Learning sensori-motor mappings using little knowledge : application to manipulation robotics

François de La Bourdonnaye

UNIVERSITÉ CLERMONT AUVERGNE
ÉCOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

DOCTORAL THESIS

Learning sensori-motor mappings using little knowledge: application to manipulation robotics

---

Thesis defended on December 18th, 2018

by

François de La Bourdonnaye

Jury

Reviewers:	David Filliat	Pr, ENSTA ParisTech
	Ghilès Mostafaoui	PhD HDR, Université de Cergy-Pontoise, ETIS
Examiners:	Atilla Baskurt	Pr, INSA Lyon, LIRIS
	Alain Dutech	PhD HDR, INRIA Grand Est, LORIA
Thesis director:	Thierry Chateau	Pr, Université Clermont Auvergne, Institut Pascal
Thesis supervisors:	Céline Teulière	PhD, Université Clermont Auvergne, Institut Pascal
	Jochen Triesch	Pr, FIAS

Acknowledgements

This work is sponsored by the French government research program "Investissements d'avenir" through the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01), by the European Union through the program Regional competitiveness and employment (ERDF Auvergne region), and by the Auvergne region.

For all the thesis, I would like to thank my supervisors Céline Teulière, Jochen Triesch and Thierry Chateau to trust me to conduct research.

I acknowledge the jury members as well to have accepted analysing all the work done during the PhD.

Furthermore, I acknowledge Laurent Lequièvre to make my work easier in adapting some of my algorithms in a real setting and also Juan Antonio Corrales Ramon and Youcef Mezouar to make available a real robotic platform to me.

Many thanks go to my colleagues and other friends Yosra Dorai, Thanh-Tin Nguyen, Rustem Abdrakhmanov, Gautier Claisse, Charles-Antoine Noury, Ange Nizard, Damien Joubert, Ruddy Theodose, Thomas Wentz, Charles Philippe and Aline Delsart (and all the persons I am forgetting) for all the convivial moments we shared together in Clermont-ferrand.

Of course, I do not forget my family, which gave me a lot of mental support.

Finally, I deeply thank my computer for its power, and meanwhile I apologize to the planet for all the carbon emissions.

Contents

List of Figures

List of Tables

Summary (in french)

Introduction

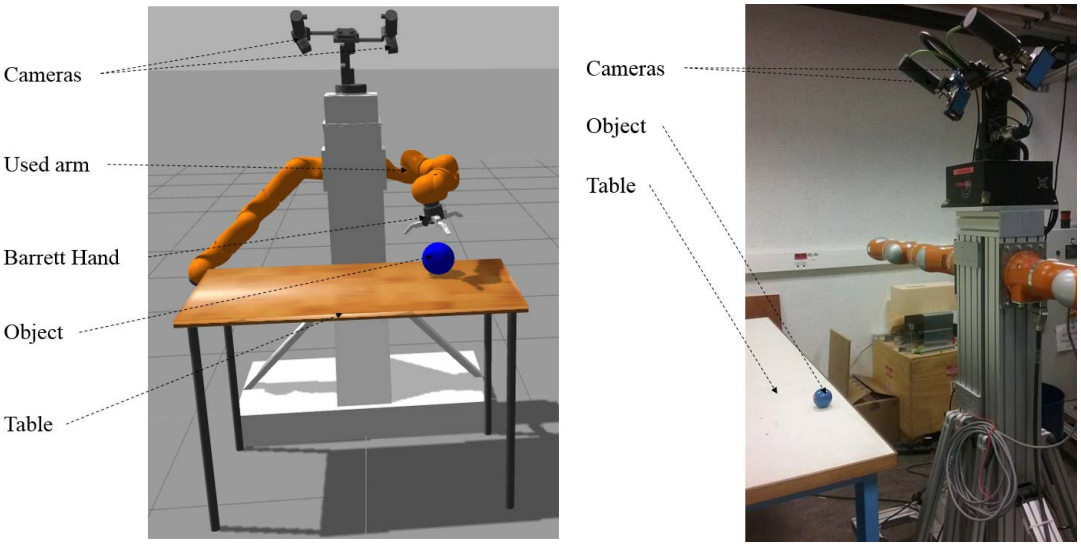
Dans de nombreux domaines, les robots ont révolutionné notre mode de vie. Dans les mondes industriel et agricole, ils ont contribué à l'automatisation de tâches pénibles et répétitives ainsi qu'à la réduction des coûts de production. Par ailleurs, certains robots sont de nos jours utilisés pour de l'assistance à la personne, pour de la maintenance de centrales nucléaires ou pour de l'exploration de milieux inaccessibles tels que les planètes ou les fonds marins.

Dans certains de ces domaines, il reste certains défis à relever. D'une part, dans des environnements ouverts et/ou complexes et/ou inaccessibles, calculer un modèle dynamique du robot dans son environnement est très compliqué. C'est pourquoi, l'utilisation de lois de commande classiques n'assure pas nécessairement une exécution parfaite des tâches robotiques si des variations non prévues se produisent. D'autre part, les tâches robotiques usuelles dépendent très souvent d'un modèle géométrique parfait. Si ce dernier varie, le robot doit en tenir compte. Enfin, certaines tâches complexes peuvent traiter un grand nombre de données, ce qui complique le processus de décision. Pour relever ces défis, il est intéressant de s'inspirer du fonctionnement animal. En effet, (1) les animaux s'adaptent à la plupart des variations d'environnement, (2) ils prennent beaucoup de données en compte pour accomplir leurs tâches (vision, proprioception, odorat, goût...), et (3) personne ne leur fournit des modèles géométriques et dynamiques clé en main, ils les apprennent. Nous nous intéressons particulièrement au dernier aspect dans le sens où fournir des modèles au robot peut être dangereux si ces derniers deviennent erronés. Par exemple, si la détection de piétons pour des "voitures autonomes" repose sur des à prioris et que ces derniers deviennent faux, les voitures risquent de ne plus détecter les piétons.

Ainsi, notre objectif est qu'un robot apprenne une tâche robotique complexe en étant indépendant de modèles pré-calculés par l'être humain. Plus précisément, nous voulons investiguer s'il est possible de se priver de paramètres de calibrage, des modèles géométriques et dynamiques, de descripteurs d'image pré-calculés ou de démonstrations d'expert. Nous appliquons cet objectif sur des tâches liées à la robotique de manipulation. Deux comportements robotiques sont appris. Le premier est une compétence d'atteinte d'objets avec la paume de l'effecteur d'un robot série, ce qui peut être considéré comme une étape de pré-saisie. Selon deux aspects, cette tâche est complexe. D'une part, l'espace d'entrée de la tâche comporte beaucoup de dimensions (images, coordonnées articulaires, et données tactiles). D'autre part, l'apprentissage est effectué pour un nombre important de conditions initiales. La deuxième compétence apprise est la capacité de prédiction d'atteignabilité d'un objet en fonction de la direction du regard du robot (on utilise pour cela un système de deux caméras). Il s'agit d'une compétence utile pour un robot mobile manipulateur. En effet, la prédiction pourrait être utilisée pour se déplacer pour rendre un objet atteignable par le robot.

Matériel

Nous utilisons une plateforme robotique munie de deux bras et d'un système de deux caméras. La figure A présente les versions simulée et réelle de la plateforme d'étude.



A - Images représentant les plateformes robotiques simulée et réelle

État de l'art

Les deux comportements précédemment mentionnés doivent être appris sans modèles pré-calculés par l'être humain. L'apprentissage par renforcement profond est une classe d'algorithmes utile à cet objectif. En effet, utiliser de l'apprentissage par renforcement permet de nous dispenser d'utiliser un modèle dynamique pré-calculé pour apprendre un comportement donné. De plus, l'apprentissage profond permet de se passer de descripteurs pré-calculés pour la représentation d'état. Ainsi, l'apprentissage peut s'effectuer directement sur des espaces d'états à haute dimensionalité.

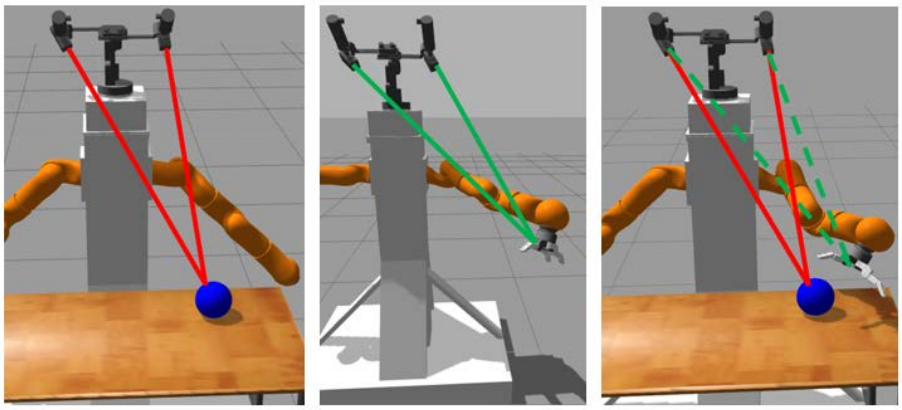
Cependant, le défi principal de ce genre d'approches est la façon de spécifier l'objectif du robot. En effet, les robots ont souvent comme seuls signaux de renforcement un signal positif quand la tâche est effectuée et un signal constant quand elle ne l'est pas, ce qui peut rendre l'apprentissage long et complexe.

Que ce soit avant ou après l'émergence de l'apprentissage profond, deux méthodes principales sont utilisées. Elles ont pour objectif de guider l'apprentissage vers l'objectif du robot. D'une part, des démonstrations d'experts peuvent être utilisées, ce que nous ne voulons pas pour notre approche. D'autre part, des signaux de récompenses informatifs additionnels ("reward shaping") peuvent être employés. Cette approche n'est pas à proscrire en soi. Seulement, ces signaux additionnels dépendent souvent de modèles géométriques et de modules visuels supervisés pour détecter une pose cible, ce que nous voulons éviter.

Des stratégies annexes peuvent également être utilisées comme l'apprentissage "du plus simple au plus compliqué" ("curriculum learning" or "learning from easy missions" en anglais), l'utilisation parallèle de nombreux robots et la décomposition

de la tâche complexe en sous-tâches plus simples. La première solution suppose que l'on connaisse ce qui est "simple" et donc que l'on ait a minima une idée sur un des états terminaux de la tâche. La deuxième solution nécessite des ressources coûteuses. Enfin, la dernière solution n'implique pas nécessairement l'absence de supervision pour les sous tâches considérées mais permet efficacement de réduire la complexité de la tâche.

Approche choisie



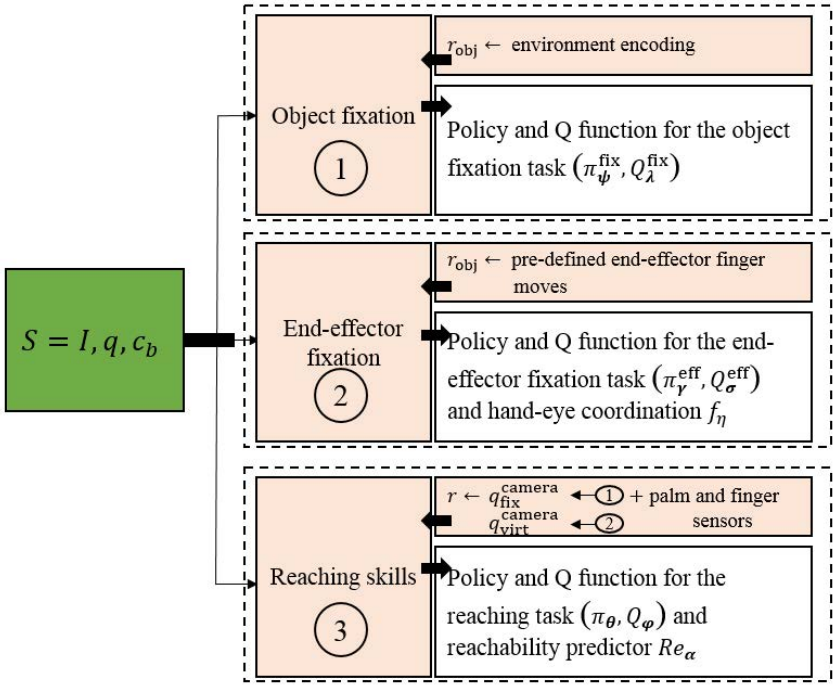
B - Images représentant les 3 étapes de la tâche d'atteinte

En s'inspirant du comportement humain (les humains généralement regardent d'abord les objets qu'ils saisissent), nous choisissons de décomposer la tâche d'atteinte en trois sous tâches.

D'une part, le robot doit apprendre à fixer un objet posé sur la table avec un système de deux caméras. L'idée derrière cette tâche est que les coordonées articulaires du système de caméras encodent dans l'espace la position de l'objet. Ensuite, le robot apprend simultanément une tâche de fixation d'effecteur et une fonction de coordination main-oeil. Cette dernière encode d'une certaine façon la position de l'effecteur dans l'espace en fonction des coordonnées articulaires du bras robotique. Enfin, le robot s'aide des deux précédentes tâches pour apprendre à toucher l'objet. Toutes ces tâches sont apprises l'une après l'autre sans paramètres de calibrage, modèles cinématiques ou dynamiques pré-calculés, et traitements d'image fait à la main. Elles seront détaillées dans les parties qui leur sont dédiées. De plus, les fonctions apprises au cours de ces étapes sont représentées par des réseaux de neurones.

Apprentissage de fixation binoculaire d'objets

La fixation d'objet consiste à bouger les caméras de telle sorte que l'objet soit au centre de l'image. L'apprentissage de la fixation d'objets se fait à l'aide d'un algorithme d'apprentissage par renforcement profond. Les états sont les images des caméras et les coordonnées articulaires du système de caméras. Les actions possibles sont les variations de coordonnées articulaires du système de caméras. Notre contribution pour cette tâche se situe dans le calcul du signal de récompense qui ne repose que sur très peu d'aprioris.



C - Schéma résumant toutes les procédures d'apprentissage

Ce signal se calcule comme suit :

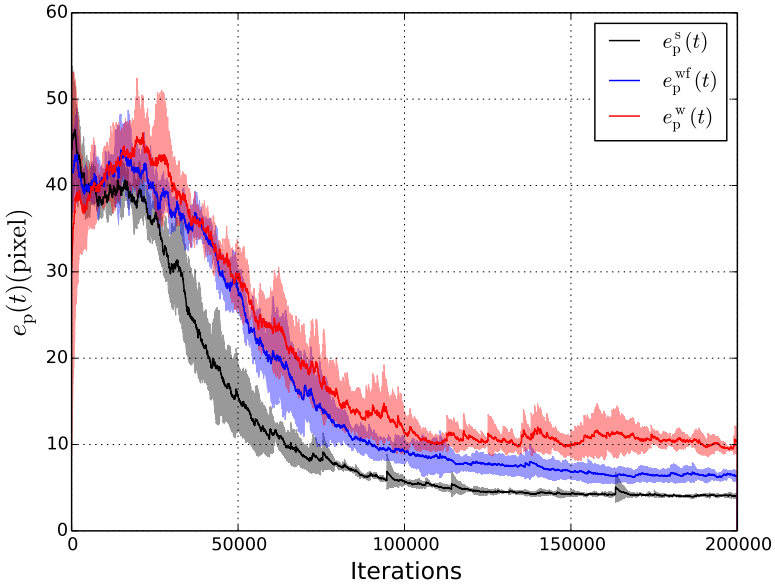
- On ne place pas d'objets dans l'environnement et un auto-encodeur est utilisé pour encoder les images de l'environnement.
- Lorsque l'on ajoute un objet dans l'environnement, la portion de l'image correspondante à l'objet est mal reconstruite. Cette propriété est exploitée pour localiser l'objet dans l'image. Ainsi, nous pouvons obtenir la distance Euclidienne entre le centre de l'image et la position de l'objet.
- Une fois cette distance obtenue, le signal de récompense se calcule comme une fonction affine décroissante de cette dernière.

La localisation de l'objet est entachée d'un bruit impulsionnel du fait de l'imperfection des auto-encodeurs appris : ceux-ci ne reconstruisent pas optimalement les zones de haute fréquence comme les pieds de table. Ainsi, de temps en temps, ce genre de zone est détectée à la place de l'objet ce qui produit un bruit impulsionnel sur le signal de récompense.

Nous établissons une approche sans aprioris pour limiter l'influence de ce bruit dans l'apprentissage. Cette méthode consiste à apprendre la variation observée des positions d'objets dans l'image comme une fonction de la norme du vecteur d'actions (variations des coordonnées articulaires du système de caméras) appliqué antérieurement. Ainsi, à chaque action appliquée, nous prédisons une variation de position d'objets dans l'image. Si la variation observée est très différente de la variation prédite, cette transition <état, action, état, récompense> n'est pas prise en compte dans l'apprentissage.

Pour le côté expérimental, nous montrons dans un domaine synthétique que l'apprentissage de la fixation avec notre signal de récompense filtré (en bleu sur la figure D) produit un apprentissage de meilleur qualité que celui non filtré (en rouge

sur la figure D). De plus, le signal filtré approche les performances d'un signal de récompense supervisé (calcul en tout point similaire mis à part la localisation de l'objet qui est réalisée par projection de la position 3D dans l'image, en noir sur la figure D).



D - Courbes d'apprentissage de la fixation d'objet pour le domaine simulé. $e_{\mathrm{p}}(t)$ représente au cours du temps la distance dans l'image entre la projection du centre de gravité de l'objet et le centre de l'image.

Dans le domaine réel, nous montrons que notre méthode de calcul de récompense filtré produit de bons résultats, comme le montre la figure E :

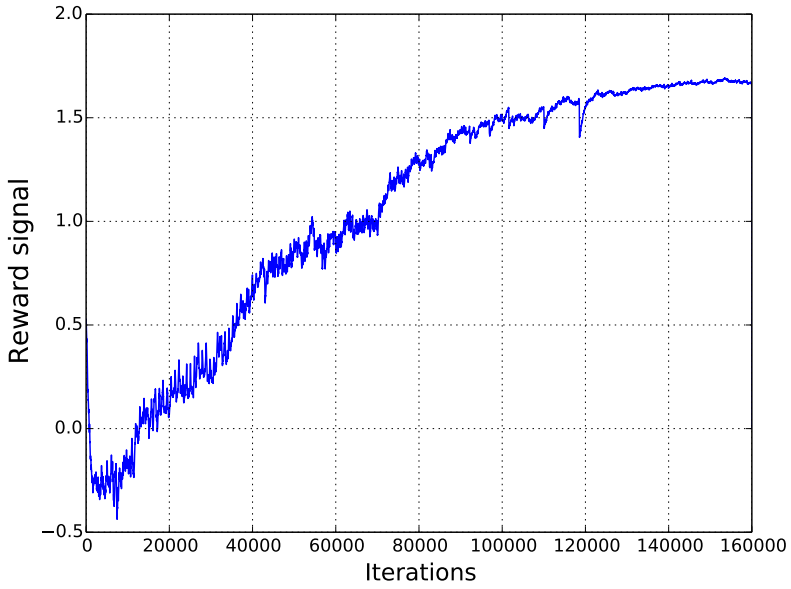
Apprentissage simultané de la fixation d'effecteur et de la fonction de coordination main-oeil

L'objectif réel de cet apprentissage simultané est l'apprentissage de la fonction de coordination main-oeil f telle que : $\boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}} = f_{\eta}(\boldsymbol{q}^{\mathbf{robot}})$. $\boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}}$ représente les coordonnées articulaires du système de caméras qui font que celui-ci fixe l'effecteur, $\boldsymbol{q}^{\mathbf{robot}}$ est le vecteur des coordonnées articulaires du bras robotique, η est le vecteur de paramètres du réseau de neurones représentant f.

Pour apprendre cette fonction sans supervision, nous utilisons une technique d'apprentissage supervisé avec des couples entrée-sortie obtenues de façon faiblement supervisée. Pour obtenir ces données $(\boldsymbol{q}^{\mathbf{robot}}, \boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}})$, une politique de fixation d'effecteur est apprise.

Cet apprentissage est effectué de la même façon que pour la fixation d'objet. Les mêmes espaces d'état et d'action sont utilisés. En revanche, le signal de récompense est différent et est calculé de la façon suivante :

- A chaque itération, le robot bouge un doigt, et la différence d'images avant et après le mouvement est obenue.



E - Courbe d'apprentissage de la fixation d'objet pour le domaine réel.
Le signal de récompense est représenté en ordonnée.

- De cette différence d'images, l'effecteur est localisé. Ainsi, on calcule une distance entre l'effecteur et le centre de l'image.
- De la même façon que pour la fixation d'objets, la récompense est calculée comme étant une fonction affine décroissante de la distance entre l'effecteur et le centre de l'image.

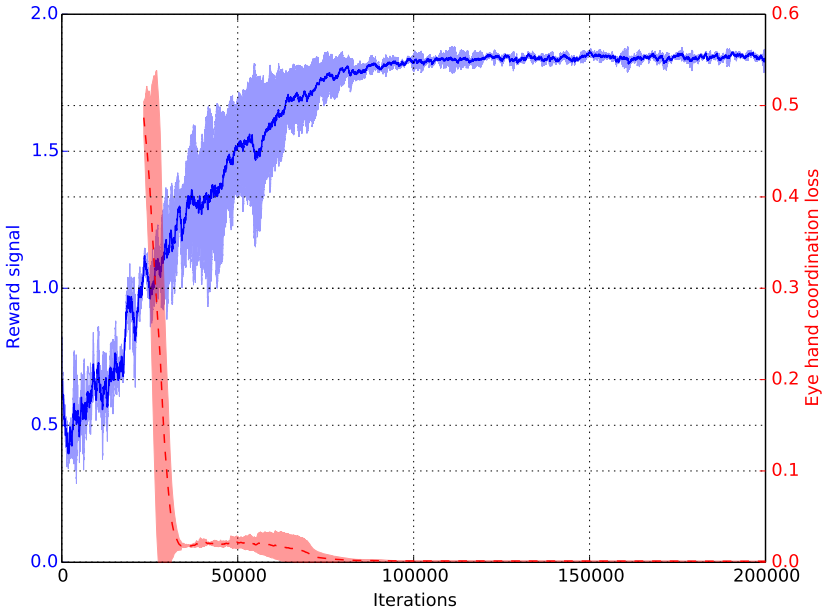
Les données $(q^{\mathbf{robot}}, q^{\mathbf{camera}}_{\mathbf{virt}})$ sont ajoutées à la base de données d'entraînement de f lorsque la récompense dépasse une valeur de seuil.

Ces apprentissages sont effectués dans le domaine simulé. La figure F représente l'entraînement parallèle de la politique de fixation d'effecteur et de la fonction de la coordination main-oeil. Nous observons la convergence du signal de récompense et du coût de la fonction de coordination main-oeil.

Apprentissage de comportements liés à l'atteinte d'objets

La tâche d'atteinte d'objets consiste à atteindre l'objet avec la paume de l'effecteur. Pour notre étude, nous considérons un seul objet. Lors de l'apprentissage de l'atteinte d'objets, le robot fixe l'objet au préalable avec la politique apprise lors de la première étape. Le vecteur d'état est composé des coordonnées articulaires du système de caméras et du bras robotique. Nous nous passons d'images parce que notre étude considère un seul objet et que les coordonnées articulaires du système de caméras localisent suffisamment bien la position de l'objet dans l'espace. Les actions sont les variations de coordonnées articulaires du bras robotique.

