pi}{9}; -\frac{\pi}{90}]$	80 N
Hanches	3 (x, y, z)	$[-\frac{5\pi}{36}; \frac{\pi}{36}] \times [-\frac{\pi}{3}; \frac{7\pi}{36}] \times [-\frac{11\pi}{18}; \frac{\pi}{9}]$	$40 \times 120 \times 40$ N
Bassin-Abdomen	1 (x)	$[-\frac{7\pi}{36}; \frac{7\pi}{36}]$	40 N
Abdomen-Thorax	2 (y, z)	$[-\frac{\pi}{4}; \frac{\pi}{4}] \times [-\frac{5\pi}{12}; \frac{\pi}{6}]$	40×40 N
Épaules	2 (x, z)	$[-\frac{17\pi}{36}; \frac{17\pi}{36}] \times [-\frac{17\pi}{36}; \frac{17\pi}{36}]$	10×10 N
Coudes	1 (y)	$[-\frac{\pi}{2}; \frac{5\pi}{18}]$	10 N

Table 3.4 – Articulations de l'environnement Humanoïde.

pour stabiliser la simulation. Le sol a un coefficient de restitution de 0.05 (rebond) si la vitesse entrante est supérieure à $0.3 \text{ m} \cdot \text{s}^{-1}$.

3.5 Conclusion

Nous avons décrit les quatre environnements qui nous serviront de validations expérimentales (Table 3.5). Ils sont tous définis sur des domaines continus. Bien que certains soient partiellement observables, nous ferons l'hypothèse que définir nos politiques sur l'espace des observations est suffisant pour les résoudre (Section 2.1.2 page 29). Ces environnements sont inspirés de OpenAI Gym, mais contrairement au simulateur Mujoco, ils sont libres et gratuits. La gestion du nombre élevé de dimensions représente une difficulté que nous espérons traiter à l'aide de nos algorithmes d'apprentissage par renforcement développemental que nous présenterons dans les chapitres suivants.

	Cartpole	Acrobot	Half-Cheetah	Humanoid
Récompense en ligne	Oui	Non	Oui	Oui
État initial aléatoire	Oui	Non	Non	Non
Articulations contrôlables	1	1	6	10
$d_{\mathcal{S}}$	2	4	18	45
$d_{\mathcal{A}}$	1	1	6	17
Dimensions espace sensori-moteur	2	4	108	765
Nombre maximal de décisions	500	75	200	1000
Temps simulation	10 sec.	15 sec.	10 sec.	15 sec.
γ	0.99	0.9	0.99	0.99

Table 3.5 – Comparaison des caractéristiques des quatre environnements servant de validation expérimentale. Le nombre de décisions et le temps de simulation sont donnés sur un épisode.

Chapitre 4

Architectures acteur-critique neuronales on-policy

Dans ce chapitre, nous allons proposer un nouvel algorithme respectant nos deux exigences principales :

— gérer les espaces continus d'états et d'actions afin de résoudre des problèmes plus réalistes

— minimiser la connaissance nécessaire du concepteur pour laisser l'agent acquérir progressivement ses propres représentations dans une perspective de robotique développementale (Asada *et al.*, 2009)

Pour cela, nous proposerons un algorithme *acteur-critique* ayant une bonne efficacité en données dans un cadre *on-policy*. À cette fin, nous utiliserons un *replay buffer* (Section 2.5.2 page 59), le principe des méthodes *least-squares* (Section 2.5.2 page 60) en nous appuyant sur CACLA (Section 2.4.3 page 56) et NFQ (Algorithme 2.6 page 45).

Nous nous intéressons à l'apprentissage de politiques déterministes pour plusieurs raisons :

— lorsque l'espace d'actions est grand, les politiques déterministes sont plus simples à apprendre (Silver *et al.*, 2014),

— la seconde raison, plus pragmatique, est qu'il est plus simple d'évaluer une politique déterministe durant la validation expérimentale. En effet, il est difficile de comparer correctement les différents algorithmes existants à cause d'un nombre de métaparamètres élevé et de la forte stochasticité (Henderson *et al.*, 2017).

Avant de proposer notre algorithme, nous proposerons une comparaison de plusieurs algorithmes *model-free online* pour justifier nos choix futurs. Tout au long du chapitre, nous comparerons notre algorithme à plusieurs algorithmes de l'état de l'art chaque fois que cela est possible.

4.1 État de l'art des algorithmes *online semi-gradient*

Dans cette première partie, nous allons mener une série d'expériences montrant les avantages de l'algorithme CACLA lors de l'apprentissage de politiques déterministes. Nous allons comparer 3 algorithmes *acteur-critique* présentés dans l'état de l'art (Section 2.4 page 52) sur l'environnement Half-Cheetah (Section 3.3 page 65). Ce sont, à notre connaissance, les seuls algorithmes *acteur-critique online* et *on-policy*.

Ils peuvent être considérés comme exécutant une politique *non stationnaire* durant la phase d'exploration, car la politique est mise à jour à chaque pas de temps : ce n'est pas la même politique qui est exécutée au long d'un épisode.

Nous distinguons la politique d'exploration (stochastique) $\mu_{\theta,\sigma}$ de la politique déterministe à apprendre π_θ. Elles sont liées par la formulation suivante : $\mu_{\theta,\sigma} = f(\pi_\theta, \sigma)$ où f est une fonction stochastique et σ contrôle la stochasticité. Par exemple, f peut prendre la forme de la politique d'exploration Gausienne (Définition 2.24 page 58). Puisqu'en phase de test nous allons évaluer une politique π_θ différente de celle d'exploration $\mu_{\theta,\sigma}$, on pourrait argumenter que le cadre n'est pas totalement *on-policy*. Dans la littérature, on considère tout de même ce cas comme étant *on-policy*, car on apprend les fonctions de valeur de la politique d'exploration μ, qui est bien celle exécutée, et, puisque σ est fixé, il est équivalent d'améliorer la politique π_θ ou $\mu_{\theta,\sigma}$. Nous nous autoriserons à réduire $\mu_{\theta,\sigma}$ à la notation μ_θ, car la stochasticité σ est fixée et considérée comme étant un métaparamètre. Nous allons maintenant décrire les 3 algorithmes résumés dans la table 4.1 page suivante.

4.1.1 SAC

Le premier algorithme repose sur le gradient du coût d'une politique stochastique (Sutton *et al.*, 1999a) décrit section 2.4.1 page 52 que nous noterons SAC[28]. Il s'agit de l'architecture acteur-critique «classique». Il est possible d'utiliser l'erreur de différence temporelle plutôt que la fonction de valeur Q. Comme détaillé précédemment, les deux variantes sont équivalentes en ce qui concerne le biais. L'apprentissage de politique déterministe avec le gradient du coût d'une politique stochastique (2.4.1 page 52) est possible en désactivant toute stochasticité lors de la phase de test et en considérant σ comme un métaparamètre qui n'est pas optimisé par le gradient.

4.1.2 DAC

La seconde approche repose sur le gradient du coût d'une politique déterministe (Théorème 2.3 page 55) que nous noterons DAC[29] décrit en section 2.4.2 page 55. Plus précisément, nous avons développé l'algorithme dénommé *On-Policy Deterministic Actor-Critic* dans les travaux de Silver *et al.* (2014).

4.1.3 CACLA

Enfin, le dernier algorithme évalué se base sur CACLA (Section 2.4.3 page 56). Cacla utilise l'erreur de différence temporelle $\delta = R(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t)$ pour mettre à jour à la fois son critique et son acteur. L'erreur temporelle contient 2 types d'information sur la façon dont la fonction de valeur se trompe : un changement récent de la politique ou un manque de précision du critique. En général, le critique cherche à faire diminuer δ pour se rapprocher de la fonction de valeur réelle. À l'inverse, l'acteur cherche, avec sa stratégie d'exploration, où δ peut être rendu large, pour ensuite agir en conséquence.

4.1.4 Bilan des mises à jour

Tous les critiques seront appris par la méthode *semi-gradient* TD(0) (Algorithme 2.5 page 44) minimisant l'erreur MSTDE (Section 2.2.7 page 40). La mise à jour des critiques est bien *on-policy*, car on apprend la fonction de valeur de la politique d'exploration μ, c'est-à-dire celle à exécuter. Dans le même temps, on cherche à améliorer la politique déterministe π. Puisque la politique $\mu_{\theta,\sigma} = f(\pi_\theta, \sigma)$, la mise à jour des acteurs fait l'hypothèse que $Q^\mu_\theta \approx Q^\pi$.

28. Pour *Stochastic Actor-Critic*
29. Pour *Deterministic Actor-Critic*

	Critique	Mise à jour de l'acteur
SAC	Q^μ	$A^\mu(s_t,a)\frac{\partial \log \mu_\theta(a\|s_t)}{\partial\theta}\Big\|_{a\sim\mu_\theta(\cdot\|s_t)}$
DAC	Q^μ	$\frac{\partial Q^\mu(s_t,a)}{\partial a}\frac{\partial\pi_\theta(s_t)}{\partial\theta}\Big\|_{a=\pi_\theta(s_t)}$
CACLA	V^μ	$\mathbb{1}_{\hat\delta_\mu(s_t,s_{t+1},a_t)>0}\Big(a_t-\pi_\theta(s_t)\Big)\frac{\partial\pi_\theta(s_t)}{\partial\theta}\Big\|_{a_t\sim\mu_\theta(\cdot\|s_t)}$

TABLE 4.1 – Comparaison des algorithmes *acteur-critique online*.

4.1.5 Validation expérimentale

En plus de la comparaison des 3 algorithmes précédents, nous ajoutons deux expériences de contrôle : la performance d'une méthode *actor-only* en utilisant l'algorithme CMA-ES (Section 1.2.3 page 19) et la performance d'une recherche aléatoire dans l'espace des paramètres. Ces deux expériences de contrôle ne sont pas *online*.

Politique d'exploration Dans cette expérience, la forme de la fonction d'exploration f liant la politique d'exploration $\mu_{\theta,\sigma}$ à celle d'exploitation π_θ, par $\mu_{\theta,\sigma}=f(\pi_\theta,\sigma)$, est une Gaussienne tronquée sur l'intervalle $[-1;1]$.

$$\mu_{\theta,\sigma}(a|s)\sim\frac{e^{-\frac{1}{2}\Big(\frac{a-\pi_\theta(s)}{\sigma}\Big)^2}}{\sigma\sqrt{2\pi}\Big(F\Big(\frac{-1-\pi_\theta(s)}{\sigma}\Big)-F\Big(\frac{1-\pi_\theta(s)}{\sigma}\Big)\Big)},$$

où $F(x)=\frac{1}{2}(1+\text{erf}(\frac{x}{\sqrt{2}}))$ est la fonction de répartition de la loi normale. La fonction $\pi_\theta(s)$ correspond à la sortie déterministe du réseau de neurones de l'acteur.

Réseau de neurones Chaque algorithme est composé de deux réseaux de neurones : un pour l'acteur et un pour le critique. Contrairement aux chapitres 1 et 2 où nous pouvions utiliser le symbole θ pour désigner les paramètres à apprendre, à partir de maintenant, nous distinguons les paramètres du critique w et les paramètres de l'acteur θ.

Comme représenté dans la figure 4.1, les réseaux des critiques disposent d'une seule unité de sortie. La dimension de la sortie de la politique est la même que celle de l'espace des actions $\mathcal{A}$.

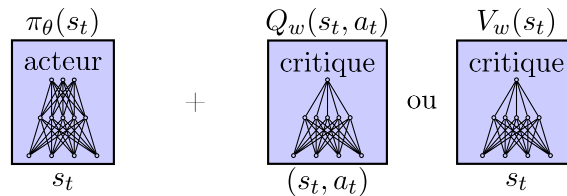


FIGURE 4.1 – Forme des réseaux de neurones avec une architecture *acteur-critique*.

Pour utiliser les formules de la table 4.1 page précédente avec des réseaux de neurones, il est possible d'utiliser l'algorithme de rétro-propagation du gradient (Algorithme 1.5 page 20). Plutôt que de rétro-propager la dérivée de l'erreur MSE, on la remplace par le terme multipliant $\nabla_\theta \pi_\theta(s_t)$. Par exemple, pour CACLA on rétro-propage le vecteur $(a - \pi_\theta(s_t))$ pour la formule complète $\big(a - \pi_\theta(s_t)\big)\nabla_\theta \pi_\theta(s_t)$, puisque l'algorithme de rétro-propagation du gradient se charge de calculer $\nabla_\theta \pi_\theta(s_t)$ en interne.

Implémentation algorithmique Pour CACLA, on rétro-propage le vecteur $(a - \pi_\theta(s_t))$ dans la politique. La fonction de valeur d'états V_w est apprise.

Dans DAC, c'est la fonction Q_w qui est apprise. Pour calculer le terme $\nabla_a Q^\mu(s_t, a)$, on utilise également l'algorithme de rétro-propagation du gradient auquel on rétro-propage la valeur 1 à la sortie du critique pour obtenir $\nabla_a Q^\mu(s_t, a)$ sur l'entrée du réseau. Puis, c'est ce vecteur qui est à son tour rétro-propagé dans le réseau de l'acteur.

Enfin, pour SAC, c'est également la fonction Q_w qui est apprise. Puisque nous utilisons une loi gaussienne tronquée pour l'exploration (la dérivée est la même qu'avec une loi gaussienne non-tronquée) :

$$
\begin{aligned}
A^\mu(s_t, a)\frac{\partial \log \mu_\theta(a|s_t)}{\partial \theta} &= A^\mu(s_t, a)\frac{a - \pi_\theta(s_t)}{\sigma^2}\frac{\partial \pi_\theta(s_t)}{\partial \theta} \\
&= \Big(Q^\mu(s_t, a) - V^\mu(s_t)\Big)\frac{a - \pi_\theta(s_t)}{\sigma^2}\frac{\partial \pi_\theta(s_t)}{\partial \theta} \\
&= \underbrace{\Big(Q^\mu(s_t, a) - \mathbb{E}\Big[Q^\mu(s_t, a')\big|a' \sim \mu_\theta(\cdot|s_t)\Big]\Big)\frac{a - \pi_\theta(s_t)}{\sigma^2}}_{\text{vecteur rétro-propagé dans l'acteur}}\frac{\partial \pi_\theta(s_t)}{\partial \theta}.
\end{aligned}
$$

Le vecteur multipliant le terme $\nabla_\theta \pi_\theta(s_t)$ est rétro-propagé dans l'acteur.

On peut déduire de cette première expérience (Figure 4.2 page suivante) que CACLA est un algorithme prometteur pour créer un algorithme plus efficace en données. Toutes les méthodes sont capables d'apprendre et de surpasser une recherche aléatoire.

Bien que l'on cherche une politique déterministe, on peut la mettre à jour avec le gradient du coût d'une politique stochastique. Même en s'appuyant sur une estimation de la fonction avantage, la méthode SAC a toujours une forte variance. On peut potentiellement expliquer ce phénomène par le fait que la fonction V est estimée par la fonction Q dans cette expérience et ne dispose pas de son propre modèle de représentation. Dans la version la plus basique de SAC, seul $Q(s, a)$ est propagé, sans estimer la fonction avantage, nous n'avons pas réussi à apprendre un comportement satisfaisant avec cette version basique sur Half-Cheetah.

Remarquons que 30000 épisodes correspondent à un maximum de 6 millions de transitions. L'ensemble des métaparamètres utilisés dans cette expérience est donné en annexe A.2 page 130. La structure des réseaux de neurones est la suivante : $18 \times 50 \times 25 \times 6$ unités pour la politique (acteur) et $18(+6) \times 50 \times 25 \times 1$ unités pour les fonctions de valeurs (critique). Nous avons procédé à une métaoptimisation sur chaque algorithme pour un réseau de neurones identique et la même stratégie d'exploration. Pour cela, nous avons utilisé une méthode de recherche en grille parcourant plusieurs ensembles de métaparamètres. Cette métaoptimisation étant très coûteuse en temps de calcul, nous avons utilisé la plateforme Grid5000 pour la réaliser. Une expérience prend en moyenne 1 heure et 30 minutes en utilisant un seul cœur sur des processeurs datant de 2008 jusqu'à 2017 (moyenne 2013). Ces résultats complètent ceux de Van Hasselt (2012) montrant que CACLA est également meilleur dans un environnement à plus haute dimension.

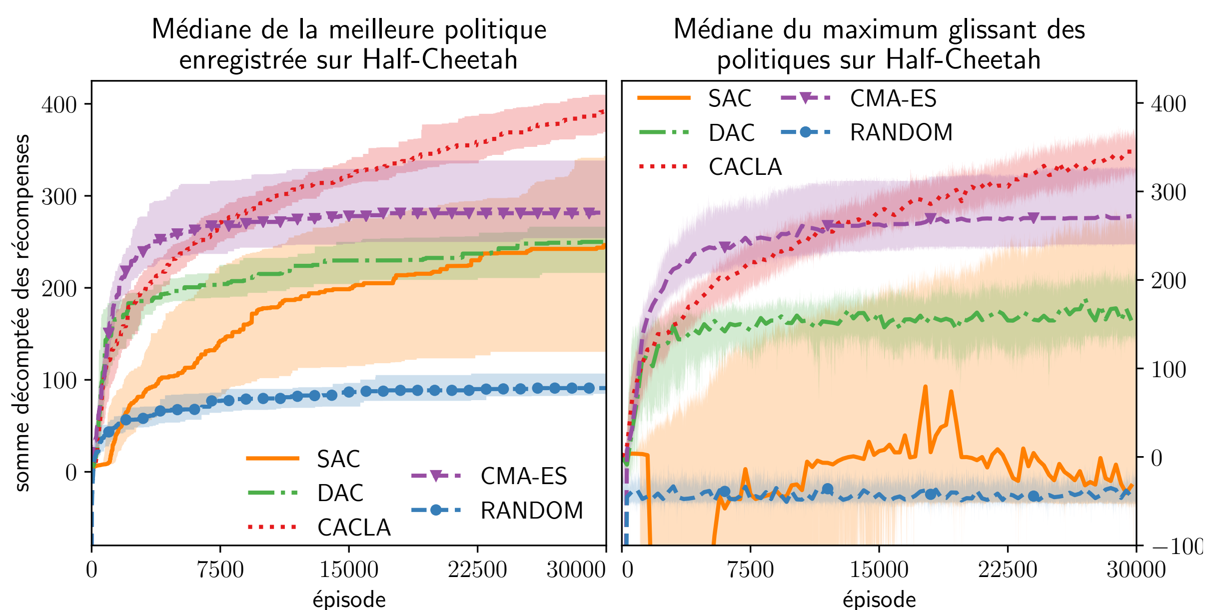


FIGURE 4.2 – Performances des algorithmes *acteur-critique online on-policy* sur 50 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Le maximum glissant est réalisé sur 10 épisodes. Les maximums sont affichés pour pouvoir comparer les algorithmes avec CMA-ES qui utilise intrinsèquement un maximum. La courbe CMA-ES de droite est laissée à titre indicatif, mais ne peut pas être utilisée à titre de comparaison, car l'algorithme implémente un maximum sur l'ensemble de ses solutions. Pour cette raison, les courbes CMA-ES de droite et gauche sont les mêmes.

4.2 Limitation de CMA-ES

Dans cette section, nous allons mettre en œuvre une expérience montrant les limites de CMA-ES lorsque le nombre de paramètres du réseau de neurones augmente. Cela doit aider à la compréhension globale du document, notamment de certaines courbes.

Les expériences les plus anciennes que nous avons réalisées au cours de cette thèse favorisaient l'algorithme CMA-ES au détriment des approches *acteur-critique* à notre insu. Lorsque c'est le cas, nous le ferons apparaître dans la légende de chacune des courbes. En effet, nous nous étions basés sur les performances de CMA-ES pour déterminer la taille de nos réseaux utilisés ensuite par nos algorithmes. Or, comme nous allons le montrer dans cette expérience, CMA-ES favorise une taille réduite de réseaux, même si une meilleure solution existe dans un espace de recherche plus large. Pour montrer cela, nous allons reproduire la même expérience que dans la section précédente sur l'environnement Half-Cheetah en réduisant la taille des réseaux.

La figure 4.3 page suivante nous montre que CMA-ES est meilleur sur les petits réseaux même s'ils sont moins expressifs. Les réseaux ayant une seule couche cachée (à 4 ou 10 unités) permettent à CMA-ES de trouver une meilleure solution qu'avec un réseau à 2 couches cachées. On pourrait alors penser que choisir un réseau de petite taille est une bonne solution ; or ce n'est pas le cas, car nous savons qu'il existe une meilleure solution dans un espace plus grand ; par exemple, avec l'algorithme CACLA sur la figure 4.2. Il faut donc relativiser les bonnes performances présentées par l'algorithme CMA-ES dans certaines courbes. Lorsque le réseau favorise CMA-ES, nous le précisons en légende.

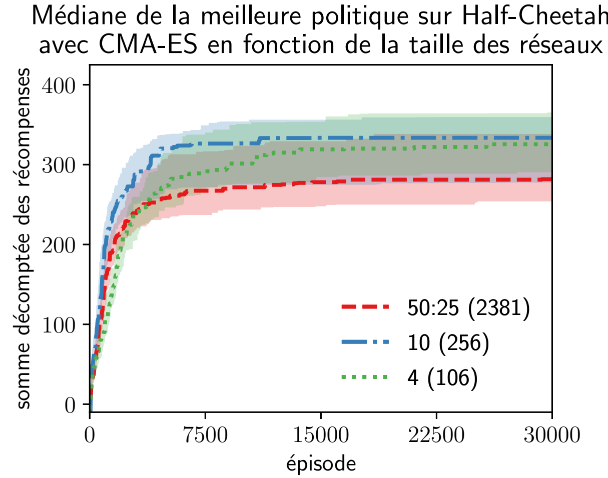


FIGURE 4.3 – Performances de CMA-ES sur Half-Cheetah sur 50 essais. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Les étiquettes décrivent la composition des couches cachées. Le nombre de paramètres correspondant est donné entre parenthèses. Les métaparamètres utilisés peuvent être trouvés en annexe A.3 page 131.

4.3 Neural Fitted Actor-Critic

Dans cette section, nous allons présenter un nouvel algorithme, NFAC[30], plus efficace en données que les algorithmes *model-free online* précédents, toujours en restant dans un cadre *on-policy* (Zimmer *et al.*, 2016a,d). L'un des points faibles pour l'efficacité en données de CACLA est qu'il n'utilise qu'une seule fois les transitions qu'il a générées. L'algorithme NFQ (Algorithme 2.6 page 45) permet justement d'itérer sur les données collectées pour apprendre un critique avec un réseau de neurones. Rappelons ici son équation de mise à jour :

$$w_{k+1} \leftarrow w_k + \alpha \sum_{t=1}^{L} \Big[\big(r_{t+1} + \gamma Q^{\pi}_{w_k}(s_{t+1}, \pi(s_{t+1})) \big) - Q^{\pi}_{w_k}(s_t, a_t) \Big] \frac{\partial Q^{\pi}_{w_k}(s_t, a_t)}{\partial w_k}. \tag{4.1}$$

Rappelons que le symbole w désigne les paramètres du critique et θ ceux de l'acteur. Le problème de NFQ est qu'il ne gère pas les espaces continus d'actions ou très difficilement. Dans sa version originale, NFQ utilise une politique gloutonne avec le terme :

$$Q^{\pi}_{w_k}(s_{t+1}, \pi(s_{t+1})) = \max_{a \in \mathcal{A}} Q^{\pi}_{w_k}(s_{t+1}, a)$$

qui est problématique avec un $\mathcal{A}$ continu. Nous allons reprendre l'idée de NFQ en l'appliquant à une architecture acteur-critique : en particulier pour l'apprentissage de la fonction de valeur V.

NFAC est un nouvel algorithme acteur-critique que nous proposons, il s'inspire à la fois de CACLA et NFQ. Il est *offline*, peut réutiliser les données et traiter les espaces continus. Il procède en deux étapes, résumées dans la Figure 4.4 page ci-contre :

— étant donné la politique courante π et sa version exploratoire μ, il échantillonne et recueille les transitions dans une base de données $\mathcal{D}_\mu$, aussi appelée *replay buffer* (Section 2.5.2 page 59),

30. Pour *Neural Fitted Actor-Critic*

— il améliore l'approximation du critique et de l'acteur à l'aide de $\mathcal{D}_\mu$.

D_μ est composé de plusieurs n-uplet $(s_t, u_t, a_t, r_{t+1}, s_{t+1})$. Pour un état s_t, l'acteur fournit l'action proposée $u_t = \pi_\theta(s_t)$ qui est potentiellement modifiée en une action exploratoire a_t selon μ (ici, par la Gaussienne tronquée) et qui conduit à un nouvel état s_{t+1} avec récompense r_{t+1}. L'action u_t proposée par le réseau de neurones de l'acteur n'est pas exécutée par l'environnement : seule a_t est exécutée.

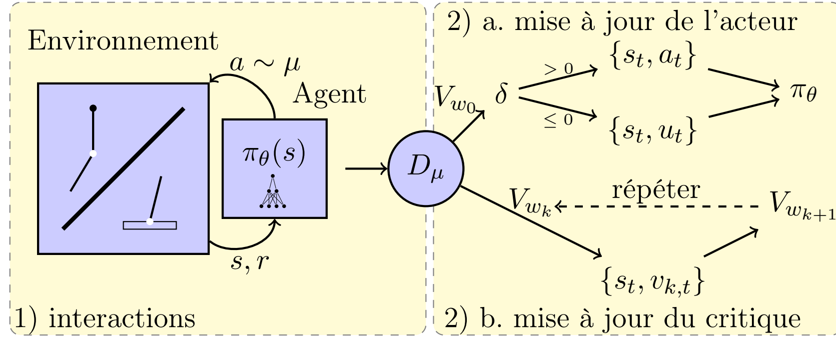


FIGURE 4.4 – Schéma de l'algorithme NFAC.

Comme lors de la première expérience (Section 4.1 page 71), à la fois l'acteur et le critique utilisent des réseaux de neurones comme modèles de représentations. La mise à jour du critique repose sur plusieurs itérations d'apprentissage supervisé, comme dans NFQ. À chaque itération k, à la fin d'un épisode de longueur $L = |\mathcal{D}_\mu|$, une base d'apprentissage $(s_t, v_{k,t})_{\{1,\dots,L\}} \in \mathcal{S}^L \times \mathbb{R}^L$ est construite à partir de $\mathcal{D}_\mu$ en utilisant l'approximation courante V_{w_k} de la fonction de valeur V^μ. Chaque état s_t de $\mathcal{D}_\mu$ est associé à une étiquette (ou cible) $v_{k,t}$ qui est soit $r_{t+1} + \gamma V_{w_k}(s_{t+1})$ ou r_{t+1} lorsque s_{t+1} est un état absorbant. Ainsi, $V_{w_{k+1}}$ est calculé comme suit :

$$V_{w_{k+1}} \leftarrow \operatorname*{argmin}_{V \in \mathcal{F}_c} \sum_{s_t \in \mathcal{D}_\mu} \Big[V(s_t) - v_{k,t} \Big]^2,$$

où $\mathcal{F}_c$ est l'espace de recherche du critique. Pour effectuer cette optimisation, on utilise la descente de gradient (Section 1.2.1 page 16) :

$$\Delta w_{k+1} = - \sum_{s_t \in \mathcal{D}_\mu} \Big(V_{w_k}(s_t) - v_{k,t} \Big) \frac{\partial V_{w_k}(s_t)}{\partial w_k}.$$

La base d'apprentissage doit être reconstruite à chaque itération puisque les cibles $\{v_{k,t}\}$ pour apprendre $V_{w_{k+1}}$ dépendent de V_{w_k}. Contrairement à NFQ qui est potentiellement *off-policy*, ici, en raison de l'architecture acteur-critique, V_w est une évaluation *on-policy* de l'acteur courant. Cela implique que D_μ doit être reconstruite et nettoyée après chaque modification de l'acteur (à la fin de chaque épisode). Il ne s'agit pas exactement de la version *offline* de l'algorithme *semi-gradient* TD(0) (Algorithme 2.5 page 44), car nous utilisons plusieurs itérations pour mettre à jour la fonction de valeur avant de modifier la politique. Cet algorithme est plus semblable à une version *incrémentale* de LSTD (Section 2.2.7 page 45). Néanmoins, puisque nous utilisons des réseaux de neurones, il n'y a plus de garanties de convergence comme avec des modèles linéaires. L'erreur minimisée par le critique sur une itération est MSTDE (Figure 2.8 page 43), mais il est plus difficile de classifier le critère au terme de plusieurs itérations.

La mise à jour de l'acteur modifie la politique afin de renforcer les actions exploratoires a_t qui ont de meilleurs résultats, à travers δ, que les actions d'exploitation u_t. Cette mise à jour repose

sur l'estimation de l'erreur de différence temporelle $\hat{\delta}_t = \big(r_{t+1}+\gamma V_w(s_{t+1})\big) - V_k(s_t)$. Étant donné que $\hat{\delta}_t$ dépend de l'approximation de V^μ, la mise à jour de l'acteur est accomplie avant celle du critique ; ce qui n'a jamais été le cas dans les algorithmes cités auparavant. Comme CACLA, la mise à jour est effectuée vers l'action exploratoire a_t seulement lorsque $\hat{\delta}_t > 0$. Cependant, contrairement à CACLA, le processus n'est pas *online* mais utilise les données de $\mathcal{D}_\mu$:

$$\pi_\theta \leftarrow \underset{\pi \in \mathcal{F}_a}{\arg\min} \sum_{(s_t,u_t,a_t)\in\mathcal{D}_\mu} \begin{cases} \big(\pi(s_t) - a_t\big)^2 & \text{si } \hat{\delta}_t > 0 \\ \big(\pi(s_t) - u_t\big)^2 & \text{sinon,} \end{cases}$$

où $\mathcal{F}_a$ est l'espace de recherche de l'acteur. Comme pour le critique, on utilise une descente de gradient pour réaliser cette optimisation :

$$\Delta\theta = - \sum_{(s_t,a_t)\in\mathcal{D}_\mu} \mathbb{1}_{\hat{\delta}_t>0} \Big(\pi_\theta(s_t) - a_t\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

Remarquons que lors de l'exploitation, où $a_t = u_t$, il n'y a pas de mise à jour de la politique. Contrairement au critique, une seule itération est effectuée par épisode. Il s'agit donc d'une version *offline* de la mise à jour de l'acteur par CACLA. Tandis que CACLA ne peut renforcer qu'une action d'exploratoire à la fois, le cadre *offline* de NFAC permet de modifier la politique avec plus de stabilité en estimant un gradient sur un échantillon complet de données (ou *mini-batch*).

NFAC est décrit en pseudo-code à l'algorithme 4.1 page 85. En plus de gagner en stabilité, il y a plusieurs avantages à avoir une méthode *offline* : pouvoir utiliser la *batch normalization* (Section 1.3.1 page 21) ou l'algorithme RPROP (Algorithme 1.2 page 18) évitant de choisir des valeurs pour les taux d'apprentissage.

4.4 Première validation expérimentale de NFAC

Dans cette section, nous espérons montrer expérimentalement que le passage de CACLA à une méthode *offline* et *fitted* permet de gagner en efficacité en données. À cette fin, CACLA et NFAC sont comparés sur deux environnements continus et déterministes : Acrobot (Section 3.2 page 64) et Cartpole (Section 3.1 page 63). Les différents métaparamètres de cette expérience sont décrits en annexe A.4 page 131. Dans cette expérience, et uniquement dans celle-ci, nous avons utilisé l'algorithme RPROP (Algorithme 1.2 page 18) pour éviter d'avoir à choisir des taux d'apprentissage à la main.