La fonction de récompense utilisée est une somme de trois termes :



F - Courbes représentation l'apprentissage de la fonction de coordination main-oeil

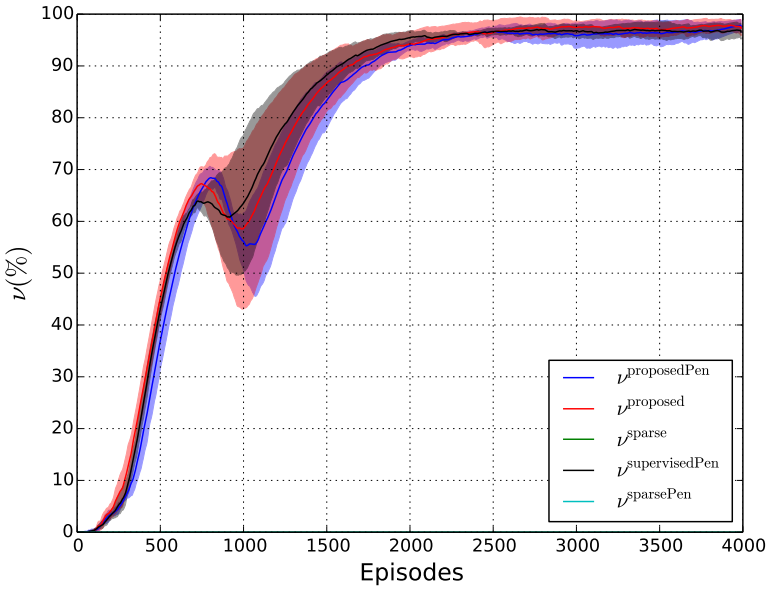
- un signal épars pour définir l'objectif de la tâche (1 lorsque l'objet est atteint et une constante négative lorsqu'il ne l'est pas.
- un signal informatif, fonction affine décroissante de la distance entre les coordonnées articulaires du système de caméras et des coordonnées articulaires "virtuelles" du système de caméras (qui feraient en sorte que les caméras fixent l'effecteur). Ce signal fait en sorte que le bras apprenne à diriger l'effecteur vers l'objet.
- un signal de pénalité négatif quand le robot touche la table.

Nous comparons cette fonction de récompense ($r_{\text{proposedPen}}$) avec 4 autres récompenses :

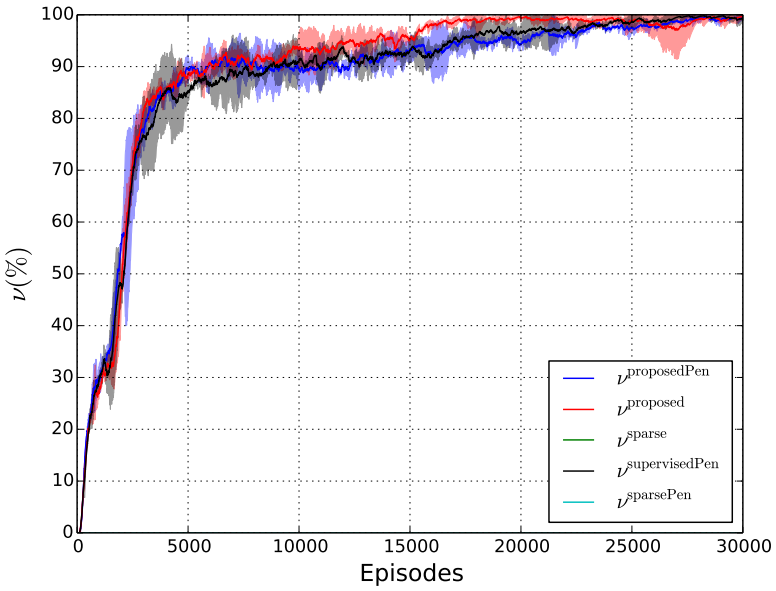
- Un signal épars seul (r_{sparse})
- Un signal épar muni du signal de pénalité négatif ($r_{\text{sparsePen}}$)
- Un signal épars muni du signal de récompense informatif précédemment mentionné (r_{proposed})
- Un signal épars muni d'un signal de récompense informatif dépendant d'un modèle géométrique direct et d'une connaissance parfaite de la position 3D de l'objet et du signal de pénalité négatif ($r_{\text{supervisedPen}}$)

Tous ces signaux ont été testés dans des configurations initiales particulières :

1. Une position initiale d'objet et une position initiale de bras (figure G)
2. Plusieurs positions initiales d'objet et une position initiale de bras (figure H)
3. Une position initiale d'objet et plusieurs positions initiales de bras (figure I)
4. Plusieurs positions initiales de bras et d'objet (figure J)

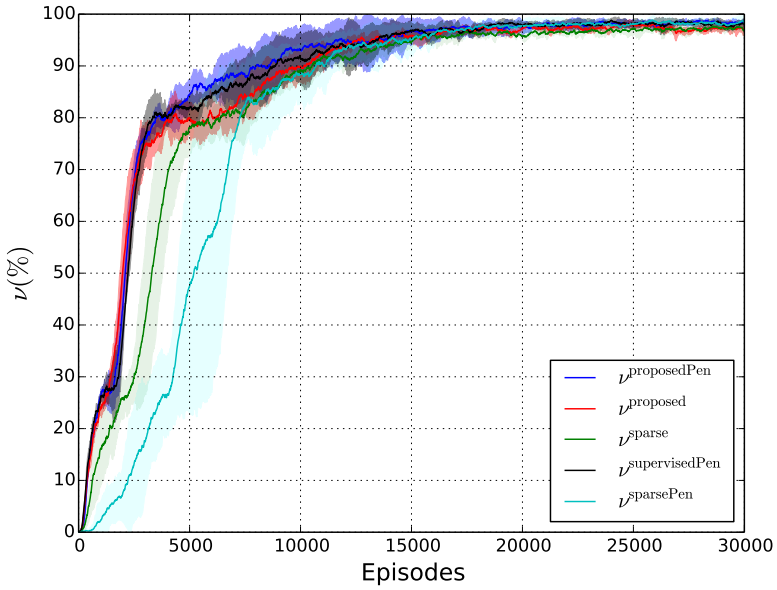


G - Courbes représentant la fréquence d'atteinte en fonction des épisodes pour diverses fonctions de récompense pour la configuration initiale 1

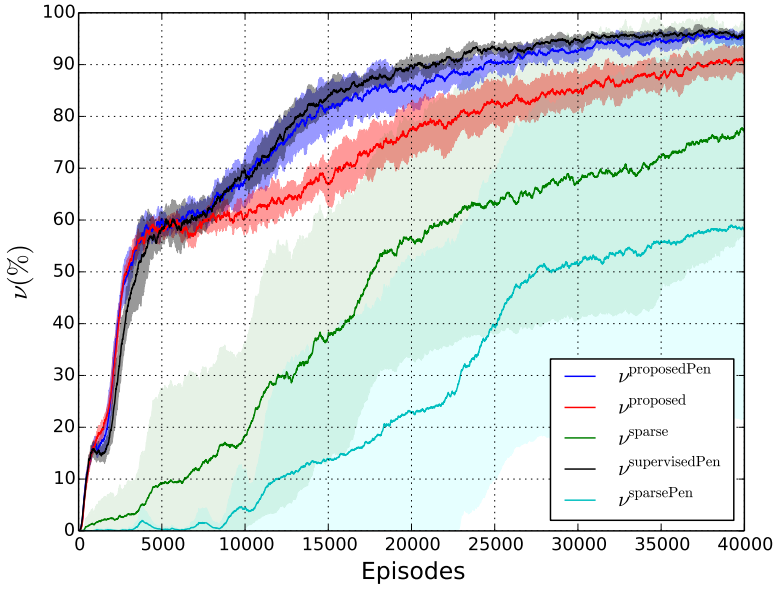


H - Courbes représentant la fréquence d'atteinte en fonction des épisodes pour diverses fonctions de récompense pour la configuration initiale 2

De ces résultats, nous tirons les conclusions suivantes. Si nous utilisons des récompenses éparses sans récompenses informatives, l'apprentissage ne fonctionne pas quand seulement une position initiale de bras est générée à chaque début d'épisode. En effet, la probabilité que la paume de l'effecteur touche l'objet est faible. En



I - Courbes représentant la fréquence d'atteinte en fonction des épisodes pour diverses fonctions de récompense pour la configuration initiale 3



J - Courbes représentant la fréquence d'atteinte en fonction des épisodes pour diverses fonctions de récompense pour la configuration initiale 4

revanche, lorsque nous utilisons plusieurs positions initiales de bras, nous augmentons la probabilité de toucher l'objet. Quand la position de l'objet est fixe, nous obtenons des performances très bonnes. Cependant, lorsqu'elle ne l'est pas, l'apprentissage n'est pas fiable du tout.

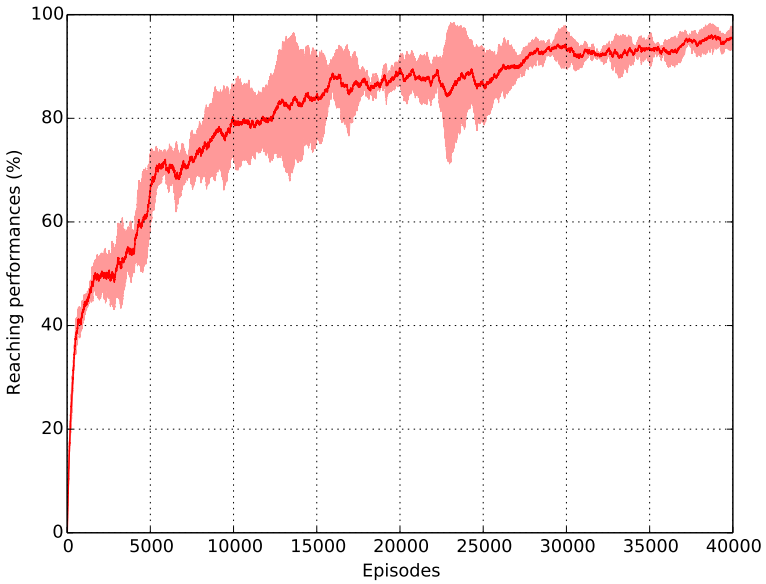
Nous observons par ailleurs que pour toutes les configurations d'intialisation d'épisode, utiliser des termes informatifs accélère l'apprentissage.

De plus, nous observons que pour les configurations où plusieurs positions intiales de bras peuvent être générées, le terme pénalisant les contacts entre l'effecteur et la table accélère l'apprentissage. En effet, il permet de guider plus efficacement le robot vers des positions d'atteinte. Nous observons cet effet particulièrement lorsque l'on utilise la récompense informative faiblement supervisée. Si nous n'utilisons pas ce terme, le terme de pénalité peut faire en sorte que l'apprentissage considère la table comme un répulsif, ce qui ralentit l'apprentissage.

Enfin, la récompense informative proposée permet d'atteindre des performances proches de celles obtenues avec la récompense informative supervisée.

La prédicteur d'atteignabilité $R = Re_\alpha(\boldsymbol{q}^{\mathbf{camera}})$ est appris en utilisant de l'apprentissage supervisé avec des données générées par l'apprentissage de l'atteinte d'objets. En effet, à chaque fin épisode (lorsque la politique d'atteinte produit déjà des bonnes performances), le couple $(\boldsymbol{q}^{\mathbf{camera}}, R), R \in \{0,1\}$ est ajouté à la base de données d'apprentissage de Re. R indique si le robot a réussi à atteindre l'objet.

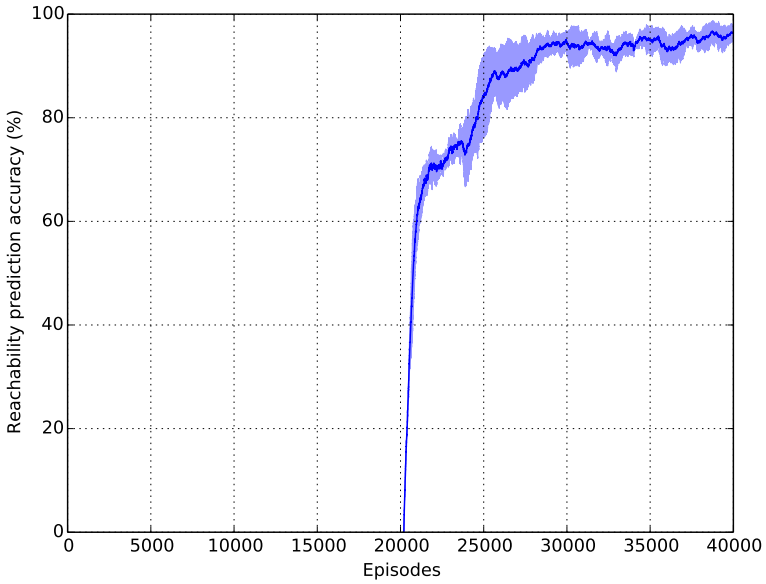
Pour apprendre l'atteignabilité, nous utilisons la configuration initiale 2, i.e. une seule position initiale de bras, plusieurs positions initiales d'objet. Deux mesures sont relevées au fil des épisodes. La première (figure K) évalue les performances de la politique d'atteinte. Pour cela, nous relevons si oui ou non, la politique d'atteinte est en accord avec la prédiction théorique d'atteignabilité (si l'objet est atteignable, le robot doit l'atteindre pour obtenir un succès). Nous observons que le robot converge vers des performances plutôt bonnes.



K - Courbe représentant l'évolution de la performance de la politique d'atteinte au fil des épisodes

La deuxième (figure L) évalue à quel point le prédicteur d'atteignabilité se rapproche de l'estimation théorique de l'atteignabilité. Nous observons également que le prédicteur atteint des performances plutôt bonnes.

De plus, nous affichons en figure M une évaluation de l'atteignabilité en fonction de la position de l'objet sur la table.



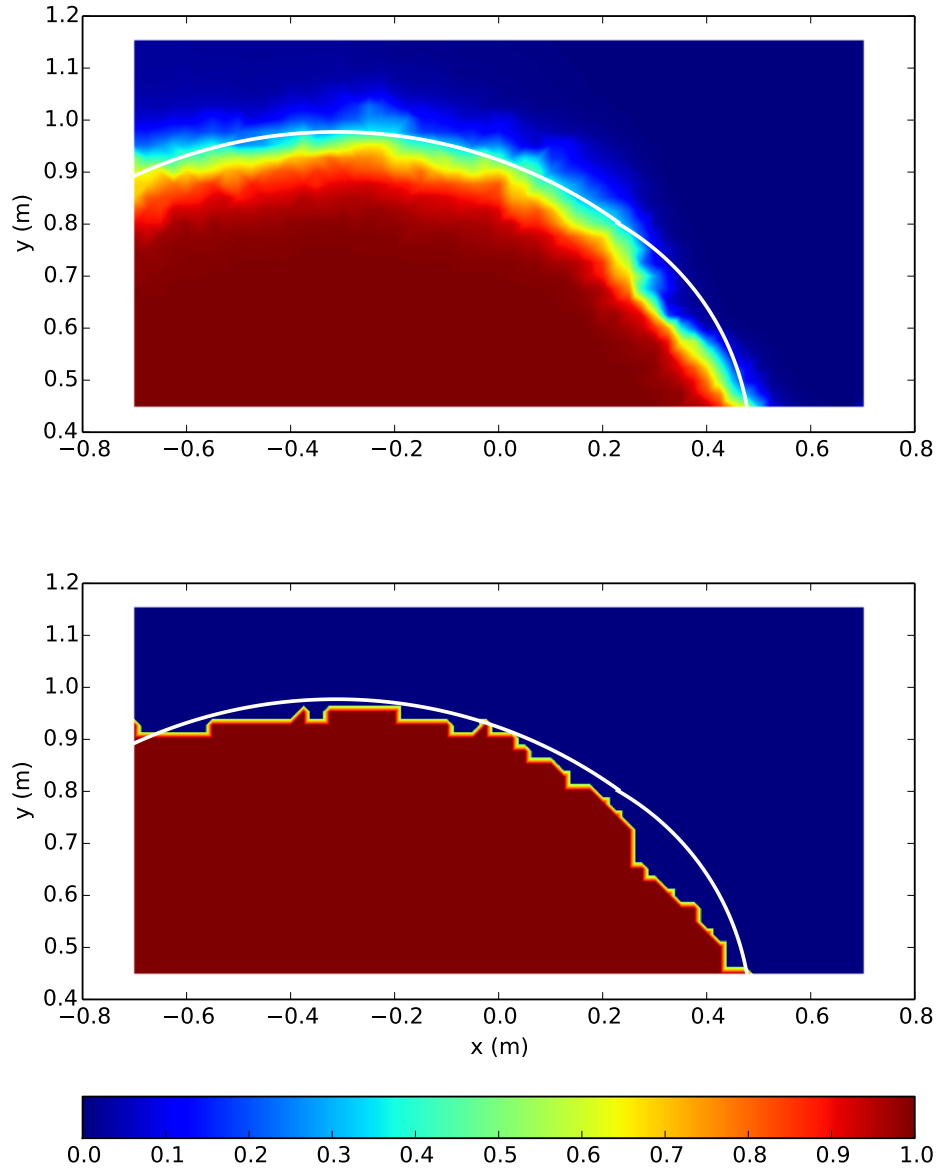
L - Courbe représentant l'évolution de la précision du prédicteur d'atteignabilité

Pour chaque position sur la table, le robot fixe l'objet, et le prédicteur d'atteignabilité sort une valeur entre 0 et 1. Nous observons que la prédiction est très bien apprise pour des positions loin de la frontière d'atteignabilité. En revanche, nous observons des imprécisions près de la frontière. Pour la plupart de ces erreurs, elles peuvent être imputées à une fixation d'objets qui ne permet pas de localiser avec une très grande précision l'objet. En revanche, une des zones (près du bout de droite de la courbe blanche) présente des erreurs plus importantes. Cela s'explique par le fait que le robot ne parvient pas à atteindre de manière régulière l'objet situé dans cette zone. Entraîner la fonction de coordination main-oeil dans cette zone devrait aider à réduire ces erreurs.

Conclusion

Les contributions majeures des travaux présentés sont liées à la faible supervision requise pour l'apprentissage des comportements robotiques. Pour les apprentissages de la fixation d'objet, de la fixation d'effecteur et de la fonction de coordination main-oeil, aucun paramètre de calibration des caméras, ni de traitement d'image supervisé n'est requis. Pour l'apprentissage de la politique d'atteinte d'objet, et du prédicteur d'atteignabilité aucun modèle géométrique, ni traitement d'image supervisé n'est utilisé. Ainsi, l'entière procédure d'apprentissage ne recquiert que très peu de supervision par l'être humain.

Ces travaux comportent quelques limites qui proviennent du peu d'aprioris utilisés dans notre méthode. En effet, pour la fixation d'objet, la limite vient du fait que nous supposons qu'il n'y a pas d'objet lors de l'encodage de l'environnement. Ainsi, si l'environnement change, la fonction de récompense risque de ne plus être valide. Par conséquent, les auto-encodeurs doivent être entraînés de nouveau sur le nouvel environnement, ce qui peut être fastidieux si l'environnement varie souvent.



M - Évaluation du prédicteur d'atteignabilité. Pour les deux figures, la courbe blanche représente la fontière d'atteignabilité. La figure du haut représente la sortie du prédicteur (entre 0 et 1) et celle du bas représente la prédiction binarisée (0 or 1).

De plus, pour l'apprentissage de la fixation d'effecteur, il est supposé que rien ne bouge mis à part l'effecteur. La fonction de récompense peut ne pas être valide si l'environnement varie énormément.

Les perspectives qu'offrent ces travaux concernent d'une part le traitement des limites de notre approche. D'autre part, ces travaux peuvent également être plus largement expérimentés sur une plateforme robotique réelle. Enfin, la tâche d'atteinte d'objets peut être étendue à divers types d'objets.

Chapter 1

Introduction

1.1 Motivations

In industrialized countries, robotics has deeply revolutionized our daily life in several aspects. It has led to increased industrial production while reducing the drudgery of some automatic tasks. It has also brightly helped many sectors such as the field of agriculture, people assistance (domestic robots), complex low-invasive surgical operations (medical robots), exploration of the seabed (underwater robots), and maintenance in nuclear plants.

Some of these tasks still present interesting challenges according to several aspects. First, in open or/and inaccessible or/and complex environments (e.g. robots exploring the seabed), designing models that perfectly describe dynamics of robots and environments through time is very challenging. Thus, with the use of classical control laws, it is difficult for the robot to reliably achieve a task if unexpected changes occur. Second, robotic tasks generally rely on a perfect model of robot geometry. If the latter changes, the robot has to take it into account. Third, many tasks require to take into account high-dimensional input spaces, e.g. learning to grasp from visual inputs, which is challenging. To solve these tasks, it is interesting to take inspiration from humans or animals since (1) they are able to adapt to most environment variations, (2) they take into account high-dimensional inputs (vision, proprioception) and (3) nobody provides the animals with models (about geometry or dynamics) since they are learning them. We particularly focus on the latter aspect in the sense providing robots with models can be particularly damaging when they become wrong.

We illustrate these thoughts by two examples which show how designing models can affect the behaviour of robots. The first is the one of a so-called autonomous car, which has to learn to drive on a lane while avoiding obstacles such as pedestrians or other cars. Let us assume that programmers made the car detect these obstacles in good weather conditions, i.e. without rain or snow. If it rains, the robot may not detect well the obstacles and avoid them, because the assumption of good weather conditions was wrong. For a more technical example, assume that a humanoid robot provided with a pair of cameras has to sort blue objects (there are two forms of objects) in a white room using inverse kinematics, calibration parameters, a blue-color-segmentation algorithm, and an algorithm outputting the shape of the object. Several changes to the robot model or the environment can affect the performance of the robot. First, inverse kinematics can simply change if the geometry of the robot varies because of damages, or if joint encoders are deficient. Thus, the robot would not be able to reach the objects. Second, if objects with different shapes and colors appear in the scene, the robot will not be able to localize the object and to manipulate it (see Figure 1.1 for an illustration). These two examples show first that it is not easy

to build models which are valid for any robot and environment variations. Second, if humans provide robots with models of kinematics, dynamics or use restricted assumptions, the robots cannot reliably achieve a task when these models become wrong. Therefore, we want to work on this specific issue by making the robot as much as possible independent on models designed by humans.

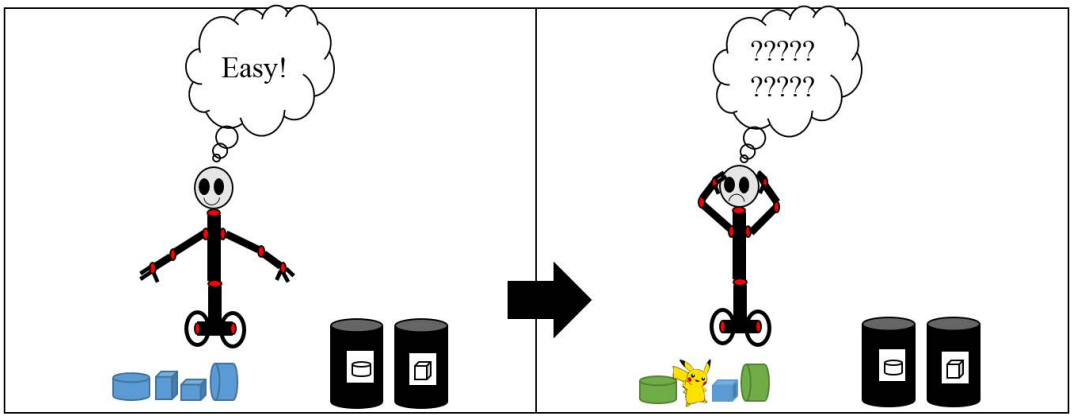


FIGURE 1.1: Scheme representing the issue regarding the use of restrictive assumptions. The robot only learned to sort some specific objects and when new objects appear, the robot may fail to achieve its task.

1.2 Objective

The objective of the thesis is to show how a complex robotic task can be learned with minimal supervision (as explained above, a complex task generally involves high-dimensional input spaces or/and complex environments). More precisely, we want to investigate the following question.

Can a robot learn a complex manipulation task without:

- priors on calibration parameters,
- priors on kinematics,
- priors on dynamics,
- knowledge about environment images (hand-crafted visual features),
- expert demonstrations?

1.3 Presentation of the robotic tasks

To address this question, we consider learning manipulation robotics tasks. This field is interesting since manipulation skills are necessary for repairing machines in dangerous environments (nuclear plants) and for assistance or exploration robots. Specifically, we focus on two reaching skills:

- A "palm-touching task": a robotic manipulator has to touch an object with its end-effector palm. This task can be considered and used as a pre-grasping task. Indeed, once the object is touched with the palm, the resulting arm posture is close to be a

grasping posture (we do not learn to grasp for the sake of simulation simplicity). As grasping skills are important in numerous manipulation tasks (clearing table or materials handling), our task is thus interesting from this point of view. Besides, this task is complex because on the one hand we only use raw sensory inputs such as image pixels, camera joint angles, arm joint angles and tactile sensors as states. This makes the problem high-dimensional. On the other hand, the initial configurations of the reaching task are both numerous and diverse and some of them imply not obvious solutions (the robot has to substantially modify its orientation to touch the object).

- A reachability prediction skill: when fixating an object, from the direction of its gaze, the robot has to predict whether the object is reachable. This skill is interesting in that it can be directly used in the situation where the manipulator has a mobile base and cannot directly reach the object without moving. In such a case, the robot would just have to move its mobile base such that it can reach the object.

We want to achieve these tasks with very few priors about the robot model and the environment geometry and without expert demonstrations. Deep reinforcement learning methods have taken important steps towards this objective. Indeed, in order to learn a required behaviour (or a policy) for a given task, reinforcement learning methods dispense with the use of modelling dynamics. The knowledge about the interaction between the agent and the environment comes from the exploration of the latter by the agent. Besides, the emergence of deep learning has offered good solutions to dispense with the use of hand-crafted features for the state representation. Consequently, many deep reinforcement learning algorithms have been proposed and applied to high-dimensional robotics tasks [1], [2].

However, in order to learn a complex manipulation robotics task with deep reinforcement learning, robots are often provided with a positive reward given only upon full completion of the task. E.g. to learn that a grasping move is good, the robot has to execute this correct grasping move and then it is rewarded. This correct move is found through exploration of the environment. In a high-dimensional state space, this can take a very long time.

To alleviate this issue, many methods use strong assumptions to make this kind of learning tractable, e.g. hand-crafted features, expert demonstrations, forward kinematics (generally used to compute shaping rewards), knowledge of some goal states. In contrast, we try to make the robot learn the chosen manipulation tasks without these kinds of prior knowledge.

1.4 Our approach

To limit the amount of requirements in robotic learning, one promising approach is to take inspiration from human or animal behaviours because they are not provided with kinematics or hand-crafted features, they are learning them. Consequently, to learn the palm-reaching task, we take inspiration from the human behaviour. Generally, to grasp an object, the humans first look at it and then grasp it. Based on this principle, we choose to decompose the whole palm-reaching task into three tasks (see Figure 1.2). First, the robot learns how to fixate objects. Second, the robot jointly learns how to fixate its end-effector and a hand-eye coordination function. Third, based on the knowledge acquired in the two previous tasks, the robot learns to touch objects. The three tasks are learned one after the other. The rationale is that the first task makes the robot implicitly locate the object in the 3D scene from

raw pixels. The second one allows the robot to implicitly locate the end-effector from robot joint angles. And in the third one, the robot uses a shaping reward function based on the knowledge acquired during the two prior tasks which helps it to reach successful areas. The key point of our approach is that each task hardly requires human supervision.

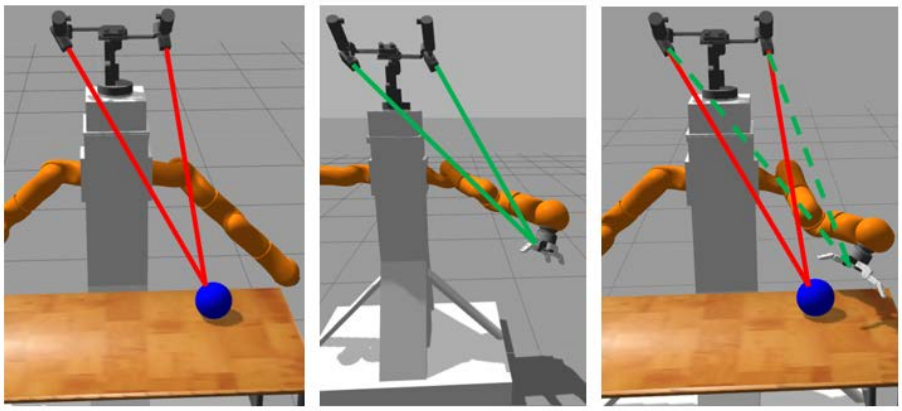


FIGURE 1.2: Scheme of the three steps of the stage-wise framework

1.5 Contributions

In the object fixation task [3], [4], two contributions are involved. First, we have designed a weakly-supervised shaping reward and second we have implemented an unsupervised denoising procedure. In the joint learning of the end-effector fixation and the hand-eye coordination, the framework built for the object fixation task is applied and produces good results, which is interesting in itself. For the reaching skills [5], [6], we present a new stage-wise deep reinforcement learning framework. In addition, the latter does not use forward/inverse kinematics, supervised visual modules, expert knowledge, dimension reduction or too costly materials and resources, which represents a step towards autonomy.

1.6 Report plan

The remainder of the thesis is as follows. Chapter 2 presents theoretical tools used in this thesis. It presents first a broad scope of machine learning and deep learning. Second, it presents ways to solve sequential decision making problems, and focuses on reinforcement learning. Finally, we present what is deep reinforcement learning and some of the algorithms suitable to learn complex robotics tasks. Chapter 3 mainly describes modern applications of deep reinforcement learning on manipulation robotics tasks. The stress is put on the way the goal is specified for each application, i.e whether it needs forward/inverse kinematics or supervised visual modules or state and action space simplifications or expensive materials. In chapter 4, we develop the idea to make the robot learn with little supervision. Then, three chapters (5, 6 and 7) follow, describing the contributions of our work. Chapter 8 summarizes the contributions and the limits of our work and provides guidelines for future work.

Chapter 2

Theoretical background

This chapter provides the theoretical background that will be required in the remainder. We present first a non exhaustive overview of machine learning with neural networks. Second, we focus on sequential decision making problems. Third, we present what is deep reinforcement learning and what algorithms can suit our requirements for learning manipulation robotic tasks.

2.1 Machine learning with neural networks

This section successively describes the general outline of machine learning, basic understanding on neural networks and recent deep learning progresses.

2.1.1 Machine learning

From a broad perspective, machine learning is a subfield of artificial intelligence in which the goal is to learn relations among data. It generally consists in learning data statistics or predicting data and can be applied to a wide range of tasks in which efficient algorithmic solutions are difficult or impossible to find such as classifying objects in images or predicting the best move in the game of Go.

2.1.1.1 Supervised learning

A supervised learning algorithm is provided with a dataset of S input-output pairs $(\mathbf{X_i}, \mathbf{Y_i})_{i\in\{1,...,S\}}$, where $(\mathbf{X_i})_{i\in\{1,...,S\}}$ are the data and $(\mathbf{Y_i})_{i\in\{1,...,S\}}$ the associated labels. Then, the algorithm tries to approximate a function M such that $\forall i \in \{1, ..., S\}, M(\mathbf{X_i}) = \mathbf{Y_i}$. The underlying objective of such an algorithm is to generalize to untrained data. The most known examples of supervised learning frameworks are classification [7] (countable outputs) and regression [8] (uncountable outputs).

2.1.1.2 Unsupervised learning

An unsupervised learning framework is provided with only data $\mathbf{X_i}$ and no labels. Then, the algorithm tries to learn the hidden structure behind data. It can be used for different purposes such as estimating a density function, grouping data into clusters (clustering), compressing data, and eventually detecting or localizing anomalies (as we will see in section 5.2.2).

2.1.1.3 Reinforcement learning

In a reinforcement learning framework, an agent learns from its interaction with the environment. The learning is driven by a reward signal and an exploration

method. The former is a real-valued number $R_i \in \mathbb{R}$ which rates a pair $(\mathbf{X_i}, \mathbf{Y_i})$ with $\mathbf{X_i} \in \mathcal{X}$ and $\mathbf{Y_i} \in \mathcal{Y}$ generally being states and actions. The algorithm picks up tuples $< \mathbf{X_i}, \mathbf{Y_i}, R_i, \mathbf{X_{i+1}} >$ called transitions through exploration. From them, it learns a sensori-motor mapping $\pi : \mathcal{X} \rightarrow \mathcal{Y}$ which makes the agent pick up high rewards in the future. Reinforcement learning objective is to deal with sequential decision making problems. Section 2.2 gives more details about this particular framework.

2.1.1.4 Generalization

In all the previously mentioned frameworks, the key problem is to learn something which generalizes from training data to all data. This problem is called "generalization" and can be addressed by different kinds of method. The first idea is to train with a large and diverse dataset. Indeed, the latter can more efficiently capture the variability of data. The second idea consists in decreasing the capacity of the model. For example, the algorithm can use less learnable parameters. Besides, regularization can be used to limit overfitting to training data. Most popular regularization methods are dropout [9] and L1 & L2 regularization [10]. Generalization is generally checked by comparing during training the performances of the framework on training, validation and test sets.

2.1.2 Artificial Neural networks

Several ways to approximate a function $M : X \rightarrow Y$ exist in the machine learning literature: Gaussian processes [11], Support vector machines [12], etc. We choose to use artificial neural networks since they have been proven efficient to approximate functions with high-dimensional inputs. The field of artificial neural networks dates back to the 40s when [13] tries to model the nervous activity by a neural network. Supervised learning [14], [15] and unsupervised learning [16], [17] problems as well as associative memories [18], [19] using simple neural networks were explored several years later (from 1969 to 1982). The reader can refer to [20] for additional references.

2.1.2.1 Neurons

Definition 2.1. *In the field of artificial neural networks, a neuron n is a computational unit which takes as input a vector $\boldsymbol{x} \in \mathbb{R}^N$ ($N \in \mathbb{N}$) and computes an output value z given a bias $b \in \mathbb{R}$, a vector of weights $\boldsymbol{w} \in \mathbb{R}^N$, and an activation function a.*

We note g the pre-activation value such that:

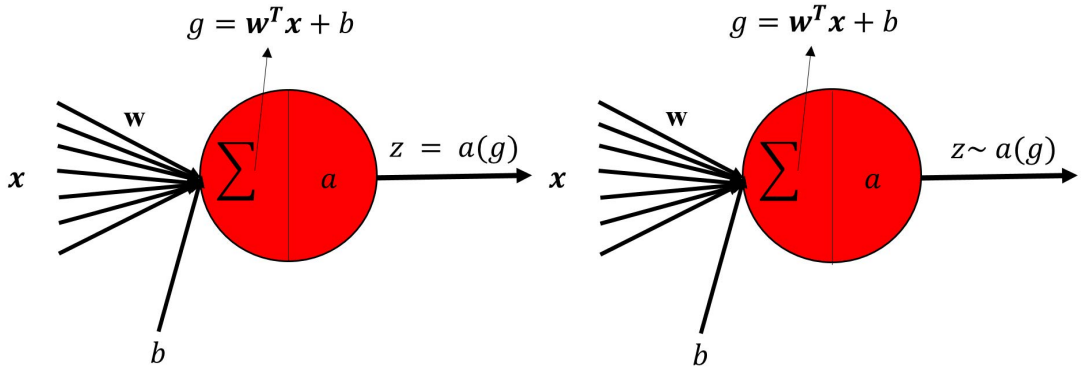
$$g = \boldsymbol{w}^T \boldsymbol{x} + b = \sum_{i=1}^{N} w_i x_i + b \tag{2.1}$$

Definition 2.2. *In the field of artificial neural networks, a deterministic neuron is a neuron whose output z is a function $a : \mathbb{R} \rightarrow \mathcal{E} \subset \mathbb{R}$ of $g : z = a(g)$.*

Definition 2.3. *In the field of artificial neural networks, a stochastic neuron is a neuron whose output z is sampled from the density function a depending on the pre-activation function g: $z \sim a(g)$.*

One widely used stochastic neuron has binary outputs $z \in \{0, 1\}$ and computes them using the sigmoid σ (see 2.1.2.6 for its definition) of the activation:

$z = \begin{cases} 1, & \text{if } r \sim U(0,1) > \sigma(g), \\ 0, & \text{otherwise,} \end{cases}$ where $\sim U(0, 1)$ means sampled from a uniform distribution of real numbers between 0 and 1.



(A) deterministic neuron (B) stochastic neuron

FIGURE 2.1: Scheme of neurons

2.1.2.2 Artificial neural networks

We define a general artificial neural network (ANN) as an undirected graphical model composed of neurons which are randomly connected through synapses. This allows self and bi-directional connections.

We can distinguish several neuron types according to their role in the network:

- Input neurons n^{I} which have the identity as the activation function and copy the information that the user provides.
- Output neurons n^{O} which provide the results of the prediction given inputs and hidden unit states.
- Hidden neurons n^{H} which process information based on hidden units states and/or inputs.

The output and hidden units can have arbitrary activation functions.

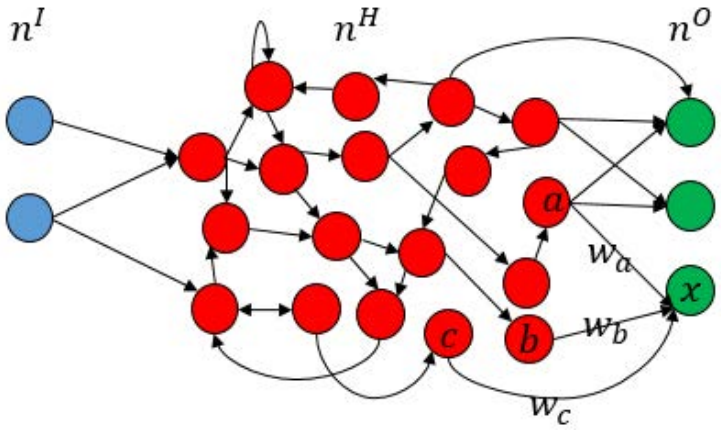


FIGURE 2.2: A random artificial neural network. To ensure a good visibility, the biases are not represented.

The pre-activation of hidden or output units is computed using equation (2.1). For example, in Figure 2.2, the pre-activation g_x of the neuron x fed by neurons a, b and c is $g_x = w_a z_a + w_b z_b + w_c z_c + b_x$. The bias is not represented in Figure 2.2 for

better visualization. Note that such a random artificial neural network is also called a recurrent network due to the presence of cycles in the neural network architecture. The presence of cycles notably allows to take into account past inputs for the output computation.

In the following paragraphs, we describe neural network structures which are used in the thesis. We willingly omit undirected graphical models such as Restricted Boltzmann machines [21], [22]. Furthermore, we do not describe recurrent neural networks since they are mainly used to address time-dependent problems that we do not consider in this thesis.

2.1.2.3 Feed-forward neural networks

Structure

A feed-forward neural network (FNN) is a directed graphical model in which neurons are organized in a layered structure. In such a structure, there is no cycle and no self-connection.

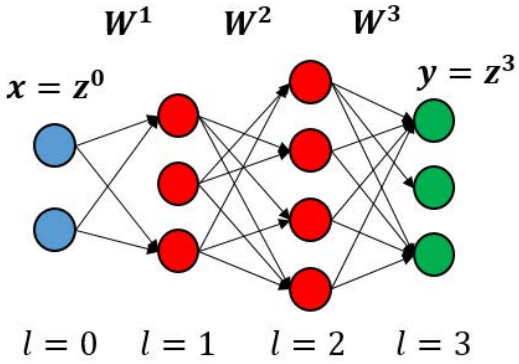
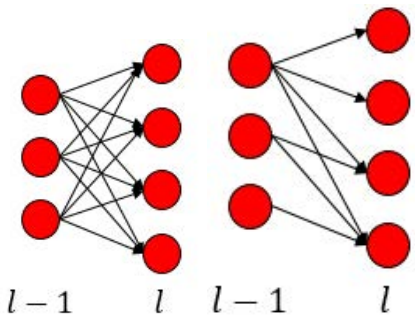


FIGURE 2.3: A 2-hidden layers FNN



(A) (B)

FIGURE 2.4: Fully (A) and partially (B) connected layers

Layers

There are several ways to compute unit pre-activations of a layer l given the outputs of the previous layer $l-1$. We present here some usual ways of doing it:

- Fully-connected layers

Definition 2.4. *A layer l is called "fully-connected" if and only if each unit of the layer l is connected to each unit of the layer $l-1$.*

The activations and outputs of fully-connected layer units are given by equations (2.2) and (2.3):

$$\forall i \in \{1, ..., N_l\}, g_i^l = \sum_{j=1}^{N_{l-1}} w_{ij}^l z_j^{l-1} + b_i^l, \tag{2.2}$$

$$\forall i \in \{1, ..., N_l\}, z_i^l = a_l(g_i^l), \tag{2.3}$$

with N_l being the number of units in the layer l, w_{ij}^l being the weight between the jth neuron of the layer $l-1$ and the ith neuron of the layer l, and b_i^l the bias of the ith neuron of the layer l.

- Partially-connected layers

Definition 2.5. *A layer l is called "partially-connected" if and only if it contains at least one unit which is not connected to at least one unit of the layer $l-1$.*

The computations of activations and outputs of partially-connected layer units can be made using equations (2.2) and (2.3). In (2.2), w_{ij}^l is replaced by 0 when there is no connection between the ith neuron of layer l and the jth neuron of the layer $l-1$.

- Convolutional layers
Let us describe first what is a feature map. This is a structure composed of neurons belonging to one specific layer l. Each neuron of a feature map is computed using the same weights (shared weights) on a specific area of feature maps belonging to the previous layer $l-1$. This area which corresponds to one neuron is called the receptive field of the neuron and the shared weights are represented by a kernel. A convolutional layer l is composed of F feature maps whose pre-activations are computed using F convolutional kernels (one per feature map) $\boldsymbol{W^{l,f}}(f \in \{1, ..., F\})$ based on the feature map outputs of the layer $l-1$.

Definition 2.6. *A layer l is called "convolutional" if and only if, for each feature map f of the layer l, the pre-activations are computed using a convolution product between a kernel $\boldsymbol{W^{l,f}}$ and the output units of the layer $l-1$: $\boldsymbol{g^{l,f}} = \boldsymbol{W^{l,f}} * \boldsymbol{z^{l-1}} + \boldsymbol{b^l}$.*

Practically, cross correlation is used instead of convolution in neural networks. However, these operators are equivalent in this context. Using convolutional layers is particularly interesting in fields in which there is a grid-like structure and the same processing is relevant in all the input space, e.g computer vision, sound analysis, natural language processing. Besides, it allows to significantly reduce the number of parameters to be optimized compared with a fully connected layer.

Figure 2.5 is a 1D example in which the same convolutional kernel can be relevant in all the input space.

Indeed, $2 \times (z_1^{l-1}, z_2^{l-1}, z_3^{l-1}) = (z_4^{l-1}, z_5^{l-1}, z_6^{l-1})$, and $W^{l,1}$ can easily describe the input space with 3 parameters (if the learning objective is to reconstruct the input

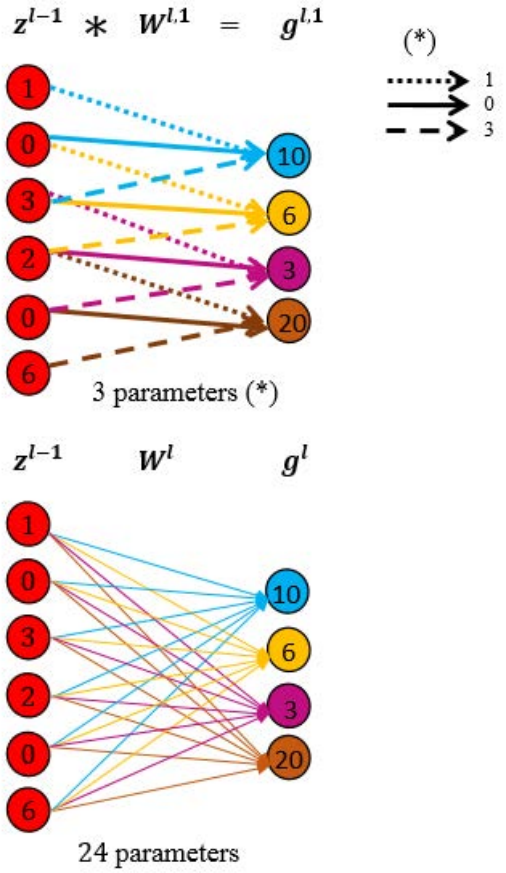


FIGURE 2.5: Convolutional layer (top) vs fully-connected layer (bottom). In the fully connected layers, all the weights can be different. In the convolutional layer, the weights represented by the same arrow style are equal.

for instance). Furthermore, for the same input-output pair, a convolution layer needs 8 times less parameters (in this example) compared with a fully-connected layer.

Forward pass

For a N-hidden-layer FNN, the output computation starts from the input layer $l = 0$ until the output layer $l = N + 1$ through all the hidden layers $l = \{1, ..., N\}$. The computation of the output given inputs is called a forward pass. The vectors $\boldsymbol{x} = \boldsymbol{z}^0$ and $\boldsymbol{y} = \boldsymbol{z}^{N+1}$ are the inputs and the outputs of the neural network. If we note $\boldsymbol{\theta}$ a vector regrouping all the learnable parameters $\boldsymbol{W}^l_{l\in\{1,...,N+1\}}$ and $\boldsymbol{b}^l_{l\in\{1,...,N+1\}}$, we note the forward pass of the neural network $N_{\boldsymbol{\theta}}$: $\boldsymbol{y} = N_{\boldsymbol{\theta}}(\boldsymbol{x})$.

2.1.2.4 Autoencoder

An autoencoder [23] is an artificial neural network trained to reconstruct its input (see Figure 2.6 for the representation of an example feed-forward autoencoder). The output of the autoencoder is written $\hat{\boldsymbol{x}}$, and thus the learning objective is that $\hat{\boldsymbol{x}} = N_{\boldsymbol{\theta}}(\boldsymbol{x})$. Since it does not require labels, it is an unsupervised learning framework. It is aimed at capturing the features of inputs. It can be notably used to compute

a compact feature vector and reduce the dimensionality of the input data [24] or for anomaly detection [25]. It is particularly relevant for data which have spatial coherence, e.g. images.

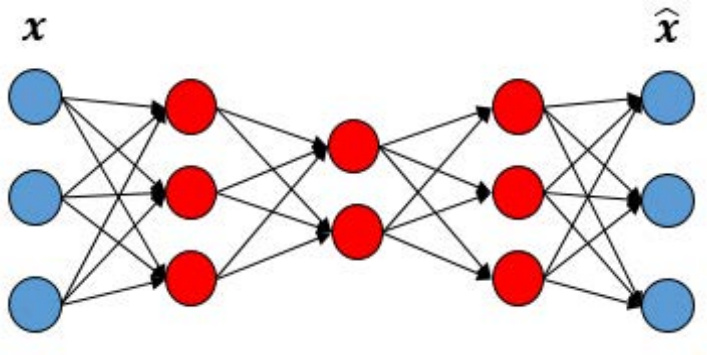


FIGURE 2.6: A feed-forward autoencoder

2.1.2.5 Learning FNN parameters

Learning the FNN parameters is a non-linear optimization problem and we describe here the most common way to learn these parameters in the literature. We give the equations only for the case of a FNN using only fully-connected layers (these equations can be easily expanded to the other layers).