Comme l'illustre la figure 4.5 page ci-contre, NFAC a de meilleures performances que CACLA pour l'apprentissage de politiques déterministes. Néanmoins, puisque c'est le maximum de la phase d'apprentissage qui est affiché, on ne compare pas ici la stabilité des algorithmes. Sur l'environnement Acrobot, 75% des agents NFAC atteignent leurs objectifs après seulement 151 épisodes là où les agents CACLA ont besoin de 544 épisodes. La médiane montre que la qualité des politiques trouvées par NFAC est meilleure. Après 1000 épisodes, 75 % des agents NFAC peuvent atteindre leur but après seulement 313 étapes contre 453 étapes pour CACLA. Sur l'environnement Cartpole, 25% des agents NFAC peuvent préserver leur but pendant 500 étapes après 623 épisodes contre 1419 épisodes pour CACLA. La médiane de NFAC est meilleure que celle de CACLA pendant les premiers épisodes, puis converge après 2200 épisodes.

Ce premier résultat montre que NFAC est prometteur, car il s'avère plus performant que CACLA avec moins de métaparamètres (grâce à RPROP). Or, dans plusieurs environnements et

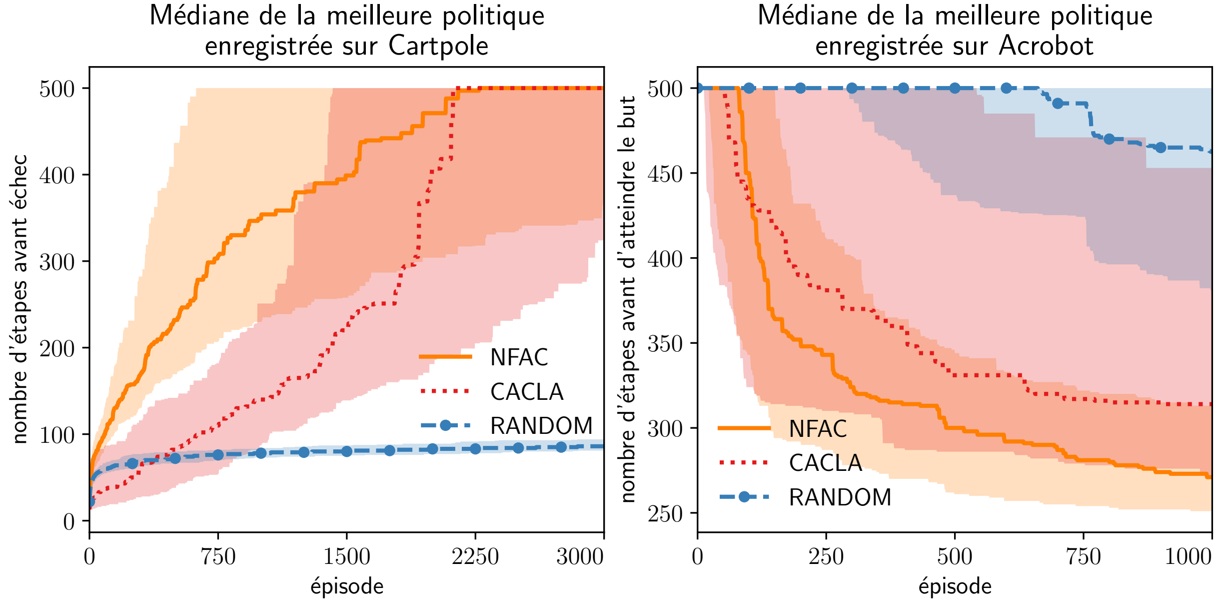


FIGURE 4.5 – Les algorithmes NFAC et CACLA sont évalués sur les environnements Cartpole et Acrobot sur 150 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase d'apprentissage. Sur le Cartpole l'agent doit maximiser son nombre d'étapes, tandis que sur l'Acrobot il doit le minimiser. Reproduction des résultats présents dans Zimmer *et al.* (2016a).

dans notre première expérience (Section 4.1 page 71), CACLA s'est montré meilleur que CMA-ES et Natural Actor-Critic (Van Hasselt, 2012). Nous allons maintenant étudier comment NFAC va pouvoir contrôler le compromis *biais-variance* lors de l'estimation du gradient pour être encore amélioré.

4.5 NFAC(λ) avec traces d'éligibilité

Puisque NFAC enregistre toute une trajectoire dans $\mathcal{D}_\mu$, il est facile d'utiliser les traces d'éligibilité TD(λ) pour améliorer l'estimation des gradients en contrôlant le compromis *biais-variance* à l'aide de λ (Section 2.2.5 page 35 et Algorithme 2.2 page 37). Cela est d'autant plus intéressant que, sur une itération, la complexité reste linéaire en nombre de passages sur chaque transition. L'utilisation des traces d'éligibilité peut être mise en œuvre à la fois pour le critique, afin d'avoir une meilleure estimation de la fonction de valeur vers laquelle il doit tendre, et pour l'acteur, afin d'avoir une meilleure estimation de l'erreur temporelle δ.

Ainsi, dans la mise à jour du critique, rappelée ici :

$$\Delta w_{k+1} = - \sum_{s_t \in \mathcal{D}_\mu} \Big(V_{w_k}(s_t) - v_{k,t} \Big) \frac{\partial V_{w_k}(s_t)}{\partial w_k},$$

$v_{k,t}$ n'est plus calculé uniquement à l'aide d'une seule transition $r_{t+1} + \gamma V_{w_k}(s_{t+1})$, mais à l'aide de l'ensemble des transitions de la trajectoire pondéré exponentiellement par λ. Rappelons que lorsque $\lambda = 1$, il s'agit de l'estimation de Monte-Carlo de V^μ avec une forte variance et sans biais. Lorsque $\lambda = 0$, on donne plus de poids à l'estimateur ce qui réduit la variance mais augmente le

biais. Pour la mise à jour de l'acteur :

$$\Delta\theta = - \sum_{(s_t,a_t)\in\mathcal{D}_\mu} \mathbb{1}_{\hat{\delta}_t>0} \Big(\pi_\theta(s_t) - a_t\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta},$$

les traces d'éligibilité peuvent également aider à l'estimation du gradient. Plutôt que d'utiliser une seule transition $\hat{\delta}_t \leftarrow r_{t+1} + \gamma V_{w_k}(s_{t+1}) - V_{w_k}(s_t)$, on peut utiliser l'ensemble des transitions présentes dans la trajectoire pour estimer $\hat{\delta}_t$ grâce à TD(λ) (Algorithme 2.2 page 37). Dans la littérature, le terme GAE est aussi utilisé pour décrire cette approche (Schulman *et al.*, 2015b). Lorsque $\lambda = 0$, on retrouve l'algorithme NFAC classique décrit en section 4.3 page 76 que nous noterons NFAC(0).

Validation expérimentale Pour vérifier que l'ajout des traces d'éligibilité est bénéfique à NFAC, nous avons reproduit la même expérience que la première du chapitre sur l'environnement Half-Cheetah (celle comparant les méthodes *online* section 4.1 page 71) sur différentes variantes de NFAC(λ). Nous voulons également savoir si les traces d'éligibilité sont plus importantes pour l'acteur ou pour le critique. Les métaparamètres utilisés sont décrits en annexe A.5 page 132. Nous comparerons également CACLA avec NFAC(λ) dans la section suivante.

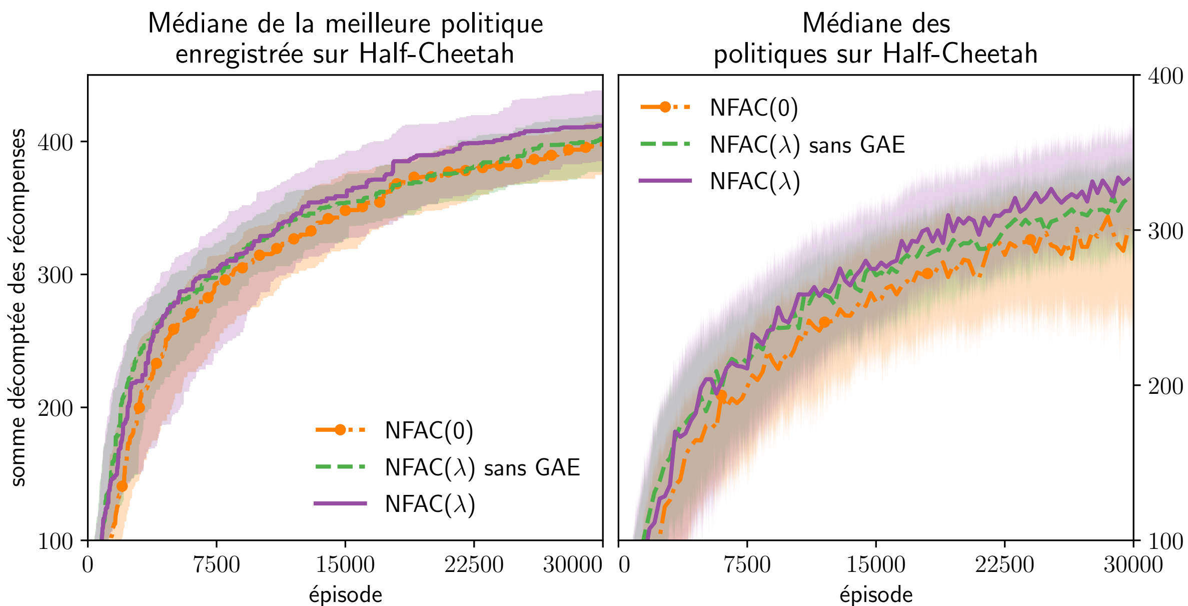


FIGURE 4.6 – Performances des variantes de NFAC(λ) sur 100 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Les maximums sont affichés sur la figure de gauche pour pouvoir comparer les résultats avec ceux de la figure 4.2 page 75 qui ont la même échelle. NFAC(λ) utilise TD(λ) à la fois pour le critique et l'acteur, tandis que NFAC(λ) sans GAE ne l'utilise qu'avec le critique.

La figure 4.6 montre que sur l'environnement Half-Cheetah, TD(λ) apporte une légère amélioration lorsqu'il est utilisé à la fois pour l'acteur et le critique. Il n'y a pas de différence significative entre ce qu'il apporte à l'acteur et au critique. Ces résultats relativement mitigés peuvent être potentiellement expliqués par le fait que NFAC itère déjà plusieurs fois sur les données, même si $\lambda = 0$ puisqu'il s'agit d'une méthode *fitted*. Néanmoins, nous avons observé dans nos expériences

qu'utiliser λ permettait de diminuer le nombre d'itérations K à chaque fin d'épisode dans NFAC (Algorithme 4.1 page 85). TD(λ) permet d'économiser du temps de calcul, ce qui n'est pas visible sur la figure 4.6 page ci-contre.

4.6 Analyses de NFAC

Nous allons nous intéresser à plusieurs aspects de NFAC afin d'entrevoir si d'autres améliorations sont possibles sans faire intervenir de données *off-policy* pour le moment. La première question que nous allons nous poser concerne les fondements théoriques de la mise à jour de l'acteur de CACLA.

4.6.1 Mise à jour de l'acteur par CACLA

Dans la table 4.1 page 73, nous avons rappelé les 3 algorithmes existants pour les acteur-critique *online*. SAC et DAC ont des fondements théoriques forts : ils découlent de théorèmes cherchant à maximiser le critère J (Définition 2.6 page 28). Rappelons que, pour une politique paramétrée, le critère J est le suivant :

$$\theta^* = \arg\max_{\theta \in \Theta} J\big(\pi_\theta\big) = \arg\max_{\theta \in \Theta} \int_{\mathcal{S}} T_0(s_0)\ \mathbb{E}\Big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \ \Big|\ \pi_\theta, s_0\Big] \mathrm{d}s_0.$$

Cependant, la mise à jour de l'acteur par CACLA, proposée dans les travaux de Van Hasselt et Wiering (2007), ne repose que sur l'idée de renforcer une action exploratoire qui a produit un meilleur résultat que celui attendu, sans avoir défini de critère à optimiser et sans faire de lien avec J. Dans un premier temps, nous avons d'abord cru qu'il s'agissait d'une approximation par différences finies du gradient du coût d'une politique déterministe (Théorème 2.3 page 55) :

$$\begin{aligned}
\frac{\partial J}{\partial \theta} =& \mathbb{E}\left[\frac{\partial Q^\pi(s,a)}{\partial a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi \right] && \text{(Théorème 2.3)} \\
=& \mathbb{E}\left[\frac{\partial A^\pi(s,a)}{\partial a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi \right] \\
=& \mathbb{E}\left[\frac{\partial \mathbb{E}[\delta^\pi(s,a,s') | s' \sim T(\cdot|s,a)]}{\partial a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi \right] \\
\approx& \mathbb{E}\left[\frac{\mathbb{E}[\delta^\pi(s,a',s'') | s'' \sim T(\cdot|s,a')] - \mathbb{E}[\delta^\pi(s,a,s') | s' \sim T(\cdot|s,a)]}{a'-a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi \right] \\
\approx& \mathbb{E}\left[\frac{\mathbb{E}[\delta^\pi(s,a',s'') | s'' \sim T(\cdot|s,a')]}{a'-a}\Big|_{a=\pi_\theta(s)} \frac{\partial \pi_\theta(s)}{\partial \theta} \Big| s \sim d^\pi \right].
\end{aligned}$$

En supposant que $a' \sim \mu(\cdot|s_t)$, le signe de la mise à jour de CACLA et de celui de cette approximation de DAC sont les mêmes. CACLA utilise le vecteur $a' - a$ pour mettre à jour sa politique et cette approximation utilise $\frac{\delta}{a'-a}$: les deux sont relativement proches, en particulier lorsque l'on utilise uniquement le signe de ces expressions dans Rprop (Algorithme 1.2 page 18) ou une moyenne glissante dans ADAM (Algorithme 1.3 page 18).

C'est en réalisant la première expérience (Section 4.1 page 71) que nous nous sommes rendus compte que CACLA ressemblait à une déclinaison du gradient du coût d'une politique stochastique (Théorème 2.2 page 53) pour certaines représentations de politiques ; en particulier les

politiques gaussiennes. En effet, si la politique exploratoire μ est gaussienne :

$$A^{\mu}(s_t, a) \frac{\partial \log \mu_\theta(a|s_t)}{\partial \theta} = \underbrace{\mathbb{E}\Big[\hat{\delta}_\mu(s_t, a, s_{t+1}) \big| s_{t+1} \sim T(\cdot|s_t, a)\Big]}_{\text{utilisé par CACLA mais non propagé}} \overbrace{\frac{1}{\sigma^2}}^{\text{négligeable}} \underbrace{\Big(a - \pi_\theta(s_t)\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}}_{\text{partie commune avec CACLA}} .$$

La constante $\frac{1}{\sigma^2}$ est négligeable, car elle peut être intégrée au taux d'apprentissage des méthodes par descente de gradient. CACLA est donc très proche de SAC, à une différence près : seul le signe de $\hat{\delta}_\mu$ est utilisé. Une dernière façon de voir la mise à jour de l'acteur par CACLA, telle que nous l'avions présenté avec NFAC (Section 4.3 page 76), est à travers la minimisation d'un critère MSE. En supposant un temps infini, que la politique d'exploration $\mu(a|s) > 0$ et que tous les états du MDP soient atteignables, CACLA minimise le critère J' suivant :

$$J'(\pi_\theta) = \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} d(s, a, s_{t+1}) \underset{\hat{\delta}_\mu(s_t, s_{t+1}, a) > 0}{\mathbb{1}} \Big(\pi_\theta(s_t) - a\Big)^2,$$

où $d(s, a, s')$ est la densité de probabilité de prendre la transition (s, a, s') en fonction de T et de la politique d'exploration μ. Le critère peut être simplifié par :

$$J'(\pi_\theta) = \int_{\mathcal{S}} \int_{\mathcal{A}} d(s, a) \mathcal{H}(A^{\mu}(s, a)) \Big(\pi_\theta(s) - a\Big)^2,$$

où $\mathcal{H}$ est la fonction de Heaviside. Néanmoins, nous n'avons pas pu montrer qu'il était équivalent à J, car J' est défini dans l'espace des actions, alors que J est défini dans l'espace des récompenses. Finalement, notre intuition nous dit que la SAC et CACLA sont très proches comme nous l'avons montré précédemment, mais nous n'avons pas la preuve qu'ils optimisent exactement le même critère dans le cas général.

4.6.2 Ordre des mises à jour de NFAC

Nous allons nous intéresser à un trait particulier de NFAC pour expliquer l'ordre des mises à jour. Contrairement à beaucoup d'algorithmes *acteur-critique*, NFAC met son acteur à jour avant son critique. Le critique de NFAC a donc un temps de retard par rapport à l'acteur. Pour bien comprendre le processus qui se joue, nous l'avons résumé dans la figure 4.7.

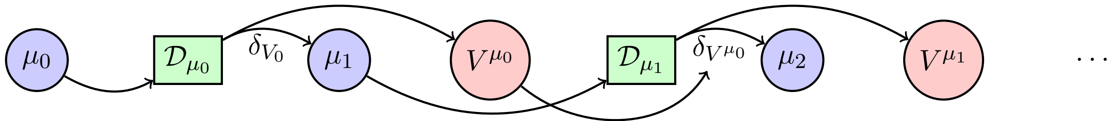


FIGURE 4.7 – Exécution temporelle de NFAC. Les politiques sont représentées par μ, les fonctions de valeurs par V et les données par $\mathcal{D}$. Le cas exceptionnel $t = 0$ n'est pas très intéressant, car V_0 est juste initialisée aléatoirement. Par contre, pour $t = 1$ on voit bien que c'est le critique V^{μ_0} qui met à jour μ_1 vers μ_2. Il a donc un temps de retard sur l'acteur.

Nous pourrions commencer par mettre à jour le critique pour calculer δ. Ainsi, dans le cas de la figure 4.7, nous aurions bien V^{μ_1} qui met à jour μ_1 vers μ_2. Sauf qu'en procédant ainsi, l'erreur temporelle δ aurait déjà été atténuée par l'apprentissage du critique, et la surprise que nous cherchons à capter par les actions exploratoires de la politique d'exploration s'en trouverait amoindrie.

Nous avons procédé à plusieurs expériences, que nous ne reporterons pas ici, car leur importance est négligeable. De ces expériences, nous avons observé qu'il était préférable de laisser la mise à jour de l'acteur avoir lieu en premier, même si le gain engendré n'est pas fondamental.

4.6.3 Comparaisons supplémentaires

Nous procédons à une dernière comparaison de NFAC avec les meilleurs algorithmes cités précédemment, en particulier CMA-ES et CACLA. À la différence que cette fois-ci, CACLA et NFAC utilisent tous les deux l'algorithme ADAM (Algorithme 1.3 page 18) pour l'optimisation des réseaux de neurones. Par la même occasion, nous comparons aussi l'estimation du gradient de A3C (Algorithme 2.9 page 54) pour la mise à jour de l'acteur.

A2C A3C est un algorithme récent et bien connu de l'état de l'art (Mnih *et al.*, 2016). Dans cette expérience, le critique reste appris avec NFAC(λ), seul l'acteur utilise la mise à jour proposée par A3C. De plus, il n'y a aucun aspect d'asynchronicité, contrairement à l'algorithme A3C classique. Rappelons que la mise à jour de l'acteur par A3C est la suivante (Algorithme 2.9 page 54) :

$$\Delta\theta = \sum_{t=0}^{L-1} \left(\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} \Big(\sum_{i=0}^{t-1} \gamma^i r_i + \gamma^t V^\pi(s_{t+1}) - V^\pi(s_t) \Big) \right).$$

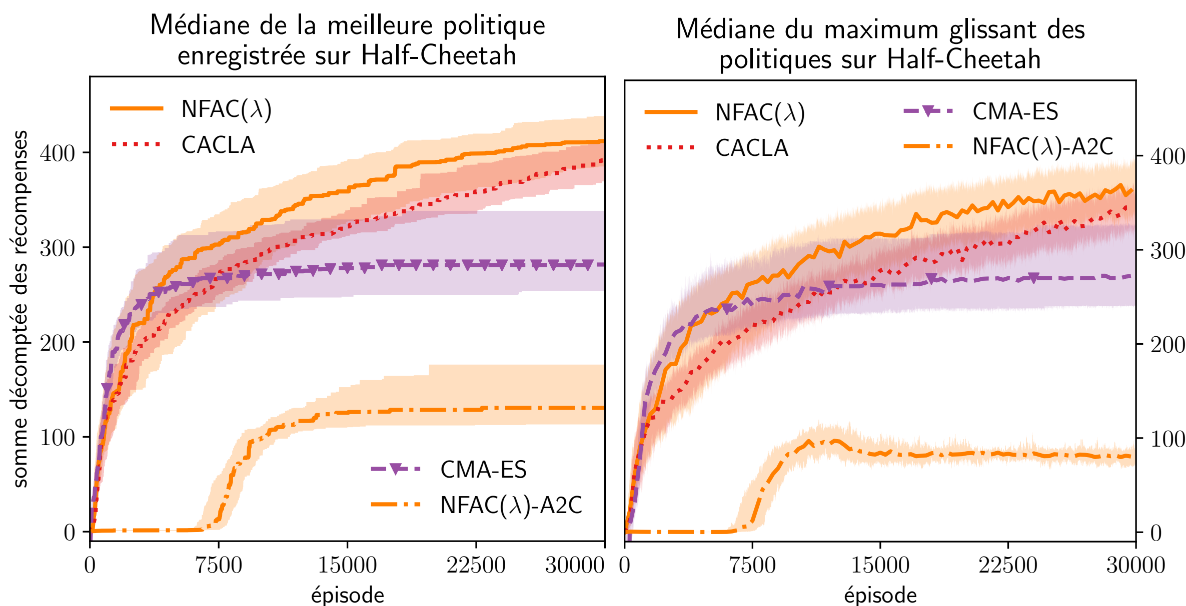


FIGURE 4.8 – Comparaison des meilleurs algorithmes acteur-critique neuronaux *on-policy* sur Half-Cheetah. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration sur 100 essais avec des graines aléatoires différentes. Le maximum glissant est réalisé sur 10 épisodes. Cette figure peut être comparée avec les figures 4.2 page 75 et 4.3 page 76 qui ont la même échelle.

Pour chaque algorithme, la structure des réseaux de neurones est la suivante : $18 \times 50 \times 25 \times 6$ unités pour la politique et $18 \times 50 \times 25 \times 1$ unités pour les fonctions de valeurs. Les métaparamètres

de chaque méthode ont été optimisés avec la même structure de réseaux de neurones et la même stratégie d'exploration. Ils sont décris en annexe A.6 page 132.

Sur la figure 4.8 page précédente, les maximums sont affichés pour pouvoir comparer les algorithmes avec CMA-ES qui utilise intrinsèquement un maximum. La courbe CMA-ES de droite est laissée à titre indicatif, mais ne peut pas être utilisée à titre de comparaison, car l'algorithme implémente un maximum sur l'ensemble de ses solutions. Pour cette raison, les courbes CMA-ES de droite et gauche sont les mêmes.

D'un point de vue implémentation, NFAC(λ) peut utiliser plusieurs cœurs pour ses mises à jour grâce à l'utilisation de *batch*. Ainsi, en moyenne dans cette expérience, NFAC(λ) prend 1 heure et 20 minutes sur des ordinateurs, disposant d'environ 10 cœurs, datant de 2008 jusqu'à 2017 (moyenne 2013). Rappelons que, sur ces mêmes nœuds de Grid5000, CACLA prend en moyenne 1 heure et 30 minutes sur un seul cœur.

On remarque, sur la figure 4.8 page précédente, que la mise à jour de A3C n'apporte pas d'amélioration à NFAC, au contraire même. NFAC(λ) a la meilleure efficacité en données sur cet environnement. CACLA est également bon, nous l'expliquons par le fait que puisque CACLA est *online* il peut s'adapter rapidement.

Nous pensons qu'il est possible d'améliorer encore légèrement NFAC en intégrant des mises à jour plus fréquentes : pas seulement à la fin d'un épisode, mais potentiellement à un intervalle d'étapes donné. Nous n'avons pas mené d'expériences dans ce sens, car nous pensions que le gain serait négligeable par rapport à l'apport de prise en compte de données *off-policy* que nous allons développer dans une seconde contribution.

4.7 Conclusion

Dans ce chapitre, nous avons introduit notre première contribution sous forme d'un algorithme : *Neural Fitted Actor-Critic* (NFAC) et l'avons comparée à de nombreux algorithmes *acteur-critique on-policy* faisant partie de l'état de l'art. Le seul algorithme, à notre connaissance qui est également *on-policy*, avec lequel nous n'avons pas eu le temps d'effectuer de comparaison est TRPO (Section 2.3.3 page 52).

NFAC est un *acteur-critique model-free on-policy offline* plus efficace en données que la plupart des algorithmes *on-policy* qui s'appuie sur des réseaux de neurones pour représenter le critique et l'acteur. Son critique utilise une méthode *fitted* (Section 2.2.7 page 45), inspirée de NFQ (Algorithme 2.6 page 45), d'où son efficacité en données, pour estimer la fonction de valeur d'état V. Son acteur utilise une version *offline* de CACLA (Section 2.4.3 page 56) qui s'appuie sur une déclinaison du gradient du coût d'une politique stochastique (Théorème 2.2 page 53). La mise à jour de l'acteur utilise l'erreur temporelle qui est un échantillonnage de la fonction avantage. Parce que NFAC est *offline*, il est possible d'utiliser facilement les traces d'éligibilité pour gérer le compromis biais-variance (Section 2.2.5 page 35), de profiter de la *batch normalization* (Section 1.3.1 page 21) et de lui appliquer l'algorithme RPROP (Algorithme 1.2 page 18). NFAC peut apprendre à la fois des politiques déterministes ou stochastiques, même si nous avons plus mis l'accent sur les politiques déterministes. Il permet de traiter à la fois des espaces continus d'états et d'actions. Les besoins en connaissances expertes sur le domaine appliqué sont minimaux grâce à l'emploi des réseaux de neurones (estimateurs non-linéaires).

Pour améliorer l'efficacité en données de cette approche, nous allons essayer d'en faire une version *off-policy*, ce qui est le sujet du chapitre suivant.

Algorithme 4.1 (Neural Fitted Actor-Critic(0))

Initialisation aléatoire du réseau du critique V_w
Initialisation aléatoire du réseau de l'acteur π_θ
pour *episode* $\leftarrow 0$ **à** M **faire**
  Percevoir l'état initial s_0
  $t \leftarrow 0$
  tant que $s_t \notin S^*$ **et** $t < L_{max}$ **faire**
    Calculer l'action d'exploitation $u_t \leftarrow \pi_\theta(s)$
    Explorer potentiellement $a_t \sim \mu(\cdot|u_t, \sigma)$
    Effectuer l'action a_t, observer r_{t+1} et s_{t+1}
    Conserver la transition $(s_t, a_t, r_{t+1}, s_{t+1})$ dans le *replay buffer* $\mathcal{D}_\mu$
    $t \leftarrow t + 1$
  fin
  pour $(s_t, r_{t+1}, s_{t+1}) \in \mathcal{D}_\mu$ **faire**

$$\hat{\delta}_t \leftarrow \begin{cases} r_{t+1} - V_w(s_t), & \text{si } s_{t+1} \in S^* \\ r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t), & \text{sinon} \end{cases}$$

  fin
  Mettre à jour l'acteur avec une descente de gradient :

$$\Delta\theta = - \sum_{(s_t,a_t)\in\mathcal{D}_\mu} \mathbb{1}_{\hat{\delta}_t > 0} \Big(\pi_\theta(s_t) - a_t\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

  $w_0 \leftarrow w$
  pour $k \leftarrow 1$ **à** K **faire**
    pour $(s_t, r_{t+1}, s_{t+1}) \in \mathcal{D}_\mu$ **faire**

$$v_{k,t} \leftarrow \begin{cases} r_{t+1}, & \text{si } s_{t+1} \in S^* \\ r_{t+1} + \gamma V_{k-1}(s_{t+1}), & \text{sinon} \end{cases}$$

    fin
    Mettre à jour le critique avec une descente de gradient :

$$\Delta w_k = - \sum_{s_t\in\mathcal{D}_\mu} \Big(V_{w_{k-1}}(s_t) - v_{k,t}\Big) \frac{\partial V_{w_{k-1}}(s_t)}{\partial w_k}.$$

  fin
  $w \leftarrow w_K$
  Vider $\mathcal{D}_\mu$
fin

Chapitre 5

Architectures acteur-critique neuronales off-policy

Dans ce chapitre, nous allons voir comment étendre notre algorithme *Neural Fitted Actor-Critic* pour qu'il puisse prendre en compte des données *off-policy*. Nous unirons différents algorithmes proposés au sein d'un même cadre.

Dans un premier temps, nous utiliserons une seule transition pour réestimer la fonction Q puis, en nous appuyant sur des techniques de l'état de l'art (Section 2.2.6 page 37), nous utiliserons des trajectoires entières. On parle d'algorithme *one-step* lorsque une seule transition est utilisée pour estimer une fonction de valeur ; ce qui correspond également au cas $\lambda = 0$. Inversement, on utilise le terme *multi-step* pour l'usage de tout ou partie de la trajectoire pour estimer une fonction de valeur. En règle générale, les algorithmes *multi-step* sont préférables, car ils améliorent l'estimation des gradients sans modifier la complexité algorithmique, néanmoins ils augmentent de fait la variance de cette estimation. Par exemple, TD(λ) (Algorithme 2.2 page 37) est un algorithme *multi-step* alors que *Semi-gradient TD(0)* (Algorithme 2.5 page 44) est *one-step.*

Dans ce chapitre, nous supposons à nouveau qu'une politique d'exploration $\mu_{\theta,\sigma} = f(\pi_\theta, \sigma)$ est utilisée et qu'on cherche une politique déterministe π_θ.

5.1 Le framework Neural Fitted Actor-Critic

Avant de passer directement au problème *off-policy*, nous allons présenter un cadre permettant de décrire nos différents algorithmes proposés au sein des chapitres 4 et 5.

Jusqu'à présent, nous avons décrit NFAC comme étant un algorithme unique, nous allons maintenant voir que la philosophie derrière NFAC peut s'étendre à différentes instanciations selon la fonction de valeur utilisée et le type de mise à jour de l'acteur. Ainsi, pour unifier les notations, nous utiliserons le terme NFAC(λ)-V ou NFAC-V pour faire référence à l'algorithme 4.1 page 85.

L'idée de NFAC, qui est d'utiliser des méthodes *fitted value iteration* (Gordon, 1995) avec des réseaux de neurones pour gagner en efficacité en données, peut s'étendre notamment à des algorithmes utilisant la fonction de valeur Q. Se pose alors la question de la mise à jour de la politique d'action. Si l'on dispose de Q, alors il est possible d'utiliser le gradient du coût d'une politique déterministe ou la mise à jour de CACLA, contrairement au cas de la fonction de valeur d'état V.

L'avantage d'utiliser la fonction de valeur Q est qu'il est plus simple de prendre en compte des données *off-policy* grâce à la définition même de Q. En effet, rappelons que $Q^\mu(s_t, a_t)$ ne fait

pas d'*a priori* sur la façon dont s_t et a_t ont été générés : il s'agit de l'estimation de la qualité de prendre l'action a_t dans l'état s_t puis de suivre la politique μ. Ainsi, même si $(s_t, a_t, s_{t+1}, r_{t+1})$ ont été générés par une politique $\mu_b \neq \mu$, il est possible d'estimer $Q^\mu(s_t, a_t)$ avec cette seule transition par $Q^\mu(s_t, a_t) \approx r_{t+1} + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))$, ce qui n'est pas aussi simple avec la fonction de valeur V. À l'inverse, la fonction de valeur V est plus simple à apprendre, car elle nécessite moins de données puisque son espace de recherche est réduit étant donné qu'il ne prend en entrée que les états.

Les choix possibles au sein du cadre NFAC sont décrits dans la figure 5.1. Il en résulte 3 algo-

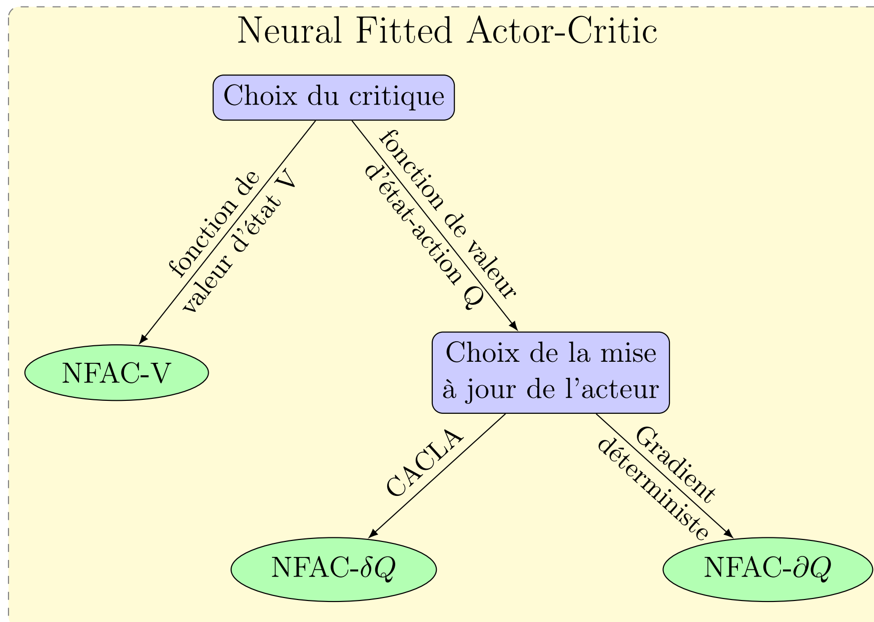


FIGURE 5.1 – Le cadre Neural Fitted Actor Critic et ses différentes instances.

rithmes différents. En réalité, il pourrait en ressortir 5 algorithmes en ajoutant la mise à jour de l'acteur provenant du gradient du coût d'une politique stochastique. Nous avons volontairement omis ce type de mise à jour car il semble être bien moins performant, d'après nos expériences réalisées dans le précédent chapitre (Figure 4.2 page 75), que le gradient du coût d'une politique déterministe ou que la mise à jour de l'acteur proposée dans CACLA pour l'apprentissage de politiques déterministes.

L'utilisation de la fonction Q ouvre la possibilité d'utiliser également des méthodes *fitted policy iteration* pour la mise à jour de l'acteur (Antos *et al.*, 2008) en plus des méthodes *fitted value iteration* (Gordon, 1995) du critique. Une analogie peut être réalisée avec les méthodes *least-square* : utiliser la fonction de valeur Q permet de passer aux algorithmes *least-square policy iteration* (Lagoudakis et Parr, 2003; Buşoniu *et al.*, 2012) pour l'acteur tout en utilisant des algorithmes *least-square value iteration* du critique. De plus, l'apprentissage *off-policy* est simplifié grâce à la définition même de Q. À condition de n'utiliser que la dernière trajectoire pour les mises à jour des acteur-critique, les algorithmes NFAC-Q peuvent potentiellement être *on-policy*. Les prochaines sections de ce chapitre vont décrire les deux nouvelles variantes NFAC(0)-∂Q et NFAC(0)-δQ.

5.2 One step avec Q et le gradient d'une politique déterministe

Dans cette partie, nous allons voir comment utiliser le gradient du coût d'une politique déterministe pour dériver un algorithme *efficace en données*. Rappelons que le gradient du coût d'une politique déterministe passe forcément par l'utilisation de la fonction Q (Théorème 2.3 page 55).

Nous présentons un nouvel algorithme *off-policy offline model-free acteur-critique* pouvant gérer les espaces continus. Il permet de gérer partiellement le compromis soulevé entre *efficacité en données* et *mise à l'échelle* (Section 2.5.2). Nous l'avions initialement nommé DENFAC [31] (Zimmer *et al.*, 2016b,c), néanmoins nous utilisons maintenant plutôt la notation NFAC(0)-∂Q pour le désigner. Cette notation prend tout son sens dans le cadre NFAC (Section 5.1 page 87).

La partie *off-policy* de l'algorithme provient essentiellement de la définition même de Q, car peu importe la politique utilisée pour générer des transitions $(s_t, a_t, s_{t+1}, r_{t+1})$, $Q^\mu(s_t, a_t)$ représente la qualité de prendre précisément l'action a_t dans l'état s_t puis de suivre la politique μ. Un second avantage d'utiliser la fonction Q, plutôt que V, est de pouvoir profiter des algorithmes *fitted policy iteration* pour l'acteur en plus de *fitted value iteration* du critique. Cela signifie que non seulement le critique peut être «*fitted*», mais l'acteur aussi.

5.2.1 DDPG

Parmi les méthodes *off-policy one-step*, l'algorithme DDPG [32] se base sur le gradient du coût J d'une politique déterministe (Théorème 2.3 page 55) pour apprendre des politiques déterministes avec des réseaux de neurones (Lillicrap *et al.*, 2015). Deux réseaux de neurones sont utilisés pour apprendre la politique et la fonction de valeur Q.

Par rapport à l'algorithme *online semi-gradient* DAC (Table 4.1 page 73), DDPG utilise un *replay buffer*, des *targets networks* et la technique de *batch normalization* (Section 1.3.1). Les *targets networks*, qui sont simplement une duplication des réseaux de la politique et de la fonction de valeur, tentent de résoudre ce que Richard Sutton appelle le «*deadly triad*» : l'instabilité de l'apprentissage due à la combinaison des approximateurs, de l'évaluation *off-policy* et du *bootstraping*. Les *targets networks* ralentissent les mises à jour des poids des réseaux. Pour ce faire, les poids des *targets networks* θ' sont mis à jour sur une échelle de temps différente de celle des réseaux clonés : $\theta' \leftarrow \theta' + \alpha\theta$.

Dans notre comparaison, pour la rendre plus juste, nous avons modifié deux aspects de DDPG : la stratégie d'exploration est une simple politique gaussienne tronquée et les actions entrent dans la même couche que les états (la première couche) dans la fonction de valeur Q. Ces deux modifications entraînent probablement une légère dégradation de l'apprentissage par DDPG, mais les autres algorithmes auxquels il est comparé y sont également soumis (sauf CMA-ES).

En supposant qu'on tire aléatoirement un échantillon $\mathcal{D}'$ dans le *replay buffer* $\mathcal{D}$, DDPG utilise les mises à jour suivantes de manière *online* :

$$\Delta w = \sum_{\substack{(s_t, a_t, \\ r_{t+1}, s_{t+1}) \in \mathcal{D}'}} \Big(r_{t+1} + \gamma Q_{w'}(s_{t+1}, \pi_{\theta'}(s_{t+1})) - Q_w(s_t, a_t) \Big) \frac{\partial Q_w(s_t, a_t)}{\partial w}.$$

$$\Delta \theta = \sum_{(s_t, a_t) \in \mathcal{D}'} \frac{\partial Q^\pi(s_t, a)}{\partial a}\Big|_{a=\pi_\theta(s_t)} \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

31. Pour *Data Efficient Neural Actor Critic*
32. Pour *Deep Deterministic Policy Gradient*

où θ représente les paramètres de la politique, w ceux de la fonction de valeur. Les paramètres des *target networks* sont θ' (politique/acteur) et w' (fonction de valeur/critique). Remarquons que c'est la fonction $Q^\pi \approx Q_w$ qui est apprise et non Q^μ. Par ailleurs, les transitions rejouées n'ont pas forcément été générées par π, d'où l'aspect *off-policy*.

5.2.2 Off-policy NFAC(0)-∂Q

NFAC(0)-∂Q fait partie de la famille des *fitted actor-critic* (Section 2.4.3 page 56). Il peut être vu comme une version neuronale de l'algorithme proposé par Antos *et al.* (2008); Hafner et Riedmiller (2011). Le critique est mis à jour à l'aide de NFQ (Algorithme 2.6 page 45), sauf que, comme avec NFAC-V, nous n'utilisons pas d'opérateur max. Pour le critique, l'unique différence avec NFAC(0)-V, qui utilise la fonction de valeur V, est que l'on utilise la fonction de valeur Q. Ainsi, le *replay buffer* $\mathcal{D}$ n'aura plus besoin d'être vidé à chaque fin d'épisode, grâce à la définition même de Q. En effet, rappelons que $Q^\pi(s,a)$ est l'estimation de la qualité d'effectuer l'action a dans l'état s puis de suivre π. Il n'y a pas de condition sur la façon dont s et a sont générés dans la définition de $Q^\pi(s,a)$. Ainsi, un échantillon $(s_t, a_t, s_{t+1}, r_{t+1})$ généré depuis μ peut être utilisé pour mettre à jour $Q^\pi(s,a)$. Cette assertion n'est en général pas vraie lors de l'utilisation d'approximateur de fonctions, car dans ce cas, Q_w peut se spécialiser sur des transitions qui ne seront jamais prises par π. Par ailleurs, cette assertion se limite au cas *one-step* où $\lambda = 0$, car dans un cas *multi step* il faut forcément ajuster les transitions futures. Malgré ce problème, dans cet algorithme, nous nous limitons au cas *one-step* et nous faisons l'hypothèse que la spécialisation de Q n'arrive pas. Un schéma récapitulatif de NFAC(0)-∂Q est fourni dans la figure 5.2.

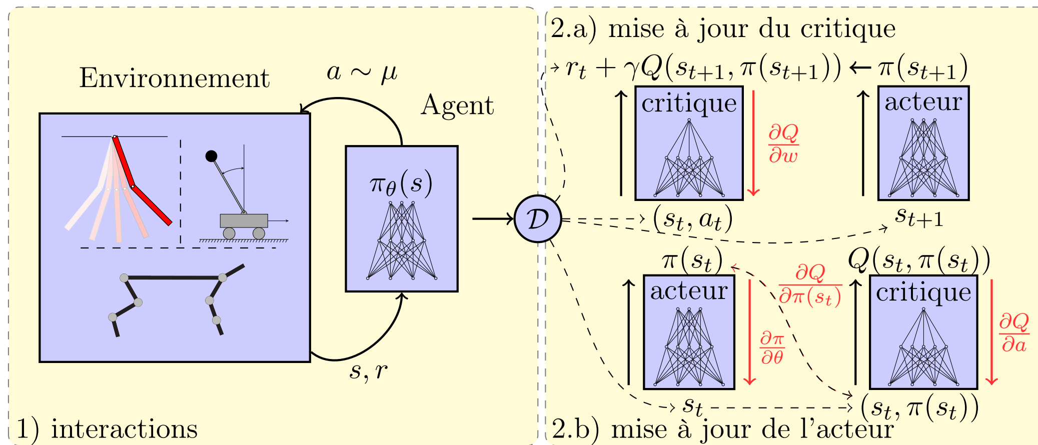


FIGURE 5.2 – Schéma de l'algorithme NFAC(0)-∂Q. Les interactions produites par la politique d'exploration $\mu_{\theta,\sigma} = f(\pi_\theta, \sigma)$ sont recueillies dans le *replay buffer* $\mathcal{D}$. À la fin d'un épisode, le critique est mis à jour à l'aide du replay buffer, et l'acteur est mis à jour à l'aide du critique.

NFAC(0)-∂Q recueille les interactions produites par la politique d'exploration dans un *replay buffer* $\mathcal{D}$. Contrairement à DDPG, les mises à jour de l'acteur et du critique sont faites sur l'ensemble des échantillons étant donnée une taille de *replay buffer* fixée. Le dilemme entre *efficacité en données* et *mise à l'échelle* peut être partiellement contrôlé via la taille du *replay buffer*. Plus il est grand, plus les données seront réutilisées, mais plus l'algorithme est gourmand en temps de calcul. Dans cet algorithme, nous considérons $\mathcal{D}$ comme étant une queue FIFO [33], ainsi

33. Pour *First-In-First-Out*

l'agent dispose d'une mémoire des interactions les plus récentes qu'il a effectuées. L'optimisation réalisée par NFAC(0)-∂Q est la suivante :

$$Q_{k+1} = \underset{Q \in \mathcal{F}_c}{\text{argmin}} \sum_{(s_t, a_t, r_{t+1}, s_{t+1}) \in \mathcal{D}} \Big[Q(s_t, a_t) - \Big(r_{t+1} + \gamma Q_k(s_{t+1}, \pi_k(s_{t+1})) \Big) \Big]^2,$$

$$\pi_{k+1} = \underset{\pi \in \mathcal{F}_a}{\text{argmax}} \sum_{s_t \in \mathcal{D}} Q_{k+1}\big(s_t, \pi_k(s_t)\big).$$

Plusieurs étapes itératives d'optimisation peuvent être appliquées sans récupérer de transitions supplémentaires : il s'agit d'une méthode efficace en données appelée *fitted actor-critic* provenant des méthodes *least-square policy iteration*. En dehors de l'aspect *fitted* et *offline*, DDPG et NFAC(0)-∂Q sont très similaires : ils utilisent tous les deux le gradient du coût d'une politique déterministe pour l'acteur.

Les détails de NFAC(0)-∂Q sont décrits dans l'algorithme 5.1 page 102. Il y est fait mention de deux aspects supplémentaires :

- La remise à zéro des poids des réseaux, qui signifie une réinitialisation aléatoire, une fois les cibles à atteindre calculées, peut aider à sortir des minimums locaux. Ce métaparamètre peut être utilisé à condition que la taille du *replay buffer* soit assez conséquente. La remise à zéro n'entraîne pas une perte totale de ce qui a été appris précédemment, car elle intervient après que les cibles à apprendre ont été calculées.
- L'inversion de gradient permet de gérer des espaces d'actions bornés (Hausknecht et Stone, 2016). Plutôt que de limiter la sortie du réseau de neurones représentant la politique (par exemple avec une fonction tangente hyperbolique à la couche de sortie qui écrase le gradient), il semble préférable d'avoir une couche de sortie non bornée en adaptant le gradient. Cette propriété n'a pas toujours été mise en œuvre dans nos expériences, il s'agit d'un métaparamètre dépendant de l'environnement.

  Pour inverser le gradient, on regarde si la sortie actuelle est en dehors des bornes souhaitées et si le gradient proposé persiste dans la mauvaise direction, alors son signe est modifié :

$$\nabla_a\big|_{a=\pi_\theta(s_t)} \leftarrow \frac{\partial Q_w(s_t, a)}{\partial a} . \begin{cases} (a_{max} - a_t)/(a_{max} - a_{min}) & \text{si } \dfrac{\partial Q_w(s_t, a)}{\partial a} < 0, \\ (a - a_{min})/(a_{max} - a_{min}) & \text{sinon.} \end{cases}$$

5.2.3 Validation expérimentale

Dans cette expérience, nous comparons plusieurs algorithmes : DDPG, CMA-ES, NFAC(0)-V, NFAC(0)-∂Q et CACLA. La comparaison est effectuée sur trois environnements : Acrobot, Cartpole et Half-Cheetah (Section 3.1 à 3.3 page 63 à 65).

Les réseaux utilisent tous la *batch normalization* (Section 1.3.1 page 21) avec ADAM (Algorithme 1.3 page 18) sauf CMA-ES qui ne s'appuie pas sur le gradient. Le critique est composé de deux couches cachées de 50 puis 7 neurones. La structure de l'acteur est la suivante : $1 \times 5 \times 1$ neurones (Acrobot), $1 \times 20 \times 1$ neurones (Cartpole), et $18 \times 20 \times 10 \times 6$ neurones (Half-Cheetah). Ces structures ont été définies en fonction de la performance de CMA-ES, nous nous sommes rendu compte plus tard que CMA-ES était meilleur sur de petits réseaux et donc favorisait l'utilisation de réseaux de petite taille (Section 4.2 page 75), ce qui implique un biais dans cette expérience favorisant CMA-ES.

Pour l'optimisation des métaparamètres, nous avons dans un premier temps cherché ceux de DDPG, puis nous avons utilisé les mêmes sur NFAC(0)-∂Q. Ils sont décrits en annexe A.7 page 133.

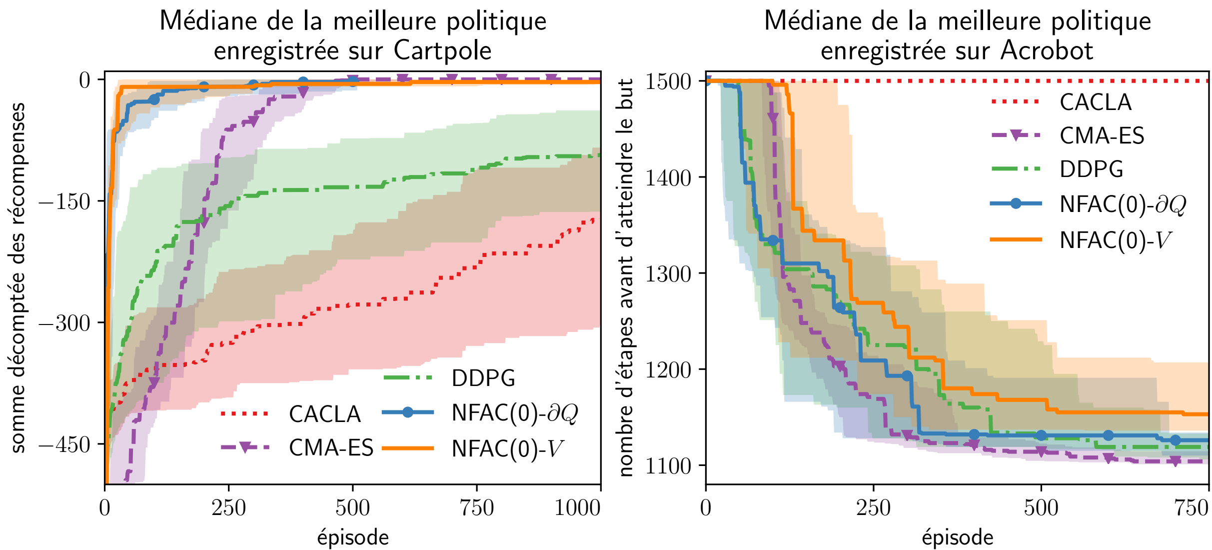


FIGURE 5.3 – Comparaisons de NFAC(0)-∂Q et DDPG sur Cartpole et Acrobot. Les médianes et les quartiles affichés sont calculés durant la phase d'apprentissage sur 40 essais avec des graines aléatoires différentes. Les maximums sont affichés pour pouvoir comparer les algorithmes avec CMA-ES qui utilise intrinsèquement un maximum. Sur Cartpole, plus la courbe est haute, meilleur est l'algorithme, à l'inverse de l'environnement Acrobot.

Les courbes des figures 5.3 et 5.4 montrent les bonnes performances des algorithmes NFAC. Sans utiliser les *target networks*, les algorithmes NFAC sont capables d'apprendre efficacement. On aperçoit que sur ces courbes NFAC(0)-V est aussi bon, voire meilleur, que NFAC(0)-∂Q. Cela laisse sous-entendre que NFAC(0)-∂Q qui est *off-policy* n'est pas plus efficace en données qu'une méthode *on-policy*. Notre intuition pour expliquer ce phénomène est que NFAC(0)-∂Q est *one-step* : il n'exploite pas tout le potentiel des transitions qu'il recueille. On peut aussi spéculer sur la différence de mise à jour de l'acteur (déterministe vs CACLA) ou la difficulté d'apprendre Q par rapport à V.

Notons que cette expérience présente des résultats réalisés dans Zimmer *et al.* (2016b). À ce moment l'environnement Half-Cheetah était relativement différent, plus proche de l'article fondateur de Wawrzyński (2007), alors que, dans le reste de thèse, l'environnement est plus proche de OpenAI (Brockman *et al.*, 2016). C'est aussi le cas pour Cartpole et Acrobot. Ainsi, ces courbes ne peuvent pas être comparées avec les autres courbes fournies dans cette thèse, car les échelles sont différentes.

Étant donné que les figures 5.3 et 5.4 affichent le maximum sur la phase d'apprentissage, on ne compare pas ici la stabilité des algorithmes, mais leur capacité à trouver rapidement une bonne solution quitte à l'oublier rapidement. Dans la suite de ce chapitre, nous aurons l'occasion de comparer leur stabilité en phase de test (Section 5.7 page 100).

Dans ces expériences, NFAC(0)$-V$ est meilleur que DDPG et NFAC(0)-∂Q alors qu'il n'utilise pas de données *off-policy*. Par ailleurs, il est plus simple à implémenter :

— il n'est pas nécessaire de maintenir de *targets networks*,

— l'apprentissage de V est moins coûteux que celui de Q et évite de choisir à quelle couche on insère les actions.

Lors de l'optimisation des paramètres, nous n'avons pas remarqué que l'ajout d'une régularisation

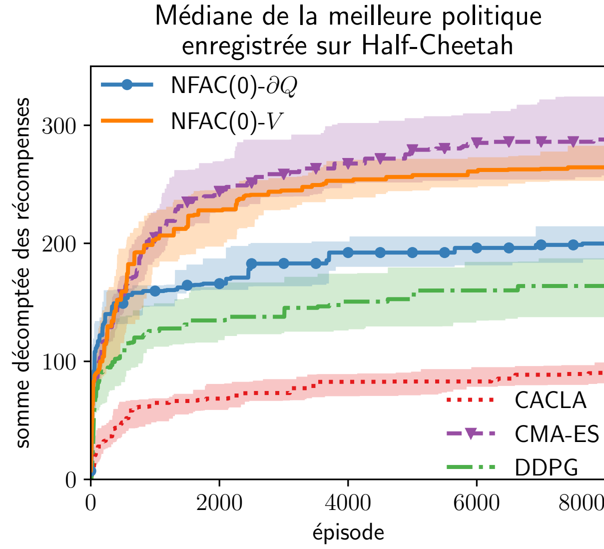


FIGURE 5.4 – Comparaisons de NFAC(0)-∂Q et DDPG sur Half-Cheetah. Les médianes et les quartiles affichés sont calculés durant la phase d'apprentissage sur 40 essais avec des graines aléatoires différentes. Les maximums sont affichés pour pouvoir comparer les algorithmes avec CMA-ES qui utilise intrinsèquement un maximum.

L2 dans le critique améliorait DDPG, contrairement à la version originale.

5.3 One step avec Q et CACLA

Nous nous intéressons maintenant à l'algorithme NFAC(0)-δQ qui utilise une mise à jour de l'acteur inspirée de CACLA et la fonction de valeur Q. Trois formulations théoriques sont légitimes pour cet algorithme. Nous nous attardons moins sur NFAC(0)-δQ car, dans la pratique, il est peu efficace, malgré nos tentatives pour trouver de bons métaparamètres.

Rappelons que la mise à jour de l'acteur proposée par CACLA dépend de V^μ et se calcule grâce à une estimation de la fonction avantage :

$$\Delta\theta = \mathbb{1}_{R(s_t,a_t)+\gamma V^\mu(s_{t+1})-V^\mu(s_t)>0} \Big(a_t - \pi_\theta(s_t)\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

Pour remplacer V par Q, dans la mise à jour de l'acteur, nous procédons de la façon suivante :

$$\gamma V^\mu(s_{t+1}) - V^\mu(s_t) \leftarrow \gamma \mathbb{E}\Big[Q^\mu(s_{t+1}, a)\Big| a \sim \mu(\cdot|s_{t+1})\Big] - \mathbb{E}\Big[Q^\mu(s_t, a')\Big| a' \sim \mu(\cdot|s_t)\Big]$$

Il y a trois façons d'apprendre le critique Q.

On-policy Dans cette première variante, nous cherchons à apprendre Q^μ à partir de données *on-policy* avec une version *fitted* de SARSA(0) (Algorithme 2.3 page 37). L'optimisation réalisée est la suivante :

$$Q_{k+1} = \underset{Q \in \mathcal{F}_c}{\text{argmin}} \sum_{(s_t,a_t,a_{t+1},r_{t+1},s_{t+1}) \in \mathcal{D}} \Big[Q(s_t, a_t) - \Big(r_{t+1} + \gamma Q_k(s_{t+1}, a_{t+1})\Big)\Big]^2.$$

Off-policy π Dans cette seconde variante, nous apprenons la fonction de valeur de la politique déterministe, de la même manière que dans NFAC(0)-∂Q :

$$Q_{k+1} = \underset{Q \in \mathcal{F}_c}{\operatorname{argmin}} \sum_{(s_t,a_t,r_{t+1},s_{t+1}) \in \mathcal{D}} \Big[Q(s_t,a_t) - \Big(r_{t+1} + \gamma Q_k(s_{t+1}, \pi(s_{t+1})) \Big) \Big]^2.$$

Off-policy μ Enfin, dans la dernière variante, nous apprenons Q^μ avec une version *fitted* de l'algorithme Expected Sarsa(0) (Formule 2.3 page 39) :

$$Q_{k+1} = \underset{Q \in \mathcal{F}_c}{\operatorname{argmin}} \sum_{(s_t,a_t,r_{t+1},s_{t+1}) \in \mathcal{D}} \Big[Q(s_t,a_t) - \Big(r_{t+1} + \gamma \mathbb{E}\Big[Q_k(s_{t+1},a') \Big| a' \sim \mu(\cdot|s_{t+1}) \Big] \Big) \Big]^2.$$

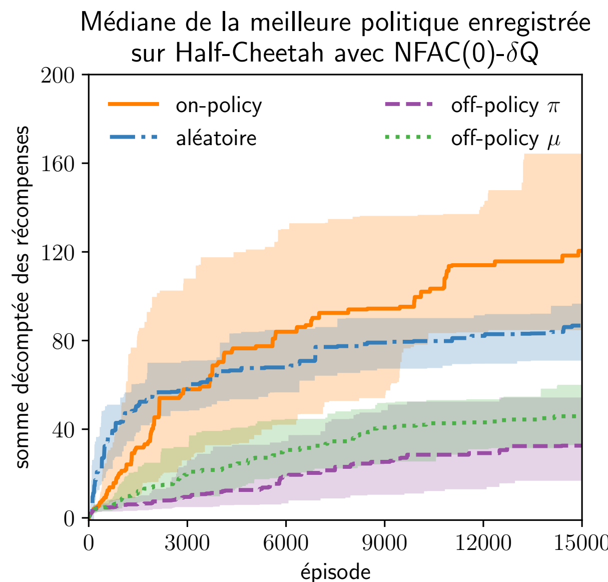


FIGURE 5.5 – Comparaison des variantes de NFAC(0)-δQ sur Half-Cheetah. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 60 essais avec des graines aléatoires différentes.

La figure 5.5 décrit les résultats des 3 variantes de NFAC(0)-δQ. Les métaparamètres de cette expérience sont décrits en annexe (Annexe A.8 page 136). La meilleure variante est celle *on-policy*, néanmoins, on peut déduire de cette figure que la mise à jour de l'acteur proposée par CACLA fonctionne très mal avec la fonction de valeur Q. Nous pensons que cela provient principalement de la difficulté à estimer la fonction avantage à l'aide seulement de Q. Pour pallier cette difficulté, il faudrait essayer une architecture neuronale dédiée à l'estimation de la fonction avantage (Wang *et al.*, 2015, 2016), où les fonctions Q et V ont à la fois des paramètres partagés pour profiter de la généralisation et des paramètres distincts pour se spécialiser.

5.4 Pondération des transitions off-policy

Jusqu'à maintenant, nous nous sommes seulement intéressés à l'apprentissage *off-policy one-step* ($\lambda = 0$) ; or il est possible d'exploiter plus efficacement les données en utilisant leur structure.

En effet, nous savons que les transitions ne sont pas indépendantes les unes des autres : il s'agit d'une trajectoire. Il est possible d'utiliser plusieurs transitions au sein d'une même trajectoire pour estimer une fonction de valeur comme avec l'algorithme TD(λ) (Algorithme 2.2 page 37). Néanmoins dans un cadre *off-policy*, puisque la trajectoire a été générée par une politique différente de celle qu'on est en train d'estimer, il faut pondérer ces transitions. Pour cela, nous utilisons les différentes variantes de l'algorithme *Tree-Backup*(λ) généralisé (Algorithme 2.4 page 39), dont nous rappelons la formulation ici :

$$\Delta\hat{Q}^\pi(s_t,a_t) = \sum_{n=t}^{\infty} \gamma^{n-t}\Bigg(\prod_{k=t+1}^{t+n} c_k\Bigg)\Bigg(r_{n+1} + \gamma\mathbb{E}\Big[\hat{Q}^\pi(s_{n+1},a)\Big|a \sim \pi(\cdot|s_{n+1})\Big] - \hat{Q}^\pi(s_n,a_n)\Bigg).$$

Dans cette formule, le terme c_k a un rôle crucial, car il correspond à la pondération des transitions *off-policy*. Si le produit des c_k est nul, alors on retombe sur le cas *one-step*, *a contrario*, si le produit des c_k est strictement positif alors la $n^{\text{ième}}$ transition est prise en compte. Nous faisons varier les valeurs de c_k pour déterminer la meilleure forme entre 5 propositions (Table 5.1 page suivante).

Dans la suite de ce chapitre, nous différencions 4 politiques. La politique déterministe π_b, et sa politique stochastique d'exploration associée μ_b, servent à générer une partie des échantillons qui sont ensuite utilisés pour estimer π (et donc sa politique d'exploration μ).

Distance L2 L'algorithme *Tree-Backup*(λ) généralisé a été conçu en faisant l'hypothèse que la politique à estimer était stochastique ; or rappelons qu'on cherche à estimer une politique π déterministe. Ainsi, le ratio $\frac{\pi(a_t|s_t)}{\mu_b(a_t|s_t)}$ provenant de l'*importance sampling* devrait correspondre à une fonction de Dirac et ne pourrait exploiter que peu de données dans l'espace des actions. Pour cette raison, nous proposons une valeur de c_k qui n'est pas basée sur ce ratio, mais sur une distance dans l'espace des actions. En supposant que $u_k = \pi_b(s_k)$ est l'action d'exploitation et $a_k \sim \mu_b(u_k, \sigma)$ est l'action effectuée pour générer la transition, nous proposons la mesure suivante pour c_k :

$$c_k \leftarrow \lambda\big(1 - \ \mathrm{min}^\pi_{\mathrm{l2}}(a_k,u_k,s_k)\big) = \lambda\Bigg(1 - \ \min\Big(||a_k - \pi(s_k)||_2, ||u_k - \pi(s_k)||_2\Big)\Bigg)$$

Cette distance donne plus de poids aux transitions dont les actions sont proches de l'action favorisée actuellement par $\pi(s_k)$. Nous pensons que cette distance a du sens lorsque les deux conditions suivantes sont vérifiées :

— la politique π est déterministe
— l'espace des actions $\mathcal{A}$ est continu

En effet, si π n'est pas déterministe, alors $\pi(a_t|s_t)$ peut être utilisée directement. Si $\mathcal{A}$ est discret, la notion d'action *proche* d'une autre n'existe pas forcément. À l'inverse, lorsque les deux conditions précédentes sont respectées, il est raisonnable de faire l'hypothèse qu'une action proche, de celle que ferait la politique déterministe, a des conséquences similaires.