Loss function

In a supervised setting, learning uses a database $\mathcal{D} = \{(\boldsymbol{x^1}, \boldsymbol{y_T^1}), ..., (\boldsymbol{x^S}, \boldsymbol{y_T^S})\}$ of S input-output pairs, $\boldsymbol{x}$ and $\boldsymbol{y_T}$ being the inputs and desired outputs provided to the algorithm. The learning goals are to adapt $\boldsymbol{\theta}$ such that $\boldsymbol{N_\theta(x)} = \boldsymbol{y_T}$ and also that the network generalizes to new test data, i.e. $\boldsymbol{N_\theta(x_{test})} = \boldsymbol{y_{test}}$. This goal is mathematically written as a loss function $L(\boldsymbol{y}, \boldsymbol{y_T}) = L(\boldsymbol{N_\theta(x)}, \boldsymbol{y_T})$. The latter should be differentiable with respect to $\boldsymbol{y}$ so that $\frac{\partial L(\boldsymbol{y}, \boldsymbol{y_T})}{\partial \boldsymbol{y}}$ can be computed. In our work, we consider the Euclidean loss:

$$L = \frac{1}{2S} \sum_{i=1}^{S} (\boldsymbol{y_T^i} - \boldsymbol{N_\theta(x^i)})^2. \tag{2.4}$$

Computing gradients

To optimize the parameters $\boldsymbol{\theta}$, we compute the gradient of the loss with respect to each parameter using the back-propagation algorithm [26], [27], [28], [29]. The latter uses equations (2.5), (2.6), (2.7), (2.8) from the layer $N+1$ to the layer 0 (see Figure 2.7).

$$\forall l \in \{1, ..., N+1\}, \forall j \in \{1, ..., N_{l-1}\}, \frac{\partial L}{\partial z_j^{l-1}} = \sum_{i=1}^{N_l} \left(\frac{\partial L}{\partial g_i^l} \frac{\partial g_i^l}{\partial z_j^{l-1}}\right) = \sum_{i=1}^{N_l} \left(\frac{\partial L}{\partial g_i^l} \times w_{ij}^l\right) \tag{2.5}$$

$$\forall l \in \{1, ..., N+1\}, \forall i \in \{1, ..., N_l\}, \frac{\partial L}{\partial g_i^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial g_i^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial a_l(g_i^l)}{\partial g_i^l} \tag{2.6}$$

$$\forall l \in \{1, ..., N+1\}, \forall i \in \{1, ..., N_l\}, \forall j \in \{1, ..., N_{l-1}\}, \frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial g_i^l} \frac{\partial g_i^l}{\partial w_{ij}^l} = \frac{\partial L}{\partial g_i^l} z_j^{l-1} \tag{2.7}$$

$$\forall l \in \{1, ..., N+1\}, \forall i \in \{1, ..., N_l\}, \frac{\partial L}{\partial b_i^l} = \frac{\partial L}{\partial g_i^l} \frac{\partial g_i^l}{\partial b_i^l} = \frac{\partial L}{\partial g_i^l} \tag{2.8}$$

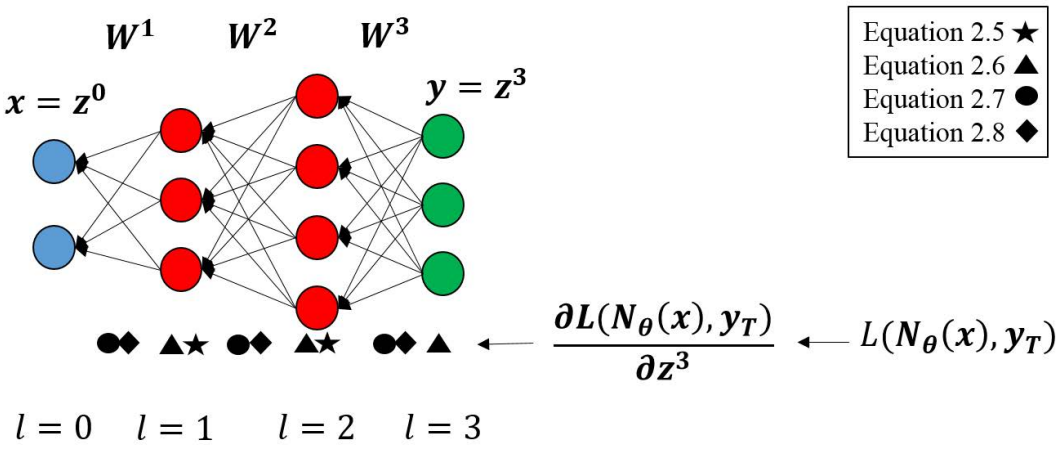


FIGURE 2.7: Back-propagation scheme

Updating parameters

Once the gradients of the loss with respect to the learnable parameters are computed, the learnable parameters can be updated using gradient descent [30] to minimize the loss function (and gradient ascent if the goal is to maximize the loss function):

$$\theta_t^j = \theta_{t-1}^j - \alpha \frac{\partial L}{\partial \theta^j}, \tag{2.9}$$

with α the learning rate, θ^j the jth component of θ and t denoting a time step. If the loss function was convex with respect to the parameters applying equation (2.9) would lead to the optimal solution. However, it is never the case in many problems. While equation (2.9) can offer decent results even in some non-convex problems, it is usually not the best way to update weights. To get a good sub-optimal solution, there are several update rules using gradients. We consider a given way to be a particular solver. (2.9) is the equation for gradient descent without momentum.

We review now several important features for updating neural networks and briefly describe the solver that we use in our work.

- Batch gradient descent and Stochastic gradient descent:
  Let us consider that the problem is to learn $M : \mathcal{X} \to \mathcal{Y}$ using the S-sized database $\mathcal{D} = \{(\boldsymbol{x^1}, \boldsymbol{y_T^1}), ..., (\boldsymbol{x^S}, \boldsymbol{y_T^S})\}$. We assume first that $\mathcal{D}$ represents all the data of the problem. Let L^i be the loss for the ith input-output pair.

Applying a batch gradient descent means updating the weight using $\frac{1}{S}\sum_{i=1}^{S}(\frac{\partial L^i}{\partial \theta^j})$, i.e. using the gradient averaged over all the dataset. In problems with a continuous input space, it is impossible to average the gradient on all the dataset.

This is why, stochastic gradient descent (SGD) is generally used. Applying a stochastic gradient descent solver means updating the weights based on an approximation of the gradient. In other terms, gradient updates happen on A samples with $A < S$: $\frac{1}{A}\sum_{i=1}^{A}(\frac{\partial L^i}{\partial \theta^j})$. The choice of the A value is based on a trade-off between computational cost and gradient variance. Indeed, the smaller is A, the larger is the variance of the gradient estimate and the smaller is the computational cost. In contrast, a higher A value decreases the variance of the gradient estimate and the computational cost.

- Using a momentum [29]:
  Using a momentum with SGD consists in keeping an exponentially weighted average m_t of the gradient and using it to update the gradient:

$$m_t^j = \beta m_{t-1}^j + (1-\beta)\frac{\partial L}{\partial \theta^j}, \tag{2.10}$$

$$\theta_t^j = \theta_{t-1}^j - \alpha m_t^j. \tag{2.11}$$

  Its role is to deal with the fact the loss functions are almost never strictly monotonous. Indeed, the loss function can reach a plateau and an update using only the gradients cannot make it escape from this flat area. To deal with this, the momentum uses the delay property of the exponential average filter and allows to still update parameters in the prior gradient direction even though the gradient is zero.

- ADAM [31]: ADAM keeps track over time of the gradient first moment estimate m_t^j and the gradient uncentered second moment estimate v_t^j:

$$m_t^j = \beta_1 m_{t-1}^j + (1-\beta_1)\frac{\partial L}{\partial \theta^j}, \tag{2.12}$$

$$v_t^j = \beta_2 v_{t-1}^j + (1-\beta_2)(\frac{\partial L}{\partial \theta^j})^2, \tag{2.13}$$

  with β_1 and β_2 being tunable hyper-parameters. After that, the parameters are updated using equation (2.14):

$$\theta_t^j = \theta_{t-1}^j - \alpha\frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}\frac{m_t^j}{\sqrt{v_t^j}+\epsilon}, \tag{2.14}$$

  with $\frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ being a factor correcting the biased estimates m_t and v_t, ϵ being a small-valued hyper-parameter, and α being an upper bound of the weight update for non-sparse gradients. The case of sparse gradients corresponds to the case where all the gradients are zero for several time steps $t-1...t-N$, and for a given iteration t it becomes non null.

2.1.2.6 Activation functions

The activation functions of neural units are of prime importance since they give the neural networks the ability to learn non linear functions. One property which

is required for the back-propagation algorithm equations (see equation (2.6)) is that they must be differentiable. Furthermore, when choosing an activation function, the values of the gradients $\frac{\partial a(g)}{\partial g}$ have to be taken into account, because they allow to stop or transfer updating information to inferior layers. E.g. a zero gradient stops the information processed by the neuron and no update information from this neuron is passed to the inferior layers. For example, we cannot use the heavyside function used in the Rosenblatt's perceptron [14]:

$$a(g) = \begin{cases} 1, & \text{if } g > 0 \\ 0, & \text{otherwise.} \end{cases}.$$

Indeed, the gradient does not exist in zero and otherwise is zero everywhere (the back-propagation algorithm was not formalized at that time).

In the following, we review the most usual activation functions among the deterministic ones and explain their use.

- Identity: $a(g) = g$. In this case, the output of the neuron is a linear combination of the outputs of the connected neurons. It is generally applied on output units in regression problems and on input units.
- Sigmoid: $a(g) = \frac{1}{1+e^{-g}}$. This is a non-linear function outputting values in the range $[0, 1]$. Using it allows to learn non linear functions. The drawback of such a function is that the gradient quickly tends to zero when the input tends to the infinite or minus the infinite. Thus, if the pre-activation of a sigmoid neuron is very high or very low, no gradient is back-propagated towards the first layers. This is called the "vanishing gradient problem".
- Tanh: $a(g) = \frac{2}{1+e^{-2g}} - 1$. This is a scaled and shifted version of the sigmoid activation with values in $[-1, 1]$.
- ReLU: $a(g) = max(0, g)$. This is the rectified linear unit [32]. As the sigmoid or tanh function, it is a non-linear function. An interesting fact about ReLU is that only few neurons are activated. Indeed, if the pre-activation value of a neuron is negative, its output is null. This makes the representation sparser and potentially leads to better generalization. The drawback is that for negative inputs, the gradient is zero. It means that no gradient is back-propagated and it potentially leads to dead neurons which are never activated in the learning process.
- Leaky ReLU [33]: $a(g) = \begin{cases} g, & \text{if } g \geq 0 \\ bg, & \text{otherwise.} \end{cases}$ This is an improved version of ReLU which addresses the problem of dead neurons. When the input is negative, the gradient is non-zero. b has a positive value and can be a learnable parameter in the parametrized version of the leaky ReLU activation.

2.1.3 Deep learning

This section describes some ideas behind deep learning and the deep structures that will be used in the thesis.

2.1.3.1 Definition

To our knowledge, there exists no consensual definition of deep learning in the literature. In practice, a neural network is generally considered as deep according to its depth i.e. the number of hidden layers. However, it is not clear from which

depth we can consider a network as deep. Moreover, this criterion does not necessarily imply a good function approximator. Indeed, we can take as an example a 11-hidden-layer FNN with one hidden unit per layer. This network is not a good function approximator. Several criteria could also be used to specify what is a deep network such as the potential to solve problems with high-dimensional input data or the average number of hidden units per layer. However, the threshold values are not easy to compute.

In this thesis, we consider that a neural network is deep if it has more than two hidden layers. We can add the adjective "complex" if the deep network was proven able to solve problems with high-dimensional input data. One interesting aspect of such a deep complex network is that a hierarchy of features is learned. Low-level features are learned in first layers and subsequent layers combine them to compute more abstract or higher-level features which are useful to optimize the loss function.

2.1.3.2 Why use deep structures?

An interesting question about deep structures is why they are chosen instead of shallow ones in various scientific fields since the latter are already universal approximators. Indeed, according to the universal approximation theorem, any continuous function on $\mathbb{R}^n$ can be approximated with any given precision by a single-hidden-layer FNN, provided enough hidden units are used [34], [35]. These results are easily expandable to deep networks since they contain universal approximators. The choice of choosing deep structures over shallow ones has theoretical reasons since many studies tend to show that shallow networks such as a single-hidden-layer FNN require exponentially more neurons than deep networks do [36], [37].

2.1.3.3 Deep convolutional neural networks (DCNN)

In this section, we describe the deep convolutional neural networks since it is relevant for our work. We make the choice of omitting other existing deep structures such as deep Boltzmann machines [38] and deep belief networks [24]. We did not use them because of implementation and computational reasons. Besides, we do not describe other popular recurrent neural networks such as long short term memory networks [39] and gated recurrent unit neural networks [40] since they are mainly used to address time-dependent problems that we do not consider in the thesis. For a more complete review of deep learning, the reader can refer to [41].

DCNNs are deep deterministic FNNs which contain at least one convolutional layer (see Figure 2.8). During training, a DCNN builds a hierarchy of features which are useful to optimize the objective. Note that, DCNNs have been generally composed of three kinds of layer. The first layers are convolutional (see 2.1.2.3) and are used to build a hierarchy of features, the last layers are fully-connected. The third kind of layer is a pooling layer and is generally placed after a convolutional layer. It reduces the dimensions of a feature map by estimating statistics on feature map areas. The average as well as the maximum of such an area can be used to build a reduced feature map and are called "mean-pooling" and "max-pooling". The main reason to use pooling is to limit over-fitting. Nowadays, it seems that more and more neural networks involve only convolutional and pooling layers.

Historically speaking, the first convolutional architecture is the neocognitron [42] which was used for unsupervised learning and already exhibited translational invariance properties. Then, in the 90s convolutional networks were trained with backpropagation from a teacher signal for handwritten zip code recognition [43],

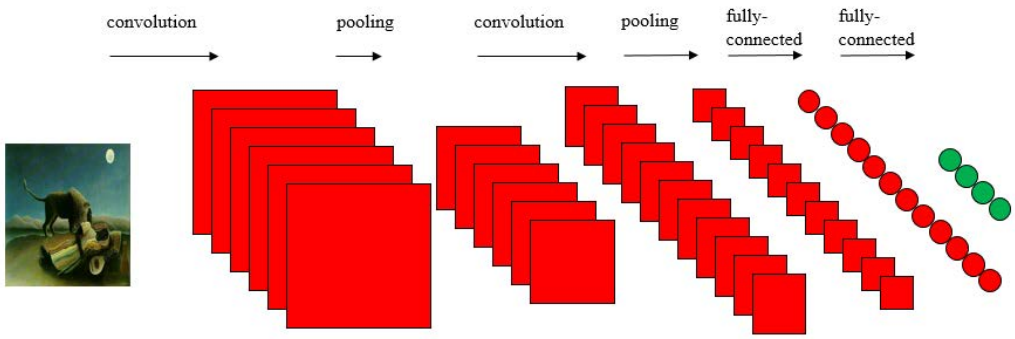


FIGURE 2.8: A convolutional neural network

[44], handwritten digit recognition [45] and fingerprint recognition [46]. Nevertheless, deep convolutional neural networks were not widely used. The main issues were pointed out by [47] and concerned the vanishing or exploding gradient problems in the back-propagation pass. One of the starting point of the successes of DCNNs is the success in the imageNet classification contest [48] with a GPU implementation. According to [20], the GPU implementation of deep convolutional learning structures is a key of their success. After that, DCNNs kept on winning contests and establishing benchmarks [49], [50] and started to be applied in many different fields such as robotics [1], sound classification [51] and natural language processing [52] (see [53] for more examples of DCNN applications).

2.1.4 Choice

We summarize here what we have chosen to use among the presented tools for this thesis.

First, we choose to use deterministic feed-forward neural networks (and convolutional when we deal with images) because they have been proved efficient to approximate high-dimensional functions with a simple and relatively fast learning procedure. Second, we mostly use leaky ReLUs for the activation functions of our neurons. Finally, in order to update the parameters using computed gradients, we use the ADAM solver. The values of the associated parameters can be found in Appendix A.

2.2 Sequential decision making problems

In this section, we focus on sequential decision making problems. The latter concern tasks that an agent has to achieved by making decisions at each time-step (sequentially). A simple example illustrating it is the game of chess.

In the following, we first discuss sequential decision making problems and traditional methods to solve them. Then, reinforcement learning basic knowledge is described as well as relevant deep reinforcement learning algorithms. We will explain our algorithm choice with the main criteria being the ability to solve high-dimensional problems and the implementation simplicity.

2.2.1 Definitions and sequential decision making problems

2.2.1.1 Definitions

We define in this paragraph some common notions related to sequential decision making problems.

Definition 2.7. *An agent is an entity which learns to make decisions for given objectives while interacting with the environment and is equipped with sensors and actuators.*

For example, humans are agents which learn to walk at the beginning of their life and possess many sensors such as eyes, force sensors and neurally-based control actuators such as arm or leg muscles. A robotic manipulator is also an agent which can learn to reach objects and can possess cameras and angle coders as sensors and motors as actuators.

Definition 2.8. *An environment is an entity which can contain agents. It can be partially modified by the agent actuators.*

Practically, the environment can be any place such as rooms, fields or streets or anything with which the agent can interact except itself.

Definition 2.9. *An observation $\boldsymbol{o_t} \in \mathcal{O} \subset \mathbb{R}^d$ ($d \in \mathbb{N}^*$) is a vector which contains a description of the environment and the agent in the environment at time-step t.*

The observation generally contains sensor data e.g for a robotic manipulator raw images, joint robot angle values or tactile sensor values.

Definition 2.10. *A state $\boldsymbol{s_t} = (\boldsymbol{o_{t-n_o+1}}, ..., \boldsymbol{o_t}) \in S = \mathcal{O}^{n_o}$ is a sequence of n_o observations.*

Definition 2.11. *An action $\boldsymbol{a_t} \in \mathcal{A}$ is a vector containing command values for the agent actuators at time-step t. Applying these command values implies potential modifications of the state after time-step t.*

In a manipulation robotic context, actions are often torques, variations of joint angles or Cartesian coordinates or velocities.

Definition 2.12. *A policy π is a representation allowing agents to choose an action $\boldsymbol{a_t}$ at state $\boldsymbol{s_t}$. It can be deterministic i.e. in form of a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$ or stochastic i.e in a form of a density function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$.*

Definition 2.13. *A transition function or model or dynamics is a way to predict new observations given an action and the most recent past state. It can be deterministic i.e. in form of function $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ or stochastic i.e in form of a density function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$.*

For example, the linear equation for dynamics $\boldsymbol{s_{t+1}} = A\boldsymbol{s_t} + B\boldsymbol{a_t}$ is often used in robotics or in other automatic systems as a transition function (the state only consists of one observation in this case).

Definition 2.14. *The reward function R is a real-valued function which rates the quality of an action $\boldsymbol{a_t}$ in a given state $\boldsymbol{s_t}$ ($R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ or $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$).*

In a manipulation reaching task, it can be a decreasing function of the distance between the current pose and a target pose. In zero-sum games such as chess or go, it can be the outcome of the game (1 for a win and 0 for a loss).

Definition 2.15. *A "sparse" or "impulse" reward function has a reward signal value r_g in a small state area $\mathcal{S}_g$, called the goal area and a constant value r_c in the remaining state area such that $r_g \gg r_c$.*

$$R : \boldsymbol{s} \rightarrow R(\boldsymbol{s}) = \begin{cases} r_g, & \text{if } \boldsymbol{s} \in \mathcal{S}_g \subset \mathcal{S}, \\ r_c, & \text{otherwise.} \end{cases} \tag{2.15}$$

Typical examples of goal state areas are the last board position of a game in chess or successful reaching positions in robotic reaching tasks.

Definition 2.16. *Let $\boldsymbol{\delta}$ be a vector of features describing the state s. $\boldsymbol{\delta_s}$ describes the current state $\boldsymbol{s}$ and $\boldsymbol{\delta_{s^*}}$ a target state $\boldsymbol{s^*}$. A "shaping" reward function is a monotonously decreasing function g of a distance between target features $\boldsymbol{\delta_{s^*}}$ and current features $\boldsymbol{\delta_s}$.*

$$R : \boldsymbol{s} \rightarrow R(\boldsymbol{s}) = g(d(\boldsymbol{\delta_{s^*}}, \boldsymbol{\delta_s})) \tag{2.16}$$

Note that the two previous definitions are given for reward functions defined on S but can be extended to other input spaces.

Definition 2.17. *A transition T_t is a tuple $< \boldsymbol{s_t}, \boldsymbol{a_t}, r_{t+1}, \boldsymbol{s_{t+1}} >$ giving local spatial and temporal information on the interaction between the agent and the environment. At time t, the action $\boldsymbol{a_t}$ is taken from state $\boldsymbol{s_t}$ and produces a reward $r_{t+1} = R(s_t, a_t)$ and a new state $\boldsymbol{s_{t+1}}$ at time $t + 1$.*

Note that in the literature, the transition is also written as follows: $< \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}' >$

Definition 2.18. *An episode E is a bounded succession of transitions in time: $E = \{T_0, ..., T_{|E|-1}\}$*

The conditions of termination of an episode can be diverse. In this thesis, either the episode length reaches its maximum or the agent reaches an "absorbing" or "terminal" state. The latter is generally a goal state in the literature.

Definition 2.19. *A return G of an episode is the sum of all the rewards in an episode: $G = \sum_{t=1}^{|E|} r_t$.*

Definition 2.20. *A return-to-go $G_{\boldsymbol{s_t},\boldsymbol{a_t}}$ of an episode at transition T_t is the sum of all the rewards of an episode from the transition $< \boldsymbol{s_t}, \boldsymbol{a_t} >$: $G_{\boldsymbol{s_t},\boldsymbol{a_t}} = \sum_{i=t}^{|E|} r_i$*

In the two latter definitions, if we want the agent to take greater account of short-term future rewards, a discount factor $\gamma \in [0, 1]$ can be used as follows: $G = \sum_{t=1}^{|E|} \gamma^t r_t$ and $G_{\boldsymbol{s_t},\boldsymbol{a_t}} = \sum_{i=t}^{|E|} \gamma^i r_i$. This parameter, when different of 1, allows to take greater account of short-term future rewards. Otherwise, all the future reward values are considered with the same weight in the optimization.

2.2.1.2 Sequential decision making problems

The problem of sequential decision making consists in choosing the action $\boldsymbol{a_t}$ at time step t given any state $\boldsymbol{s_t}$ in an optimal way w.r.t to an objective.

An example of sequential decision making problem is the game of chess in which the agent has to make the move which ensures better winning chances from a given chessboard position (and the last 8 positions because of drawing-by-repetition rules). In a robotic reaching manipulation task, the problem can be to make the right arm joint angle move which makes the robot closer to a touching position without collision with the environment (see Figure 2.9 for the pictures). For this thesis, we focus on complex problems where the action and the state spaces are continuous and the state space is high-dimensional.