Récapitulatif Nous avons résumé les différentes propositions pour les valeurs de c_k au sein de la table 5.1. Les 4 premières variantes estiment la fonction de valeur V^μ de la politique stochastique d'exploration. Puisque μ est stochastique, il s'agit du cadre classique de l'algorithme *Tree-Backup*(λ) généralisé. À l'inverse, notre proposition basée sur la distance dans l'espace des actions tente d'estimer V^π.

Algorithmes	c_k	Variance estimation	Convergence	Utilise tous les retours
Importance Sampling	$\frac{\mu(a_k\|s_k)}{\mu_b(a_k\|s_k)}$	forte	pour tout μ_b et μ	oui
Tree-backup(λ)	$\lambda\mu(a_k\|s_k)$	faible	pour tout μ_b et μ	non
$Q^\pi(\lambda)$ *off-policy*	λ	faible	μ_b proche de μ	oui
Retrace(λ)	$\lambda \min\Big(1, \frac{\mu(a_k\|s_k)}{\mu_b(a_k\|s_k)}\Big)$	faible	pour tout μ_b et μ	oui
Distance L2(λ)	$\lambda\big(1 - \min_{l2}^{\pi}(a_k, u_k, s_k)\big)$	?	?	oui

TABLE 5.1 – Récapitulatif des avantages et inconvénients des variantes de *Tree-Backup(λ)*. Adapté depuis Munos *et al.* (2016).

5.5 Multi-step avec la fonction de valeur d'état V

Dans cette section, nous allons voir comment NFAC(λ)-V qui est déjà *multi-step*, mais *on-policy*, peut prendre en compte plusieurs trajectoires. Concrètement, le *replay buffer* de l'algorithme 4.1 page 85 n'est plus vidé à chaque fin d'épisode.

5.5.1 Tree-Backup généralisé avec V

Les méthodes *off-policy* s'appuyant sur *Tree-Backup*(λ) généralisé (Algorithme 2.4 page 39) utilisent la fonction Q. Rappelons la formule générale :

$$\Delta\hat{Q}^\pi(s_t, a_t) = \sum_{n=t}^{\infty} \gamma^{n-t} \prod_{k=t+1}^{t+n} c_k \bigg(r_{n+1} + \gamma \mathbb{E}\Big[\hat{Q}^\pi(s_{n+1}, a) \Big| a \sim \pi(\cdot|s_{n+1})\Big] - \hat{Q}^\pi(s_n, a_n) \bigg).$$

Le terme $\mathbb{E}\big[\hat{Q}^\pi(s_{n+1}, a)\big|a \sim \pi(\cdot|s_{n+1})\big]$ peut facilement être remplacé par $\hat{V}^\pi(s_{n+1})$, car la politique π qu'on cherche à évaluer est déterministe :

$$\Delta\hat{Q}^\pi(s_t, a_t) = \sum_{n=t}^{\infty} \gamma^{n-t} \prod_{k=t+1}^{t+n} c_k \bigg(r_{n+1} + \gamma\hat{V}^\pi(s_{n+1}) - \hat{Q}^\pi(s_n, a_n) \bigg).$$

Nous faisons maintenant l'hypothèse que la formule précédente fonctionne avec V^π en ajustant le premier tirage de $a_t \sim \mu$ par $\frac{c_t}{\lambda}$:

$$\Delta\hat{V}^\pi(s_t) = \frac{c_t}{\lambda} \sum_{n=t}^{\infty} \gamma^{n-t} \prod_{k=t+1}^{t+n} c_k \bigg(r_{n+1} + \gamma\hat{V}^\pi(s_{n+1}) - \hat{V}^\pi(s_n) \bigg).$$

Nous voulons voir si cette formulation permet à NFAC(λ)-V de profiter de l'apprentissage *off-policy* sans passer par l'utilisation de la fonction de valeur Q.

5.5.2 Validation expérimentale

Dans cette expérience, nous voulons savoir si la pondération des transitions par les facteurs c_k peuvent aider l'algorithme NFAC(λ)-V pour le rendre *off-policy*. Pour cela, nous procédons à

deux expériences indépendantes, l'une avec une mise à jour du critique *off-policy* et une seconde avec une mise à jour de l'acteur *off-policy*.

Pour ne pas surcharger les figures, nous affichons toujours la performance de Retrace(λ), de la distance L2 que nous proposons, puis la meilleure variante parmi les 3 restantes.

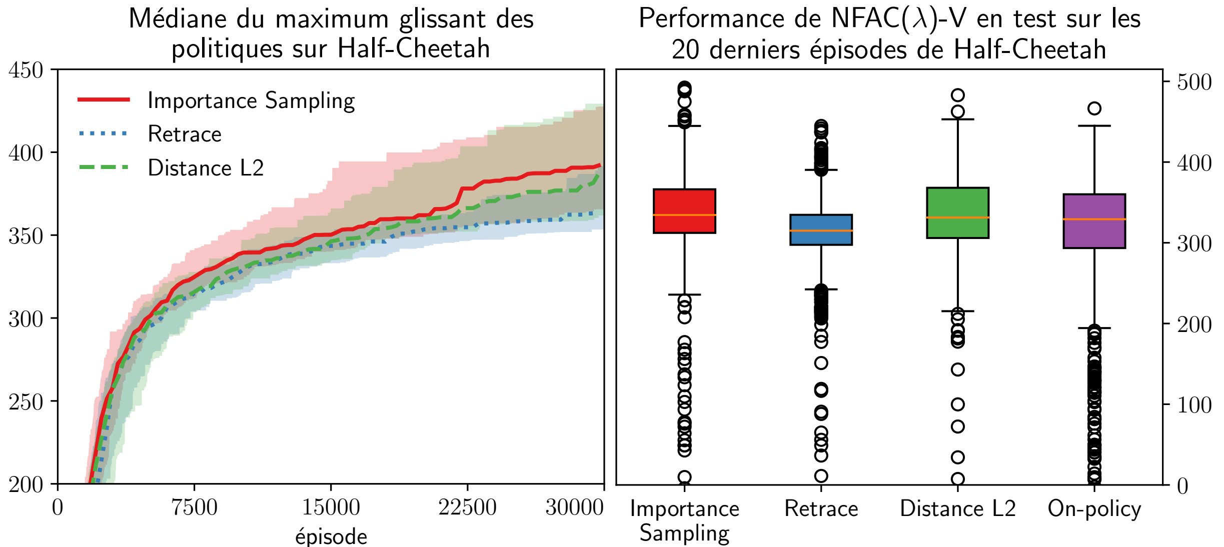


FIGURE 5.6 – Comparaisons des variantes *off-policy* pour la mise à jour de l'acteur dans NFAC(λ)-V. Le maximum glissant des courbes de gauche est réalisé sur 10 épisodes. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 60 essais avec des graines aléatoires différentes.

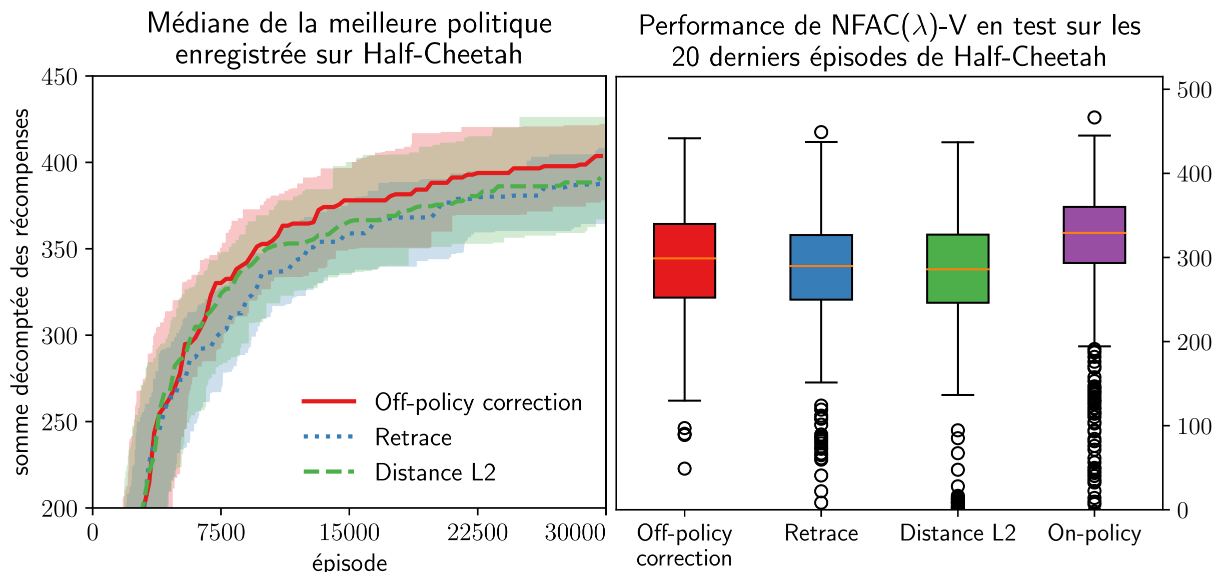


FIGURE 5.7 – Comparaisons des variantes *off-policy* pour la mise à jour du critique dans NFAC(λ)-V. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 60 essais avec des graines aléatoires différentes.

Dans ces expériences, le nombre maximal de trajectoires rejouées est de 25. On remarque dans les figures 5.6 et 5.7 qu'aucune variante ne se distingue clairement de la version *on-policy.*

Rappelons que la version *on-policy* de NFAC(λ)-V prenait en moyenne 1 heure 20 minutes en rejouant une seule trajectoire. Ici, pour la prise en compte de 25 trajectoires, le temps d'exécution d'une expérience passe à 9 heures 40 minutes, soit une augmentation de facteur 7 pour un résultat plutôt mitigé. Nous en concluons que l'utilisation de *Tree-backup*(λ) généralisé avec la fonction de valeur V permet seulement de légèrement augmenter la stabilité des performances pour un coût en calcul bien trop important.

La distance L2 que nous proposons atteint des performances similaires aux autres approches sans les dépasser, ce qui nous laisse penser qu'estimer V^π avec notre distance dans l'espace des actions, ou apprendre V^μ avec les autres variantes de l'état de l'art, est équivalent pour l'algorithme NFAC-V.

Les résultats ne varient que très faiblement entre mises à jour *off-policy* de l'acteur et mise à jour *off-policy* du critique. Il est possible que l'augmentation de la variance engendrée par la prise en compte des transitions *off-policy* puisse expliquer cette stagnation de la performance. D'autres expériences où la valeur de λ ne serait pas la même que dans le cas *on-policy*, mais une valeur plus faible pour réduire la variance, pourrait potentiellement mener à d'autres résultats.

5.6 Multi-step avec la fonction de valeur Q

Puisque la pondération n'a pas donné d'amélioration significative en utilisant la fonction de valeur d'état V, nous allons voir s'il est possible d'obtenir une amélioration avec la fonction Q.

5.6.1 Off-policy NFAC(λ)-∂Q

Nous comparons les différentes pondérations des transitions *off-policy* pour l'apprentissage du critique dans l'algorithme off-policy NFAC(0)-∂Q. Pour appliquer correctement l'algorithme *Tree-Backup*(λ) généralisé (Algorithme 2.4 page 39), nous n'apprendrons plus la fonction de valeur Q^π, mais Q^μ, car *Tree-Backup*(λ) généralisé fait l'hypothèse que la politique courante est stochastique.

La mise à jour du critique est alors définie telle que :

$$Q_{m+1} = \underset{Q \in \mathcal{F}_c}{\operatorname{argmin}} \sum_{(s_t,a_t,r_{t+1},s_{t+1}) \in \mathcal{D}} \Big[Q(s_t,a_t) - \Big(Q_m(s_t,a_t) + \Delta\hat{Q}_m(s_t,a_t) \Big) \Big]^2,$$

où $\Delta\hat{Q}_m(s_t,a_t)$ est l'estimation faite par *Tree-Backup*(λ) de la direction de modification :

$$\Delta\hat{Q}_m(s_t,a_t) = \sum_{n=t}^{\infty} \gamma^{n-t} \prod_{k=t+1}^{t+n} c_k \Big(r_{n+1} + \gamma \mathbb{E}\Big[Q_m(s_{n+1},a) \Big| a \sim \mu(\cdot|s_{n+1}) \Big] - Q_m(s_n,a_n) \Big).$$

La mise à jour de l'acteur ne change pas et s'effectue toujours à travers le gradient de Q.

5.6.2 Validation expérimentale

Pour réaliser cette expérience, nous avons fait varier λ dans l'ensemble $\{0.6, 0.8, 0.9\}$ pour chacune des variantes de c_k. Nous n'affichons que la configuration avec le meilleur λ. Sachant qu'avec l'algorithme NFAC(λ)-V, sur l'environnement Half-Cheetah, le meilleur λ est 0.6.

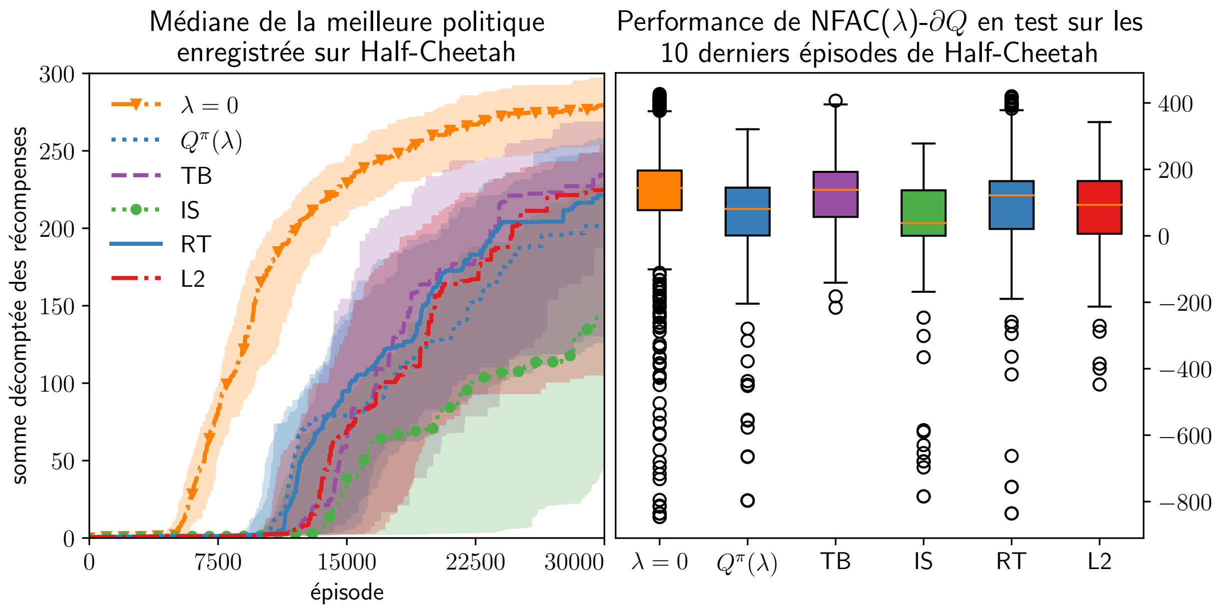


FIGURE 5.8 – Comparaisons des variantes de NFAC(λ)-∂Q sur Half-Cheetah. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 50 essais avec des graines aléatoires différentes.

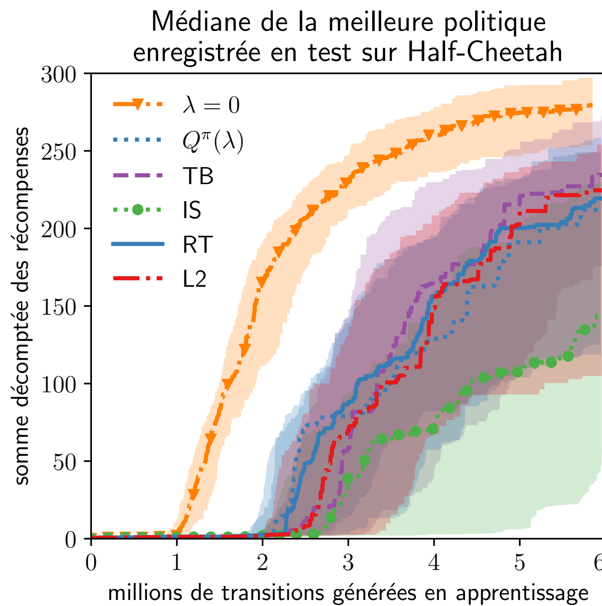


FIGURE 5.9 – Comparaisons des variantes de NFAC(λ)-∂Q sur Half-Cheetah en matière de données. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 50 essais avec des graines aléatoires différentes.

Nous ajoutons une expérience de contrôle *one-step* où $\lambda = 0$. Cette expérience n'est pas *on-policy*, mais bien *off-policy* en utilisant la définition même de la fonction de valeur Q.

Dans la figure 5.8, on peut remarquer, à nouveau, qu'aucune des variantes de pondération

ne permet une nette amélioration par rapport au cas *one-step off-policy*. Le cas *off-policy multi-step* permet une légère amélioration de la stabilité de la mise à jour, sans rendre possible un apprentissage plus rapide.

Dans la figure 5.9, nous affichons ces résultats sous une échelle différente, qui est le nombre de données produites par chacune des variantes, mais les conclusions restent les mêmes.

La distance L2 que nous proposons a des performances semblables aux autres méthodes, seul l'*importance sampling* est moins efficace.

Le cas *off-policy multi-step* engendre plus de variances que le cas *off-policy one-step*, aussi il est possible que notre recherche d'un λ approprié n'ait pas été suffisante. À l'aide d'une recherche en grille, nous avons essayé les valeurs suivantes $\lambda \in \{0.3, 0.6, 0.8, 0.9\}$, qui nous semblent tout de même couvrir une partie significative des valeurs usuelles, pour garder finalement $\lambda = 0.6$.

5.7 Comparaison finale

Les résultats de nos tentatives de passage à un apprentissage *off-policy* ou *multi-step* sont mitigés. Nous procédons à une comparaison finale des meilleurs algorithmes sur l'environnement Half-Cheetah. Tous les algorithmes comparés utiliseront la *batch normalization*, sauf CACLA qui est *online*. Les métaparamètres utilisés dans cette expérience se trouvent en annexe A.9 page 136.

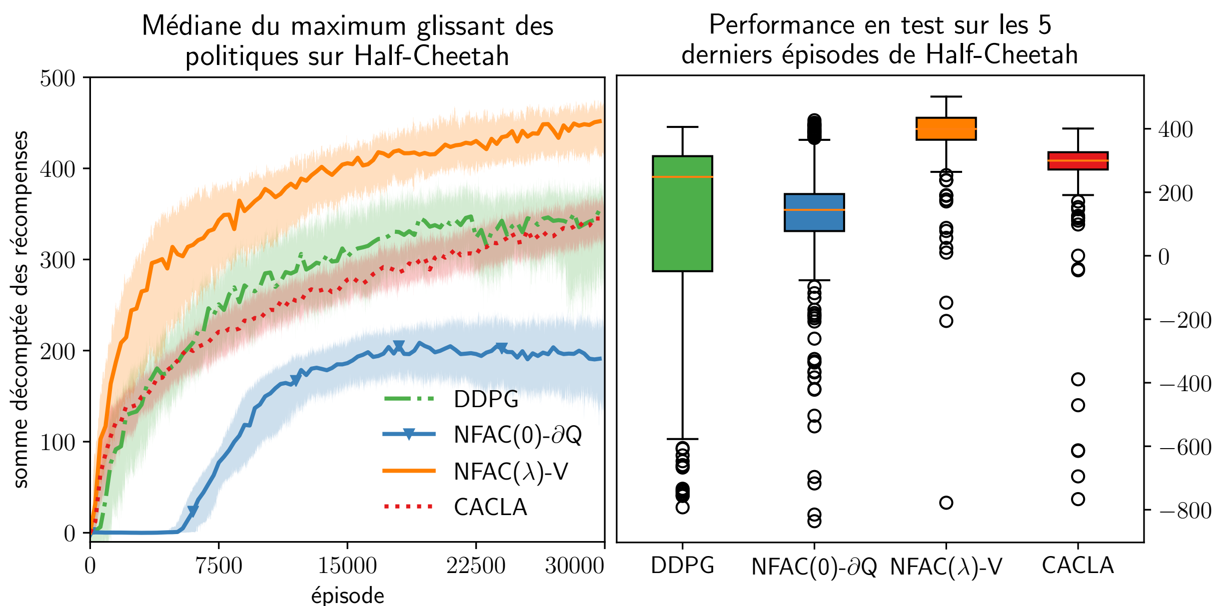


FIGURE 5.10 – Comparaisons des meilleurs algorithmes sur Half-Cheetah. Les médianes et les quartiles affichés sont calculés durant la phase de test sur 60 essais avec des graines aléatoires différentes. Le maximum glissant des courbes de droite est réalisé sur 10 épisodes.

La figure 5.10 nous conforte dans l'idée que NFAC(λ)-V est un algorithme efficace en données, malgré le fait qu'il soit *on-policy*. Cette dernière comparaison corrobore les bons résultats obtenus précédemment par NFAC(λ)-V (Section 5.2 page 89) avec la *batch normalization*. Il est à la fois meilleur que DDPG et NFAC(0)-∂Q qui sont eux *off-policy*. En fin de compte, ce qui a été le plus bénéfique pour l'algorithme NFAC(λ)-V n'est pas l'apprentissage *off-policy*, mais l'utilisation de la *batch normalization* dans le réseau de l'acteur.

La seconde conclusion concerne NFAC(0)-∂Q. Contrairement aux expériences menées précédemment (Section 5.2 page 89), il s'avère qu'ici, il n'est pas si efficace. Nous l'expliquons par plusieurs raisons :

— l'environnement Half-Cheetah a changé entre-temps,
— enfin, dans les expériences précédentes (Section 5.2 page 89), on affichait la meilleure performance durant la phase d'apprentissage, ce qui montrait que NFAC(0)-∂Q pouvait trouver rapidement un bon ensemble de paramètres, mais n'était pas stable pour autant. Par exemple, explorer rapidement et intelligemment l'espace des paramètres, sans garantir de stabilité, permettait d'obtenir une bonne performance pour la mesure précédemment utilisée, ce qui n'est plus le cas dans la figure 5.10.

5.8 Conclusion

Dans ce chapitre qui détaille notre seconde contribution, nous avons proposé une nouvelle version *off-policy* de notre algorithme *Neural Fitted Actor-Critic* : NFAC(0)-∂Q. Comme DDPG, il utilise le gradient d'une politique déterministe pour mettre à jour son acteur. Le critique et l'acteur utilisent des mises à jour *incremental least-square* pour l'efficacité en données. Le dilemme *efficacité en données* ou *mise à l'échelle* peut être partiellement contrôlé par la taille du *replay buffer* qui permet de contrôler combien de données sont rejouées. Nous avons vu que NFAC(0)-∂Q pouvait trouver rapidement de bonnes politiques, mais qu'il manquait de stabilité dans son apprentissage, or la stabilité est importante sur les environnements à forte dimensionnalité ayant besoin d'une recherche à long terme, l'utilisation de *target networks*, comme le fait DDPG, pourrait potentiellement aider à stabiliser cet apprentissage.

Que ce soit avec la fonction de valeur V ou Q, il est difficile d'améliorer CACLA, et par la même occasion NFAC(λ)-V, pour qu'il puisse prendre en compte des données *off-policy.* Il vaut mieux utiliser la *batch normalization* pour l'améliorer.

Enfin, nous avons observé que l'utilisation de *multi-step* pour l'apprentissage de politiques déterministes ne permettait pas d'amélioration dans le cas *off-policy*, contrairement au cas *on-policy*, où NFAC(λ)-V peut en bénéficier.

Dans la suite de la thèse, nous utilisons les algorithmes NFAC(λ)-V et DDPG pour nous intéresser à une exploration guidée de l'espace sensorimoteur en apprentissage par renforcement. Ce n'est que maintenant, avec ces algorithmes, que nous pouvons attaquer cette seconde problématique qu'il n'aurait pas été possible de traiter en début de thèse. En effet, à l'époque, il y avait peu d'algorithmes de la littérature permettant de gérer de manière satisfaisante des espaces continus avec des réseaux de neurones.

Algorithme 5.1 (Neural Fitted Actor-Critic(0)-∂Q)

Initialisation aléatoire du réseau du critique Q_w
Initialisation aléatoire du réseau de l'acteur π_θ
pour *episode* $\leftarrow 0$ **à** M **faire**
    Percevoir l'état initial s_0
    $t \leftarrow 0$
    tant que $s_t \notin S^*$ **et** $t < L_{max}$ **faire**
        Calculer l'action d'exploitation $u_t \leftarrow \pi_\theta(s)$
        Explorer potentiellement $a_t \sim \mu(\cdot|u_t, \sigma)$
        Effectuer l'action a_t, observer r_{t+1} et s_{t+1}
        Conserver la transition $(s_t, a_t, r_{t+1}, s_{t+1})$ dans le *replay buffer* $\mathcal{D}$
        $t \leftarrow t + 1$
    fin
    pour $k \leftarrow 1$ **à** K **faire**
        pour $g \leftarrow 1$ **à** G_c **faire**
            pour $(s_t, r_{t+1}, s_{t+1}) \in \mathcal{D}$ **faire**

$$q_t \leftarrow \begin{cases} r_{t+1}, & \text{si } s_{t+1} \in S^* \\ r_{t+1} + \gamma Q_w(s_{t+1}, \pi_\theta(s_{t+1})), & \text{sinon} \end{cases}$$

            fin
            Initialiser aléatoirement le réseau critique Q_w **si** critique_remis_à_zéro
            Mettre à jour le critique avec une descente de gradient :

$$\Delta w = \sum_{(s_t,a_t)\in\mathcal{D}} \Big(q_t - Q_w(s_t, a_t)\Big) \frac{\partial Q_w(s_t, a_t)}{\partial w}.$$

        fin
        Initialiser aléatoirement le réseau acteur π_θ **si** acteur_remis_à_zéro
        pour $g \leftarrow 1$ **à** G_a **faire**
            pour $s_t \in \mathcal{D}$ **faire**

$$\nabla_a \leftarrow \frac{\partial Q_w(s_t, a)}{\partial a}\Big|_{a=\pi_\theta(s_t)}$$

                si *stratégie_inversion_gradient* **alors**

$$\nabla_a \leftarrow \nabla_a . \begin{cases} (a_{max} - a)/(a_{max} - a_{min}) & \text{si } \nabla_a < 0 \\ (a - a_{min})/(a_{max} - a_{min}) & \text{sinon} \end{cases}$$

                fin
            fin
            Mettre à jour l'acteur avec une descente de gradient :

$$\Delta\theta = \sum_{s_t \in \mathcal{D}} \nabla_a \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

        fin
    fin
fin

Chapitre 6

Exploration guidée de l'espace sensorimoteur

Nous avons maintenant des algorithmes assez robustes pour passer à la seconde problématique de cette thèse. L'apprentissage à long terme reste un problème difficile, surtout lorsque la fonction de récompense est peu informative ou tardive dans le temps. L'hypothèse que nous faisons dans cette thèse est que l'augmentation progressive de l'espace sensorimoteur de l'agent peut aider l'agent à apprendre au long terme.

Dans ce chapitre, nous proposons une nouvelle approche de *transfer learning* (Taylor et Stone, 2009; Wiering et Van Otterlo, 2012), inspirée du développement de l'enfant, où le nombre de degrés de liberté de l'espace sensorimoteur d'un agent augmente en même temps qu'il apprend une politique. Nous combinons l'apprentissage par renforcement profond, décrit dans les deux chapitres précédents, qui est déjà une approche développementale, parce que réalisé dans un environnement continu décrivant un corps physique, avec une augmentation progressive de l'espace sensorimoteur de l'agent (Figure 6.1).

Nous validons expérimentalement notre approche, sur deux environnements comportant de nombreux degrés de liberté, en montrant que le développement morphologique peut être bénéfique pour l'apprentissage.

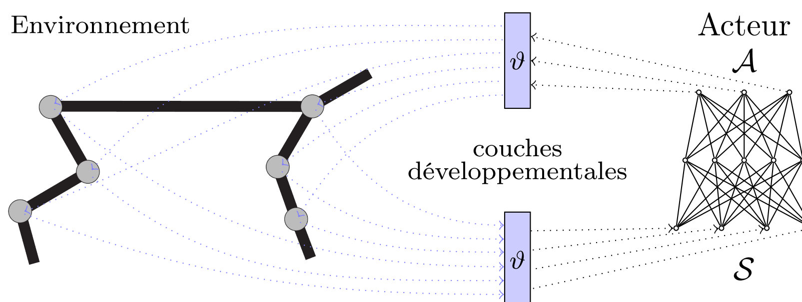


FIGURE 6.1 – Illustration de notre solution pour l'augmentation de l'espace sensorimoteur de l'agent dans l'environnement Half-Cheetah. Les couches développementales servent à choisir quelles dimensions de l'espace sensorimoteur sont perceptibles ou contrôlables.

6.1 Robotique développementale

La robotique développementale (Guerin, 2011; Asada *et al.*, 2009) s'intéresse à résoudre des problèmes où des robots disposent de moins d'*a priori* que la «robotique classique» qui repose parfois sur une description logique et symbolique complète de l'environnement.

Par exemple, certains robots, qui utilisent de l'apprentissage par renforcement, s'appuient sur des actions et états prédéfinis (Riedmiller *et al.*, 2009; Stone *et al.*, 2005). Cela implique que le concepteur puisse prévoir plusieurs situations que le robot pourrait rencontrer et les actions importantes qu'il devrait entreprendre. Dans le cas des descriptions logiques, les limitations proviennent du problème de l'*ancrage des symboles* (Searle, 1980; Brooks, 1991).

La robotique développementale tente, au contraire, de faire découvrir automatiquement les représentations des états et des actions au robot : il doit être capable de se développer à l'image des enfants. En effet, l'intelligence n'est pas nécessairement innée, mais peut se développer au cours de la vie d'un individu (Piaget, 1948). L'apprentissage de nouvelles compétences est graduel et repose sur la maîtrise des interactions plus simples. Plusieurs travaux se sont intéressés à ce problème (Laflaquière, 2013; Baranes et Oudeyer, 2013), mais peu à son application avec l'apprentissage par renforcement.

L'apprentissage profond et les réseaux de neurones sont à même de construire progressivement des représentations efficaces avec peu de connaissances *a priori* (Sigaud et Droniou, 2016; LeCun *et al.*, 2012). Ainsi, l'apprentissage par renforcement profond qui utilise ces deux derniers en est également capable. Pour Guerin (2011), apprendre les concepts physiques de base dans des environnements continus en laissant l'agent construire ses représentations correspond à une approche *développementale*. Pour cette raison, les algorithmes traités dans les deux chapitres précédents peuvent être considérés comme suivant une approche développementale, car ils font partie de *l'apprentissage par renforcement profond en environnement continu* où l'agent contrôle un corps.

6.2 Augmentation de l'espace sensorimoteur de l'agent

L'idée principale de ce que nous appelons «augmentation de l'espace sensorimoteur» est d'augmenter progressivement la dimension, et ainsi la difficulté, du problème à apprendre, à l'instar du *curriculum learning* (Bengio *et al.*, 2009).

Alors que l'apprentissage par renforcement développemental peut être général, nous nous basons ici sur une des contraintes de cette thèse : les espaces continus. L'apprentissage par renforcement développemental peut aussi avoir du sens dans un cadre discret. À ce moment, c'est le nombre d'actions ou d'états qui augmente (Dutech, 2012). Nous définirons donc essentiellement ce que nous entendons par augmentation de l'espace sensorimoteur dans des espaces continus.