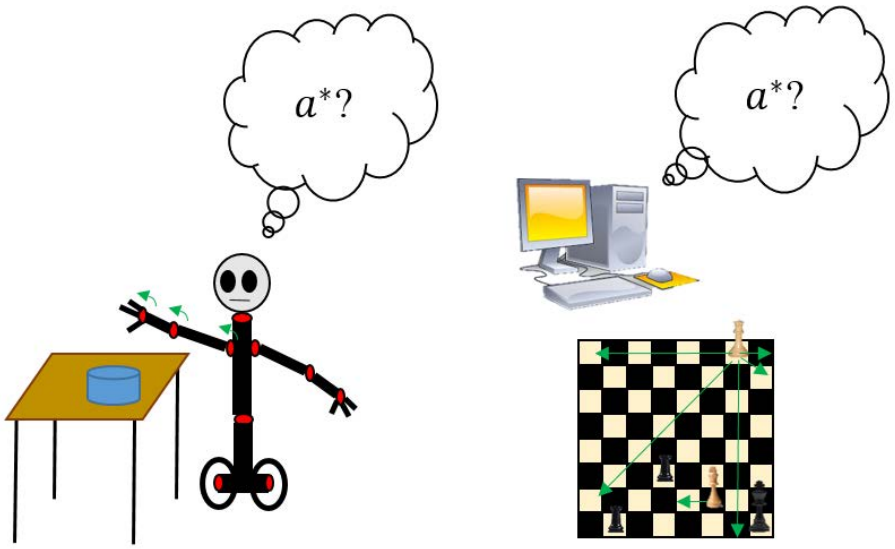


FIGURE 2.9: Example of sequential decision making problems

2.2.2 Solve sequential decision making problems

There are several ways [54] to solve sequential decision making problems (see Figure 2.10).

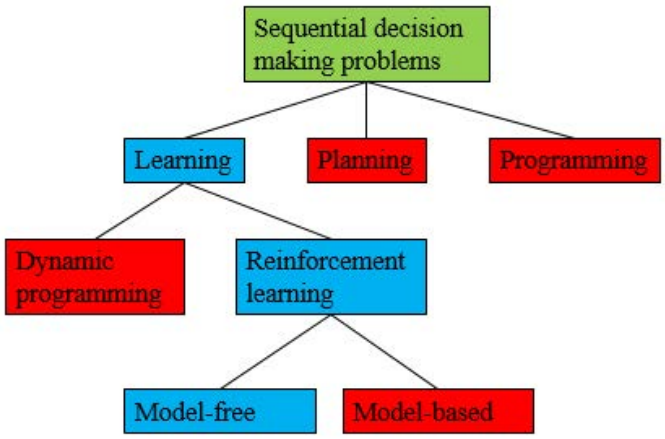


FIGURE 2.10: Solve sequential decision making problems. The blue boxes are the classes of methods used in our work.

2.2.2.1 Programming

This class of methods consists in hand-specifying the policy π. As a consequence, for each state s_t, the optimal action a_t^* is known. This setting is impossible to apply in noisy environments such as most robotic environments and in many high-dimensional problems such as go or chess games.

2.2.2.2 Search or planning

If the dynamics of the problem are perfectly known i.e. from a given state-action pair, the next state can be computed without uncertainty, a brute force search algorithm can be efficient to decide which action to take to reach the objective. This principle has been programmed in a chess software called deep Blue which wins a serie of games against the world champion at that time Garry Kasparov [55]. The obvious drawback is that the dynamics need to be known without uncertainty to ensure a high search depth which is not the case in robotics. Furthermore, this class of methods does not scale well to too high-dimensional problems.

2.2.2.3 Learning

A third category of approach consists in learning the policy with the help of the interaction between the agent and the environment. The goal of the problem is specified by a reward function and the agent optimizes its policy so that it receives higher rewards in the future. There are two classes of algorithms which make the agent learn a policy through interaction with the environment: dynamic programming and reinforcement learning. The latter are described in 2.2.5.

2.2.3 Markov decision processes and optimality criteria

2.2.3.1 Markov Decision processes

A Markov decision process (MDP) is a tuple $< \mathcal{S}, \mathcal{A}, R, T >$ where T is defined on $\mathcal{S} \times \mathcal{A}$ and outputs a value in $\mathcal{S}$ (in the deterministic case), i.e. the next state $\boldsymbol{s_{t+1}}$ can be predicted from the action $\boldsymbol{a_t}$ and the state $\boldsymbol{s_t}$ at time t, the previous states $\boldsymbol{s_{t-n}}(n \in \mathbb{N}^*)$ not being required. As defined in 2.2.1.1, $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and R is the reward function.

This condition on T is the basis of many reinforcement learning and dynamic programming algorithms and will be assumed in any task of the presented work.

2.2.3.2 Optimality criteria

In dynamic programming and in reinforcement learning, a policy is optimized with respect to a criterion taking into account future rewards. In other terms, the goal is to obtain an optimal policy π^* while optimizing a criterion J_π. Let r_t be the reward picked at time step t and $E_\pi[x]$ be the expectation of a random variable x under the policy, i.e. the expectation of the considered random variable if the policy π is applied. We present three different criteria used in the literature [54]:

- the finite horizon criterion:

$$J_\pi = E_\pi \left[\sum_{t=0}^{h-1} r_t \right]. \tag{2.17}$$

  This criterion makes the agent consider with the same weight rewards for only h steps. It can be used in an episode setting. Two ways of using it can be considered. First, in a h-length episode, we can consider the criterion using h steps at the beginning transition, $h-1$ steps at the second transition and so on. One drawback of such a method is that for a same state, the criterion can be different according to its place in the episode. Second, for a given transition, we can always consider using the optimal h value. Indeed, for a MDP, this value can vary. For example, a

robot which has to reach using (bounded) actions an object has only few moves to make if it close to the object. Thus, h can be small. In contrast, if the robot is far from the object, the optimal h value should be higher. However, this value is not always known

- the infinite discounted criterion:

$$J_\pi = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]. \tag{2.18}$$

$\gamma \in [0, 1]$ is the discount factor. A discount factor inferior to 1 ensures that short-term rewards are taken into greater account whereas a discount factor of 1 considers all the rewards with the same weight. Furthermore, it does not have the drawbacks of the previous criterion, e.g. finding the optimal "evaluation depth".

- the average criterion:

$$J_\pi = \lim_{h \to \infty} E_\pi \left[\frac{1}{h} \sum_{t=0}^{h} r_t \right]. \tag{2.19}$$

It considers all the rewards with the same weight. One problem with this criterion is that we cannot distinguish policies that favour short-term reward from ones that favour long-term rewards.

In our tasks, we choose to use the infinite discounted criterion because it is the most flexible one. Indeed, among our applications, some tasks require to favour short-term rewards and other ones favour more long-term rewards. We only need to modify the discount factor in our algorithm to switch from a task to another one.

For each task, the values of γ are given in Appendix B.

2.2.4 Usual functions for learning sequential decision making problems

We present here the functions that are used in dynamic programming and reinforcement learning algorithms. We define them for the infinite discounted criterion (equation (2.18)).

2.2.4.1 The value (V) function

$$\forall \pi, \forall t \in \mathbb{N}, \forall \boldsymbol{s} \in \mathcal{S}, V^\pi(s) = E_\pi \left[\sum_{i=t}^{\infty} \gamma^{i-t} r_{i-t} | \boldsymbol{s_t} = \boldsymbol{s} \right] \tag{2.20}$$

The V^π function gives the value of the state $\boldsymbol{s}$ under the policy π, i.e the expectation of future rewards (the optimization criterion) from state $\boldsymbol{s}$ if the agent follows the policy π.

Note that the V^π function can be computed recursively via the equation (2.21) (in this equation, we consider deterministic dynamics) called the Bellman equation.

$$\forall \pi, \forall \boldsymbol{s} \in \mathcal{S}, V^\pi(\boldsymbol{s}) = R(\boldsymbol{s}, \boldsymbol{\pi(s)}) + \gamma V^\pi(\boldsymbol{T(s, \pi(s))}) \tag{2.21}$$

2.2.4.2 The quality (Q) function

$$\forall \pi, \forall t \in \mathbb{N}, \forall \boldsymbol{s} \in \mathcal{S}, \forall \boldsymbol{a} \in \mathcal{A}, Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = E_{\pi}\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_{i-t} | \boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a}\right] \tag{2.22}$$

The Q^{π} function assesses the quality of a state-action pair under the policy π, i.e. the expectation of future rewards (the optimization criterion) from state $\boldsymbol{s}$ if the agent applies the action $\boldsymbol{a}$ and follows the policy afterwards.

The Q^{π} function can also be computed recursively via the Bellman equations by assuming deterministic dynamics:

$$\forall \pi, \forall \boldsymbol{s} \in \mathcal{S}, \forall \boldsymbol{a} \in \mathcal{A}, Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = R(\boldsymbol{s}, \boldsymbol{a}) + \gamma \max_{\boldsymbol{b} \in A} Q^{\pi}(T(\boldsymbol{s}, \boldsymbol{a}), \boldsymbol{b}), \tag{2.23}$$

$$\forall \pi, \forall \boldsymbol{s} \in \mathcal{S}, \forall \boldsymbol{a} \in \mathcal{A}, Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = R(\boldsymbol{s}, \boldsymbol{a}) + \gamma V(T(\boldsymbol{s}, \boldsymbol{a})). \tag{2.24}$$

2.2.4.3 The advantage (A) function

$$\forall \pi, \forall \boldsymbol{s} \in \mathcal{S}, \forall \boldsymbol{a} \in \mathcal{A}, A^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) - V^{\pi}(\boldsymbol{s}). \tag{2.25}$$

The advantage function is nothing but the subtraction of the Q and the V function. Learning such a function compared with the Q function can be interesting when V values are much bigger than advantage values. Consequently, the algorithm would focus more on what is useful to discriminate actions.

2.2.4.4 Optimal value functions

In the literature, we note optimal value, quality and advantage functions respectively V^*, Q^* and A^*. These notations mean that the functions take into account the optimal policy π^*.

2.2.5 Dynamic programming vs reinforcement learning

2.2.5.1 Dynamic programming

Dynamic programming is a class of algorithms which seeks for an optimal policy with the perfect knowledge of the transition function and the reward function.

The two most famous dynamic programming algorithms are:

- Policy iteration [56]:
  This algorithm alternates between two steps. In the first one called policy evaluation, the goal is to evaluate the value function of a fixed policy π with the value function. For that, the equation (2.21) can be used, forming $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknowns (if we assume that $\mathcal{S}$ is finite). An iterative procedure is used to solve the system. The rationale is to apply for each iteration and for each state, the following update:

  $$\forall s \in \mathcal{S}, V_{k+1}^{\pi}(\boldsymbol{s}) = R(\boldsymbol{s}, \boldsymbol{\pi(s)}) + \gamma V_{k}^{\pi}(T(\boldsymbol{s}, \boldsymbol{\pi(s)})), \tag{2.26}$$

  where k denotes an algorithm iteration. This step ends when V converges.

The second step called policy improvement consists first of computing the Q value for each state action pair using equation (2.24). Then, for each state $\boldsymbol{s}$, the policy is improved by linking this state with the action $\boldsymbol{a}$ having the higher Q value $Q(\boldsymbol{s}, \boldsymbol{a})$:

$$\boldsymbol{\pi_{k+1}(s)} = \arg\max_{\boldsymbol{a}} Q^{\boldsymbol{\pi_k(s)}}(\boldsymbol{s}, \boldsymbol{a}), \tag{2.27}$$

where k denotes an algorithm iteration.

The algorithm stops when the improvement step does not improve the policy anymore.

- Value iteration [57]:
  This algorithm does not consider two separate steps and is more dedicated to the V function estimation.

  For each state $\boldsymbol{s}$, and for each action $\boldsymbol{a}$, the Q value $Q(\boldsymbol{s}, \boldsymbol{a})$ is estimated using equation (2.24). Then, V is deduced from each state using $V(\boldsymbol{s}) = \max_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a})$. The loops over states and actions stops when V converges. The policy is not explicitly represented in this algorithm but it can be deduced from the intermediate Q estimations $\boldsymbol{\pi(s)} = \arg\max_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a})$.

Many variations of these algorithms exist in the literature. However, they will not be described since we cannot apply them to our problems. As a matter of fact, in complex manipulation robotics problems, it is hard if not impossible to compute a perfect model for the reward and dynamics.

2.2.5.2 Reinforcement learning (RL)

Reinforcement learning is a class of algorithms which seeks for an optimal policy without precomputed model of dynamics. This is a particularly well-suited framework for manipulation robotics because a model of dynamics is often hard to compute.

Since models are not available, reinforcement learning requires to learn statistics about rewards or/and dynamics. To this end, the agent needs to explore its environment by taking actions and collecting transitions $< \boldsymbol{s_t}, \boldsymbol{a_t}, r_{t+1}, \boldsymbol{s_{t+1}} >$. One approach of RL is to learn models using these transitions and to use a dynamic programming method afterwards. This features the model-based RL methods. To the contrary, model-free RL methods directly learn policies without model estimation. We specifically focus on this class of methods in the report since it is very resource-demanding to learn a model with high-dimensional state spaces (though some approaches managed to learn dynamics for MDPs with a reduced state space [58]). Both approaches are illustrated in Figure 2.11.

2.2.6 Reinforcement learning issues

The paragraphs which follow aim at explaining usual reinforcement learning issues.

2.2.6.1 The exploration-exploitation trade-off

In a reinforcement learning setting, we do not have access to a model of the agent interacting with the environment. Thus, to learn a task, the agent needs to explore its environment through a trial-and-error process. However, exploring too much without using the learned knowledge about the actions leads to bad performances.

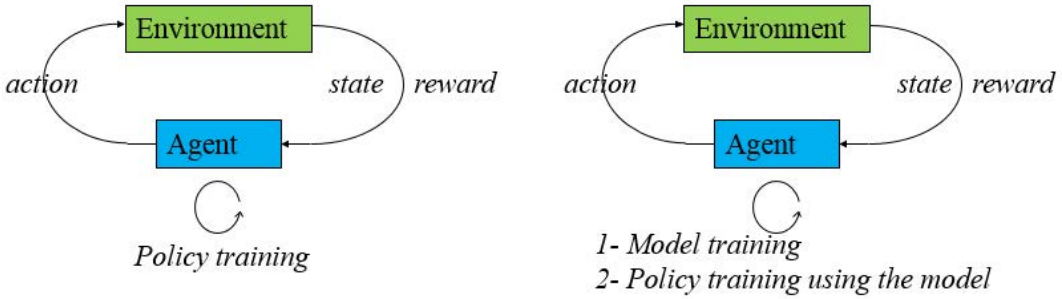


FIGURE 2.11: Model-free (left) vs model-based (right) Reinforcement learning

Indeed, in already explored state-action pairs, applying exploratory actions can lead to worse actions than the one given by the policy. Then, the agent has to explore to learn new actions and it has to exploit to perform. This problem is called the exploration-exploitation trade-off.

2.2.6.2 Credit assignment

In some reinforcement learning problems, goals are defined by "sparse-only" (or "impulse") rewards as defined in 2.2.1.1.

Note that here the reward function is defined on S as we assume deterministic dynamics.

When the agent reaches the goal area, the difficulty is to clearly distinguish the actions which allowed the agent to reach the area and the other ones. Indeed, the effect of actions on the way to reach the goal is delayed a lot. Deciding which actions to reinforce is the temporal credit assignment problem. Obvious examples are games of go or chess. The sparse reward is the outcome of the game and there are many key moves in some opening phases which have to be played to maintain winning chances [1]. Another example which fits in the subject thesis is the reaching problem for a 7-DOF manipulator. The latter has to reach a target and receives a reward only in case of success. From this success, the agent has to discriminate precisely what were the good actions.

2.2.6.3 Low probability of the first success

Another issue that can occur when the task is defined by a sparse reward is the low probability of the first success. The problem generally appears when the algorithm is initialized far from the solution and thus reaching the goal is very unlikely. For instance, reaching a target with a 7-DOF manipulator from a fixed end-effector position, reaching the goal state with uniform random action selection is simply very unlikely and lots of trials have to be made before entering it. It is also the case in the maze problem of Figure 2.12 where the agent from the red area is very unlikely to reach the blue area with a uniform exploration. To deal with it, a learning-from-easy-mission (or curriculum learning) set-up [59] can be used but requires to know where the goal is in the state space. In a nutshell, the strategy consists in learning easier tasks at the beginning (e.g. initial and goal states are made close) and continuously learning tasks of increasing difficulty. This problem does not occur in the

[1] In chess, $c5$ for black pieces in the accepted queen's gambit

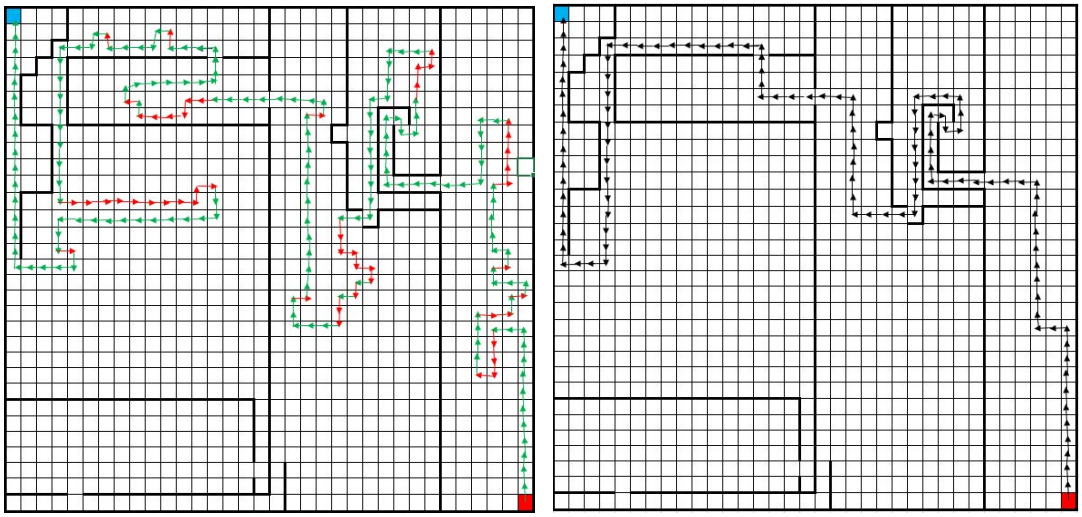


FIGURE 2.12: Maze example to illustrate the credit assignment problem: from several sub-optimal trajectories such as the one on the left figure, the agent has to learn what were the adequate (in green) and inadequate (in red) actions and find the optimal trajectory (figure on the right).

games of go or chess since there is always an outcome at the end of the game, and thus a sparse reward.

2.2.6.4 Data efficiency

Solving or getting an optimal/suboptimal policy from complex problems such as go, or some manipulation robotics tasks can require a large amount of transitions. For real-world robotics, this can be prohibitively costly. Improving the data-efficiency or sample-efficiency of an RL algorithm means making the algorithm produce an optimal/suboptimal policy with fewer transitions. Model-based RL algorithms can offer good opportunities for the data efficiency when learning dynamics is less hard than learning the policy, which is typically the case with low-dimensional problems [60], i.e problems with low-dimensional state and action spaces. In our work, we did not focus on this specific issue.

2.2.6.5 Representations and Curse of dimensionality

Until now, we did not mention how the policy and the value functions can be represented. Intuitively, for small and discrete state and action spaces, representing these functions with tables can be tractable. However, for continuous state and action spaces (and even for large discrete state and action spaces, e.g in Figure 2.12), it is intractable to use a tabular representation (curse of dimensionality, [57], [61]). As a consequence, there is a necessity to approximate these functions. Before the popularization of deep learning, using non-linear approximators in reinforcement learning algoritms was not so popular because it was very long to train and prone to instability. The most common representations for RL functions in robotics were the linear approximators or dynamic movement primitives. However, interesting innovations made learning stabler with deep non linear neural networks and yield impressive performances in complex high-dimensional games [62] and robotics [63]. This is why, we choose to use neural networks as they have been proven efficient for this purpose. Consequently, from now on, the subscripts in greek letters attached

to the RL functions represent vectors grouping all the learnable parameters of the neural network approximating the considered function. For instance, $\pi_{\boldsymbol{\theta}}$ is a neural network approximating the policy π with a parameter vector θ.

2.2.6.6 On-policy vs Off-policy

An on-policy RL method assumes that the policy is optimized based on the evaluation of the policy followed by the agent (through value function estimation). To the contrary, off-policy RL methods can solve problems based on the evaluation of a policy not followed by the agent.

For example, the policy followed by the agent might be an exploratory policy. In this case, the on-policy method will take into account the exploratory policy whereas the off-policy method will take into account another policy, e.g. the estimated optimal (or greedy) policy. The main advantage of using an on-policy method is that it generally offers less variance for the value function evaluation [64]. However, off-policy methods are more flexible. Indeed, since they can be used with other policies, they can be provided with data coming from various sources such as human expert or other demonstration data.

2.2.7 Model-free reinforcement learning

The class of model-free RL methods can be divided into two categories, the critic-only and the actor methods:

2.2.7.1 The critic-only methods

These methods directly estimate one of the value functions presented in 2.2.4. When the model is not present or not learned in the method, the learning of Q is generally privileged over V or A. This is why we focus on presenting some classical ways to learn the Q function (they can also be used to learn the V function).

Monte-Carlo methods

The Monte-Carlo estimation of the Q function at each state-action pair $< \boldsymbol{s}, \boldsymbol{a} >$ is the average of returns-to-go from this pair:

$$Q^{MC}(\boldsymbol{s}, \boldsymbol{a}) = \frac{1}{N_{\boldsymbol{s},\boldsymbol{a}}} \sum_{i=1}^{N_{\boldsymbol{s},\boldsymbol{a}}} G_{\boldsymbol{s},\boldsymbol{a}}, \tag{2.28}$$

with $N_{\boldsymbol{s},\boldsymbol{a}}$ being the number of occurrence of the state-action pair $< \boldsymbol{s}, \boldsymbol{a} >$. The Monte-Carlo method assumes that the return-to-go $G_{\boldsymbol{s},\boldsymbol{a}}$ is a random variable and computes an unbiased estimate of its expectation under the policy. An important feature of the method is that the variance of $Q^{MC}(\boldsymbol{s}, \boldsymbol{a})$ can be high (decreasing it requires to do many experiments).

Temporal difference (TD) learning

TD learning consists in learning Q based on the Q estimates at subsequent time steps. To that end, the two most famous methods are Q-learning [65], [66] (the latter refers to a specific algorithm even if the name simply suggests learning of the Q function) and SARSA [67], [68], [69]. Q-learning is an off-policy algorithm which

consists in using the Bellman equation (2.23) for an estimation:

$$Q_{k+1}(\boldsymbol{s_t}, \boldsymbol{a_t}) = Q_k(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha\delta_t, \tag{2.29}$$

$$\delta_t = r_t + \gamma \max_{\boldsymbol{a}} Q_k(\boldsymbol{s_{t+1}}, \boldsymbol{a}) - Q_k(\boldsymbol{s_t}, \boldsymbol{a_t}), \tag{2.30}$$

with α being a decay parameter. Note that the difference δ_t is called the TD-error. Theoretically, this algorithm converges if each state-action is visited an infinite number of times and α is properly decreased [66]. SARSA which is an on-policy algorithm uses a similar update:

$$Q_{k+1}(\boldsymbol{s_t}, \boldsymbol{a_t}) = Q_k(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha(r_t + \gamma Q(\boldsymbol{s_{t+1}}, \boldsymbol{a_{t+1}}) - Q_k(\boldsymbol{s_t}, \boldsymbol{a_t})). \tag{2.31}$$

However, instead of maximizing the Q function at state $\boldsymbol{s_{t+1}}$, the agent computes the action $\boldsymbol{a_{t+1}}$ according to the policy followed by the agent (including exploration) and replaces the maximum by the Q value of the state-action pair at time $t+1$.