6.2.1 Définition

Le développement de la difficulté du problème à résoudre se traduit par une augmentation de l'espace sensorimoteur de l'agent (Figure 6.2 page suivante). Par exemple, pour un agent composé de plusieurs membres, tels les quatre environnements décrits au chapitre 3 page 63, l'augmentation de l'espace sensorimoteur peut être vue comme le choix des membres qui sont perçus et contrôlés par l'agent.

Formellement, en utilisant le vocabulaire du *transfer learning*, on suppose un MDP cible $\mathcal{M}_c = \langle \mathcal{S}_c, \mathcal{A}_c, T_c, R_c \rangle$, celui qu'on cherche à résoudre en définitive, et qu'il existe $n \in \mathbb{N}^*$ MDP

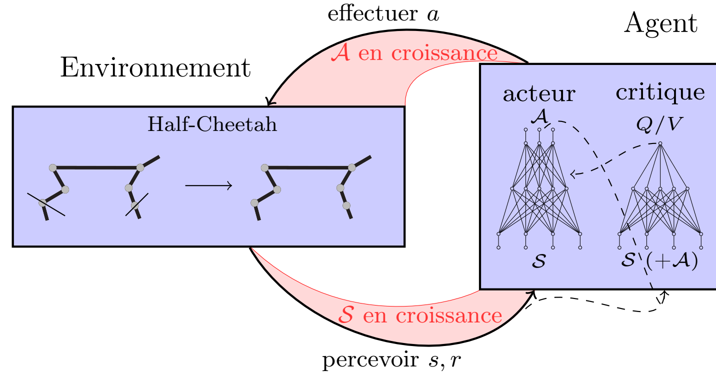


FIGURE 6.2 – Illustration de l'augmentation de l'espace sensorimoteur de l'agent dans un cadre continu avec un algorithme *acteur-critique*. Les dimensionnalités $d_{\mathcal{S}}$ et $d_{\mathcal{A}}$ de l'espace des états et des actions augmentent pendant que l'agent apprend. Le nombre d'entrées et de sorties de l'agent augmente : il apprend sur une succession de différents MDP pour résoudre, en fin de compte, le plus important et le plus complexe.

sources $\mathcal{M}_1 = \langle \mathcal{S}_1, \mathcal{A}_1, T_1, R_1 \rangle, \cdots, \mathcal{M}_n = \langle \mathcal{S}_n, \mathcal{A}_n, T_n, R_n \rangle$, ceux sur lesquels l'agent peut progresser, tels que :

$$\begin{cases} \mathcal{S}_1 \subseteq \cdots \subseteq \mathcal{S}_n \subseteq \mathcal{S}_c, \\ \mathcal{A}_1 \subseteq \cdots \subseteq \mathcal{A}_n \subseteq \mathcal{A}_c, \\ \mathcal{S}_c \neq \mathcal{S}_1 \text{ ou } \mathcal{A}_c \neq \mathcal{A}_1. \end{cases}$$

En langage naturel, l'espace des actions ou l'espace des états des MDP sources est contenu dans ceux du MDP cible. En fonction du développement, plusieurs tâches cibles peuvent être obtenues avec plusieurs étapes de transfert. Les tâches sources ont une dimension réduite par rapport à la tâche cible.

Tout MDP ne peut être décomposé de la sorte sur son espace sensorimoteur en gardant des fonctions de transitions et de récompenses cohérentes. Néanmoins, cela peut être possible. Par exemple, l'environnement Half-Cheetah (Section 3.3 page 65) peut être décomposé par un premier MDP où les chevilles sont désactivées comme dans la figure 6.2.

6.2.2 Définition alternative

Il serait possible de fournir une définition moins restrictive que celle proposée précédemment, qui inclurait le *reward shaping* (Ng *et al.*, 1999), en remplaçant la dernière condition par $\mathcal{M}_c \neq \mathcal{M}_1$, ce qui signifie qu'une modification de la fonction de récompense entre le MDP source et le MDP cible suffirait. Néanmoins, dans ce cas, il est difficile de définir les conditions requises sur les fonctions de récompenses. En effet, deux concepts s'opposent :

- On peut argumenter que, si l'espace sensorimoteur est équivalent, la fonction de récompense R_1 doit être plus informative que R_c sinon il n'y a aucun intérêt à passer par une telle décomposition.
- À l'inverse, sur un espace sensorimoteur croissant, la fonction de récompense R_1, définie sur un espace réduit, peut contenir moins d'informations et être plus simple à apprendre que la fonction R_n définie sur l'espace complet.

Pour cette raison, nous gardons la définition précédente excluant le *reward shaping* et forçant une modification de l'espace des actions ou de l'espace des états.

6.2.3 Problématique

De cette définition résultent deux problèmes cruciaux :
- Il est possible que plusieurs décompositions soient possibles, dans ce cas, il faut en définir l'ordre. Par exemple, pour l'environnement Half-Cheetah, est-il préférable de commencer à contrôler ses chevilles ou ses genoux ?
- Le second problème consiste à décider *quand* s'opère le changement de MDP. S'il s'effectue trop tôt, l'agent peut ne pas avoir assez profité de ce préapprentissage. Au contraire, s'il se fait trop tard, il peut n'y avoir aucun intérêt par rapport au fait d'apprendre directement sur la tâche cible sans passer par les tâches sources.

Ces questions restent encore ouvertes. Dans la suite, nous essayons de montrer expérimentalement que l'augmentation de l'espace sensorimoteur peut aboutir à un apprentissage plus rapide et de meilleure qualité. Les deux questions précédentes ne sont gérées que comme étant des métaparamètres de l'apprentissage.

6.2.4 Changement d'espace de recherche

Lors d'un transfert entre deux MDP, l'espace de recherche est forcément modifié, car certaines dimensions sont ajoutées. Les solutions trouvées dans le sous-espace de recherche devraient être exploitées au mieux dans l'espace de recherche complet. Néanmoins, ce n'est pas simple avec des réseaux de neurones à cause du problème du *catastrophic forgetting* (French, 1999). Parce que nous voulons être générique et ne pas modifier l'algorithme d'apprentissage en profondeur, l'approche que nous proposons est très simple : la solution trouvée dans le sous-espace de recherche $\theta^s \in \mathbb{R}^k$ est utilisée comme initialisation dans l'espace de recherche plus important $\mathbb{R}^K$ avec $k < K$. Formellement, la solution initiale dans l'espace de recherche complet θ^c est définie telle que :

$$\forall i \in \{1, \cdots, K\}, \theta_i^c \leftarrow \begin{cases} \theta_i^s & \text{si } i \leq k, \\ \text{initialisation aléatoire} & \text{sinon.} \end{cases}$$

Pour que cette initialisation ait du sens, il faut que les k dimensions soient alignées et gardent la même signification entre les deux MDP. Des approches plus élaborées avec de l'inertie sur les k dimensions pourraient être également utilisées, mais elles introduisent des paramètres supplémentaires.

6.3 Couche neuronale développementale

Dans cette proposition, nous avons cherché à être les plus génériques possibles en minimisant les modifications faites sur l'environnement et sur les algorithmes d'apprentissage. Pour ce faire, nous avons uniquement travaillé sur l'architecture des réseaux de neurones. Le développement morphologique n'a pas été simulé depuis l'environnement. Nous avons créé un nouveau type de couche au sein des réseaux des neurones, appelé *developmental layer* (DL) ou «couche développementale». Il permet de désactiver des entrées et des sorties du réseau. De la même manière que la régularisation *dropout* (Section 1.3.1 page 23), chaque entrée est connectée à une seule sortie où le poids de la connexion détermine l'activation de la sortie (Figure 6.3 page ci-contre).

Comme pour la régularisation *dropout* ou la *batch normalization*, nous distinguons ϑ, qui correspond aux paramètres des couches développementales, de θ qui représentent les poids des produits scalaires du réseau de neurones, car ces deux ensembles de paramètres ne sont pas appris de la même manière.

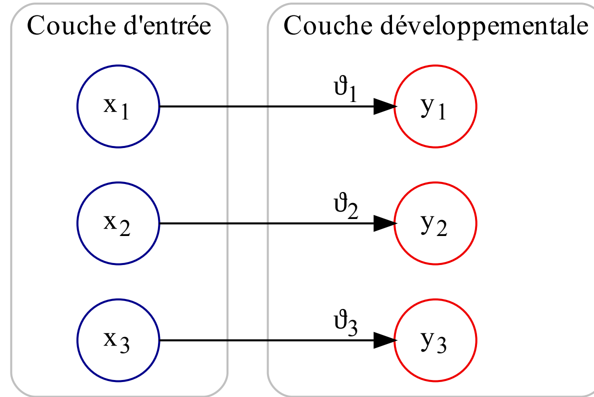


FIGURE 6.3 – Exemple d'une couche développementale. Les paramètres ϑ déterminent si la sortie y est égale à x ou désactivée.

Étant donné un vecteur $x \in \mathbb{R}^n$ en entrée et les paramètres $\vartheta \in \mathbb{R}^n$ d'une couche développementale, nous définissons deux types différents d'activations pour calculer la sortie $y \in \mathbb{R}^n$:

— déterministe avec $y_i = \begin{cases} x_i & \text{si } \vartheta_i \geq \text{constante de seuil,} \\ 0 & \text{sinon,} \end{cases}$

— stochastique avec $\begin{cases} \mathbb{P}(y_i = x_i) = \vartheta_i & \text{si } x_i \neq 0, \\ y_i = 0 & \text{sinon.} \end{cases}$

Dans les deux cas, la sortie y ne peut que prendre la valeur d'entrée x (nous dirons alors que la sortie est activée), ou la valeur 0. Pour l'activation déterministe, si le poids est assez élevé la sortie est activée. Pour l'activation stochastique, le poids détermine la probabilité d'activation de la sortie. L'activation déterministe permet de simuler un passage discret depuis un MDP source au prochain. Tandis que l'activation stochastique permet de simuler un passage continu entre les deux MDP.

6.3.1 Emplacement

Dans les architectures de contrôle neuronales, la couche développementale devrait être ajoutée au début et à la fin du réseau représentant la politique (Figure 6.4 page suivante). Si l'espace des états reste le même entre les différents MDP sources et cibles, $\mathcal{S}_c = \mathcal{S}_1$, il n'est pas nécessaire d'ajouter une couche développementale en entrée. De même, la couche développementale à la sortie du réseau n'est pas nécessaire si $\mathcal{A}_c = \mathcal{A}_1$. Si l'algorithme utilise également un critique avec un réseau de neurones, deux couches développementales devraient être ajoutées aux entrées : une pour les états et la seconde pour les actions.

Grâce aux DL, il est possible de simuler le passage d'un MDP à un autre, sans modifications de l'environnement ni des algorithmes d'apprentissage. La sortie des dimensions désactivées sera 0. Ainsi, les poids correspondants, ceux présents au sein du produit scalaire qui déterminent l'entrée de la couche développementale, ne sont pas modifiés durant la rétropropagation du gradient. L'espace de recherche est donc plus restreint. Si l'algorithme d'optimisation n'utilise pas la rétropropagation du gradient, par exemple CMA-ES, la réduction de l'espace de recherche n'est pas aussi évidente : l'algorithme doit comprendre lui-même que changer les poids qui sont reliés à des neurones développementaux qui restent désactivés n'a aucun effet.

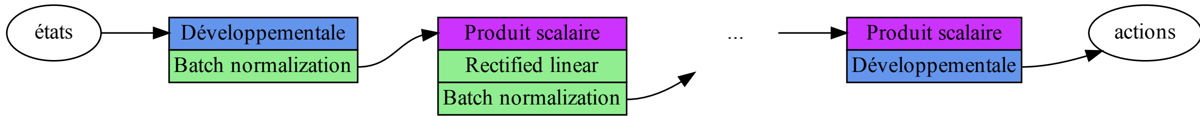


FIGURE 6.4 – Une architecture de contrôle neuronale utilisant des couches développementales. La première couche développementale permet de désactiver certaines dimensions de l'espace des états et la dernière de désactiver le contrôle de certaines dimensions de l'espace des actions.

6.3.2 Action neutre

Nous avons expliqué que les dimensions désactivées de l'espace des actions auront pour valeur 0. Par exemple, lorsqu'un des paramètres d'activations vaut 0 dans la couche développementale. Ainsi, une des limitations principales des couches développementales est de s'appuyer sur des actions neutres, ce qui signifie que les dimensions de l'espace d'action qui sont nulles n'influent pas sur les fonctions de transition et de récompense. Nous définissions un MDP $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ comme supportant des actions neutres si et seulement si :

$$\forall (s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S}, \begin{cases} T(s'|s, a_{1 \dots d_{\mathcal{A}}}) = T(s'|s, a_{i \in \{1 \dots d_{\mathcal{A}}\} | a_i \neq 0}), \\ R(s, a_{1 \dots d_{\mathcal{A}}}, s') = R(s, a_{i \in \{1 \dots d_{\mathcal{A}}\} | a_i \neq 0}, s'). \end{cases}$$

Cette propriété signifie que les dimensions des actions valant 0 n'apportent pas d'information supplémentaire au MDP. Ces actions neutres sont nécessaires si l'on ne veut pas modifier l'environnement pour implémenter le passage entre deux MDP. Les actions neutres existent souvent sur les environnements continus, par exemple, elles consistent à n'appliquer aucun couple sur un joint. Si un MDP supporte des actions neutres, il est possible, en général, de le décomposer pour y appliquer une augmentation de l'espace sensorimoteur.

6.3.3 Changement d'espace de recherche

Comme expliqué précédemment, lors du passage à un prochain MDP, il s'opère une modification de l'espace de recherche. En utilisant des couches développementales, cette opération est transparente. Si l'on suppose que l'espace de recherche des paramètres θ passe de $\mathbb{R}^k$ à $\mathbb{R}^K$ avec $k \leq K$, les $K - k$ nouvelles dimensions existaient déjà au sein du réseau, mais n'avaient jamais été modifiées : elles ont gardé leurs poids initiaux. Il s'agit des poids effectuant un produit scalaire avec des neurones précédemment désactivés par les couches développementales. L'initialisation aléatoire de ces poids est importante si l'on tient à ne pas trop perturber le transfert de solution. S'ils ont des valeurs trop grandes, il n'y aura aucune garantie que la solution trouvée dans l'espace restreint puisse servir dans l'espace complet.

Remarquons que dans ce passage de k à K dimensions, on ne compte pas les paramètres ϑ présents au sein même des couches développementales, car ils ne sont pas optimisés de la même façon que les autres poids θ du réseau.

6.3.4 Limitations

Nous allons résumer les différentes limitations à l'utilisation des couches développementales pour réaliser une augmentation progressive de l'espace sensorimoteur de l'agent :

— L'espace sensorimoteur du MDP doit pouvoir être décomposable tout en gardant du sens : une ou plusieurs tâches sources doivent exister lorsque des dimensions d'entrées ou de

sorties sont retirées (Section 6.2.1 page 104). Par exemple, les environnements où l'agent contrôle un corps composé de différents membres sont décomposables.
— Il doit exister des actions neutres sur les dimensions qu'on cherche à contrôler avec des couches développementales (Section 6.3.2 page précédente).
— Les fonctions de récompenses des tâches sources doivent contenir des informations et être non nulles sur une partie du sous-domaine, sinon la solution transférée agira de façon aléatoire dans le problème complet.
— Les paramètres supplémentaires des modèles, lors du transfert dans un espace plus grand, ne doivent pas perturber complètement la solution transférée (Section 6.3.3 page ci-contre).

Enfin, s'il existe des correspondances dans l'espace sensorimoteur entre des dimensions d'entrées et de sorties, cela peut favoriser la réduction de l'espace de recherche des paramètres des couches développementales. Par exemple, dans l'environnement Half-Cheetah, nous savons qu'il existe 2 dimensions d'entrées dans l'état pour percevoir une cheville et une dimension de sortie dans l'action pour contrôler cette cheville. Il est donc possible de partager les poids de perception de la cheville, et de maintenir le poids de contrôle à une valeur supérieure (ou égale) à ceux de perception pour garantir que l'agent perçoive avant d'agir.

6.3.5 Recherche connexe

Les couches développementales partagent plusieurs similarités avec le *pre-training* (Erhan *et al.*, 2010) utilisé en *deep learning* : les poids des modèles sont initialisés par une solution calculée sur un ensemble de données préliminaires. Ici, la phase de *pre-training* est réalisée sur une tâche aux dimensions plus réduites que la tâche finale. Néanmoins, le *pre-training* en *deep learning* est réalisé de manière non supervisée avec des données sans-étiquettes, tandis que la solution proposée par les couches développementales s'appuie sur la fonction de récompense, comme dans le *reward shaping* (Ng *et al.*, 1999).

Plusieurs travaux précédents se sont attaqués à l'amélioration de la représentation neuronale pour l'apprentissage par renforcement. Les *progressive neural networks* (Rusu *et al.*, 2016) créent de nouvelles couches à chaque nouvelle tâche. Ils sont utilisés essentiellement sur des tâches similaires, où la politique est calculée à partir de l'ensemble des pixels d'une image : les dimensions des entrées et des sorties ne changent pas. L'hypothèse utilisée est la suivante : les caractéristiques extraites de l'image dans les couches cachées peuvent servir à la prochaine tâche. Les *progressive neural networks* permettent de transférer de la connaissance d'un réseau à un autre en évitant le problème de *catastrophic forgetting*. Il est possible de combiner à la fois les couches développementales et les *progressive neural networks* pour effectuer une augmentation progressive de l'espace sensorimoteur de l'agent : cela pourrait permettre de mieux exploiter la solution produite sur l'espace réduit, mais ne répond pas aux limitations précédemment réécrites (Section 6.3.4 page précédente).

PathNet (Fernando *et al.*, 2017) permet de faire évoluer la structure d'un réseau de neurones de façon *online* à l'aide d'algorithmes génétiques pour exploiter des connaissances précédemment apprises sur d'autres tâches en évitant le *catastrophic forgetting*. Utiliser les idées de *PathNet* pourrait permettre d'apprendre les poids des couches développementales.

6.4 Démonstration de faisabilité des couches développementales

Nous voulons maintenant montrer qu'il peut y avoir un intérêt à utiliser les couches développementales. Néanmoins, apprendre les poids de ces couches n'est pas trivial. Ainsi, nous

procédons simplement à une recherche dans une grille pour les déterminer.

6.4.1 Recherche dans une grille

Pour chaque dimension ϑ_i des poids des couches développementales, un point de développement pdd_i est associé. La valeur des ϑ_i est déterminée au début de chaque épisode en fonction du nombre d'épisodes ep et de l'activation :

— déterministe où $\vartheta_i \leftarrow \begin{cases} 1 & \text{si } ep \geq pdd_i, \\ 0 & \text{sinon,} \end{cases}$

— stochastique où $\vartheta_i \leftarrow \min(1, ep \cdot l_i) = \min\left(1, \dfrac{ep}{\max(1, pdd_i)}\right)$,

où $pdd_i \in \mathbb{N}$ sont des points de développement et $l_i \in [0; 1]$ sont des coefficients linéaires tels que $l_i \leftarrow \frac{1}{max(1, pdd_i)}$. La recherche en grille détermine les points de développement pdd_i qui servent ensuite à calculer les poids des couches développementales (Figure 6.5). Dans le cas de l'activation stochastique, le point de développement ppd_i représente le moment où la dimension i est totalement activée.

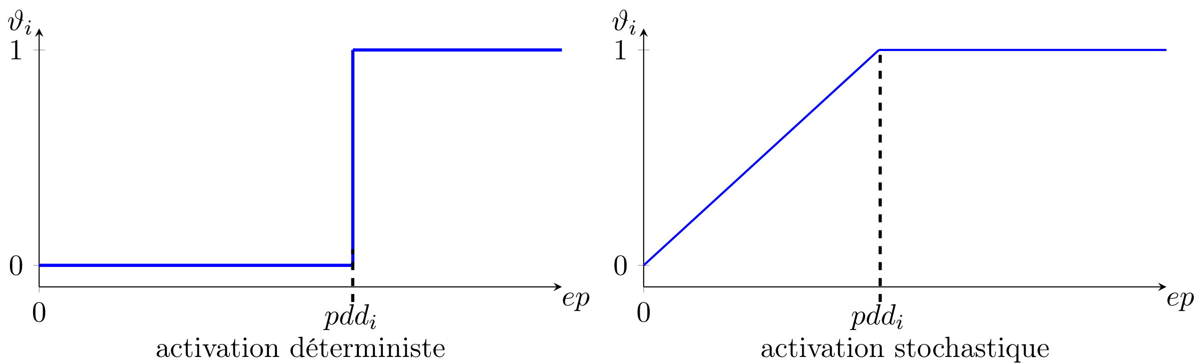


FIGURE 6.5 – Illustration de la dépendance entre les points de développement pdd_i, le nombre d'épisodes ep et les paramètres des couches développementales.

La recherche en grille doit donc déterminer les différents points de développement pour chacune des dimensions d'entrées et de sorties qui pourraient être désactivées. Pour réduire l'espace de recherche de la grille, on peut suivre la ligne directrice suivante : les dimensions de sorties ne doivent pas être activées avant les entrées équivalentes. Par exemple, dans l'environnement Half-Cheetah, l'agent ne devrait pas contrôler ses chevilles avant de les percevoir.

6.4.2 Environnements

Pour démontrer la faisabilité des couches développementales, nous avons utilisé deux environnements qui comportaient un nombre important de degrés de liberté : Half-Cheetah (Section 3.3 page 65) et Humanoïde (Section 3.4 page 67).

Sur l'environnement Half-Cheetah, les points de développement concernent les deux chevilles et les deux genoux (Figure 6.6 page suivante). Nous avons défini 7 valeurs possibles pour les points de développement : $pdd_i \in \{0, 500, 1000, 1500, 2500, 4000, 10000\}$. Pour réduire l'espace de recherche, la contrainte suivante est ajoutée : les deux chevilles sont contrôlées par les mêmes

points de développement, de même pour les deux genoux. Par exemple, si un genou est perçu, le second sera également perçu dans le cas des activations déterministes. La recherche en grille doit donc trouver 4 points de développement parmi les 7 valeurs, en respectant le fait qu'on ne peut pas contrôler un membre avant de le percevoir.

Sur l'environnement Humanoïde, les points de développement concernent les deux épaules et les deux coudes (Figure 6.6). Nous avons défini 7 valeurs possibles pour les points de développement : $pdd_i \in \{0, 5000, 10000, 15000, 25000, 50000\}$. Parce que le coût de calcul est élevé, la recherche en grille doit trouver seulement 2 points de développement : quand percevoir les épaules et les coudes, et quand les contrôler.

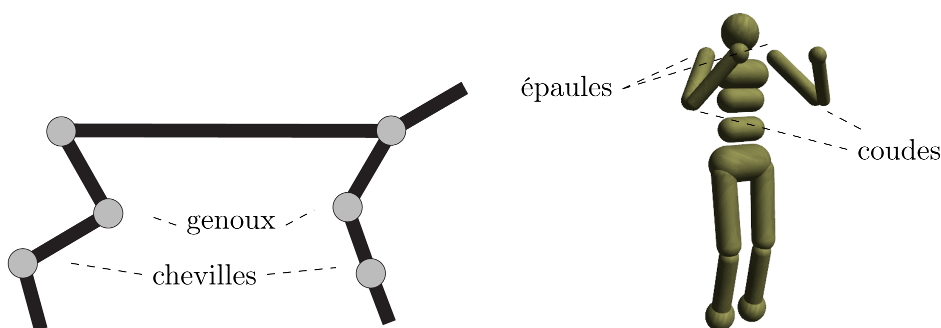


Figure 6.6 – Illustration des environnements Half-Cheetah (à gauche - reproduit depuis (Wawrzyński, 2007)) et Humanoïde (à droite). Les membres décrits sont contrôlés à l'aide des couches développementales.

6.4.3 Validation expérimentale

Pour cette validation expérimentale, nous avons utilisé trois algorithmes bien différents pour montrer que les couches développementales peuvent être bénéfiques dans ces trois cas (Table 6.1).

	Offline	Exploration	Acteur	Critique	Off-policy replay
CMA-ES	oui	espace des paramètres	oui	non	non
NFAC(λ)-V	oui	espace des actions	oui	oui	potentiellement
DDPG	non	espace des actions	oui	oui	oui

Table 6.1 – Les caractéristiques des algorithmes CMA-ES, NFAC(λ)-V et DDPG.

L'ensemble des métaparamètres utilisés dans cette expérience peut être retrouvé en Annexe A.10 page 137. Les métaparamètres n'ont pas été optimisés sur l'environnement Humanoïde, mais nous avons réutilisé ceux trouvés sur l'environnement Half-Cheetah lorsqu'ils permettaient un apprentissage : cela permet de mesurer la robustesse d'un algorithme (Duan *et al.*, 2016). Dans cette expérience, nous ne cherchons pas à comparer les différents algorithmes, mais seulement à montrer qu'avec des couches développementales ils apprennent plus rapidement.

Dans le cas de DDPG et de CMA-ES, nous avons ajouté un paramètre à la recherche en grille pour les activations déterministes : l'activation d'un opérateur de remise à zéro. Puisque DDPG et CMA-ES ont une certaine mémoire des précédentes expériences, qui se déroulaient dans des espaces de recherches réduits, il faut potentiellement en oublier une partie. Par exemple, pour

DDPG, il n'est pas évident de savoir si rejouer des données où les membres étaient désactivés ne perturberait pas l'apprentissage.

Pour DDPG, l'opérateur de remise à zéro consiste à simplement vider le *replay buffer*. Pour CMA-ES, une nouvelle population est créée à partir de la meilleure solution trouvée avant l'activation de l'opérateur de remise à zéro. NFAC(λ)-V n'a pas d'opérateur de remise à zéro, car nous utilisons ici la version *on-policy*. Dans le cas des activations stochastiques, l'opérateur de remise à zéro n'est pas utilisé, car il s'agit d'un transfert continu.

Des couches développementales (DL) sont ajoutées au début (pour les états) et à la fin (pour les actions) des réseaux de neurones des politiques. Pour les algorithmes utilisant des critiques, les couches développementales sont ajoutées au début du réseau. Chaque poids des réseaux, sauf ceux des couches développementales, est initialisé à partir d'une distribution gaussienne $\mathcal{N}(0, 0.01)$. Il est important d'initialiser les paramètres à des valeurs faibles pour ne pas trop modifier la solution lorsque le transfert s'opère (Section 6.3.3 page 108). Nous n'avons pas eu le temps d'analyser les effets d'une modification de l'écart-type de la distribution gaussienne servant d'initialisation aux poids θ.

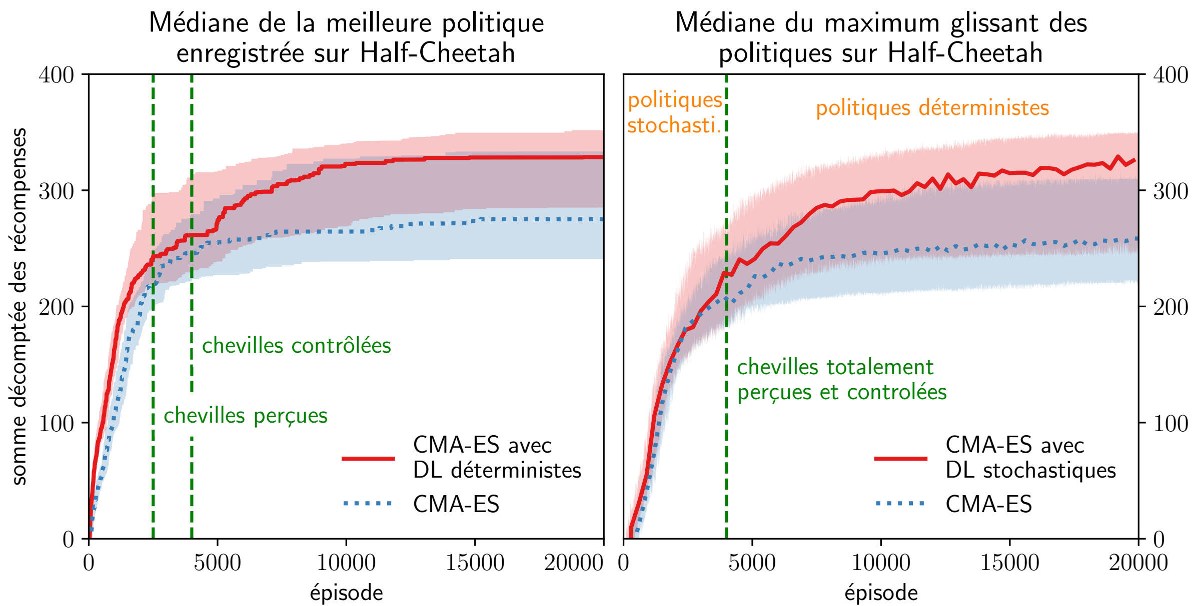


FIGURE 6.7 – Performances de CMA-ES avec des couches développementales (DL) sur l'environnement Half-Cheetah sur 50 essais avec des graines aléatoires différentes. Le maximum glissant, de la courbe de droite, est réalisé sur 10 épisodes. Il est utilisé, à la place du maximum, pour ne pas biaiser la partie où les politiques sont stochastiques. Les points de développement ppd_i sont affichés en vert. Pour les DL stochastiques, après le point de développement à l'épisode 4000, l'activation des neurones développementaux vaut 1.

Pour les DL à activation stochastique, nous ne montrons ici que les résultats de CMA-ES, car les algorithmes DDPG et NFAC ont été conçus pour apprendre des politiques déterministes : l'ajout des DL stochastiques dans ces algorithmes n'a permis ni amélioration ni détérioration, néanmoins les DL stochastiques ajoutent du bruit au *replay*.

Dans la figure 6.7, nous avons affiché les meilleurs points de développement trouvés par la recherche en grille avec l'algorithme CMA-ES. On remarque que l'ajout des couches développementales permet d'améliorer la vitesse d'apprentissage. Avec les couches déterministes, l'opérateur

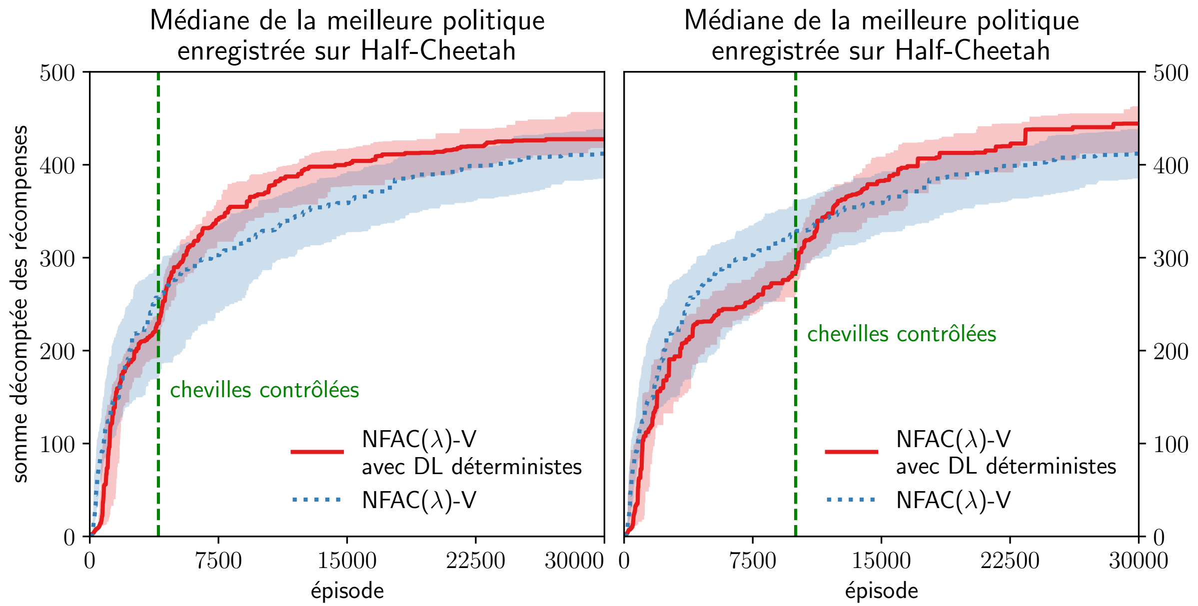


Figure 6.8 – Performances de NFAC(λ)-V avec des DL déterministes sur l'environnement Half-Cheetah sur 50 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Les points de développement ppd_i sont affichés en vert : il s'agit des deux meilleurs trouvés par la recherche en grille.

de remise à zéro est utilisé deux fois à chaque point de développement. Parmi tous les points de développement testés, il en existe diminuant la vitesse d'apprentissage, mais ces résultats n'ont pas été montré ici.

Dans la figure 6.8, la version utilisée de NFAC(λ)-V est *on-policy*. Les chevilles sont perçues dès le départ. Il est intéressant de remarquer que cette figure contredit l'une de nos hypothèses, à savoir que l'agent devrait apprendre plus rapidement dans un espace sensorimoteur réduit. Or, dans cette figure, on voit qu'avant les points de développement, l'apprentissage est plus lent que dans l'espace complet. Pourtant, après les points de développement, l'apprentissage redevient, tout de même, de meilleure qualité. Ce phénomène est difficile à expliquer. Notre hypothèse étant que la solution trouvée dans l'espace réduit n'obtient pas d'aussi bonnes récompenses, mais découvre des caractéristiques plus essentielles dans ces réseaux que dans l'espace complet. Plus d'analyses seraient nécessaires pour comprendre ce qui se passe.

Dans la figure 6.9 page suivante, l'opérateur de remise à zéro n'est utilisé dans aucun des environnements. Dans l'environnement Half-Cheetah, les chevilles sont perçues dès le départ comme avec NFAC(λ)-V. C'est avec l'algorithme DDPG que les couches développementales apportent le plus de bénéfices.

Dans la figure 6.10 page suivante, l'opérateur de remise à zéro de CMA-ES est utilisé. La performance médiane est légèrement meilleure avec les couches développementales, mais il y a surtout une différence de variance. D'après toutes les expériences, l'opérateur de remise à zéro ne semble être utile qu'à CMA-ES et non à DDPG.

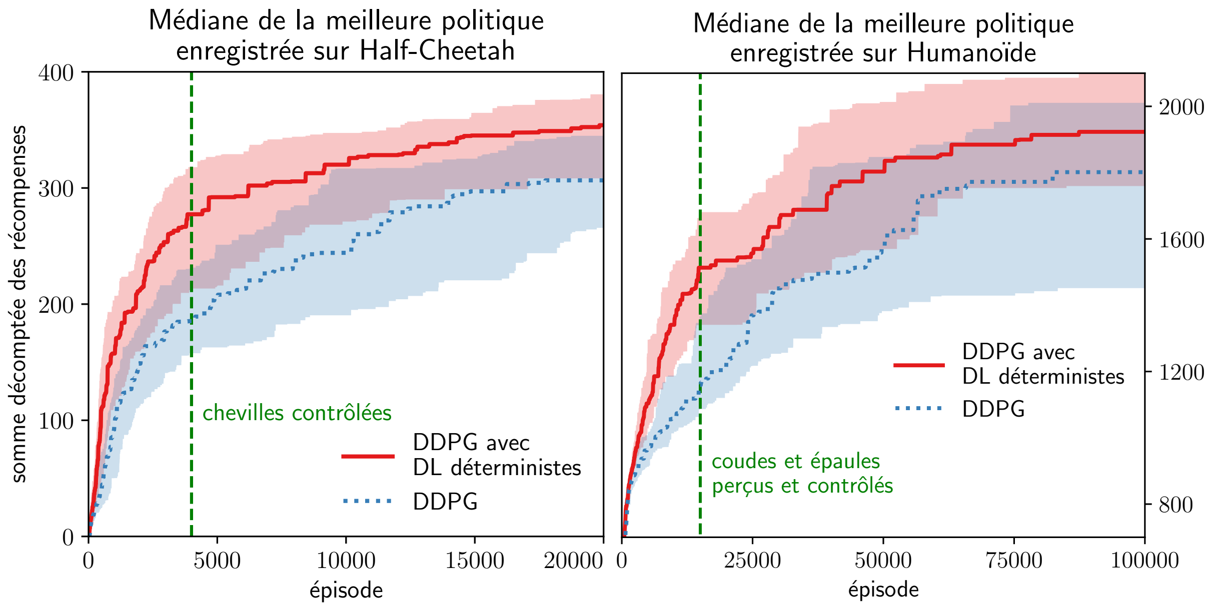


FIGURE 6.9 – Performances de DDPG avec des DL déterministes sur 50 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Les meilleurs points de développement sont affichés en vert.

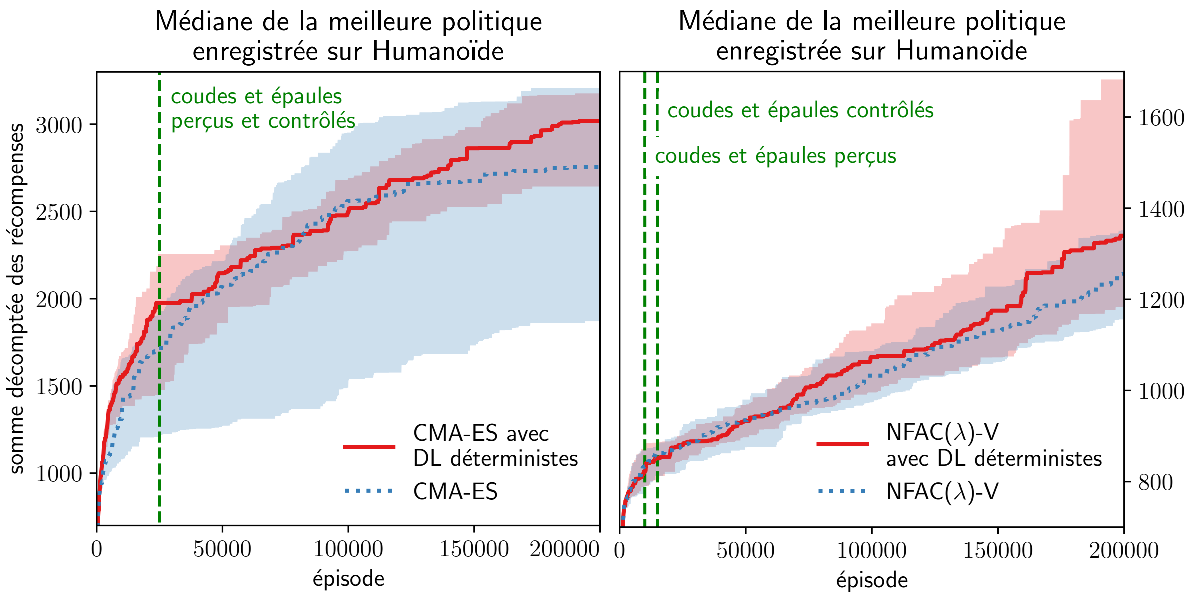


FIGURE 6.10 – Performances de CMA-ES et NFAC(λ)-V avec des DL déterministes sur l'environnement Humanoïde sur 50 essais avec des graines aléatoires différentes. Les médianes et les quartiles affichés sont calculés durant la phase de test sans exploration. Les meilleurs points de développement ppd_i sont affichés en vert.

6.4.4 Discussion

Décider *a priori* quels membres devraient se développer est un problème morphologique dépendant beaucoup de l'environnement. Un expert peut fournir, à cet égard, des connaissances sur le domaine en précisant des contraintes ou en donnant l'ordre de développement complet des membres. L'une des contraintes évidentes étant qu'un agent ne devrait pas contrôler un membre qu'il ne perçoit pas, au risque d'ajouter de la stochastique à l'apprentissage.

Tenter de répondre *a priori* à la question de *quand* définir les points de développement entre les différents MDP est également un problème difficile. Cette seconde question ne dépend pas uniquement de l'environnement, mais aussi de l'algorithme d'apprentissage : chaque algorithme apprend à une vitesse différente. Les points de développement ne sont pas partagés entre algorithmes, comme nous pouvons le voir dans la validation expérimentale précédente. Néanmoins, cette seconde question concernant le moment où développer des membres est probablement plus simple à résoudre à l'aide de méta heuristiques que l'ordre de développement des membres. Par exemple, il serait intéressant d'utiliser la «motivation intrinsèque» (Oudeyer et Kaplan, 2008), ou de compter les états explorés (Bellemare *et al.*, 2016) pour procéder au changement de MDP en apprenant les poids des couches développementales de manière *online*.

6.5 Conclusion

Dans le cadre de l'apprentissage par renforcement développemental, nous avons introduit un nouveau paradigme basé sur l'augmentation progressive de l'espace sensorimoteur de l'agent et proposé une solution algorithmique basée sur ce que nous appelons des couches développementales. Nous avons prouvé expérimentalement la validité de ce concept en montrant que le simple ajout de couches développementales, aux réseaux de neurones, pouvait aider l'agent à apprendre plus rapidement de meilleures politiques, sur deux environnements continus de grande dimension, avec des algorithmes différents.

Néanmoins, de nombreuses limitations à cette approche existent. Le MDP cible doit pouvoir être décomposable en tâche de dimensions inférieures. Des actions neutres doivent être présentes dans la fonction de transition et de récompense des MDP sources. La fonction de récompense des MDP sources doit contenir des informations, au risque de transférer une solution aléatoire. Enfin, l'initialisation des nouveaux paramètres de l'espace complet ne doit pas trop perturber la solution transférée.

Il nous semble que le fait de faire grossir progressivement les dimensions de l'espace sensorimoteur est assez original. Cette idée n'est d'ailleurs pas mentionnée ou évoquée dans les travaux du domaine (Bengio *et al.*, 2009; Narvekar *et al.*, 2016). Cette troisième contribution ouvre de nouvelles perspectives et s'inscrit de manière complémentaire aux diverses méthodes utilisées dans l'apprentissage par renforcement développemental.

Bilan

Dans cette thèse, nous nous sommes attaqués à un problème intéressant de l'intelligence artificielle : l'apprentissage d'un comportement par un agent en interaction avec un environnement dont il ne connaît rien. Nous avons fait le postulat que l'environnement de l'agent était continu pour des raisons de cohérence avec l'approche développementale à laquelle nous aspirons. Aussi, dans notre travail, durant la phase d'exécution dans l'environnement, le comportement d'un agent est entièrement régi par une fonction mathématique, appelée *politique*.

Puisque l'espace sensorimoteur de l'agent est continu, il nous faut passer par des approximations. Au cours du chapitre 1, nous avons décrit différents modèles de représentation de fonctions pouvant être utilisés comme approximations de *politiques*. Parmi les modèles présentés, les réseaux de neurones sont les plus à même de correspondre à l'approche développementale : ils nécessitent très peu de connaissances préalables pour être appris. En plus de cela, ils sont capables de gérer des espaces continus, ont une bonne capacité de *mise à l'échelle* et disposent d'un large *pouvoir de représentation*. Cependant, par rapport aux autres modèles, ils nécessitent un nombre assez important de données d'apprentissage pour être efficaces. Nous avons ensuite répertorié différentes méthodes d'optimisation pour apprendre les paramètres de ces réseaux de neurones. Nous nous sommes plus particulièrement intéressés à deux algorithmes faisant partie de l'état de l'art, la *descente de gradient* et CMA-ES. Enfin, nous avons mis en exergue le dilemme *biais-variance*, présent dans l'ensemble des algorithmes d'apprentissage par renforcement détaillés ultérieurement. Une estimation *non biaisée* est, en moyenne, conforme à la réalité, là où une estimation à faible *variance* jouit de peu de dispersion autour de sa moyenne impliquant une meilleure stabilité.

Pour nous inscrire dans un apprentissage développemental, nous avons fait l'hypothèse que l'agent apprenait à partir de récompenses. Définir une fonction de récompense est moins restrictif et moins informatif que fournir directement la sortie à associer à une entrée, comme dans l'*apprentissage supervisé*. Au regard de la diversité des comportements possibles, apprendre à partir de récompenses est peu contraignant tout en rendant possible une évaluation de ces comportements. L'agent n'a pas connaissance des états buts à atteindre, ni des conséquences de ses actions, ni de solutions initiales fournies : il apprend de sa propre expérience par ses interactions, ce qui est essentiel dans une approche développementale. Aussi, dans le chapitre 2, nous avons présenté les algorithmes existants en apprentissage par renforcement. Dans un premier temps, nous avons exposé, à partir de l'expérience passée, les différentes façons d'évaluer une situation à l'aide des *fonctions de valeur* ; comment intégrer la connaissance provenant de comportements distincts à l'aide de l'apprentissage *off-policy*, et, par quels moyens profiter des approximations présentées dans le chapitre 1, pour manipuler des espaces d'états continus. Dans un second temps, nous avons vu de quelle manière une *politique* pouvait être approximée sur un espace continu d'actions et les façons d'explorer de multiples comportements. Enfin, en combinant les deux approches précédentes, nous avons montré que les algorithmes *acteur-critique* pouvaient

être une solution efficace au problème des environnements continus. Pour finir, nous avons mis en évidence le dilemme entre *mise à l'échelle* et *efficacité en données*.

Au cours du chapitre 3, nous avons simplement exposé les différents environnements et problèmes auxquels l'agent serait confronté au cours des expériences de validation dans cette thèse. Ils sont au nombre de quatre, ont un espace sensorimoteur de dimension élevée pour certains, et sont tous continus. Nous les avons implémentés à l'aide d'un moteur physique (ODE) pour des simulations plus réalistes.

Au fil du chapitre 4, nous nous sommes globalement intéressés aux méthodes *acteur-critique on-policy*. En début de thèse, les seuls algorithmes efficaces, permettant de gérer des espaces continus et d'utiliser des réseaux de neurones, étaient CACLA et CMA-ES. Dans une première expérience, nous avons corroboré la bonne efficacité de CACLA dans un environnement ayant de nombreuses dimensions. Nous avons montré les limites de CMA-ES lorsque le nombre de paramètres à optimiser était élevé. Le point faible de CACLA est qu'il n'utilise qu'une seule fois les données qu'il produit, car il fonctionne de manière *online*. En partant de ce constat, nous avons proposé un nouvel algorithme *Neural Fitted Actor-Critic* (NFAC) qui reprend la mise à jour de l'acteur proposée par CACLA, mais qui rejoue les données à l'aide d'un *replay buffer* pour apprendre le critique. Nous avons ensuite étendu NFAC avec les traces d'éligibilité pour le rendre plus efficace en contrôlant le dilemme *biais-variance*. Finalement, nous avons validé expérimentalement la meilleure *efficacité en données* de NFAC par rapport à CACLA ou A3C sur trois environnements.

Pendant la thèse, le domaine de l'apprentissage par renforcement profond a fortement évolué donnant lieu à de nouveaux algorithmes : DDPG, Retrace, TRPO, A3C, Q-Prop, etc. En s'appuyant sur une mise à jour de l'acteur moins efficace que celle de CACLA, DDPG propose d'utiliser la fonction de valeur Q pour faire de l'apprentissage *off-policy* en rejouant les données précédemment générées, le rendant plus efficace en données que CACLA qui est *on-policy*. Au sein du chapitre 5, nous avons voulu étendre notre algorithme NFAC au cas *off-policy*. Nous avons observé expérimentalement que l'algorithme Retrace, qui permet de pondérer des transitions *off-policy*, pouvait être appliqué à la fonction de valeur V, mais n'engendrait pas de gains significatifs en matière d'*efficacité en données*. Nous avons alors décidé de passer à l'apprentissage de la fonction de valeur Q pour proposer l'algorithme *off-policy* NFAC-∂Q, abandonnant par la même occasion la mise à jour de l'acteur suggérée par CACLA, car nous avons relevé qu'elle était inefficace avec Q dans NFAC-δQ. Nous avons montré empiriquement que NFAC-∂Q pouvait être *efficace en données* pour trouver des solutions plus rapidement que DDPG, sur trois environnements, grâce à l'utilisation de méthode *fitted*, mais qu'il manquait de stabilité : il favorise trop l'exploration. À nouveau, Retrace n'a pas permis d'amélioration significative de NFAC-∂Q. Enfin, nous avons établi une comparaison expérimentale finale montrant que NFAC-V, l'algorithme *on-policy* proposé dans le chapitre 4, était, en fin de compte, le plus efficace parmi DDPG et NFAC-∂Q.

À ce moment, nous disposions d'algorithmes assez robustes pour passer à la problématique de l'exploration guidée de l'espace sensorimoteur de l'agent. Dans l'optique de parvenir à un apprentissage à long terme, nous conjecturons que pour réussir à trouver une solution satisfaisante dans un espace de recherche vaste, il est plus aisé de se confronter au préalable à des sous-espaces. À l'intérieur du chapitre 6, nous avons cherché à travailler sur les représentations des fonctions plutôt que sur le fond des algorithmes. Nous avons suggéré une définition possible pour l'augmentation progressive de l'espace sensorimoteur de l'agent à travers par une succession de MDP analogues. Pour répondre à ce nouveau problème, et montrer qu'il pouvait être bénéfique

à l'apprentissage, nous avons imaginé une manière de ne pas altérer l'environnement et l'algorithme d'apprentissage à l'aide de *couches développementales* qui contrôlent quelles dimensions de l'espace sensorimoteur sont perceptibles et contrôlables. Finalement, nous avons mis en œuvre cette solution et avons montré empiriquement que les *couches développementales* pouvaient permettre un meilleur apprentissage, sur deux environnements à forte dimensionnalité, plutôt que d'apprendre directement dans l'espace de recherche complet.

Conclusion

Deux idées s'opposent dans cette thèse, l'une est celle de l'*efficacité en données*, car elle permet d'apprendre plus rapidement et de s'approcher de problèmes réalistes, l'autre est celle de la capacité à construire des représentations avec peu d'*a priori*. Les réseaux de neurones font partie de la seconde catégorie, que nous privilégions, car nous nous inscrivons dans une approche développementale. Aussi, si l'*efficacité en données* est primordiale, il convient de remettre en cause le choix des réseaux de neurones au profit d'autres modèles disposant de moins de paramètres à apprendre, car il existe en apprentissage automatique un principe selon lequel plus un modèle dispose de paramètres plus il aura besoin de données pour les apprendre. Dans nos travaux, nous avons cherché à rendre les réseaux de neurones plus efficaces en données, il s'avère que rejouer des données, qui sont coûteuses à produire, est fondamental dans cette recherche.

Dans la recherche de l'*efficacité en données*, le choix de la méthode d'optimisation des réseaux de neurones est contestable. Si la *descente de gradient* permet de garder une bonne capacité de *mise à l'échelle* grâce à son faible temps de calcul, $O(n)$ en nombre de paramètres, des méthodes de second ordre (ou CMA-ES) peuvent atteindre de meilleures solutions avec moins de données, au prix d'un coût en $O(n^2)$ en nombre de paramètres. Ici, nous avons, à nouveau, fait le choix de sacrifier l'*efficacité en données* au profit de la *mise à l'échelle*. Ce choix fut pragmatique car, dans nos expériences en simulation, produire des données n'était pas si coûteux, à l'inverse de situations réelles, par exemple en robotique. En choisissant de rejouer des données avec la *descente de gradient*, nous établissons un compromis entre les méthodes en $O(n)$ et celles en $O(n^2)$. Ce compromis a un coût en $O(n \cdot r)$, où $r \ll n$ est le nombre de répétitions.

En apprentissage par renforcement, pour les contraintes que nous nous sommes fixées, à savoir l'utilisation de réseaux de neurones avec *la descente de gradient*, les méthodes *acteur-critique* sont les plus à même d'être *efficaces en données* en environnement continu. La généralisation, présente dans les approximations des fonctions de valeur du critique, permet de réduire la variance et d'évaluer des situations encore jamais rencontrées. De plus, l'utilisation par l'acteur de politiques approximées permet de garder une certaine continuité dans les actions et d'éviter une étape d'optimisation coûteuse lorsque l'espace des actions est continu.

Utiliser un *replay buffer* est un compromis entre les méthodes *model-based* et celles *model-free*. L'ensemble des données du *replay buffer* constitue un modèle non paramétrique. Avec des réseaux de neurones, il n'est pas évident de déterminer l'approche la plus *efficace en données* entre le *replay buffer* et les méthodes *model-based*. La taille et le nombre de données utilisées du *replay buffer*, paramétrables dans les algorithmes DDPG et NFAC-∂Q déterminent la préférence entre *efficacité en données* et *mise à l'échelle*. Néanmoins, il n'y a pas de garantie que rejouer plus de données permette d'être toujours plus *efficace en données*, à cause de la variance engendrée les approximateurs peuvent devenir instables (*surapprentissage*). Quant aux méthodes purement *model-based* utilisant des réseaux de neurones, il leur est difficile de prédire avec exactitude le prochain état dans des espaces ayant une dimensionnalité élevée.

Les méthodes *fitted*, sur lesquelles se basent nos algorithmes NFAC, sont une forme spécifique de *replay*. Plutôt que de rejouer des données tirées aléatoirement du *replay buffer*, elles sont choisies comme faisant partie de la même trajectoire. Puis sur ces mêmes trajectoires, plutôt que de les rejouer une seule fois, la minimisation du critère MSTDE est opérée plusieurs fois, ce que nous appelons le nombre de *fitted* itérations. Dans les fonctions de valeur, sachant que la cible à apprendre est mouvante, car dépendante de son approximation actuelle, répéter une trajectoire permet de maintenir une certaine stabilité, car toutes les cibles feront partie de la prochaine mise à jour (sauf celle de la dernière transition si elle n'atteint pas un état absorbant). Ainsi, les méthodes *fitted*, qui sont une relaxation des méthodes *least-square*, permettent de tendre vers l'*efficacité en données*.

La mise à jour *online* de l'acteur proposée par CACLA, se basant sur une estimation de la fonction avantage, peut être effectuée de manière *offline* tout en maintenant son efficacité dans notre algorithme NFAC-V. Disposer d'une bonne estimation de la fonction avantage est essentiel pour mettre à jour efficacement une politique paramétrée à partir d'un critique. Là où les gradients du coût d'une politique déterministe ou stochastique vont aussi s'éloigner des actions ayant un avantage négatif, CACLA ne se rapproche que des actions déjà réalisées où la fonction avantage est positive, gardant ainsi une bonne stabilité dont nous profitons dans NFAC-V. Enfin, pouvoir être *offline* et donc utiliser des *mini-batch*, permet de bénéficier de la *batch normalization* ou de *Rprop*.

Nos expérimentations sur la pondération des transitions *off-policy* nous ont suggéré que la meilleure façon de prendre en compte des données *off-policy* pour l'apprentissage de politiques déterministes était d'apprendre la fonction de valeur Q en *one-step* en tirant parti de sa définition même. De ces échecs à bénéficier des liaisons entre les transitions, nous inférons que l'augmentation de la variance provoquée par les méthodes *multi-step off-policy* est trop importante, d'autant plus avec une stratégie de *replay* augmentant encore cette variance.

Pour perfectionner nos algorithmes, nous nous intéressons alors à améliorer les représentations de fonctions à travers le problème de l'augmentation progressive de l'espace sensorimoteur. L'apprentissage profond utilise le principe selon lequel trouver une solution dans un vaste espace de recherche peut se faire à l'aide d'un nombre de données satisfaisant : l'algorithme de rétropropagation du gradient se chargera de définir l'importance des différentes connexions et, dans les architectures véritablement profondes, permet la construction de représentations intermédiaires, plus ou moins génériques et réutilisables. L'idée de l'augmentation progressive de l'espace sensorimoteur n'est pas de remettre en cause ce principe valide, mais de répondre au problème que, lors d'un apprentissage réaliste à long terme, le nombre de données à récupérer est trop important : nous proposons alors de restreindre l'espace de recherche. Pour effectuer cette restriction, les couches développementales sont une solution intéressante, car l'environnement et l'algorithme d'apprentissage ne sont pas altérés. De nos expériences, nous concluons qu'une exploration guidée de l'espace sensorimoteur peut aboutir à un apprentissage plus efficace et rapide. Nous sommes conscients que ces recherches sur les couches développementales n'en sont qu'à un stade préliminaire, mais nous pensons qu'elles peuvent s'inscrire dans le dessein plus grand du *curriculum learning* et de l'apprentissage développemental où les agents contrôlent la complexité de ce qu'ils apprennent.

En fin de compte, nous pensons que les avancées sur les représentations que l'agent utilise ont plus d'impact que les avancées concernant directement le domaine de l'apprentissage par renforcement. Par exemple, profiter des différentes techniques provenant de l'apprentissage profond (fonctions d'activation ReLU, *batch normalization*, ADAM, etc.), a plus d'influence que choisir

si l'apprentissage doit être *on-policy* ou *off-policy*, *one-step* ou *multi-step* ou s'il est préférable d'utiliser la mise à jour de l'acteur proposée dans CACLA ou celle provenant du gradient du coût d'une politique déterministe. Par ailleurs, ce sentiment est conforté par les récents travaux de Henderson *et al.* (2017). De ce fait, nous pensons qu'il faut réussir à produire des représentations plus efficaces, dédiées à l'apprentissage par renforcement.

Dans le futur, pour justement développer des représentations plus efficaces, nous postulons que les paramètres de ces représentations devraient interagir encore plus entre eux. Les perceptrons multicouches sont une première avancée dans ce sens, alors que dans les modèles linéaires, les paramètres sont totalement indépendants, avec les perceptrons multicouches et les réseaux profonds, certains paramètres sont liés et en interaction. Les couches développementales vont encore plus loin dans cette idée de l'interaction des paramètres, car elles permettent à certains paramètres d'en désactiver d'autres. En suivant cette idée, nous pensons que d'autres structures de réseaux de neurones sont prometteuses. Néanmoins, l'augmentation de l'interaction entre les paramètres demande de trouver de nouveaux algorithmes d'apprentissage, de la même façon que les perceptrons multicouches ont eu besoin de l'algorithme de rétroprogation du gradient pour se démocratiser.

Perspectives

Efficacité en données en maintenant une bonne mise à l'échelle La recherche de l'*efficacité en données* atteint rapidement ses limites en rejouant toutes les données présentes dans le *replay buffer*, le coût en calcul devenant de plus en plus important avec le temps, ce qui oblige à oublier certaines transitions. Or, oublier des données est contradictoire avec la recherche de l'*efficacité en données*. Pour dépasser ce problème, il faudrait pouvoir choisir quelles sont les données les plus importantes à rejouer, sans en supprimer du *buffer*. Par exemple, Schaul *et al.* (2015) proposent de rejouer les données où le critique fait le plus d'erreurs, néanmoins, en pratique, cette approche est réalisée de façon approximée, car il est trop coûteux d'évaluer tout le *buffer* à l'aide du critique.

Aujourd'hui, deux échelles sont principalement utilisées en apprentissage par renforcement : celle de la transition, bas niveau, utilisée dans les algorithmes *critic-based*, et celle des paramètres de la politique, plus haut niveau, souvent utilisée dans les algorithmes *actor-only*. Or, à l'échelle de la transition, utiliser un *replay buffer* est très coûteux numériquement, à cause du problème décrit dans le paragraphe précédent, tandis qu'à l'échelle des paramètres de la politique l'*efficacité en données* est faible. Utiliser une échelle intermédiaire pourrait être une solution, par exemple les *Semi Markov Decision Processes* (Sutton *et al.*, 1999b) permettent de compresser plusieurs transitions au sein d'une *option*, ainsi il serait potentiellement moins coûteux d'évaluer quelles *options* rejouer tout en garantissant une certaine *efficacité en données*.

D'après notre dernière expérience du chapitre 5, NFAC-V et DDPG sont les deux algorithmes les plus efficaces. Or NFAC-V est *on-policy* en apprenant V et DDPG est *off-policy* en apprenant Q, combiner ces deux algorithmes semble prometteur et relativement simple à mettre en œuvre. Effectivement, l'idée de combiner une mise à jour *on-policy* avec une mise à jour *off-policy* a récemment été utilisée par plusieurs algorithmes (Q-prop, ACER, etc.) (Gu *et al.*, 2016a; Wang *et al.*, 2016; Gu *et al.*, 2017).

Représentation Pour augmenter la capacité de *mise à l'échelle* des algorithmes d'apprentissage par renforcement utilisant des réseaux de neurones, une alternative, plus spéculative, serait d'employer un algorithme capable d'optimiser les poids du réseau en ayant une complexité $O(\log(n))$ en nombre de paramètres, plus faible que *la descente de gradient*. Pour cela, il faudrait probablement remettre en cause l'architecture des perceptrons multicouches, pour la remplacer par un réseau où l'ajout d'une connexion ne signifie pas forcément de calculs supplémentaires pour la sortie, par exemple en rendant les connexions utilisées dépendantes des entrées.

Plusieurs améliorations des réseaux de neurones sont dédiées à l'apprentissage par renforcement, en particulier au niveau de l'organisation des réseaux pour apprendre les fonctions de valeurs (Wang *et al.*, 2015; Gu *et al.*, 2016b; Tamar *et al.*, 2016).

Exploration Un progrès important pouvant être bénéfique aux méthodes *acteur-critique*, concerne la stratégie d'exploration. L'exploration naïve et non informée utilisée au cours de cette thèse

n'est pas la plus efficace. Pour rester dans le cadre d'une approche développementale, l'exploration devrait rester non-informée, mais elle peut être plus sophistiquée, par exemple en comptant ou estimant la fréquence des états où l'agent est déjà passé (Bellemare *et al.*, 2016). Une autre amélioration possible de l'exploration consiste à explorer l'espace des paramètres dans les méthodes *acteur-critique* (Sehnke *et al.*, 2010).

Évaluation Comme soulevé dans plusieurs articles récents (Henderson *et al.*, 2017; Islam *et al.*, 2017), les algorithmes d'apprentissage par renforcement profond en environnements continus sont difficiles à comparer. Contrairement à l'apprentissage par renforcement avec actions discrètes où l'existence d'une série d'environnements ouverts a permis plusieurs progrès (Bellemare *et al.*, 2013), il n'existe pas d'environnements continus équivalents. Le simulateur le plus utilisé, Mujoco (Todorov *et al.*, 2012), n'est ni ouvert ni gratuit. Les environnements ouverts, que nous avons proposés avec ODE, sont une proposition allant en ce sens, mais plus de travail est nécessaire pour les rendre utilisables avec d'autres langages.

Augmentation de l'espace sensorimoteur Nous avons seulement effleuré l'exploration guidée de l'espace sensorimoteur. De nombreuses possibilités sont envisageables, par exemple en choisissant d'autres articulations à développer sur nos environnements (les genoux sur l'humanoïde) ou d'autres schémas de développement des réseaux de neurones en tirant profit des *progressive neural networks* (Rusu *et al.*, 2016) et en diminuant le *catastrophic forgetting* (Kirkpatrick *et al.*, 2017).