TD(λ)

This method is an intermediate approach between Monte-Carlo methods ($\lambda = 1$) and one-step methods such as Q learning ($\lambda = 0$) or SARSA. It uses N-step truncated returns G_t^N:

$$G_t^N = \sum_{i=1}^{N-1} \gamma^i r_{t+i} + \gamma^N \max_{\boldsymbol{a}} Q(\boldsymbol{s_{t+n}}, \boldsymbol{a}). \tag{2.32}$$

The rationale is to combine these returns with a weight λ^{N-1} to trade-off the variance and the bias of the Q estimation. Computing these returns according to the definition is impractical but there is an efficient way to calculate it: using eligibility traces. An eligibility trace is the output of a function $e : S \times A \rightarrow \mathbb{R}^+$ allowing to keep in memory first that a transition has been explored and second at which time-steps of the episode it has been explored. The algorithm consists in updating at each time-step t every eligibility trace of state-action pair $< \boldsymbol{s_t}, \boldsymbol{a_t} >$ (they are all set to zero at the beginning of an episode):

$$e_t(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} \lambda\gamma e_{t-1}(\boldsymbol{s}, \boldsymbol{a}), & \text{if } < \boldsymbol{s}, \boldsymbol{a} > \neq < \boldsymbol{s_t}, \boldsymbol{a_t} >, \\ \lambda\gamma e_{t-1}(\boldsymbol{s}, \boldsymbol{a}) + 1, & \text{otherwise.} \end{cases}$$

Then, with the already defined TD-error δ_t, Q can be updated this way for each state-action pair:

$$Q_{t+1}(\boldsymbol{s}, \boldsymbol{a}) = Q_t(\boldsymbol{s}, \boldsymbol{a}) + \alpha e_t(\boldsymbol{s}, \boldsymbol{a})\delta_t \tag{2.33}$$

Efficient versions of Q(λ) have been proposed in [65], [70], [71].

Policy computation

Using estimates of Q allows to deduce the policy with the formula $\boldsymbol{\pi(s)} = \arg\max_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a})$. This is applied without difficulty in MDPs with discrete action spaces such as Atari games [72]. When the action space is continuous, estimating the policy is less simple. In this case, finding the optimal action requires a potentially costly optimization procedure. Thus, even though finding actions for complex robotic problems has been done efficiently in [73], we have chosen to learn the policy such that an action can be computed in one policy forward pass.

2.2.7.2 The actor methods

The actor methods learn the policy π. Among them, we distinguish the ones which use a value function (actor-critic) and the other ones (actor-only). Generally, an actor-critic algorithm learns a value function (Q, A or V) which is called the critic and from this critic computes policy (actor) updates. An actor-only algorithm directly learns the policy without learning any value function, e.g. it can learn the policy from return or return-to-go values.

In all the examples, we consider a policy $\pi_{\boldsymbol{\theta}}$ parametrized by a neural network of learnable parameters $\boldsymbol{\theta}$. We can distinguish several classes of policy search methods. As we cannot be exhaustive for this class of algorithms, we focus on methods that (1) can be applied in robotics (based on [74]) and (2) have been expanded to deal with high-dimensional problems. And, in section 2.3, we will describe some algorithms able to deal with high-dimensional and continuous state and action spaces.

Policy gradient methods

This class of methods consists in applying gradient ascent to maximize the expected return $E\left[G_{\boldsymbol{\theta}}\right]$: $\boldsymbol{\theta_{k+1}} = \boldsymbol{\theta_k} + \alpha \frac{\partial E[G_{\boldsymbol{\theta}}]}{\partial \boldsymbol{\theta}}$, where α is a learning rate.
The policy gradient definition is the following:

$$\frac{\partial E\left[G_{\boldsymbol{\theta}}\right]}{\partial \boldsymbol{\theta}} = \int_{\tau} \frac{\partial p_{\boldsymbol{\theta}}(\boldsymbol{\tau})}{\partial \boldsymbol{\theta}} G(\boldsymbol{\tau}) d\tau \tag{2.34}$$

$\boldsymbol{\tau}$ represents a trajectory, i.e a succession of state-action pairs in a L-length episode $\boldsymbol{\tau} = \boldsymbol{s_1}, \boldsymbol{a_1} ... \boldsymbol{a_L}, \boldsymbol{s_{L+1}}$, $G(\boldsymbol{\tau})$ is the associated return and $p_{\boldsymbol{\theta}}(\boldsymbol{\tau})$ is the probability of having the trajectory $\boldsymbol{\tau}$ given the parameters (the initial state has also an influence).
The algorithms differ in the gradient $O_{\boldsymbol{\theta}} = \frac{\partial E[G_{\boldsymbol{\theta}}]}{\partial \boldsymbol{\theta}}$ computation.

We review the following methods:

- Finite difference method [75], [76]:
  This method applies small perturbations $\Delta\boldsymbol{\theta}$ in the parameter vector $\boldsymbol{\theta}$ and updates the parameters according to the changes in the return ΔG.

- Use of the likelihood-ratio policy gradient trick:
  This trick uses the property of the derivative of the log inside equation (2.34):

  $$\frac{\partial p_{\boldsymbol{\theta}}(\boldsymbol{\tau})}{\partial \boldsymbol{\theta}} = p_{\boldsymbol{\theta}}(\boldsymbol{\tau}) \frac{\partial \log p_{\boldsymbol{\theta}}(\boldsymbol{\tau})}{\partial \boldsymbol{\theta}} \tag{2.35}$$

  Several algorithms are based on this principle among which REINFORCE (equation (2.36)) [77] and G(PO)MDP [78].

  $$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s} \sim p^{\pi}, \boldsymbol{a} \sim \pi_{\boldsymbol{\theta}}} \left[\frac{\partial \log(\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}))}{\partial \boldsymbol{\theta}} (G(\boldsymbol{\tau}) - b) \right], \tag{2.36}$$

  where b is a baseline reducing the variance of the gradient estimation. Note that in the expectation, we do not consider $p_{\boldsymbol{\theta}}(\boldsymbol{\tau})$ the trajectory probability distribution. Instead, we consider state and action distribution probabilities (p^{π} and $\pi_{\boldsymbol{\theta}}$).

  REINFORCE and G(PO)MDP assume a stochastic policy and estimate returns using Monte-Carlo samples. G(PO)MDP is based on the observation that past rewards do no depend on future actions. Thus, for a given reward value, only the past actions are taken into account in the optimization. This reduces the variance of the REINFORCE policy gradient estimate.

- Actor-critic algorithms:
  Another way to estimate the policy gradient is to learn one of the RL functions to estimate the expectation of the return (V, Q, or A) instead of computing Monte-Carlo returns. A policy gradient is derived from the RL functions afterwards. Among famous methods, there is the policy gradient theorem [79] which is based on the same rationale as G(PO)MDP:

$$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^{\pi},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[\frac{\partial log(\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}))}{\partial\boldsymbol{\theta}}Q^{\pi}(\boldsymbol{s},\boldsymbol{a})\right], \tag{2.37}$$

  where Q^{π} can be learned using a function approximator. Another popular method is the natural actor critic algorithm [80], [81]. Based on value function approximation, the policy is updated using natural policy gradients such that the distribution of trajectories $p_{\boldsymbol{\theta}}(\boldsymbol{\tau})$ does not change too much after an update. The rationale is to improve the stability of the algorithms. Other more recent actor-critic algorithms are reviewed in section 2.3.

Information-theoretic approaches

This class of algorithms shares the idea with natural policy gradient algorithms that when a policy update occurs, the trajectory distribution should not change brutally. In practice, these approaches bound the distance between the old and the new trajectory distributions (respectively $p_{\text{old}}(\boldsymbol{\tau})$ and $p_{\text{new}}(\boldsymbol{\tau})$) while updating the policy. The most famous algorithm is the relative entropy policy search (REPS) [82]. The policy search problem has the form of a constrained optimization problem in which we maximize the return given the policy parameters. This problem is subject to an inequality constraint on the Kullback-Leibler divergence between the new policy and the old one which cannot overpass a given value. Recent algorithms [83], [84] also use similar updates and are described in the next section.

Miscellaneous approaches

We do not precisely review some classes of policy search methods such as expectation-maximization policy search [85], stochastic optimization [86] or path integral methods [87], [88] because to our knowledge they were not applied in complex high-dimensional problems.

2.3 Deep Reinforcement Learning

In section 2.1, we have described neural network structures used in this thesis. In section 2.2, we have focused on the common knowledge of reinforcement learning. In addition, we have also described some policy search algorithms which are the basis of some deep reinforcement learning algorithms. In this section, we focus on deep reinforcement learning and more precisely on the algorithms which have been proven able to scale to high-dimensional and continuous state and action spaces.

2.3.1 Idea

A deep reinforcement learning (DRL) algorithm is an RL algorithm which uses deep networks to approximate one or several RL functions: V, Q, A, π or dynamics. Its goal is to learn how to solve complex problems with potentially high-dimensional

state spaces. Neural networks in RL have been used for a long time in backgammon [89], in robotics [90], [91], [92] and in elevator dispatching [93]. However, to our knowledge, the use of multi-layered neural networks to approximate functions using high-dimensional state spaces dates from 2010s where [94] used a deep autoencoder to reduce the dimensionality of images ($30 \times 30 = 900$ pixels) in a simple continuous grid-world problem. After that, a Q function is learned on these features. Since then, the field of deep reinforcement learning has quickly developed: lots of methods dedicated to complex problems have been established, e.g. for Atari games [62], for complex problems with continuous action spaces [95], [96], very complex games such as the game of go [97], and robotics [84].

2.3.2 Deep Deterministic policy gradient

We focus now on the method we have chosen for our robotic applications, namely the deep deterministic policy gradient (DDPG) algorithm [95]. This algorithm is from the actor-critic category and mixes the deep Q network algorithm [62] and the off-policy version of deterministic policy gradient algorithms [98]. In the following, we omit the subscript t for a better readability unless it is necessary. By convention, the subscript i refers to a transition and t refers to an iteration.

2.3.2.1 Deep Q network

The deep-Q-Network algorithm [62] consists in learning a Q function with a Q learning update (see equation (2.29)) using a deep network as a function approximator. For a given transition $< \boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}^{'} >$, the corresponding input-output pair $< \boldsymbol{x}, y >$ is given according to the equations:

$$\boldsymbol{x} = (\boldsymbol{s}, \boldsymbol{a}), \tag{2.38}$$

$$y = r + \gamma \max_{\boldsymbol{b}} Q_{\phi}(\boldsymbol{s}^{'}, \boldsymbol{b}), \tag{2.39}$$

where ϕ represents the vector of learnable parameters of the Q network.

To make the learning of Q stable (Q learning with non-linear neural networks suffered a lot from instability [99], [100]), two key strategies are implemented:

- The experience replay mechanism [101]:
  DQN like Deep Fitted Q-iteration [94], uses batch updates to learn the neural network representing Q. It means that Q is not updated transition by transition but rather uses a set of N_{b} transitions. If we choose the last N_{b} transitions experimented by the agent, the i.i.d assumption common to machine learning algorithms is broken. Indeed, these transitions are correlated. This is why, these transitions are chosen such that the batch contains independent and identically distributed input-output pairs. To that end, N_{b} transitions are uniformly chosen in a N_{trans}-size memory buffer D: $\forall i \in \{1, ..., N_{\text{b}}\}, < \boldsymbol{s_i}, \boldsymbol{a_i}, r_i, \boldsymbol{s_i^{'}} > \sim D$. N_{trans} is chosen high enough to limit the correlation between the chosen transitions. The buffer $\mathcal{D}$ is updated according to a first-in first-out process.

- The use of a target network:
  This strategy consists in modifying equation (2.39):

$$y = r + \gamma \max_{\boldsymbol{b}} Q_{\phi'}(\boldsymbol{s}^{'}, \boldsymbol{b}), \tag{2.40}$$

where $\phi^{'}$ groups the learnable parameters of a the target network. The target network is updated at each iteration t by following the Q_ϕ network:

$$\phi^{'}_{t+1} = \alpha\phi_t + (1-\alpha)\phi^{'}_t, \tag{2.41}$$

where α is a decay parameter. The rationale of the use of a target network is the same as for the use of natural gradients and the use of constraints in trajectory distribution change: for stability purposes, the neural network should not vary too much.

2.3.2.2 Deterministic policy gradient algorithms

Deterministic policy gradient methods (DPG) [98] are actor-critic algorithms able to deal with continuous action spaces.

This algorithm considers a deterministic version of the policy gradient theorem [79] which assumes a stochastic policy: $O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^\pi, \boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[\frac{\partial log(\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}))}{\partial\boldsymbol{\theta}} Q^\pi_\phi(\boldsymbol{s},\boldsymbol{a})\right]$ (see equation (2.37)), where $\boldsymbol{\theta}$ is the set of learnable parameters of the policy and $\boldsymbol{\phi}$ is the set of learnable parameters of Q.

Note that the original version of this theorem uses Q^π instead of Q^π_ϕ because Q^π can be estimated using Monte-Carlo sample returns instead of being parametrized. This gradient is known to have a large variance because the expectation is under the action and state probability distributions. The deterministic policy gradient theorem considers the expectation of the gradient only under the state probability distribution because the policy is deterministic:

$$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^\pi}\left[\frac{\partial\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s})}{\partial\boldsymbol{\theta}}\frac{\partial Q^\pi_\phi(\boldsymbol{s},\boldsymbol{a})}{\partial\boldsymbol{a}}|\boldsymbol{a}=\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s})\right]. \tag{2.42}$$

From this theorem, two algorithms have been developed: an off-policy and an on-policy versions which differ only by the way the critic is updated. In the on-policy version, SARSA is used to update the critic, i.e for a given transition, the target of the Q network is $r + \gamma Q_\phi(\boldsymbol{s}^{'}, \boldsymbol{a}^{'})$. In the off-policy case, the target is:

$$y = r + \gamma Q_\phi(\boldsymbol{s}^{'}, \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}^{'})). \tag{2.43}$$

For these two algorithms, the policy is updated using a stochastic gradient ascent to maximize the Q function, i.e. to maximize the expected return:

$$\Delta\boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}}\frac{\partial\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s})}{\partial\boldsymbol{\theta}}\frac{\partial Q^\pi_\phi(\boldsymbol{s},\boldsymbol{a})}{\partial\boldsymbol{a}}|\boldsymbol{a}=\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}), \tag{2.44}$$

where $\alpha_{\boldsymbol{\theta}}$ is the learning rate parameter.

2.3.2.3 Deep deterministic policy gradient (DDPG)

The DDPG [95] algorithm is an off-policy actor-critic DPG algorithm which uses the DQN algorithm for the critic, i.e. with the two key innovations described in 2.3.2.1. To summarize, the critic is updated based on a mini-batch of transitions which are uniformly chosen in a memory buffer. Equation (2.43) is used:

$$y_i = r_i + \gamma Q_{\phi'}(\boldsymbol{s}^{'}_i, \boldsymbol{\pi}_{\boldsymbol{\theta}'}(\boldsymbol{s}^{'}_i)), \tag{2.45}$$

with i being an integer denoting a transition, $\boldsymbol{\theta}'$ and $\boldsymbol{\phi}'$ being the sets of learnable parameters of the policy and Q target networks. The latter are updated using equation (2.41). The critic is updated by considering all the transitions of the mini-batch:

$$\Delta\boldsymbol{\phi} = \alpha_{\boldsymbol{\phi}} \frac{1}{N_\text{b}} \sum_{i=1}^{N_\text{b}} (y_i - Q_{\boldsymbol{\phi}}(\boldsymbol{s_i}, \boldsymbol{a_i})) \frac{\partial Q_{\boldsymbol{\phi}}(\boldsymbol{s_i}, \boldsymbol{a_i})}{\partial \boldsymbol{\phi}} \tag{2.46}$$

Note that we consider a least square error as a loss function:

$$L = \frac{1}{2N_\text{b}} \sum_{i=1}^{N_\text{b}} (y_i - Q_{\boldsymbol{\phi}}(\boldsymbol{s_i}, \boldsymbol{a_i}))^2. \tag{2.47}$$

Once the critic is updated, the actor is updated using 2.44 and considering the batch of the N_b chosen transitions:

$$\Delta\boldsymbol{\theta} = \alpha_{\boldsymbol{\theta}} \frac{1}{N_b} \sum_{i=1}^{N_\text{b}} \frac{\partial \boldsymbol{\pi_\theta}(\boldsymbol{s_i})}{\partial \boldsymbol{\theta}} \frac{\partial Q_{\boldsymbol{\phi}}(\boldsymbol{s_i}, \boldsymbol{\pi_\theta}(\boldsymbol{s_i}))}{\partial \boldsymbol{a}}. \tag{2.48}$$

Figure 2.13 shows how this update is computed from an algorithmic point of view. First, $Q_{\boldsymbol{\phi}}(\boldsymbol{s_i}, \pi_{\boldsymbol{\theta}}(\boldsymbol{s_i}))$ is computed (forward step). Second, a backward step is achieved using the backpropagation algorithm and the policy gradient is derived.

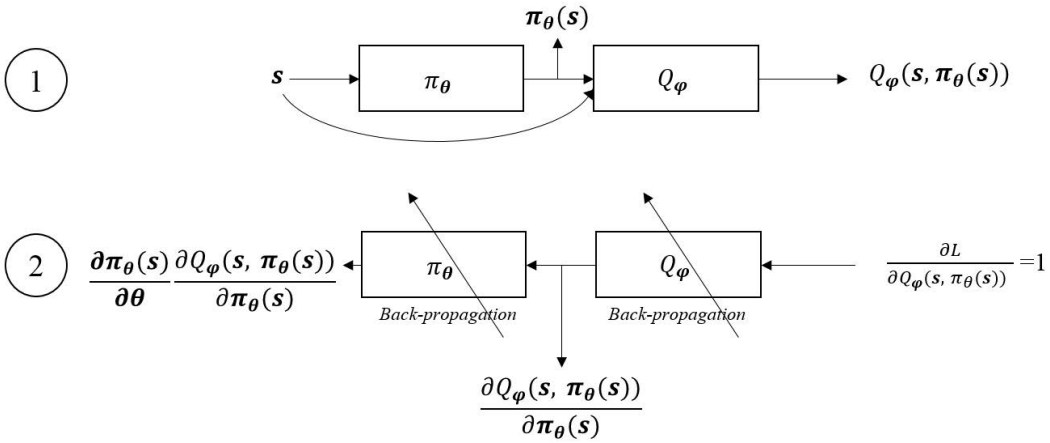


FIGURE 2.13: Scheme of the policy update using DDPG. The first step is the forward pass: from a state $\boldsymbol{s}$, we compute $Q_{\boldsymbol{\phi}}(\boldsymbol{s}, \boldsymbol{\pi_\theta(s)})$. The second step is the backward pass: we set the derivation of the loss with respect to the Q value to 1, we apply the back-propagation algorithm on Q to get $\frac{\partial Q_{\boldsymbol{\phi}}(\boldsymbol{s}, \boldsymbol{\pi_\theta(s)})}{\partial \boldsymbol{a}}$. From the latter, we apply again the back-propagation algorithm on the policy network and it gives us the policy gradient.

2.3.3 Alternate algorithms

In the following, we briefly describe some alternate algorithms which are also suitable to high-dimensional and continuous state-action spaces. We particularly describe the model-free settings.

2.3.3.1 Trust Region Policy Optimization (TRPO)

TRPO [83] is a model-free RL algorithm which gives a way to update a stochastic policy given an estimation of the advantage function. This problem is formalized as a constrained optimization problem:

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{maximize}} \quad & E_{\boldsymbol{s}\sim p^{\boldsymbol{\theta}_{\text{old}}},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{a}|\boldsymbol{s})}A_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{s},\boldsymbol{a})\right]. \\ \text{subject to} \quad & E_{\boldsymbol{s}\sim p^{\boldsymbol{\theta}_{\text{old}}}}\left[\hat{\text{D}}_{\text{KL}}(\boldsymbol{\theta}_{\text{old}},\boldsymbol{\theta})\right] \leq \epsilon \end{aligned}$$

$\boldsymbol{\theta}_{\text{old}}$ stands for the vector of parameters before the policy update, $p^{\boldsymbol{\theta}_{\text{old}}}$ denotes the distribution of states given the policy parameters before the update and ϵ is a parameter bounding the parameter update amplitude. $\hat{\text{D}}_{\text{KL}}(\boldsymbol{\theta}_{\text{old}},\boldsymbol{\theta})$ is an estimation of the KL divergence between the parameter distributions before and after the update. The method consists in updating the policy parameters so that it optimizes the probability of having low-cost values in the future. The constraint has stability purposes and makes the policy parameters not vary too much. This is an idea we can find in REPS and in algorithms using natural gradients. The difference with REPS [82] (see section 2.2.7.2) is that TRPO constraints conditional probabilities $\pi(\boldsymbol{a}|\boldsymbol{s})$ whereas REPS constraints joint distribution $p(\boldsymbol{a},\boldsymbol{s})$. The key feature of the TRPO method is that policy improvements are theoretically guaranteed. In practice, A has been replaced by Q Monte-Carlo estimation in the original paper. Results are comparable to the ones of DQN on Atari games and this algorithm has been used in problems with continuous actions spaces such as robotic locomotion (these applications are described in [83]).

2.3.3.2 Generalized Advantage Estimation (GAE)

GAE [102] is a way to estimate an advantage function which takes inspiration from $TD(\lambda)$. The advantage estimator takes two parameters γ and λ:

$$A_t(\gamma,\lambda) = \sum_{l=t}^{\infty}(\lambda\gamma)^{t+l}\delta_{t+l}^V, \tag{2.49}$$

where δ_{t+l}^V is the temporal difference using the V function at time l:

$$\delta_{t+l}^V = -V(s_{t+l}) + r_{t+l} + \gamma V(s_{t+l+1}). \tag{2.50}$$

λ is the parameter trading-off variance and bias. For example, $A_t(\gamma,0) = \delta_0^V$ is an advantage estimation with very low variance but with bias. To the contrary, $A_t(\gamma,1) = -V(s_t) + \sum_{l=t}^{\infty}\gamma^{t+l}r_{t+l}$ has low bias but high variance.

In practice, the V function is approximated and advantage estimators are computed using 2.49. From this, a policy gradient can be estimated (inspired by the policy gradient theorem):

$$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^{\pi},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[\frac{\partial log(\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}))}{\partial\boldsymbol{\theta}}A(\gamma,\lambda)\right] \tag{2.51}$$

Note that [102] proposes to update V based on a trust region update and π using TRPO [83]. This algorithm has been proved efficient on a set of robotic tasks such as 3d-biped and quadruped locomotion tasks (described in [102]).

2.3.3.3 Proximal Policy Optimization (PPO)

PPO [103] qualifies a family of policy gradient methods for reinforcement learning. This algorithm is inspired by ideas of TRPO regarding stabilility of updates. The latter should not change too much the policy parameter distribution. Two new policy gradients are proposed. The first one proposes to clip $r_{\boldsymbol{\theta}} = \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\boldsymbol{\theta_{\text{old}}}}(\boldsymbol{a}|\boldsymbol{s})}$ between $1-\epsilon$ and $1+\epsilon$ such that $r_{\boldsymbol{\theta}}$ is close to 1 (and the policy does not change too much):

$$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^{\pi},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[\min(r_{\boldsymbol{\theta}}A(\boldsymbol{s},\boldsymbol{a}), \text{clip}(r_{\boldsymbol{\theta}}, 1-\epsilon, 1+\epsilon)A(\boldsymbol{s},\boldsymbol{a}))\right], \tag{2.52}$$

where $A(s,a)$ can be a Monte-Carlo return or an advantage estimation (e.g. using GAE).

The second one proposes to include a Kulback-Leibler penalty directly in the objective.

$$O_{\boldsymbol{\theta}} = E_{\boldsymbol{s}\sim p^{\pi},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[r_{\boldsymbol{\theta}}A(\boldsymbol{s},\boldsymbol{a}) - \beta\hat{\text{D}}_{\text{KL}}(\boldsymbol{\theta_{\text{old}}},\boldsymbol{\theta})\right], \tag{2.53}$$

with β being a parameter trading-off the variation amplitude of policy parameters against the objective optimization. The first idea (clipping) has been found more efficient on benchmark continuous control tasks (including reaching and walking tasks) than the second one and has been successfully tested on Atari games (described in [103]).