Pour les couches développementales, l'une des pistes les plus importantes concerne l'apprentissage automatique des poids de ces couches sans recherche en grille. Pour ce faire, il est possible d'imaginer des heuristiques modifiant les poids lorsque l'apprentissage de l'agent stagne ou en se basant sur la motivation intrinsèque (Oudeyer et Kaplan, 2008). De telles métriques permettraient de choisir le moment où un membre est perceptible et contrôlable. Néanmoins, elles ne permettent pas de définir *a priori* l'ordre de développement des membres. En réalité, il nous semble difficile de pouvoir prédire quel ordre de développement sera le plus efficace. Pour les êtres vivants, cet ordre est prédéfini par l'ADN et les lois physiques. Si l'on voulait déterminer cet ordre de manière *online*, il faudrait laisser la possibilité aux couches développementales de «redésactiver» un membre pour essayer plusieurs ordres au sein du même agent par essai-erreur, mais il semble peu probable que cette approche soit alors plus efficace qu'un apprentissage direct dans l'espace de recherche complet sans couches développementales, sauf, potentiellement, à plus long terme.

Nous pourrions également imaginer la possibilité de réaliser une augmentation progressive de l'espace sensorimoteur au sein même d'un épisode, en remettant en cause le caractère répétitif des épisodes. En fixant une durée d'épisode bien plus longue, et en réinitialisant uniquement l'environnement si un état absorbant est atteint, nous nous rapprocherions plus d'un apprentissage au long terme. Ainsi, l'échelle de comparaison en serait modifiée, ce qui pourrait potentiellement mettre plus en valeur l'apprentissage à l'aide des couches développementales.

Vers l'intelligence artificielle générale ? De nombreux défis attendent encore d'être résolus en apprentissage par renforcement, par exemple, celui concernant le jeu Starcraft 2 semble particulièrement intéressant (Vinyals *et al.*, 2017). Néanmoins, nous pensons que les approches actuelles sont limitées, car elles s'attaquent directement aux problèmes finaux, ce qui, poussé à son paroxysme, aboutit à contredire l'idée de Turing sur la façon dont l'intelligence doit s'apprendre

et se développe petit à petit. C'est pour cette raison que nous accordons tant d'importance à l'aspect développemental de l'apprentissage par renforcement.

Grâce à la complexité des réseaux de neurones et l'interaction de leurs paramètres, nous avons partiellement perdu la capacité à expliquer les représentations développées par l'agent. Une étape vers l'intelligence artificielle générale, que nous pensons importante, est de perdre maintenant la capacité de pouvoir expliquer les algorithmes utilisés au sein d'un agent. De façon analogue aux réseaux de neurones, en imaginant que l'agent dispose de plusieurs briques algorithmiques de base, comme les fonctions d'activation dans les réseaux de neurones, il pourrait construire ses propres algorithmes. Se pose toujours la question de savoir comment un agent peut décider, de lui-même, de son objectif et de ce qu'il veut apprendre. Au-delà du mécanisme de la curiosité, qui pourrait aider à répondre à ces questions, nous pensons que l'émergence d'une intelligence artificielle générale ne peut avoir lieu que dans un environnement continu, riche, aux interactions multiples, potentiellement simulé, mais avec une notion de corps physique.

Annexe A

Métaparamètres

L'ensemble des résultats que nous avons obtenus n'a pas été répertorié dans ce manuscrit, en en particulier ceux concernant l'optimisation des métaparamètres et leur sensibilité. Sur chacune des courbes du manuscrit, nous avons fait au mieux pour optimiser de façon rigoureuse les métaparamètres pour chacun des algorithmes.

Pour avoir un ordre d'idée des calculs effectués, d'après les statistiques de Grid5000, nous avons lancé plus de 70000 tâches pour l'équivalent d'environ 900 années de calculs sur un ordinateur disposant un seul CPU. Ainsi, les courbes proposées ne représentent qu'environ 2% de l'ensemble des calculs réellement effectués.

Malgré cette puissance de calcul à disposition, en réalité, elle n'était pas suffisante et nous nous refusons à tirer des conclusions totalement définitives sur la supériorité de tel ou tel algorithme, car le nombre de métaparamètres et leur domaine sont tellement importants qu'il est difficile d'avoir des certitudes sur l'optimalité des métaparamètres trouvés. Les travaux de Henderson *et al.* (2017) vont également dans en ce sens.

A.1 Métaparamètres fixés

Les temps d'exécutions sont affichés sur des nœuds de Grid5000 disposant de processeurs datant de 2008 jusqu'à 2017 (en moyenne 2013). Pour les algorithmes pouvant être exécutés en parallèle, le nombre moyen de cœurs est de 10.

Réseaux de neurones

- coefficient linéaire négatif des couches *leaky ReLU* : 0.01
- température des tangente hyperbolique : 1
- inertie de la moyenne glissante pour la *batch normalization* : 0.999
- paramètre évitant la division par zéro pour la *batch normalization* : $\zeta = 10^{-9}$
- initialisation des poids : $\mathcal{N}(0, 0.01)$
- les produits scalaires contiennent toujours une unité supplémentaire pour le biais

L'application de la *batch normalization* à l'apprentissage par renforcement n'est tout à fait évidente, car la sortie des réseaux en phase de test ou d'apprentissage n'est pas la même lorsque des couches de *batch normalization* sont présentes. En effet, la *batch normalization* dispose de deux phases : celle d'apprentissage où les statistiques sont calculés sur le *mini-batch* courant, et celle de test où les statistiques ont été calculés de façon glissante sur l'ensemble des *mini-batch* précédemment présentés. Ainsi, nous utiliserons la phase de test de la *batch normalization*

pour l'exploration et l'évaluation des algorithmes. À l'exception de NFAC-V, qui, si la *batch normalization* est activée, utilise également la phase de test de la *batch normalization* pour calculer $\pi_\theta(s_t)$ dans :

$$\Delta\theta = - \sum_{(s_t,a_t)\in\mathcal{D}_\mu} \mathbb{1}_{\hat{\delta}_t>0} \Big(\pi_\theta(s_t) - a_t\Big) \frac{\partial \pi_\theta(s_t)}{\partial \theta}.$$

Dans ce cas, il ne faut pas oublier de calculer $\pi_\theta(s_t)$ en phase d'apprentissage, même si la sortie n'est pas utilisée, pour mettre à jour les statistiques.

ADAM

- inertie moyenne glissante $\beta_1 = 0.$
- inertie variance glissante $\beta_2 = 0.999$
- éviter division par zéro $\zeta = 10^{-8}$

Dans l'article introduisant ADAM, Kingma et Ba (2015) proposent d'utiliser les valeurs suivantes : $(\beta_1, \beta_2) = (0.9, 0.999)$. Ce sont des métaparamètres qui sont utilisés en apprentissage supervisé. Néanmoins, en apprentissage par renforcement, les réseaux de neurones doivent s'adapter plus rapidement, car la cible à apprendre est changeante. Ainsi, nous utiliserons une inertie pour la moyenne glissante valant $\beta_1 = 0$.

Pour être sûr que cette hypothèse est valide, nous avons procédé à plusieurs expériences sur les algorithmes NFAC(λ)-V, NFAC(0)-∂Q et DDPG où les inerties ont été choisies parmi l'ensemble suivant : $(\beta_1, \beta_2) \in \{(0.9, 0.999); (0, 0.999); (0.9, 0); (0, 0)\}$. Le couple $(\beta_1, \beta_2) = (0., 0.999)$ a été le plus performant sur tous les algorithmes.

iRrop-

- variation maximale sur une itération $\Delta_{\max} = 50.$
- variation minimale sur une itération $\Delta_{\min} = 0.$
- augmentation de la variation sur une itération $\Delta^+ = 1.2$
- diminution de la variation sur une itération $\Delta^- = 0.5$

A.2 Algorithmes acteur-critique online on-policy

Configuration commune

- réseau de neurones de l'acteur : 18 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 6 unités (TanH)
- stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$
- taux d'apprentissage acteur (sauf CMA-ES et RANDOM) $\alpha_a = 0.0001$
- méthode d'optimisation (sauf CMA-ES et RANDOM) : ADAM online avec Caffe

CMA-ES

- population : 25
- écart-type initial : $\sigma = 0.3$
- temps moyen d'exécution (1 cœur) : 1 heure

DAC

- réseau de neurones du critique Q : 24 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 1 unité (linéaire)

— taux d'apprentissage critique $\alpha_q = 0.001$
— temps d'exécution moyen (1 cœur) : 2 heures 10 minutes

SAC
— réseau de neurones du critique Q : 24 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 1 unité (linéaire)
— taux d'apprentissage critique $\alpha_q = 0.001$
— temps d'exécution moyen (1 cœur) : 1 heure 30 minutes

CACLA
— réseau de neurones du critique V : 18 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 1 unité (linéaire)
— taux d'apprentissage critique $\alpha_v = 0.001$
— temps d'exécution moyen (1 cœur) : 2 heures

A.3 Limitations de CMAES

Configuration 50×25 unités Il s'agit de la même que dans l'expérience précédente (Annexe A.2 page ci-contre).

Configuration 4 unités
— réseau de neurones de l'acteur : 18 unités (entrées) $\times$ 4 unités (Leaky ReLU) $\times$ 6 unités (TanH)
— population : 50
— écart-type initial : 0.5
— temps d'exécution moyen (1 cœur) : 30 minutes

Configuration 10 unités
— réseau de neurones de l'acteur : 18 unités $\times$ 10 unités (Leaky ReLU) $\times$ 6 unités (TanH)
— population : 20
— écart-type initial : 0.5
— temps d'exécution moyen (1 cœur) : 30 minutes

A.4 CACLA et NFAC sur Cartpole et Acrobot

Cette expérience est plus ancienne que les autres, elle a été réalisée avec la bibliothèque FANN. Les poids du réseau sont initialisés par la formule suivante : $\mathcal{N}(0, \frac{\sqrt{n}}{2})$ où n est le nombre de connexions entrantes. Le code des environnements est également différent, il peut être retrouvé à l'aide de git (tag : «esann2016_code»). La fonction d'activation «Lecun» correspond à la définition suivante :

$$y(x) \leftarrow \sqrt{3}\Big(\frac{2}{1+e^{-2x}} - 1\Big).$$

Acrobot
— $\gamma = 0.6$
— fréquence de décision : tous les 35 pas de simulation
— stratégie d'exploration : ϵ-greedy avec $\epsilon = 0.1$

Cartpole
- $\gamma = 0.95$
- stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$

CACLA
- réseau de neurones de l'acteur : 1 unité $\times$ 5 unités (Lecun) $\times$ 1 unité (TanH)
- réseau de neurones du critique : 1 unité $\times$ 10 unités (Lecun) $\times$ 1 unité (linéaire)
- taux d'apprentissage de l'acteur : $\alpha_a = 0.003$ (Acrobot) $\alpha_a = 0.006$ (Cartpole)
- taux d'apprentissage critique : $\alpha_v = 0.0015$
- méthode d'optimisation : descente de gradient stochastique online
- temps d'exécution moyen (1 cœur) : 1 minute

NFAC(0)-V
- réseau de neurones de l'acteur : 1 unité $\times$ 5 unités (Lecun) $\times$ 1 unité (TanH)
- réseau de neurones du critique : 1 unité $\times$ 25 unités (Lecun) $\times$ 1 unité (linéaire)
- méthode d'optimisation : Rprop sur la taille d'une trajectoire
- temps d'exécution moyen (1 cœur) : 1 minute

A.5 NFAC(λ) sur Half-Cheetah

Configuration commune
- réseau de neurones de l'acteur : 18 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 6 unités (TanH)
- réseau de neurones du critique V : 18 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 25 unités (Leaky ReLU) $\times$ 1 unité (linéaire)
- stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$
- taux d'apprentissage de l'acteur : $\alpha_a = 0.001$
- taux d'apprentissage du critique : $\alpha_v = 0.001$
- méthode d'optimisation : ADAM sur la taille d'une trajectoire avec Caffe

NFAC(0)-V
- traces d'éligibilité : $\lambda = 0$
- nombre de *fitted* itérations pour le critique : $K = 20$
- temps d'exécution moyen (multicœur) : 1 heure 35 minutes

NFAC(λ)-V
- traces d'éligibilité : $\lambda = 0.6$
- nombre de *fitted* itérations pour le critique : $K = 15$
- GAE : oui
- temps d'exécution moyen (multicœur) : 1 heure 20 minutes

A.6 Algorithmes *on-policy* sur Half-Cheetah

La configuration de CACLA et CMAES a été précédemment décrite (Annexe A.2 page 130) ainsi que celle de NFAC(λ)-V également (Annexe A.5). NFAC(λ)-V avec A3C n'implique pas de nouveau paramètre.

A.7 NFAC(0) – ∂Q

Configuration commune

- stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.05$
- batch normalization : avec sauf CMA-ES et CACLA

Configuration commune pour Acrobot et Cartpole

- réseau de neurones de l'acteur : 4 unités (entrées) $\times$ 5 unités (Leaky ReLU) $\times$ 1 unité (linéaire)
- réseau de neurones du critique V (CACLA et NFAC-V) : 4 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 7 unités (Leaky ReLU) $\times$ 1 unité (linéaire)
- réseau de neurones du critique Q (DDPG et NFAC-Q) : 5 unités (entrées) $\times$ 50 unités (Leaky ReLU) $\times$ 7 unités (Leaky ReLU) $\times$ 1 unité (linéaire)

CACLA sur Acrobot

- taux d'apprentissage du critique : $\alpha_v = 0.01$
- taux d'apprentissage de l'acteur : $\alpha_a = 0.01$
- méthode d'optimisation : ADAM online avec Caffe
- temps d'exécution moyen (1 cœur) : 1 minute

NFAC(0)-V sur Acrobot

- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_v = 0.01$
- nombre de *fitted* itérations pour le critique : $K = 25$
- remise à zéro du critique : oui
- méthode d'optimisation : 100 itérations de ADAM sur la taille d'une trajectoire avec Caffe
- temps d'exécution moyen (multicœur) : 2 heures 25 minutes

CMA-ES sur Acrobot et Cartpole

- écart-type initial : $\sigma = 0.3$
- population : 14
- temps d'exécution moyen (1 cœur) : 1 minute

DDPG sur Acrobot

- taille d'un minibatch : 64
- taille maximale du replay buffer : 10^6
- mise à jour des target networks : $\tau = 10^{-3}$
- régularisation L2 : sans
- inverting gradient : non
- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_q = 0.1$
- répétition de la mise à jour : 8 minibatch différents
- méthode d'optimisation : ADAM sur minibatch avec Caffe
- temps d'exécution moyen (multicœur) : 25 minutes

NFAC(0)-∂Q sur Acrobot

- taille maximale du replay buffer : 6000
- inverting gradient : non
- remise à zéro du critique : oui
- remise à zéro de l'acteur : oui
- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_q = 0.1$
- nombre de *fitted* itérations pour le critique : $G_c = 1$
- nombre de *fitted* itérations pour l'acteur : $G_a = 25$
- nombre global de *fitted* itérations : $K = 10$
- méthode d'optimisation de l'acteur : ADAM sur la taille d'une trajectoire
- méthode d'optimisation du critique : 10 itérations de ADAM sur la taille d'une trajectoire
- temps d'exécution moyen (multicœur) : 7 heures

CACLA sur Cartpole

- taux d'apprentissage du critique : $\alpha_v = 0.1$
- taux d'apprentissage de l'acteur : $\alpha_a = 0.001$
- temps d'exécution moyen (1 cœur) : 1 minute

NFAC(0)-V sur Cartpole

- taux d'apprentissage de l'acteur : $\alpha_a = 0.001$
- taux d'apprentissage du critique : $\alpha_v = 0.1$
- nombre de *fitted* itérations pour le critique : $K = 25$
- remise à zéro du critique : oui
- méthode d'optimisation : 50 itérations de ADAM sur la taille d'une trajectoire avec Caffe
- temps d'exécution moyen (multicœur) : 11 heures 35 minutes

DDPG sur Cartpole

- taille d'un minibatch : 64
- taille maximale du replay buffer : 10^6
- mise à jour des target networks : $\tau = 10^{-3}$
- régularisation L2 : sans
- inverting gradient : oui
- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_q = 0.1$
- répétition de la mise à jour : 8 minibatch différents
- temps d'exécution moyen (multicœur) : 10 minutes

NFAC(0)-∂Q sur Cartpole

- taille maximale du replay buffer : 3000
- inverting gradient : oui
- remise à zéro du critique : non
- remise à zéro de l'acteur : oui
- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_q = 0.1$
- nombre de *fitted* itérations pour le critique : $G_c = 1$
- nombre de *fitted* itérations pour l'acteur : $G_a = 25$

- nombre global de *fitted* itérations : $K = 10$
- méthode d'optimisation de l'acteur : ADAM sur la taille d'une trajectoire
- méthode d'optimisation du critique : 10 itérations de ADAM sur la taille d'une trajectoire
- temps d'exécution moyen (multicœur) : 5 heures 25 minutes

Configuration commune pour Half-Cheetah

- réseau de neurones de l'acteur : 18 unités (entrées) × 20 unités (Leaky ReLU) × 10 unités (Leaky ReLU) × 6 unités (linéaire)
- réseau de neurones du critique V (CACLA et NFAC-V) : 18 unités (entrées) × 50 unités (Leaky ReLU) × 7 unités (Leaky ReLU) × 1 unité (linéaire)
- réseau de neurones du critique Q (DDPG et NFAC-Q) : 24 unités (entrées) × 50 unités (Leaky ReLU) × 7 unités (Leaky ReLU) × 1 unité (linéaire)

CACLA sur Half-Cheetah

- taux d'apprentissage du critique : $\alpha_v = 0.01$
- taux d'apprentissage de l'acteur : $\alpha_a = 0.5$
- temps d'exécution moyen (1 cœur) : 10 minutes

NFAC(0)-V sur Half-Cheetah

- taux d'apprentissage de l'acteur : $\alpha_a = 0.3$
- taux d'apprentissage du critique : $\alpha_v = 0.1$
- nombre de *fitted* itérations pour le critique : $K = 25$
- remise à zéro du critique : oui
- méthode d'optimisation : 10 itérations de ADAM sur la taille d'une trajectoire avec Caffe
- temps d'exécution moyen (multicœur) : 7 heures 35 minutes

CMA-ES sur Half-Cheetah

- écart-type initial : $\sigma = 0.3$
- population : 23
- temps d'exécution moyen (1 cœur) : 5 minutes

DDPG sur Half-Cheetah

- taille d'un minibatch : 64
- taille maximale du replay buffer : 10^6
- mise à jour des target networks : $\tau = 10^{-3}$
- régularisation L2 : sans
- inverting gradient : non
- taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
- taux d'apprentissage du critique : $\alpha_q = 0.3$
- répétition de la mise à jour : 8 minibatch différents
- temps d'exécution moyen (multicœur) : 12 heures 30 minutes

NFAC(0)-∂Q sur Half-Cheetah

- taille maximale du replay buffer : 6000
- inverting gradient : non
- remise à zéro du critique : oui
- remise à zéro de l'acteur : oui

— taux d'apprentissage de l'acteur : $\alpha_a = 0.1$
— taux d'apprentissage du critique : $\alpha_q = 0.3$
— nombre de *fitted* itérations pour le critique : $G_c = 1$
— nombre de *fitted* itérations pour l'acteur : $G_a = 25$
— nombre global de *fitted* itérations : $K = 10$
— méthode d'optimisation de l'acteur : ADAM sur la taille d'une trajectoire
— méthode d'optimisation du critique : 10 itérations de ADAM sur la taille d'une trajectoire
— temps d'exécution moyen (multicœur) : 11 heures 45 minutes

A.8 NFAC(0) $- \delta Q$

Configuration commune sur Half-Cheetah

— réseau de neurones de l'acteur : 18 unités (entrées) × 50 unités (ReLU) × 25 unités (ReLU) × 6 (TanH)
— réseau de neurones du critique Q : 24 unités (entrées) × 50 unités (ReLU) × 25 unités (ReLU) × 1 (linéaire)
— trajectoires utilisées : 1
— inverting gradient : non
— remise à zéro du critique : non
— remise à zéro de l'acteur : non
— taux d'apprentissage de l'acteur : $\alpha_a = 0.001$
— taux d'apprentissage du critique : $\alpha_q = 0.001$
— nombre de *fitted* itérations pour le critique : $G_c = 20$
— nombre de *fitted* itérations pour l'acteur : $G_a = 1$
— nombre global de *fitted* itérations : $K = 1$
— méthode d'optimisation : ADAM sur la taille d'une trajectoire
— stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$

Le nombre d'actions générées pour le cas *off-policy* μ était de 10.

A.9 Comparaison finale

NFAC(λ)-V utilise les mêmes métaparamètres que dans l'annexe A.5 page 132, sauf qu'il utilise la *batch normalization* sur la première couche de son acteur. CACLA utilise les mêmes métaparamètres que dans l'annexe A.2 page 130.

Configuration commune à DDPG et NFAC(0)-∂Q sur Half-Cheetah

— réseau de neurones de l'acteur : 18 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 6 unités (TanH)
— réseau de neurones du critique Q : 24 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 1 unité (linéaire)
— stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$
— taux d'apprentissage du critique : $\alpha_q = 0.001$
— batch normalization : première couche de l'acteur
— inverting gradient : non

Tous ces paramètres, à l'exception du réseau du critique et de la batch normalization pour CACLA, sont partagés avec NFAC(λ)-V et CACLA.

NFAC(0)-∂Q sur Half-Cheetah

- taux d'apprentissage de l'acteur : $\alpha_a = 0.00001$
- trajectoires utilisées : 10
- reward scale : 10
- remise à zéro du critique : non
- remise à zéro de l'acteur : non
- nombre de *fitted* itérations pour le critique : $G_c = 10$
- nombre de *fitted* itérations pour l'acteur : $G_a = 1$
- nombre global de *fitted* itérations : $K = 1$
- méthode d'optimisation : ADAM sur la taille de 10 trajectoires avec Caffe
- nombre d'actions générées pour estimer la cible du critique : 10
- temps d'exécution moyen (multicœur) : 8 heures 40 minutes

DDPG sur Half-Cheetah

- taille d'un minibatch : 32
- taille maximale du replay buffer : 10^6
- mise à jour des target networks : $\tau = 0.01$
- reward scale : 1
- régularisation L2 : sans
- taux d'apprentissage de l'acteur : $\alpha_a = 0.0001$
- méthode d'optimisation : ADAM sur minibatch avec Caffe
- temps d'exécution moyen (multicœur) : 4 heures 20 minutes

A.10 Couches développementales

CMA-ES

- écart-type initial : $\sigma = 0.5$
- population : 50

CMA-ES sur Half-Cheetah

- réseau de neurones de l'acteur : 18 unités (entrées) $\times$ 25 unités (ReLU) $\times$ 6 (linéaire)
- points de développement pdd_i (activations déterministes) : épisode 0 (genoux perçus et contrôlés), épisode 2500 (chevilles perçues) et épisode 4000 (chevilles contrôlées)
- points de développement pdd_i (activations stochastiques) : épisode 0 (genoux perçus et contrôlés), épisode 4000 (chevilles totalement perçues et contrôlées)
- opérateur de remise à zéro : oui avec les activations déterministes
- temps d'exécution moyen (1 cœur) : 1 heure

CMA-ES sur Humanoïde

- réseau de neurones de l'acteur : 45 unités (entrées) $\times$ 25 unités (ReLU) $\times$ 17 (linéaire)
- points de développement pdd_i : épisode 0 (coudes et épaules perçus) et épisode 25000 (coudes et épaules contrôlés)
- opérateur de remise à zéro : oui
- temps d'exécution moyen (1 cœur) : 4 heures 10 minutes (200000 épisodes)

DDPG

- batch normalization : sans

— taille d'un minibatch : 64
— taille maximale du replay buffer : 10^6
— mise à jour des target networks : $\tau = 10^{-3}$
— reward scale : 1
— régularisation L2 : sans
— inverting gradient : oui
— taux d'apprentissage de l'acteur : $\alpha_a = 0.01$
— taux d'apprentissage du critique : $\alpha_q = 0.001$
— stratégie d'exploration : loi gausienne tronquée avec $\sigma = 10^{-4}$
— méthode d'optimisation : ADAM sur minibatch avec Caffe

DDPG sur Half-Cheetah
— réseau de neurones de l'acteur : 18 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 6 (linéaire)
— réseau de neurones du critique Q : 24 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 1 (linéaire)
— points de développement pdd_i : épisode 0 (chevilles perçues) et épisode 4000 (chevilles contrôlées)
— opérateur de remise à zéro : non
— temps d'exécution moyen (multicœur) : 4 heures 50 minutes

DDPG sur Humanoïde
— réseau de neurones de l'acteur : 45 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 17 (linéaire)
— réseau de neurones du critique Q : 62 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 1 (linéaire)
— points de développement pdd_i : épisode 15000 (coudes et épaules perçus et contrôlés)
— opérateur de remise à zéro : non
— temps d'exécution moyen (multicœur) : 13 heures 5 minutes (100000 épisodes)

La configuration de NFAC(λ)-V sur Half-Cheetah a déjà été décrite (Annexe A.5 page 132). Il ne manque que les points de développement pdd_i qui sont définis aux épisodes : 4000 (figure de gauche) et 10000 (figure de droite).

NFAC(λ)-V sur Humanoïde
— trace d'éligibilité : $\lambda = 0.8$
— stratégie d'exploration : loi gausienne tronquée avec $\sigma = 0.1$
— taux d'apprentissage de l'acteur : $\alpha_a = 0.0001$
— taux d'apprentissage du critique : $\alpha_v = 0.001$
— nombre de *fitted* itérations pour le critique : $K = 5$
— batch normalization : sans
— réseau de neurones de l'acteur : 45 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 17 (TanH)
— réseau de neurones du critique V : 45 unités (entrées) × 50 unités (Leaky ReLU) × 25 unités (Leaky ReLU) × 1 (linéaire)
— points de développement pdd_i : épisode 10000 (coudes et épaules perçus) et épisode 15000 (coudes et épaules contrôlés)
— temps d'exécution moyen (multicœur) : 5 heures 45 minutes (200000 épisodes)

L'ensemble des paramètres de NFAC(λ)-V sur l'environnement Humanoïde peut probablement être amélioré. Néanmoins, par manque de temps, nous avons utilisé ceux-ci.

Annexe B

Outils informatiques

Code source Le code source des algorithmes et des environnements développés au cours de cette thèse est librement disponible à l'adresse suivante : `https://github.com/matthieu637/ddrl`.

Grid5000 Nous avons grandement utilisé la grille de calcul Grid5000 pour réaliser nos expériences. Conformément à la charte d'utilisation, nous faisons apparaître ici le remerciement officiel : «Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`)».

Caffe Nous avons principalement utilisé la bibliothèque Caffe pour implémenter nos réseaux de neurones (Jia *et al.*, 2014).

FANN Au cours de la première année et dans notre première contribution, nous nous sommes servis de la bibliothèque *Fast Artificial Neural Network* (Nissen, 2003) pour implémenter nos réseaux de neurones.

ODE Le moteur physique *Open Dynamic Engine* (Smith, 2005) a été utilisé pour réaliser nos différents environnements.

GNU Octave Nous avons utilisé Octave (Eaton *et al.*, 2016) pour analyser les données générées par nos expériences.

Autres Nous remercions également les développeurs (souvent bénévoles) :
- des systèmes d'exploitation Arch Linux et Debian,
- des langages de programmation C++ et Python,
- des outils GNU GCC, CMake, Valgrind, Git, Kdevelop, Latex, TexStudio, etc.

Bibliographie

James S. Albus : A theory of cerebellar function. *Mathematical Biosciences*, 10(1-2):25–61, 1971.

Shun-ichi Amari : Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10 (2):251–276, 1998. ISSN 0899-7667.

András Antos, Csaba Szepsvari et Rémi Munos : Fitted Q-iteration in continuous action-space MDPs. *Advances in neural information processing systems*, pages 9–16, 2008.

Minoru Asada, Koh Hosoda, Yasuo Kuniyoshi, Hiroshi Ishiguro, Toshio Inui, Yuichiro Yoshikawa, Masaki Ogino et Chisato Yoshida : Cognitive Developmental Robotics : A Survey. *IEEE Transactions on Autonomous Mental Development*, 1(1):1–44, 2009. ISSN 19430604.

Peter Auer : Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

Anne Auger et Nikolaus Hansen : A restart CMA evolution strategy with increasing population size. *In IEEE Congress on Evolutionary Computation*, volume 2, pages 1769–1776. IEEE, 2005.

Leemon C. Baird III et Andrew W. Moore : Gradient descent for general reinforcement learning. *In Advances in neural information processing systems*, pages 968–974, 1999.

Adrien Baranes et Pierre-Yves Oudeyer : Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, jan 2013. ISSN 09218890.

Horace B. Barlow : Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

Andrew G. Barto, Richard S. Sutton et Charles W. Anderson : Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, (5):834–846, 1983.

Marc G. Bellemare, Yavar Naddaf, Joel Veness et Michael Bowling : The arcade learning environment : An evaluation platform for general agents. *In International Joint Conference on Artificial Intelligence*, volume 47, pages 253—-279, 2013. ISBN 9781577357384.

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton et Remi Munos : Unifying count-based exploration and intrinsic motivation. *In Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

Richard Bellman : Dynamic programming and Lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

Richard Bellman : *Dynamic Programming.* Princeton University Press, 1957. ISBN 9780691079516.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert et Jason Weston : Curriculum learning. *In Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

Shalabh Bhatnagar, Doina Precup, David Silver, Richard S. Sutton, Hamid R. Maei et Csaba Szepesvári : Convergent temporal-difference learning with arbitrary smooth function approximation. *In Advances in Neural Information Processing Systems*, pages 1204–1212, 2009.

Christopher M. Bishop : *Pattern recognition and machine learning.* Springer, 2006.

Avrim L. Blum et Pat Langley : Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997.

Justin A. Boyan : Least-Squares Temporal Difference Learning. *Proceedings of the 16th International Conference on Machine Learning*, (August):49–56, 1998.

Justin A. Boyan et Andrew W. Moore : Generalization in reinforcement learning : Safely approximating the value function. *Advances in neural information processing systems*, pages 369–376, 1995.

Steven J. Bradtke, Andrew G. Barto et Pack Kaelbling : Linear least-squares algorithms for temporal difference learning. *In Recent Advances in Reinforcement Learning*, pages 33–57. 1996.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang et Wojciech Zaremba : OpenAI gym. *arXiv preprint arXiv :1606.01540*, 2016.

Rodney A. Brooks : Intelligence without representation. *Artificial intelligence*, 1991.

David S. Broomhead et David Lowe : Radial basis functions, multi-variable functional interpolation and adaptive networks. Rapport technique, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

Robert R. Bush et Frederick Mosteller : Stochastic models for learning. *John Wiley & Sons, Inc.*, 1955.

Lucian Buşoniu, Alessandro Lazaric et Mohammad Ghavamzadeh : Least-squares methods for policy iteration. *Reinforcement Learning*, pages 75–109, 2012.

Augustin Cauchy : Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

Tony F. Chan, Gene H. Golub et Randall J. LeVeque : Algorithms for computing the sample variance : Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

Yuhu Cheng, Huanting Feng et Xuesong Wang : Actor-Critic algorithm based on incremental least-squares temporal difference with eligibility trace. *In International Conference on Intelligent Computing*, pages 183–188. Springer, 2011.

Rémi Coulom : *Reinforcement learning using neural networks, with applications to motor control.* Thèse de doctorat, Institut National Polytechnique de Grenoble-INPG, 2002.

Thomas Degris, Martha White et Richard S. Sutton : Off-policy actor-critic. *arXiv preprint arXiv :1205.4839*, 2012.

Marc P. Deisenroth, Gerhard Neumann, Jan Peters et Others : A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1–2):1–142, 2013.

Marc P. Deisenroth et Carl E. Rasmussen : PILCO : A model-based and data-efficient approach to policy search. *In International Conference on Machine Learning*, pages 465–472, 2011.

Stephane DONCIEUX : Creativity : A driver for research on robotics in open environments. *Intellectica*, (65):205–219, 2016.