2.3.3.4 Q-Prop

Q-Prop [104] is an algorithm which combines advantages of sample efficient off-policy algorithms such as DDPG and stability of on-policy algorithms such as TRPO. It also reduces the variance of Monte-Carlo gradient estimators without adding bias. Formally, the Q-prop gradient takes the form of a mix between a DDPG update and a residual Policy Gradient Theorem update:

$$\begin{aligned} O_{\boldsymbol{\theta}} = {} & E_{\boldsymbol{s}\sim p^{\pi},\boldsymbol{a}\sim\pi_{\boldsymbol{\theta}}}\left[\frac{\partial\log(\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}))}{\partial\boldsymbol{\theta}}(Q^{MC}(\boldsymbol{s},\boldsymbol{a}) - Q(\boldsymbol{s},\boldsymbol{a}))\right] \\ & + E_{\boldsymbol{s}\sim p^{\pi}}\left[\frac{\partial Q(\boldsymbol{s},\mu_{\boldsymbol{\theta}}(\boldsymbol{s}))}{\partial\boldsymbol{a}}\frac{\partial\mu_{\boldsymbol{\theta}}(\boldsymbol{s})}{\partial\boldsymbol{\theta}}\right], \end{aligned} \tag{2.54}$$

with $Q^{MC}(s,a)$ being a Monte-Carlo estimation of Q. As stochastic policies are considered, $\mu_{\boldsymbol{\theta}}(\boldsymbol{s}) = E_{\pi_{\boldsymbol{\theta}}}[\boldsymbol{a}]$. This algorithm has been proven more sample-efficient than TRPO with GAE in various robotic tasks and outperforms DDPG in very challenging robotic tasks such as humanoid and walking tasks.

2.3.3.5 Asynchronous Advantage Actor-Critic (A3C)

Asynchronous advantage actor-critic (A3C) [96] is a model-free algorithm suitable to learn complex visuo-motor problems such as games in the Atari platform or robotics tasks such as walking or reaching (experiments summarized in [96]) . It is asynchronous because it mixes gradients from several agents which act in separate threads. It is an actor-critic algorithm because it learns both the value function V and the policy π. More precisely, the V function is learned using N-step returns (see section 2.2.7.1) as targets. Then, a policy gradient can be built based on a Policy Gradient Theorem update and an advantage estimation (see equation 2.51). The latter is based on the computed N-step returns which act as Q estimates and the learned V function. The value and policy updates are based on the values and policy gradients of all the parallel agents.

Algorithm	Simplicity	Existent at the beginning of 2016	Ability to deal with high-dimensional problems
DDPG	**yes**	**yes**	**yes**
TRPO	no	**yes**	**yes**
GPS	no	**yes**	**yes**
A3C	**yes**	no	**yes**
PPO	**yes**	no	**yes**
Q-Prop	**yes**	no	**yes**

TABLE 2.1: Reasons for the algorithm choice. We try our best to be as objective as possible to evaluate the criterion "simplicity".

2.3.3.6 Guided Policy Search (GPS)

GPS is a class of policy search methods which guides policy learning by giving "adequate samples" for learning. The assumption made by the authors is that the distribution of explored trajectories has the following shape:

$$p(\boldsymbol{\tau}) \propto \exp(r(\boldsymbol{\tau})). \tag{2.55}$$

Initially [105], [106], guided policy search is a method which trains policies in two phases. First, it uses known dynamics and trajectory examples to build local controllers. Then, the local controllers can provide samples of the distribution indicated by 2.55 to a policy search learning framework (importance policy search [107] for [105] and variational inference [108] for [106]).

Under the same assumption, a constrained guided policy search [109] algorithm has been developed and outperforms the previous ones. It takes the form of a constrained optimization problem where the objective is to make the trajectory distribution $q(\boldsymbol{\tau})$ induced by local controllers match the distribution of equation 2.55. The constraints ensure that $q(\boldsymbol{\tau})$ is consistent with known dynamics and that the policy matches the local controllers. Dynamics are no longer required in [84] where they are learned via a crude prior model and linear local ones. Furthermore, a constraint about the change between trajectory distributions during the local controller optimization appears, ensuring that the further local controller optimization does not occur in areas where dynamics is not accurate enough.

Finally, the GPS algorithm has been extended to visuo-motor policies [1], [110] making it practical to learn complex visuo-motor manipulation robotic tasks such as screwing on a bottle cap or placing a coat hanger on a clothes rack.

2.3.4 Choice

We have chosen the DDPG algorithm because (1) it is an algorithm made for high-dimensional and continuous state and action spaces, (2) its implementation is relatively straightforward and (3) the algorithm existed at the beginning of our programming work. Alternate algorithms (including those which are not mentioned in the section) could have been implemented. They were not either because they lack simplicity or because they did not exist at the beginning of our work. Details of the choice are presented in Table 2.1.

2.4 Conclusion

This chapter has reviewed important tools used in this thesis. At first, we have described the neural network structures which are utilized in our work. Second, we have focused on important knowledge about reinforcement learning and in parallel we have described important policy search methods. Third, we have presented several deep reinforcement learning algorithms which are suitable to problems using a high-dimensional state space and a continuous action space. In addition, we have given reasons for the choice of the DRL algorithm used in our work: the DDPG algorithm.

Chapter 3

State of the art

The previous chapter has presented theoretical tools that we use in our work, notably deep reinforcement learning and the DDPG algorithm. The latter allows to learn high-dimensional manipulation robotics tasks without requiring to model dynamics and hand-crafted features. In this chapter, we focus on any kind of manipulation robotics task which involves an end-effector and interaction of the latter with its environment, e.g reaching, touching, grasping, pushing and placing objects. In such manipulation tasks using deep reinforcement learning, the success of the task is often defined by a sparse reward (see section 2.2.1.1 for the definition) in a high-dimensional state space. With such a setting, learning is difficult because the probability of getting the first success is often low (see section 2.2.6.3 for a discussion on the issue). In this chapter, we discuss several solutions found in the state of the art to make learning of such a task tractable. Particularly, we focus on what kind of information is required to learn manipulation robotics tasks in the state of the art whatever the deep reinforcement learning algorithm used.

First, we describe some research studies on manipulation robotics using reinforcement learning which were published before the emergence of deep learning and we analyse their main requirements for the goal specification and the state and action space representation. Second, we present methods which use deep reinforcement learning to learn manipulation robotics tasks. Since the reward function design is the core of our contributions, we describe both what the reward function requires to be computed, e.g. human supervision, model computation, calibration parameters and if required the way learning is made practical, e.g. massive uses of computational resources or materials, simplification of state and/or action spaces or use of expert demonstration.

3.1 Manipulation robotics with reinforcement learning before the emergence of deep learning

In this section, we present a non exhaustive list of manipulation robotics applications which use reinforcement learning. We want to identify what are the priors used to learn the robotics skills. Consequently for these studies, we specifically note how the state and action spaces are represented, what are their dimensions and what are the kinds of requirements for the reward computation. In addition, we want to identify whether learning needs expert demonstrations. Table 3.3 summarizes the previously stated details for the studies. This table uses the following conventions:

- In the column "State", the results are written $N_{\text{state}}|$continuity where N_{state} is the dimension of the state space and continuity indicates if the state space is continuous, discrete or mixed (if the state space contains both discrete and continuous

State	Dim(S) | (continuous (c) or mixed (m) or discrete (d))
Action	Dim(A) | (continuous (c) or mixed (m) or discrete (d))

TABLE 3.1: Notations for the state and action specifications

Reward function	**sh**: shaping	**sp**: sparse	**pe**: penalties on robot efforts		
Require-ments	**F(I)K**: forward (inverse) kinematics	**TT**: target tracking **TM**: target measure	**UD**: use of demonstra-tions	**IRR**: Random initial robot positions **TR**: Random target position	**HVP**: Hand-crafted visual processing

TABLE 3.2: Notations for the reward function and requirements

states).

- In the column "Action", the results are written in a similar way $N_{\text{action}}|$continuity.
- The column "Reward function" describes the types of reward terms used in the reward function. Specifically, "sh" means shaping term, "sp" means sparse term, "pe" means penalties on robot efforts. For instance, a reward function which uses a shaping term and penalties on robot efforts is encoded as sh | pe.
- The column "Requirements" tells about the constraints which allow to make learning tractable or/and to compute the reward function. Specifically, "IK" and "FK" mean inverse kinematics and forward kinematics, "TM" represents target measure, "TT" stands for target tracking. The latter criteria are generally used to compute reward shaping terms or to simplify the action space (inverse kinematics). "IRR" denotes initial robot position randomly generated, "TR" signifies target position randomly generated. These criteria give an indication of the complexity of the task. "UD" means use of demonstrations which is generally used to guide exploration. Finally, "HVP" means hand-crafted visual pre-processing and is generally used to simplify the state space or to detect targets in images for shaping reward term computations. For example, the set of requirements of an application which uses forward kinematics and target tracking is encoded as FK, TT.

These notations are summarized in Table 3.1 and 3.2. From Table 3.3, we can notice several important facts:

- The state spaces are low-dimensional (the dimensions are all inferior to 21) with respect to work using raw pixels as inputs (the state space generally being composed of measured forces [111], Cartesian coordinates [111], [112], [114], [116], [117], [119], [60], joint angles [113], [115], [118] and associated velocities [112], [115]). These studies could not use high-dimensional state spaces such as images because at this time it was intractable due to the curse of dimensionality (see section 2.2.6.5). Progress in terms of hardware (appearance of GPUs) and in terms of algorithm allowed to switch to more high-dimensional tasks afterwards. They will be presented in section 3.2.

Year	Paper	State	Action	Reward function	Task	Requirements
1994	[111]	11 | c	5 | c	sh | pe	peg-in-hole insertion	FK, IK, IRR, TM
1994	[112]	5 | c	5 | c	sh	devil sticking	FK
2004	[113]	7 | c	7 | c	sh | pe	reaching a joint configuration	TM, UD
2006	[114]	3 | d	3 | d	sp	reaching an object	FK, TT, UD, HVP
2009	[115]	20 | c	7 | c	sh	ball-in-a-cup	FK, TT, UD, HVP
2010	[116]	2 | c	2 | c	sh	reaching with obstacle avoidance	FK, TM, UD
2011	[117]	3 | c	3 | c	sp | sh	pouring a liquid	FK
2011	[118]	3 | c	3 | c	sp | sh | pe	flipping a light switch	FK, UD, TM
2011	[119]	1 | c	1 | c	sp | pe	picking up a pen	
2011	[60]	3 | c	4 | c	sh	block stacking	FK, TT, HVP
2013	[120]	6 | c	3 | d	sp	grasping object	IK, TT, HVP, IRR, TR

TABLE 3.3: Summary of manipulation robotics applications using reinforcement learning before the deep learning emergence. See Tables 3.1 and 3.2 for the notations.

- Lots of action spaces are continuous (all the presented papers except [114] and [120]) and can already be high-dimensional [113], [115]. For this criterion, we just notice that the researchers rarely use inverse kinematics to reduce the dimension of the action space [111], [120]. We do not want to use the knowledge of inverse kinematics in our work because we assume it can vary during some applications (due to encoder offset or damages to the arm structure).

- Some tasks do not use neither demonstrations nor shaping terms [119], [120]. In [119], learning a picking-up pen task is tractable because the state space is very low-dimensional. If the state space is high-dimensional, learning with sparse rewards is difficult because of two problems mentioned in chapter 2. First, the probability to reach a first success can be very low. Second, the algorithm has to solve the complex credit assignment problem. [120] learns a grasping task but uses object position information in the state space (requiring a supervised segmentation module) and inverse kinematics to generate actions. Furthermore, a sparse reward function is built based on a grasp success predictor (map between object position in camera images and success) learned with supervised learning. Note that no precise information is given on the way the targets are generated in the paper.

- Numerous studies use demonstrations to initialize reinforcement learning policies to make learning tractable. We do not want to use demonstrations in our work since this depends on an expert knowledge or a supervised controller.
- Lots of studies use a shaping reward term. This term is aimed at driving the agent exploration towards success areas and can considerably accelerate learning. We observe in the studied papers that the shaping reward computation often requires forward kinematics and target tracking or measure to compute it. On the one hand, we do not want to assume the knowledge of kinematics in our work because it could vary during learning (in the real-world, some damages to the robotic structure or endcoder offsets can make kinematics vary) and suddendly make the reward function incorrect. On the other hand, target tracking is not something we want to avoid at any cost. It depends on the way the target is tracked on the images. We simply do not want to track a target with hand-crafted features. Indeed, the latter can efficiently operate on a fixed scene (even a complex one), but we have no idea if they are efficient for a scene with unexpected variations. If those features become wrong, the target tracking step as well as the reward function computation produce wrong results. Finally, if the shaping reward function depends on the manual measurement of a target, the policy is restricted to learn to reach or manipulate only at positions which have been measured, which makes the policy not valid for other target positions.

3.1.1 Partial conclusion

We have described manipulation robotics learning studies which were conducted before the emergence of deep reinforcement learning. We have noticed first that the state space dimensions are rather low. Second, we have raised the fact that some robotic tasks were learned using expert demonstrations or shaping reward terms depending on kinematics, target measure or hand-crafted features. We want to avoid these hypotheses in our work.

3.2 Learning manipulation tasks using deep reinforcement learning

In the following paragraphs, we describe manipulation robotics studies using deep reinforcement learning. We also focus on the requirements for the state and action space representation and the reward computation. We show both that many of these methods share common requirements with the previous ones and that modern methods which are interested in learning with sparse rewards require additional strategies (that we describe in the following). We divide this section into three parts. First, we briefly describe modern methods which learn manipulation robotics tasks using demonstrations. However, we do not aim at being exhaustive on this subject. Second, we describe DRL manipulation robotics tasks which use shaping rewards to guide robots towards goal state areas. Third, we relate some approaches which have used sparse rewards only to learn a manipulation robotics task.

3.2.1 Learning manipulation tasks using demonstrations

As previously mentioned, in the sole presence of sparse rewards, learning can be particularly slow in high-dimensional state spaces. By assuming the knowledge of

expert behaviours, demonstrations can be used to overcome this issue as an initialization to a reinforcement learning run or by conducting an apprenticeship learning routine. This paragraph describes state-of-the-art results of high-dimensional manipulation tasks using demonstrations. The goal of this paragraph is not to exhaustively list the manipulation robotics applications using demonstrations because this kind of methods assumes the presence of an expert, which is in contradiction with our unsupervised-related objectives. Instead, we want to show the potential of this class of methods. Thus, we give a brief overview of the most recent approaches using demonstrations on high-dimensional manipulation robotics tasks.

We can divide the use of demonstrations into two categories. First, demonstrations can simply be used as supervised targets. The agent tries to mimic the expert behaviour. This category is named "behaviour cloning" and has been used recently for complex manipulation tasks such as a block-stacking task [121], [122], a pick-and-place task [123] or a peg insertion task [124]. Note that this can be used to speed-up reinforcement learning if a reward function is designed, like in section 3.1 or in [123] and [124]. Indeed, in the case the robot has only access to sparse rewards, demonstrations can guide the robot towards high-success probability areas, which can alleviate the RL issue of the first success (described in 2.2.6.3).

The other category is inverse reinforcement learning (IRL) and assumes that no reward function is designed. Indeed, in opposition to reinforcement learning which outputs a policy from a reward function, IRL outputs a reward function from demonstrations. Then, the task can be learned based on the reward function. Numerous high-dimensional robotics applications have been implemented using this principle. For instance, [125] learns a pouring task, [126] and [127] learn a door-opening task, [128] learns to slide a box on a table to a given target area and [129] learns a gripper-pusher task.

3.2.2 Learning manipulation tasks using deep reinforcement learning with shaping rewards

This paragraph describes DRL manipulation tasks which are learned with reward functions using shaping reward terms. We present a table (Table 3.4) with the notations summarized in Tables 3.1 and 3.2. We notice several important facts:

- We observe first that the state space dimensions are considerably higher than before the emergence of deep learning. This makes policy learning significantly more complex.
- We notice that the action space dimensions become higher as well and the tasks are more complex for this reason.
- All the tasks using a shaping reward require forward kinematics to track the end-effector or some parts of it and the 3D information about a corresponding target pose (obtained by measure or tracking). Indeed, shaping reward functions are often defined as a decreasing function of a distance between a current pose and a target pose. For instance, [132], [131], [2], [134] and [135], [136], [137] compute a distance measure between a current and a target pose which gives an informative shaping reward provided that robot kinematics and target position are known. In a similar way, in [63] , [110] and [1], informative shaping rewards have been computed using a distance measure between current end-effector parts or manipulated object positions and their corresponding target positions which requires

Year	Paper	State	Action	Reward function	Task	Requirements
2015	[130]	$\sim$ 50000 | d	3 | d	sp | sh	2D reaching	FK,TT
2015	[63]	33 | c	7 | c	sh	inserting a shoe tree into a shoe	FK, TM, IRR, TR
2016	[131]	4096 | m	20 | d	sh | sp	grasping a cube	FK, TT, IRR, TR
2016	[132]	100 | c	24 | c	sh | pe	hand positioning and object manipulation	FK, TT, HVP
2016	[1]	$\sim$ 200000 | m	7 | c	sh	inserting a block into a shape sorting cube	FK, TM, IRR, TR
2016	[133]	$\sim$ 200000 | m	7 | c	sh	picking up a bag of rice using a spatula	FK, TM
2016	[110]	$\sim$ 250000 | m	7 | c	sh | pe	pick and place	FK, TM, UD, TR
2017	[2]	$\sim$ 120000 | d	7 | c	sh | sp	grasping	FK, TT
2017	[134]	180 | c	7 | c	sh | pe	pick and place	FK,TT
2017	[135]	$\sim$ 21200 | d	15 | d	sh	reaching	FK, TM
2017	[136]	$\sim$ 30 | c	9 | c	sh | sp	grasping	FK, TM, TR, IRR
2017	[137]	$\sim$ 30 | c	9 | c	sh | sp	block stacking	FK, TM

TABLE 3.4: Summary of manipulation robotics applications using deep reinforcement learning and shaping rewards. See Tables 3.1 and 3.2 for the notations.

knowledge of kinematics or non-trivial visual modules. Furthermore, a more sophisticated set-up has been proposed in [133]: the shaping reward is based on the distance between current visual features and target features, both of them being computed by an autoencoder. This requires to place the robot at the target position and extract target visual features each time the target location changes. Finally, to learn a reaching task, [130] uses an unusual kind of reward function which rewards 1 when the end-effector is closer to the target, -1 if it gets further and 0 if it does not make progress changes. However, to compute the distance between the end-effector and the target, forward kinematics and target tracking are required.

To conclude, these approaches present massive changes in terms of task complexity. However, they do not focus on making the reward functions independent of kinematics, target measures or hand-crafted visual modules. They mainly contribute towards deep reinforcement learning algorithms adapted to high-dimensional state space and continuous action spaces.

3.2.3 Learning manipulation tasks with deep reinforcement using only sparse rewards

This paragraph describes manipulation tasks which are learned with reward functions which only consist of a sparse reward function. We particularly enhance the strategies which make learning with sparse rewards tractable.

In [138], a real WAM robotic arm equipped with a Barrett hand learns to grasp objects in clutter using sparse rewards. The state space is composed of RGBD data and potential actions are generated for each state. There are two kinds of actions, pushing to make objects graspable and grasping. A grasping action consists of an approaching direction of the hand, the rotation angle of the wrist and the initial distance between the tips of the two fingers and the thumb. These parameters are generated based on centres of objects obtained by a hand-crafted segmentation. A pushing action has the same parameters as a grasping action with the exception of the three fingers which have to be aligned. Note that these kinds of actions require also to compute torque outputs using inverse kinematics and calibration parameters. To our understanding, simplifying the action space was the key of the success.

In [73], 14 robots learn in parallel how to grasp objects in clutter. The robots have seven degrees of freedom and a compliant two-finger gripper. For each robot, a camera is mounted behind the arm and can look at a box containing objects. The state space is composed only by images of the box (current and at the beginning of a learning episode) and the action space by the 3D end-effector translation and a sine-cosine encoding of the change of end-effector wrist angle (the gripper is made vertical). Sparse rewards are used to train a deep grasp predictor (7 convolutional layers), which outputs a probability value given a pair of images (as previously mentioned) and an action. Actions are generated using a cross-entropy method consisting in sampling and re-sampling actions with high probability of success. Several strategies make learning the full task with sparse rewards tractable. First, the action space is simplified by the used of inverse kinematics. Second, several robots learn in parallel, which make the probability of success higher. Third, for a given episode, each transition has the same reward which is the outcome of the episode (0 or 1), and the actions are re-computed using pose difference. This significantly increases the number of successful transitions but makes an assumption on dynamics (the scene is fixed and there are no obstacles).

In [59], very precise manipulation tasks such as inserting a key into a lock or putting a ring onto a peg are learned using sparse rewards. The task is learned from a large variety of initial states using "learning from easy missions" [139]. This class of methods, also called "curriculum learning" makes an agent or a robot learn easy tasks at the beginning of learning. During the latter, it focuses on tasks of increasing difficulty and ends-up mastering all the tasks including the most difficult ones. In this context, it it assumed that one element $s_g \in S_g$ of the goal state space is used. From this goal state, they first start to learn to reach s_g from close states. And, as learning progresses, initial states are generated further and further with respect to s_g. To our understanding, one of the limitations of the method comes from its key assumption: "one element of the goal state space is known". This assumes to measure or compute a target pose, which is difficult to ensure in all the robotic scenarii, e.g manipulation task in the sea bed.

In [136], among their diverse experiments, a brick grasping task is learned with a sparse reward with random brick and arm initialisations. The state space contains 9 (6 for the arm and 3 for the finger) joint angles and its associated velocities and the 6D pose of the brick (which is often unavailable for real-world tasks). The arm is always initialized with the end-effector close to the brick and well-oriented for grasping, i.e. the required move is just a translation of the end-effector followed by a grasp. This alleviates the issue of the probability of first success and makes the task much simpler. The focus is put on the issue of credit assignment with the following strategy: several RL updates per interaction between the agent and its environment are applied, which improves a lot the data-efficiency of the RL algorithm.

In [140], a planar pushing task is learned using a model-based RL method. The state space is composed of an RGB image, 5D pose of a gripper (3D Cartesian coordinates + yaw and pitch angles) and an action corresponds to a controlled 5D gripper pose. A high-dimensional dynamics model (convolutional and LSTM modules are involved) is used, predicting a next image from a previous and current image, a previous and current gripper pose, and a sequence of future actions. Formally, the way the goal is specified requires a human user to specify some pixels and to associate target positions to them. The actions are generated using an optimization planning procedure which involves the dynamic model: choose the sequence of actions which maximizes the probability of the designated pixels moving to their goals. Note that the method is interesting in it is allowed to do prediction only for the pixels of interest and not for the whole image (because each pixel coordinate is made independent of other pixel values). The way the goal is specified in this method makes the reward rather sparse. The main assumption of the method is that the goal of a manipulation task would have to be specified in terms of target pixel, which may not be suitable to all the manipulation tasks. For example, for a grasping task it is difficult to specify the goal with pixels and target locations.

In [141], a pick-and-place is learned using sparse rewards. The state and action spaces are simplified by the use of placing and grasping motion planning algorithms. The state space is composed of potential grasps obtained from a supervised RGBD grasp sensor, a set of allowed hand configuration placements, a grasp selected in a previous step, and a believed object position in the previous time-step. The action space is composed of chosen place and grasp actions and is discrete. The motion planning algorithms requiring dynamics as well as the black-box models (requiring RGBD hand-crafted processing blocks) to compute potential grasps make learning of the pick-and-place task tractable.