Kenji DOYA : Reinforcement learning in continuous time and space. *Neural computation*, 2000.

Yan DUAN, Xi CHEN, John SCHULMAN et Pieter ABBEEL : Benchmarking Deep Reinforcement Learning for Continuous Control. *arXiv preprint arXiv :1604.06778*, 2016.

Sahibsingh A. DUDANI : The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, (4):325–327, 1976.

Alain DUTECH : Self-organizing developmental reinforcement learning. *From Animals to Animats 12*, pages 1–11, 2012.

John W. EATON, David BATEMAN, Soren HAUBERG et Rik WEHBRING : *GNU Octave version 4.2.0 manual : a high-level interactive language for numerical computations.* 2016. URL `http://www.gnu.org/software/octave/doc/interpreter`.

Bradley EFRON : Bootstrap methods : another look at the jackknife. *The annals of Statistics*, pages 1–26, 1979.

Dumitru ERHAN, Yoshua BENGIO, Aaron COURVILLE, Pierre-Antoine MANZAGOL, Pascal VINCENT et Samy BENGIO : Why does unsupervised pre-training help deep learning ? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

Damien ERNST, Pierre GEURTS et Louis WEHENKEL : Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556, 2005.

Chrisantha FERNANDO, Dylan BANARSE, Charles BLUNDELL, Yori ZWOLS, David HA, Andrei A. RUSU, Alexander PRITZEL et Daan WIERSTRA : Pathnet : Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv :1701.08734*, 2017.

Ra FISHER : *Statistical methods for research workers.* Numéro V. 1925. ISBN 0050021702.

Robert M. FRENCH : Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

Matthieu GEIST et Olivier PIETQUIN : Revisiting natural actor-critics with value function approximation. *In International Conference on Modeling Decisions for Artificial Intelligence*, pages 207–218. Springer, 2010.

Stuart GEMAN, Elie BIENENSTOCK et René DOURSAT : Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

Alborz GERAMIFARD, Michael BOWLING et Richard S. SUTTON : Incremental least-squares temporal difference learning. *In Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 356. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2006.

Federico GIROSI, Michael JONES et Tomaso POGGIO : Regularization theory and neural networks architectures. *Neural computation*, 7(2):219–269, 1995.

Ian GOODFELLOW, Yoshua BENGIO et Aaron COURVILLE : *Deep learning.* MIT press, 2016.

Geoffrey J. GORDON : Stable function approximation in dynamic programming. *In Proceedings of the twelfth international conference on machine learning*, pages 261–268, 1995.

Shixiang GU, Timothy LILLICRAP, Zoubin GHAHRAMANI, Richard E. TURNER et Sergey LEVINE : Q-Prop : Sample-Efficient Policy Gradient with An Off-Policy Critic. *arXiv preprint arXiv :1611.02247*, 2016a.

Shixiang GU, Timothy LILLICRAP, Zoubin GHAHRAMANI, Richard E. TURNER, Bernhard SCHÖLKOPF et Sergey LEVINE : Interpolated Policy Gradient : Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning. *arXiv preprint arXiv :1706.00387*, 2017.

Shixiang GU, Timothy LILLICRAP, Ilya SUTSKEVER et Sergey LEVINE : Continuous Deep Q-Learning with Model-based Acceleration. *arXiv preprint arXiv :1603.00748*, 2016b.

Frank GUERIN : Learning like a baby : a survey of artificial intelligence approaches. *The Knowledge Engineering Review*, 26(02):209–236, 2011. ISSN 0269-8889.

Hirotaka HACHIYA, Takayuki AKIYAMA, Masashi SUGIAYMA et Jan PETERS : Adaptive importance sampling for value function approximation in off-policy reinforcement learning. *Neural Networks*, 22(10):1399–1410, 2009.

Roland HAFNER et Martin RIEDMILLER : Reinforcement learning in feedback control. *Machine Learning*, 84(1-2):137–169, 2011. ISSN 0885-6125.

Alon HALEVY, Peter NORVIG et Fernando PEREIRA : The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

Nikolaus HANSEN et Andreas OSTERMEIER : Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001. ISSN 1063-6560.

Anna HARUTYUNYAN, Marc G. BELLEMARE, Tom STEPLETON et Rémi MUNOS : Q (λ) with Off-Policy Corrections. *In International Conference on Algorithmic Learning Theory*, pages 305–320. Springer International Publishing, 2016.

Trevor HASTIE, Robert TIBSHIRANI et Jerome FRIEDMAN : Overview of supervised learning. *In The elements of statistical learning*, pages 9–41. Springer, 2009.

Matthew HAUSKNECHT et Peter STONE : Deep Reinforcement Learning in Parameterized Action Space. *arXiv preprint arXiv :1511.04143*, 2016.

Peter HENDERSON, Riashat ISLAM, Philip BACHMAN, Joelle PINEAU, Doina PRECUP et David MEGER : Deep Reinforcement Learning that Matters. *ArXiv e-prints*, 2017.

Geoffrey E. HINTON : Distributed representations. 1984.

Sepp HOCHREITER et Juergen SCHMIDHUBER : Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. ISSN 0899-7667.

Kurt HORNIK, Maxwell STINCHCOMBE et Halbert WHITE : Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Christian IGEL et Michael HÜSKEN : Improving the Rprop learning algorithm. *In International Symposium on Neural Computation*, pages 115–121, 2000.

Sergey IOFFE et Christian SZEGEDY : Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv :1502.03167*, 2015.

Riashat ISLAM, Peter HENDERSON, Maziar GOMROKCHI et Doina PRECUP : Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv :1708.04133*, 2017.

Yangqing JIA, Evan SHELHAMER, Jeff DONAHUE, Sergey KARAYEV, Jonathan LONG, Ross GIRSHICK, Sergio GUADARRAMA et Trevor DARRELL : Caffe : Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv :1408.5093*, 2014.

Donald R. JONES, Matthias SCHONLAU et William J. WELCH : Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998. ISSN 09255001.

Leslie P. Kaelbling, Michael Littman et Andrew W. Moore : Reinforcement learning : A survey. *Journal of artificial intelligence research*, 1996.

Sham M. Kakade : A natural policy gradient. *In Advances in neural information processing systems*, pages 1531–1538, 2002.

Michael J. Kearns et Satinder P. Singh : Bias-Variance Error Bounds for Temporal Difference Updates. *In COLT*, pages 142–147, 2000.

Diederik P. Kingma et Jimmy L. Ba : Adam : a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho et Agnieszka Grabska-Barwinska : Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, page 201611835, 2017.

Jens Kober, J. Andrew Bagnell et Jan Peters : Reinforcement Learning in Robotics : A Survey. *International Journal of Robotics Research*, 2013.

Ron Kohavi : A study of cross-validation and bootstrap for accuracy estimation and model selection. *In Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995.

Vijay R. Konda et John N. Tsitsiklis : Actor-Critic Algorithms. *Neural Information Processing Systems*, 13:1008–1014, 1999. ISSN 0363-0129.

Alban Laflaquière : *Approche sensorimotrice de la perception de l'espace pour la robotique autonome*. Thèse de doctorat, Université Pierre et Marie Curie-Paris VI, 2013.

Michail G. Lagoudakis et Ronald Parr : Least-squares policy iteration. *Journal of machine learning research*, 4(Dec):1107–1149, 2003.

Yann A. LeCun : Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). *In Proceedings of Cognitiva 85, Paris, France.* 1985.

Yann A. LeCun, Léon Bottou, Genevieve B. Orr et Klaus-Robert Müller : Efficient backprop. *In Neural networks : Tricks of the trade*, pages 9–48. Springer, 2012.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver et Daan Wierstra : Continuous control with deep reinforcement learning. *arXiv preprint arXiv :1509.02971*, 2015.

Long J. Lin : Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992. ISSN 08856125.

Ashique R. Mahmood, Huizhen Yu et Richard S. Sutton : Multi-step Off-policy Learning Without Importance Sampling Ratios. *CoRR*, abs/1702.0, 2017.

James G. March : Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.

Francisco S. Melo et Manuel Lopes : Fitted natural actor-critic : A new algorithm for continuous state-action MDPs. *In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5212 LNAI, pages 66–81, 2008. ISBN 3540874801.

Volodymyr Mnih, Adria P. Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver et Koray Kavukcuoglu : Asynchronous methods for deep reinforcement learning. *In International Conference on Machine Learning*, pages 1928–1937, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland et Others : Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Rémi Munos, Tom Stepleton, Anna Harutyunyan et Marc G. Bellemare : Safe and Efficient Off-Policy Reinforcement Learning. *arXiv preprint arXiv :1606.02647*, 2016.

Kumpati S. Narendra et Mandayam A. L. Thathachar : Learning automata-a survey. *IEEE Transactions on systems, man, and cybernetics*, (4):323–334, 1974.

Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti et Peter Stone : Source task creation for curriculum learning. *In Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 566–574. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

Andrew Y. Ng, Daishi Harada et Stuart Russell : Policy invariance under reward transformations : Theory and application to reward shaping. *In ICML*, volume 99, pages 278–287, 1999.

S Nissen : Implementation of a Fast Artificial Neural Network Library (fann). Rapport technique, Department of Computer Science University of Copenhagen (DIKU), 2003. URL `http://leenissen.dk/fann/wp/`.

Pierre-Yves Oudeyer et Frederic Kaplan : How can we define intrinsic motivation ? *8th Conf on Epigenetic Robotics*, (July):93–101, 2008. ISSN 1098-6596.

Jason Pazis et Michail G. Lagoudakis : Binary action search for learning continuous-action control policies. *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 793–800, 2009.

Jan Peters, Katharina Mülling et Yasemin Altun : Relative Entropy Policy Search. *In Association for the Advancement of Artificial Intelligence*, pages 1607–1612. Atlanta, 2010.

Jan Peters et Stefan Schaal : Policy gradient methods for robotics. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. IEEE, 2006.

Jan Peters et Stefan Schaal : Natural Actor-Critic. *Neurocomputing*, 71(7-9):1180–1190, 2008. ISSN 09252312.

Jean Piaget : La naissance de l'intelligence chez l'enfant. 1948.

Doina Precup : Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series*, page 80, 2000.

Danil V. Prokhorov et Donald C. Wunsch : Adaptive critic designs. *IEEE Transactions on Neural Networks*, 8(5):997–1007, 1997. ISSN 10459227.

Martin Puterman : Markov Decision Processes : Discrete Stochastic Dynamic Programming. 1994.

Martin Riedmiller : Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. *In Lecture Notes in Computer Science*, volume 3720 LNAI, pages 317–328, 2005. ISBN 3540292438.

Martin Riedmiller et Heinrich Braun : RPROP - A Fast Adaptive Learning Algorithm. *In International Symposium on Computer and Information Science VII*, 1992.

Martin Riedmiller, Thomas Gabel, Roland Hafner et Sascha Lange : Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009. ISSN 0929-5593, 1573-7527.

Martin Riedmiller, Jan Peters et Stefan Schaal : Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark. *In IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 254–261, apr 2007. ISBN 1-4244-0706-0.

Frank Rosenblatt : The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Reuven Y. Rubinstein et Dirk P. Kroese : *Simulation and the Monte Carlo method*, volume 10. John Wiley & Sons, 2016.

Gavin A. Rummery et Mahesan Niranjan : *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu et Raia Hadsell : Progressive Neural Networks. *arXiv preprint arXiv :1606.04671*, 2016.

Warren S. Sarle : Stopped training and other remedies for overfitting. *Computing science and statistics*, pages 352–360, 1996.

Tom Schaul, John Quan, Ioannis Antonoglou et David Silver : Prioritized Experience Replay. *arXiv preprint arXiv :1511.05952*, pages 1–23, 2015.

Bruno Scherrer : Should one compute the Temporal Difference fix point or minimize the Bellman Residual ? The unified oblique projection view. *In 27th International Conference on Machine Learning*, 2010.

Bruno Scherrer et Boris Lesner : On the use of non-stationary policies for stationary infinite-horizon Markov decision processes. *In Advances in Neural Information Processing Systems*, pages 1826–1834, 2012.

John Schulman, Sergey Levine, Michael Jordan et Pieter Abbeel : Trust Region Policy Optimization. *International Conference on Machine Learning*, page 16, 2015a. ISSN 2158-3226.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan et Pieter Abbeel : High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv :1506.02438*, 2015b.

John R. Searle : Minds, brains, and programs. *Behavioral and brain sciences*, 3(3):417–424, 1980.

Frank Sehnke, Christian Osendorfer, Thomas Rückstiess, Alex Graves, Jan Peters et Jürgen Schmidhuber : Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

Olivier Sigaud et Alain Droniou : Towards deep developmental learning. *IEEE Transactions on Cognitive and Developmental Systems*, 8(2):99–114, 2016.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra et Martin Riedmiller : Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning*, pages 387–395, 2014.

Satinder P. Singh et Richard S. Sutton : Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

Russell Smith : Open dynamics engine. 2005.

Mark W. Spong : Swing up control problem for the acrobot. *IEEE Control Systems Magazine*, 15(1):49–55, 1995. ISSN 02721708.

Nitish SRIVASTAVA, Geoffrey E. HINTON, Alex KRIZHEVSKY, Ilya SUTSKEVER et Ruslan SALAKHUTDINOV : Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. ISSN 15337928.

Peter STONE, Richard S. SUTTON et Gregory KUHLMANN : Reinforcement Learning for RoboCup Soccer Keepaway. *Adaptive Behavior*, 13(3):165–188, 2005. ISSN 1059-7123, 1741-2633.

Freek STULP et Olivier SIGAUD : Robot Skill Learning : From Reinforcement Learning to Evolution Strategies. *Paladyn, Journal of Behavioral Robotics*, 4(1):49–61, 2013.

Richard S. SUTTON : Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Richard S. SUTTON : Introduction to reinforcement learning with function approximation. *In Tutorial at the Conference on Neural Information Processing Systems*, 2015.

Richard S. SUTTON et Andrew G. BARTO : *Reinforcement Learning : An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998. ISBN 0262193981.

Richard S. SUTTON et Andrew G. BARTO : *Reinforcement learning : An introduction*, volume 1. MIT press Cambridge, 2 édition, 2017.

Richard S. SUTTON, Hamid R. MAEI, Doina PRECUP, Shalabh BHATNAGAR, David SILVER, Csaba SZEPESVÁRI et Eric WIEWIORA : Fast gradient-descent methods for temporal-difference learning with linear function approximation. *In Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000. ACM, 2009.

Richard S. SUTTON, Ashique R. MAHMOOD et Martha WHITE : An Emphatic Approach to the Problem of Off-policy Temporal-Difference Learning. *The Journal of Machine Learning Research*, 17, 2016.

Richard S. SUTTON, David MCALLESTER, Satinder SINGH et Yishay MANSOUR : Policy Gradient Methods for Reinforcement Learning with Function Approximation. *In Advances in Neural Information Processing Systems 12*, pages 1057–1063, 1999a. ISSN 0047-2875.

Richard S. SUTTON, Doina PRECUP et Satinder SINGH : Between MDPs and semi-MDPs : A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112 (1–2):181–211, 1999b. ISSN 0004-3702.

Aviv TAMAR, Yi WU, Garrett THOMAS, Sergey LEVINE et Pieter ABBEEL : Value iteration networks. *In Advances in Neural Information Processing Systems*, pages 2154–2162, 2016.

Yuval TASSA, Tom EREZ et Emanuel TODOROV : Synthesis and stabilization of complex behaviors through online trajectory optimization. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913. IEEE, 2012.

Matthew E. TAYLOR et Peter STONE : Transfer Learning for Reinforcement Learning Domains : A Survey. *Journal of Machine Learning Research*, 10:1633–1685, 2009. ISSN 15324435.

Emanuel TODOROV, Tom EREZ et Yuval TASSA : Mujoco : A physics engine for model-based control. *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

Alan M. TURING : Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

Hado VAN HASSELT : Reinforcement Learning in Continuous State and Action Spaces. *In Reinforcement Learning*, pages 207–251. Springer Berlin Heidelberg, 2012.

Hado VAN HASSELT, A. Rupam MAHMOOD et Richard S. SUTTON : Off-policy TD (λ) with a true online equivalence. *In Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence, Quebec City, Canada*, 2014.

Hado Van Hasselt et Marco A. Wiering : Reinforcement learning in continuous action spaces. *In Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 272–279, 2007. ISBN 1424407060.

Harm Van Seijen, A. Rupam Mahmood, Patrick M. Pilarski, Marlos C. Machado et Richard S. Sutton : True online temporal-difference learning. *Journal of Machine Learning Research*, 17(145):1–40, 2016.

Harm Vanseijen et Richard S. Sutton : A Deeper Look at Planning as Learning from Replay. *In* Francis Bach et David Blei, éditeurs : *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 de *Proceedings of Machine Learning Research*, pages 2314–2322, Lille, France, 2015. PMLR.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander S. Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser et Others : Starcraft II : A new challenge for reinforcement learning. *arXiv preprint arXiv :1708.04782*, 2017.

Niklas Wahlström, Thomas B. Schön et Marc P. Deisenroth : From Pixels to Torques : Policy Learning with Deep Dynamical Models. *arXiv preprint arXiv :1502.02251*, 2015.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu et Nando DE Freitas : Sample Efficient Actor-Critic with Experience Replay. *arXiv preprint arXiv :1611.01224*, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot et Nando De Freitas : Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv :1511.06581*, 2015.

Christopher JCH. Watkins et Peter Dayan : Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Paweł Wawrzyński : Learning to control a 6-degree-of-freedom walking robot. *In International Conference on Computer as a Tool*, pages 698–705, 2007. ISBN 142440813X.

BP. Welford : Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

Paul J. Werbos : Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22(12):25–38, 1977.

Bernard Widrow et Marcian E. Hoff : Adaptive switching circuits. Rapport technique, STANFORD UNIV CA STANFORD ELECTRONICS LABS, 1960.

Marco Wiering et Martijn Van Otterlo : Reinforcement learning. *Adaptation, Learning, and Optimization*, 12, 2012.

L'encyclopédie libre Wikipédia : A schematic drawing of the inverted pendulum on a cart, 2017a. URL `https://commons.wikimedia.org/wiki/File:Cart-pendulum.svg`.

L'encyclopédie libre Wikipédia : Diagramme sur l'apprentissage par renforcement, 2017b. URL `https://en.wikipedia.org/wiki/File:Reinforcement_learning_diagram.svg`.

L'encyclopédie libre Wikipédia : Double compound pendulum, 2017c. URL `https://commons.wikimedia.org/wiki/File:Double-compound-pendulum-dimensioned.svg`.

L'encyclopédie libre Wikipédia : Processus décisionnels de Markov, 2017d. URL `https://en.wikipedia.org/wiki/File:Markov_Decision_Process.svg`.

Ronald J. Williams : Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229—-256, 1992.

Matthieu Zimmer, Yann Boniface et Alain Dutech : Neural Fitted Actor-Critic. *In European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2016a.

Matthieu Zimmer, Yann Boniface et Alain Dutech : Off-Policy Neural Fitted Actor-Critic. *In Deep Reinforcement Learning Workshop, Neural Information Processing Systems*, 2016b.

Matthieu Zimmer, Yann Boniface et Alain Dutech : Toward a data efficient neural actor-critic. *In 13th European Workshop on Reinforcement Learning*, 2016c.

Matthieu Zimmer, Yann Boniface et Alain Dutech : Vers des architectures acteur-critique neuronales efficaces en données. *In Journées Francophones sur la Planification, la Décision et l'Apprentissage pour la conduite de systèmes*, 2016d.

Matthieu Zimmer et Stephane Doncieux : Bootstrapping Q-Learning for Robotics from Neuro-Evolution Results. *IEEE Transactions on Cognitive and Developmental Systems*, 2017.

Table des figures

Introduction **1**
1 Un des problèmes que nous souhaitons résoudre. 4
2 Une politique satisfaisante apprise par nos algorithmes. 5

Chapitre 1 : Apprentissage automatique **9**
1.1 Les trois grandes classes d'apprentissage automatique. 10
1.2 Exemple de différence entre discret et continu. 11
1.3 Exemple d'un problème non discriminable par un modèle linéaire. 12
1.4 Un *neurone formel* avec $X = \mathbb{R}^3$. 14
1.5 Un perceptron multicouche avec $X = \mathbb{R}^4$ et une couche cachée de 3 neurones. . . 15
1.6 Surapprentissage ou généralisation. 21
1.7 Exemple d'un modèle sujet au surapprentissage. 22
1.8 Couche de régularisation : *batch normalization* ou *dropout*. 22
1.9 Illustration du compromis biais-variance en fonction de la complexité du modèle. 23

Chapitre 2 : Apprentissage par renforcement **25**
2.1 Illustration du cadre général de l'apprentissage par renforcement. 25
2.2 Illustration d'un MDP à espace discret. 27
2.3 Différence entre l'apprentissage par renforcement, au sens général, avec l'apprentissage par renforcement classique de Sutton. 28
2.4 Découpage des méthodes en apprentissage par renforcement. 30
2.5 Exemple de deux fonctions de valeurs V^π et V^*. 32
2.6 Illustration du *n-step return* comparé à TD(0) et Monte-Carlo. 36
2.7 Paysage des fonctions de valeurs d'état paramétrées pour un modèle Ψ donné. . . 41
2.8 Critères qualifiants l'optimalité des paramètres d'une fonction de valeur approximée. 43
2.9 Illustration du cadre *Generalized Policy Iteration* 47
2.10 Paysage des politiques paramétrées pour un modèle Ψ_θ donné. 49
2.11 Un environnement où sans exploration la politique trouvée ne sera jamais optimale. 57
2.12 État de l'art sur la résolution des MDP continus. 61

Chapitre 3 : Environnements continus **63**
3.1 Illustration du problème du Cartpole. 63
3.2 Illustration de l'environnement Acrobot. 64
3.3 Illustration de l'environnement Half-Cheetah. 65

3.4 Illustration de l'environnement Humanoïde. 67

Chapitre 4 : Architectures acteur-critique neuronales on-policy **71**

4.1 Forme des réseaux de neurones avec une architecture *acteur-critique*. 73
4.2 Performances des algorithmes *acteur-critique online on-policy*. 75
4.3 Performances de CMA-ES sur Half-Cheetah. 76
4.4 Schéma de l'algorithme NFAC. 77
4.5 NFAC et CACLA sur le Cartpole et l'Acrobot. 79
4.6 Performances des variantes de NFAC(λ). 80
4.7 Exécution temporelle de NFAC. 82
4.8 Comparaison des meilleurs algorithmes acteur-critique neuronaux on-policy. . . . 83

Chapitre 5 : Architectures acteur-critique neuronales off-policy **87**

5.1 Le cadre Neural Fitted Actor Critic 88
5.2 Schéma de l'algorithme NFAC(0)-∂Q. 90
5.3 Comparaisons de NFAC(0)-∂Q et DDPG sur Cartpole et Acrobot. 92
5.4 Comparaisons de NFAC(0)-∂Q et DDPG sur Half-Cheetah. 93
5.5 Comparaison des variantes de NFAC(0)-δQ sur Half-Cheetah. 94
5.6 Comparaisons des variantes *off-policy* pour la mise à jour de l'acteur dans NFAC-V. 97
5.7 Comparaisons des variantes *off-policy* pour la mise à jour du critique dans NFAC-V. 97
5.8 Comparaisons des variantes de NFAC(λ)-∂Q sur Half-Cheetah. 99
5.9 Comparaisons des variantes de NFAC(λ)-∂Q sur Half-Cheetah en matière de données. 99
5.10 Comparaisons des meilleurs algorithmes sur Half-Cheetah. 100

Chapitre 6 : Exploration guidée de l'espace sensorimoteur **103**

6.1 Illustration de notre solution pour l'augmentation de l'espace sensorimoteur de l'agent. 103
6.2 Illustration de l'augmentation de l'espace sensorimoteur de l'agent. 105
6.3 Exemple d'une couche développementale. 107
6.4 Une architecture de contrôle neuronale utilisant des couches développementales. . 108
6.5 Dépendance entre les points de développement, le nombre d'épisodes et les paramètres des couches développementales. 110
6.6 Illustration des environnements Half-Cheetah et Humanoïde. 111
6.7 Performances de CMA-ES avec des couches développementales sur l'environnement Half-Cheetah. 112
6.8 Performances de NFAC(λ)-V avec des couches développementales sur l'environnement Half-Cheetah. 113
6.9 Performances de DDPG avec des couches développementales déterministes. . . . 114
6.10 Performances de CMA-ES et NFAC(λ)-V avec des couches développementales déterministes. 114

Liste des tableaux

Chapitre 1 : Apprentissage automatique **9**
1.1 Récapitulatif des méthodes d'optimisation et leurs hypothèses. 16

Chapitre 2 : Apprentissage par renforcement **25**
2.1 Récapitulatif des avantages et inconvénients des variantes de *Tree-Backup(λ)* . . 40
2.2 Récapitulatif des méthodes d'évaluation de politiques avec leurs hypothèses. . . . 46
2.3 Les politiques d'exploration à adopter en fonction des politiques à apprendre et de l'espace d'exploration. 57

Chapitre 3 : Environnements continus **63**
3.1 Poids et dimensions de Half-Cheetah. 66
3.2 Articulations de Half-Cheetah. 67
3.3 Poids et dimensions de l'environnement Humanoïde. 68
3.4 Articulations de l'environnement Humanoïde. 69
3.5 Caractéristiques des 4 environnements servant de validation expérimentale. 69

Chapitre 4 : Architectures acteur-critique neuronales on-policy **71**
4.1 Comparaison des algorithmes *acteur-critique online*. 73

Chapitre 5 : Architectures acteur-critique neuronales off-policy **87**
5.1 Récapitulatif des avantages et inconvénients des variantes de *Tree-Backup(λ)*. . . 96

Chapitre 6 : Exploration guidée de l'espace sensorimoteur **103**
6.1 Les caractéristiques des algorithmes CMA-ES, NFAC(λ)-V et DDPG. 111

Table des définitions et des théorèmes

Chapitre 1 : Apprentissage automatique
1.1 Définition – Modèle tabulaire . . . 13
1.2 Définition – Modèle linéaire . . . 13
1.3 Définition – Modèle linéaire avec fonctions de base fixées . . . 13
1.4 Définition – Neurone formel . . . 14
1.5 Définition – Perceptron multicouche . . . 15
1.6 Définition – Mean Square Error . . . 16
Chapitre 2 : Apprentissage par renforcement
2.1 Définition – Séquence markovienne . . . 26
2.2 Définition – Processus décisionnel de Markov . . . 26
2.3 Définition – Processus décisionnel de Markov continu en espace . . . 27
2.4 Définition – Politique stochastique . . . 27
2.5 Définition – Politique déterministe . . . 28
2.6 Définition – Critère γ-pondéré . . . 28
2.7 Définition – Ordre total sur espace des politiques . . . 29
2.8 Définition – Politique optimale . . . 29
2.9 Définition – Politique stationnaire . . . 29
2.10 Définition – Fonction de valeur d'état . . . 31
2.11 Définition – Fonction action-valeur . . . 31
2.12 Définition – Politique gloutonne par rapport à V^π . . . 31
2.13 Définition – Politique gloutonne par rapport à Q^π . . . 32
2.14 Définition – Fonction de valeur optimale . . . 32
2.1 Théorème – Équation de Bellman . . . 33
2.15 Définition – États absorbants . . . 34
2.16 Définition – Fonction de valeur paramétrée . . . 41
2.17 Définition – Mean Square Value Error . . . 42
2.18 Définition – Mean Square Temporal Difference Error . . . 42
2.19 Définition – Politique paramétrée stochastique . . . 48
2.20 Définition – Politique paramétrée déterministe . . . 48
2.21 Définition – Politique paramétrée optimale . . . 48
2.22 Définition – Densité probabilité d'une trajectoire . . . 50
2.23 Définition – Densité de probabilité d'un état . . . 52
2.2 Théorème – Policy gradient avec fonction de valeur . . . 53
2.3 Théorème – Policy gradient déterministe avec fonction de valeur . . . 55
2.24 Définition – Politique d'exploration gaussienne . . . 58

Table des algorithmes

Chapitre 1 : Apprentissage automatique **9**
1.1 Descente de gradient 17
1.2 Resilient backpropagation iRPROP- 18
1.3 Adaptive Moment Estimation Optimizer 18
1.4 Méthode de Newton pour l'optimisation de paramètres 19
1.5 Rétropropagation du gradient 20

Chapitre 2 : Apprentissage par renforcement **25**
2.1 TD(0) 35
2.2 TD(λ) 37
2.3 Sarsa(λ) 37
2.4 Tree-Backup(λ) généralisé 39
2.5 Semi-gradient TD(0) 44
2.6 Neural Fitted Q-Iteration 45
2.7 Model-free Policy Search 49
2.8 REINFORCE 51
2.9 Asynchronous Advantage Actor-Critic 54

Chapitre 4 : Architectures acteur-critique neuronales on-policy **71**
4.1 Neural Fitted Actor-Critic(0) 85

Chapitre 5 : Architectures acteur-critique neuronales off-policy **87**
5.1 Neural Fitted Actor-Critic(0)-∂Q 102

Résumé

L'apprentissage par renforcement permet à un agent d'apprendre un comportement qui n'a jamais été préalablement défini par l'homme. L'agent découvre l'environnement et les différentes conséquences de ses actions à travers des interactions avec celui-ci : il apprend de sa propre expérience, sans avoir de connaissances préétablies des buts ni des effets de ses actions.

Cette thèse s'intéresse à la façon dont l'apprentissage profond peut aider l'apprentissage par renforcement à gérer des espaces continus et des environnements ayant de nombreux degrés de liberté dans l'optique de résoudre des problèmes plus proches de la réalité. En effet, les réseaux de neurones ont une bonne capacité de *mise à l'échelle* et un large *pouvoir de représentation*. Ils rendent possible l'approximation de fonctions sur un espace continu et permettent de s'inscrire dans une approche développementale nécessitant peu de connaissances *a priori* sur le domaine.

Nous cherchons comment réduire l'expérience nécessaire à l'agent pour atteindre un comportement acceptable. Pour ce faire, nous avons proposé le cadre *Neural Fitted Actor-Critic* qui définit plusieurs algorithmes *acteur-critique efficaces en données*. Nous examinons par quels moyens l'agent peut exploiter pleinement les transitions générées par des comportements précédents en intégrant des données *off-policy* dans le cadre proposé. Finalement, nous étudions de quelle manière l'agent peut apprendre plus rapidement en tirant parti du développement de son corps, en particulier, en procédant par une augmentation progressive de la dimensionnalité de son espace sensorimoteur.

Mots-clés: apprentissage par renforcement, acteur-critique, réseaux de neurones, environnement continu, approche développementale, apprentissage profond.

Abstract

Reinforcement learning allows an agent to learn a behavior that has never been previously defined by humans. The agent discovers the environment and the different consequences of its actions through its interaction : it learns from its own experience, without having pre-established knowledge of the goals or effects of its actions.

This thesis tackles how deep learning can help reinforcement learning to handle continuous spaces and environments with many degrees of freedom in order to solve problems closer to reality. Indeed, neural networks have a good *scalability* and *representativeness.* They make possible to approximate functions on continuous spaces and allow a developmental approach, because they require little *a priori* knowledge on the domain.

We seek to reduce the amount of necessary interaction of the agent to achieve acceptable behavior. To do so, we proposed the *Neural Fitted Actor-Critic* framework that defines several *data efficient actor-critic* algorithms. We examine how the agent can fully exploit the transitions generated by previous behaviors by integrating *off-policy* data into the proposed framework. Finally, we study how the agent can learn faster by taking advantage of the development of his body, in particular, by proceeding with a gradual increase in the dimensionality of its sensorimotor space.

Keywords: reinforcement learning, actor-critic, neural networks, continuous environment, developmental approach, deep learning