In [142], three manipulation tasks are learned (pushing a randomly located box to a random location, hitting a puck such that it slides and stops to a target location,

picking a randomly located object and place it at random location). For all these tasks, the initial position of the gripper is fixed. The state space is composed of robot joint angles and associated velocities as well as positions, rotations and velocities of all objects. The actions consist of 3D desired gripper Cartesian position and desired distance between two grippers, which requires inverse kinematics. The state space of this application is rather low-dimensional. However, the method used to learn the policy, "Hindsight Experience Replay" (HER), produces impressive results using only sparse rewards. This consists in extending off-policy DRL algorithms (DDPG included) by making them take goals as inputs (in the application, this is the object 3D Cartesian position). The idea of HER is to store transitions with the true goal and also with a subset of other goals. Several strategies were tested to select the subset of goals, the best one was to select k random positions which occur in the same episode as the considered transition. This strategy applied with DDPG resulted in very good performances while DDPG without it totally fails. Overall, the goal sampling is interesting and was a decisive factor to make the task learned. However, some modelling parts are present such as inverse kinematics and the knowledge of the object 3D position.

In [143], a complex block stacking task is learned in simulation using a hierarchical RL architecture which involves auxiliary and external sparse rewards. The state space is composed of a pair of images, proprioceptive information including arm and finger joint angles and velocities and end-effector position. The idea of this hierarchical architecture is to jointly learn external tasks (the targeted tasks) and auxiliary tasks. The latter are only used to provide good exploration samples for the external tasks and do not bias their policies. The auxiliary reward functions are all sparse in simulation (in real-world tasks, some shaping reward auxiliary functions are used) and give positive rewards if touch sensors are maximized or minimized, if objects are close to each other, if one object is higher or lower than the other one, if one object is on the left (resp on the right) of the other one. Some other auxiliary rewards output positive values if the end-effector touches something, if it does not touch anything, if it makes the object moving. These rewards are sparse, however, object-centric rewards (10 among 13) require object tracking in the two used camera images and are mainly valid for simple fully-coloured objects given the segmentation algorithms used in the experiments. To conclude with, this method is tractable because each of the auxiliary sparse reward positive values can be experimented with a relatively high probability, which makes reaching the external sparse reward positive values more likely afterwards.

3.2.4 Partial conclusion

In this section, we have reviewed manipulation robotics tasks using deep reinforcement learning. We have briefly described work on learning from demonstrations. We have summarized deep RL applications using shaping rewards and have noticed that many of these applications require forward kinematics as well as a way to know where the target is (by measure or tracking). We have also described deep RL manipulation applications which use only sparse rewards. We can enhance that the issue of data-efficiency arising with the sole use of sparse rewards was correctly addressed because presented papers have not major problems learning with them when they occur. The major difficulty is to make the first successes likely. To this end, the state and/or the action spaces can be simplified using inverse kinematics [140], [142], [73] or supervised pre-processing blocks [141], [138]. Furthermore, using several agents in parallel makes the probability of success more likely [73] but

requires expensive materials. And finally, decomposing a complex task defined by a sparse reward can be made tractable by learning meanwhile other simpler auxiliary tasks defined as well by sparse rewards [143].

3.3 Conclusion

In this chapter, we have presented manipulation robotics tasks using reinforcement learning. First, we have reviewed some approaches developed before the emergence of deep learning. We have concluded that the state spaces were rather low-dimensional and that many approaches used expert demonstrations or/and forward kinematics or/and hand-crafted visual modules. The emergence of deep learning has mainly changed the dimensionality of the problems since state and action spaces have become more high-dimensional. However, they still have major requirements regarding the way the goal is specified. Some approaches use expert knowledge or shaping rewards using kinematics to guide the exploration. Other studies use state and/or action space reduction techniques as well as costly computational and material resources to learn with sparse-only rewards. Furthermore, another solution is to divide the whole complex task into simpler tasks. This helps a lot to reduce the complexity of the whole task but does not imply to reduce the requirement of external knowledge for the sub-tasks.

In the next chapter, we will present our approach and we will describe how we can dispense with kinematics, hand-crafted visual modules and expert knowledge.

Chapter 4

Approach overview

In the previous chapter, we have reviewed deep RL work on manipulation robotics and we describe now an overview of our approach and how it relates to the state-of-the-art.

4.1 Learning reaching skills using binocular fixation and hand-eye coordination

In this section, we develop the idea which helps us partially overcome the need for supervision and we describe at the same time previous work from which we get inspiration.

4.1.1 Objective

For manipulation robotics tasks using deep reinforcement learning, the use of hand-crafted features for the state representation is limited. However, we have shown that the usual ways to specify a goal often require forward kinematics and/or supervised visual modules for shaping rewards, expert knowledge for imitation learning, a lot of material resources and/or state and action space reduction and/or knowledge of goal states for sparse-only rewards. Our objective is to learn a complex manipulation robotics task without forward kinematics, supervised visual modules, calibration parameters and expert knowledge. Specifically, we are interested in a palm-reaching task. A seven-degree-of-freedom robot has to touch with its palm an object, which is put on a table. This task can be viewed as a natural predecessor of grasping or other manipulation tasks and is interesting from this perspective. Furthermore, we are interested in learning to predict reachability while learning to reach using the gaze as the only input.

4.1.2 Idea

4.1.2.1 Development

The idea of our approach comes from the observation of human behaviour: to grasp an object, humans first look at it and then grasp it. Inspired by that fact, we design a stage-wise learning framework which is composed of three steps (as shown in Figure 4.1):

1. **Fixating objects:** the robot learns to fixate an object put on a table with a two-camera pan-tilt system. This step makes the robot localize the object in a 3D space.

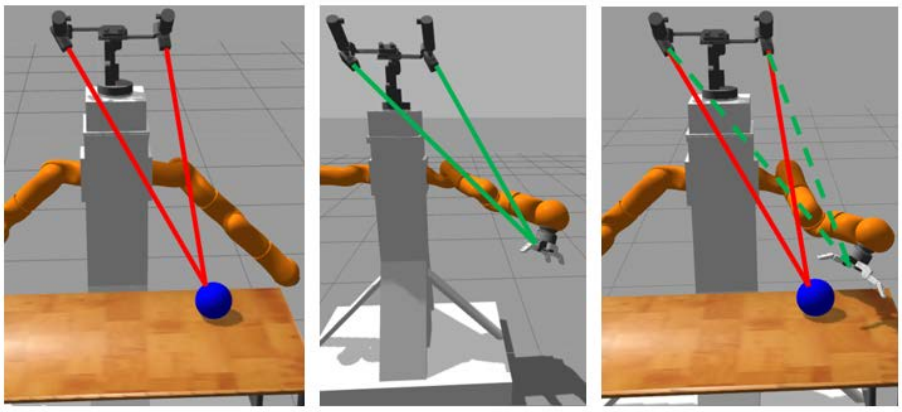


FIGURE 4.1: Scheme of the three steps of the stage-wise framework

2. **Fixating the end-effector and learning hand-eye coordination:** the robot learns to fixate (using the same stereo system) its own end-effector and meanwhile it learns a hand-eye coordination function. The latter allows at any time the robot to localize from its arm joint angles the end-effector position in terms of camera joint angles. The hand-eye coordination is considered as a proprioceptive ability.

3. **Touching:** from the knowledge acquired in the previous steps, i.e. the ability to localize objects by fixating them and the end-effector from the hand-eye coordination, the robot learns both to touch an object and to predict its reachability from its gaze.

Decomposing the whole problem into small sub-problems helps to learn the whole task in a most efficient way. However, as shown by [137] and [143], using a hierarchy of simpler tasks does not imply the absence of visual modules and forward kinematics for the considered sub-tasks. Our work focuses on reducing the required prior knowledge for the three considered tasks.

For the object fixation task, we use an autoencoder to learn to reconstruct the environment (with an unsupervised procedure) without object, i.e. through the images captured by the stereo system. Besides, we assume that if an object is added to the environment, it will be badly reconstructed. From this assumption, the object can be localized in the images because it corresponds to the area which has the largest pixel reconstruction errors. Using this anomaly localization principle, the robot learns (through deep reinforcement learning) to move its cameras such that the object is at the image center.

To learn the end-effector fixation task, we define periodic finger moves of the end-effector and localize the latter by using an image difference technique. Using this localization principle, the robot learns (through deep reinforcement learning) to move its cameras to bring the end-effector to the image center. Meanwhile, each time the end-effector is at the image center, a link is made between the arm joint angles and the camera joint angles. From these links, a hand-eye coordination mapping is learned (through supervised learning), and this allows to localize the end-effector from the arm joint angles. For this learning process, the labels are not given by humans but are found through experimentation.

The touching task is learned through reinforcement learning and uses a shaping reward directly depending on the knowledge of the two previous tasks: the camera

joint angles which make the stereo system fixate the object and the hand-eye coordination function are used. This task also assumes that the robot knows whether it is touching the object with its palm (knowledge of the sparse reward) and also whether it is touching the table.

To summarize, we list the assumptions used in our experiments per task:

1. No object is present when the autoencoder is encoding the environment and an object is added when the robot learns object fixation. Furthermore, there is no environment variation after the autoencoder training step.

2. Nothing is moving in the background.

3. We assume the robot has the tactile sensing ability to know whether it is touching the object with its palm and whether it is touching the table.

However, unlike prior work, we do not use forward kinematics, hand-crafted visual modules, expert demonstrations or other forms of external supervision. A more technical description is given at the end of the chapter.

4.1.2.2 Links to the human behaviour

We have developed our idea based on a basic observation of the human behaviour: humans generally fixate an object before manipulating it. We describe here how it is related to studies on reaching behaviour. We are interested first in knowing how a target location for reaching is encoded in the brain. Second, we briefly describe how infants learn saccades, hand-eye coordination, and some manipulation skills and we conclude on how much our method is related to the way infants learn.

How is a location encoded in the brain?

For an agent (human, primate or robot), reaching or manipulating an object requires encoding the 3D location of the object with respect to its body. Primates or humans can generally encode a 3D location with the binocular gaze direction. Indeed, as shown in a study on macaques [144], some neurons of the Brodmann area 9 (dorsolateral prefrontal cortex) are responsible for encoding variations of the head direction. Thus, the gaze direction together with the vergence angle of the eyes can encode a 3D location relative to the body. However, the gaze direction only encodes the location of those stimuli which are in the fovea. And, as humans can also reach for objects that are not in the fovea (e.g. expert jugglers use information in the periphery of vision to track juggling balls [145]), other ways of localization are involved. In a study on monkeys, [146] observed that 81 % of the neurons of the parietal reach region (in the posterior parietal cortex) encode extra-foveal locations in eye-centered coordinates. In other terms, these neurons are used to encode extra-foveal positions relative to the fovea.

In our work, we do not consider encoding locations of stimuli in extra-foveal areas. We only encode the location of the target in the foveal area. This position is only represented by the gaze of the robot, i.e. the camera joint angles, as the robot is fixating the object.

Infant learning

We discuss here how newborns may learn to reach targets and draw comparisons with our approach.

To start with, babies (and generally humans) learn their skills in form of sensori-motor mappings. This is the case in our work since the output of reinforcement learning is a sensori-motor mapping.

Besides, newborns may not learn any skills from scratch. They are provided with movements (called primitive reflexes) which help them to interact with the environment and acquire skills [147]. They progressively disappear as their sensori-motor skills improve and the brain becomes more mature. For example, the asymmetric tonic neck reflex [148] is a known movement that considerably helps babies to learn hand-eye coordination and reaching [149]. When the baby's head is turned to a side (left or right), his arm and leg from this side extend while his arm and leg from the opposite side flex. This set of moves may allow the babies to see their hands and to establish links between proprioception and vision.

Babies learn to saccade to visual stimuli directly after birth and it takes generally about four months [150] to reach a good control of the head and the eyes. However, as shown by [151], children are continuously improving this skill since their primary saccades are less precise than those of adults.

In the first few months, newborns try to reach stimulating targets (from vision and sounds) and can already make contact with objects at around 12 weeks. According to [152], the interesting point is that vision of the hand may not have an influence on these early reaching contacts. Indeed, in this study, infants make contacts with objects in two experimental conditions. In the first one, objects are presented in the light and in the second one, glowing or sounding objects are presented in complete darkness. The results show that infants reach and grasp objects at comparable ages for both environmental conditions ($\sim$ 12 weeks for reaching and $\sim$ 15 weeks for grasping). This indicates that infants do not seem to do visual correction for reaching at this age. From five months, infants may correct their reaching movements using vision and start to develop hand-eye coordination [153], [149].

With respect to these studies, we do not take into account that the learning of the tasks are intertwined e.g. reaching happens before a good development of proprioception. In a nutshell, infants autonomously learn what is the right sequence of tasks. In our case, by observing the human behaviour, we know that the target fixation and the hand-eye coordination function both help a lot to reach a target and we implement this sequence of tasks.

In our work, the first task consists in fixating environment anomalies that we consider as objects. Infants (and also adults) are interested in novel visual stimuli [154], [155] and we can argue that from this point of view, we respect this principle in that the novelties are the objects. However, we did not implement a curiosity mechanism [156] to make the robot continuously encode novel stimuli. Instead, we considered the environment as fixed and the objects the only novel stimuli. Furthermore, there is no evidence of prior encoding of the outside environment in the brain of infants.

The second task which consists in learning hand-eye coordination is biologically plausible in the sense it uses sensori-motor learning. However, the question about the nature of inputs and feedbacks is still open.

To conclude, to learn the object reaching task, we do not take into account the intertwining of the task learnings. Our approach takes inspiration from the emergent behaviour of humans (fixate the object, then manipulate it) and from its sensori-motor learning aspect.

4.1.3 Related work

The general idea of learning to reach with binocular fixations and/or hand-eye coordination has been implemented several times in the developmental robotics literature. In this section, we only describe applications which learn to reach with the help of binocular fixations and/or hand-eye coordination. The applications which learn reachability prediction skills are described in Chapter 7.

In [157], a six degree-of-freedom robot arm equipped with a stereo system learns to fixate and to grasp an object. To do so, a fixation controller is approximated using a neural network which is trained from random saccade moves using supervised learning. Afterwards, a recurrent neural network is trained to learn the grasping controller using supervised learning and requires inverse kinematics to compute the targeted arm postures. Moreover, the targets are detected via a hand-designed visual module. In [158], the same experimental environment is used. The object localization in the images is still computed with a simple color-segmentation algorithm and no hand-eye coordination is computed since the robot learns to grasp using some solutions of the robot inverse kinematics.

In [159], a seven degree-of-freedom robot is used jointly with a four degree-of-freedom stereo system. The latter is used to fixate a target in the image using a proportional command (requiring to tune gains). In this study, a hand-eye coordination function is learned, mapping arm joint angles to camera yaw, pitch and vergence angles, which is close to our hand-eye coordination function. Reaching is learned by approximating the inverse Jacobian (this allows to compute the arm joint angle velocities given the hand position in the image) with a neural network. This Jacobian is used to bring the hand to the image center. However, most samples required to learn the hand-eye mapping and the inverse Jacobian were collected offline. This issue is addressed in [160] where the hand-eye coordination mapping and the inverse Jacobian are jointly learned in an online process. The main requirement implied by the method is the visual knowledge of the target and the hand model. Indeed, a color attention mechanism is used for the hand and target detections.

In [161], target fixation and reaching are learned using radial basis functions and supervised learning. The main contribution of the paper is the fact that the presented algorithm is biologically plausible. However, the target and the end-effector are still segmented with a color segmentation algorithm and the use of markers.

In [150], target fixation, reaching and torso controllers are learned. The method focuses on making the learning architecture human-like. The targets are detected as salient areas using a novelty principle. In principle, such a method suits our requirements provided no hand-crafted feature is used. However, to our understanding, it seems that a model of proprioception is given to the robot: "firstly a series of arm movements are generated with proprioceptive information to model movements learnt prenatally".

To summarize, the idea of learning a reaching task using stereoscopic fixation and hand-eye coordination has been explored by the developmental robotics community. The majority of these papers focus on making the learning architecture human-like. In our case, we take inspiration from this kind of work because object fixation and hand-eye coordination efficiently help to learn object reaching. However, as stated in Chapter 3, we want to get rid of external information such as forward/inverse kinematics or hand-designed visual modules. This is not fully achieved in the mentioned developmental robotics papers because the objects or/and effectors are often detected using a basic color segmentation algorithm. Our approach is to design a stage-wise reinforcement learning framework where all the

tasks are learned one by one using weakly-supervised reward shaping terms. The next section describes in more technical details the chosen methods to overcome the use of kinematics or visual modules.

4.2 Technical overview for the learning of reaching skills

In this section, we present a more technical overview of the tasks with the mathematical notations that we use for the following chapters (see Figure 4.1 and 4.2 for a schematic view of the stage-wise learning process).

For our work, we use a 7 DOF arm with a pair of cameras as shown in Figure 4.1. The task consists in touching an object on a table with the end-effector palm of the robot. In the following, we use the notations:

- $\mathbf{I} = (\boldsymbol{I}^{\mathbf{left}}, \boldsymbol{I}^{\mathbf{right}})$ represents the images from the left and right cameras.
- $\mathbf{q} = (\boldsymbol{q}^{\mathbf{camera}}, \boldsymbol{q}^{\mathbf{robot}})$ represents the 3 camera joint angles (one common tilt angle and two independent pan angles) and the 7 robot arm joint angles.
- $\boldsymbol{c_b}$ is a vector composed of 8 binary values associated with 8 areas of robot fingers. The 8 areas correspond to the proximal, medial and distal areas of the three fingers, with the exception of the proximal area of one finger which is linked to the palm. One binary value becomes 1 when its associated area is in contact with the object and 0 otherwise (see Figure 4.3 for an illustration).

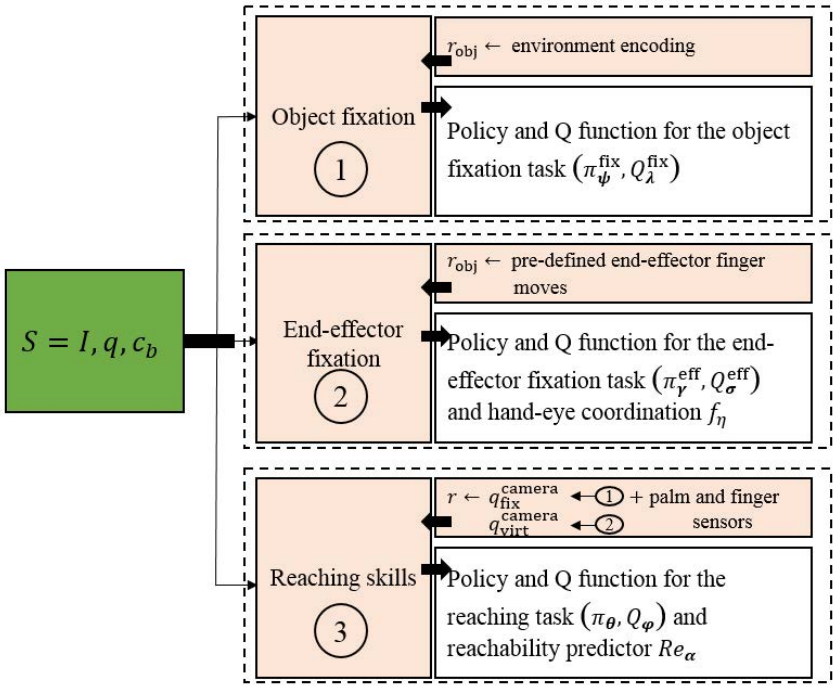


FIGURE 4.2: Overall scheme of the reaching skill learning procedure. Greek subscripts represent neural network parameters.

As mentioned above, the proposed method involves three successive tasks. First, the robot learns from raw pixels to fixate the object with a two-camera system. The

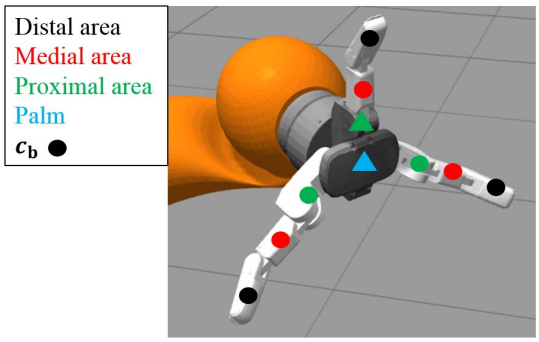


FIGURE 4.3: Representation of the areas of the robot fingers. The 8 areas associated with circles correspond to the binary values of the vector $\boldsymbol{c}_\mathbf{b}$.

resulting policy outputs from raw images $\mathbf{I}$ and camera joint angles $\boldsymbol{q}^{\mathbf{camera}}$ variations of camera joint angles $\boldsymbol{\Delta q}^{\mathbf{camera}}$. At the end of the fixation, the camera system coordinates $\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{fix}}$ implicitly encode the object position in 3D space.

Second, the robot learns a hand-eye coordination function f_η which maps robot joint coordinates to virtual camera coordinates:

$$\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{virt}} = f_\eta(\boldsymbol{q}^{\mathbf{robot}}). \tag{4.1}$$

These virtual camera coordinates correspond to the camera coordinates which would make the camera system look at the end-effector. The hand-eye coordination function is learned while learning the end-effector fixation policy, which has the same types of input-output pairs as the object fixation policy.

Finally, for the third task, a reward signal using $\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{fix}}$ and $\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{virt}}$ to make the end-effector close to the object is computed. It is combined with a sparse reward, indicating if the end-effector palm touches the object or not and a term penalizing contacts between the end-effector and the table (which assumes that the robot has the touching ability to distinguish the object from the table). While learning to touch the object, a reachability prediction network Re_α is learned using $(q^{\text{camera}}_{\text{fix}}, \text{Success})$ as input-output pairs. Indeed, as $\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{fix}}$ encodes the 3D location of a point, it should be a sufficient criterion to decide whether a point is reachable. In the following chapters, we describe each of the three steps in details with experiments.

4.3 Conclusion

In this chapter, we introduced our proposed approach to learn a touching (or pre-grasping) task using weak requirements about kinematics and the visual scene and without expert knowledge. To that end, we divide our whole complex reaching task into several sub-tasks. The way we divide the reaching task is inspired from the human behaviour and existing developmental robotics approaches. Humans generally fixate first an object before grasping it. Thus, the robot first learns to fixate objects. Second, it learns a hand-eye coordination function by learning to fixate its end-effector. And then, it learns to reach the objects. In addition to the specific division of tasks which is common to existing developmental robotics methods for reaching, our sub-tasks have the advantage of requiring only minimal prior knowledge. Using only little supervision for the goal specification of a complex manipulation skill

is a contribution itself with respect to the state of the art. The other important contribution is that we have designed a coherent deep reinforcement learning framework for the learning of the reaching task from raw sensor values, i.e. robot and camera joint angles, tactile finger sensors, and image pixels.

The following chapters describe each of the learned tasks: Chapter 5 describes the way the binocular fixation of objects is learned as well as the results in simulation and in the real world. Chapter 6 relates how the hand-eye coordination function is learned and the results in simulation. Chapter 7 presents how the knowledge of the two prior tasks are combined to learn the whole reaching task and the reachability prediction skill.

Chapter 5

Learning binocular object fixations using an anomaly localization principle

This chapter presents our proposed approach to solve the object fixation task. We first present our method from a broad perspective. Then, the method is technically explained. After that, we present experiments on simulated and real environments and discuss the results.

5.1 Introduction

As mentioned in chapter 4, the object fixation task consists in bringing the object to the center of each image (left and right) captured by the camera system as shown in Figure 5.1. Our goal is to make the robot learn this task with as little supervision

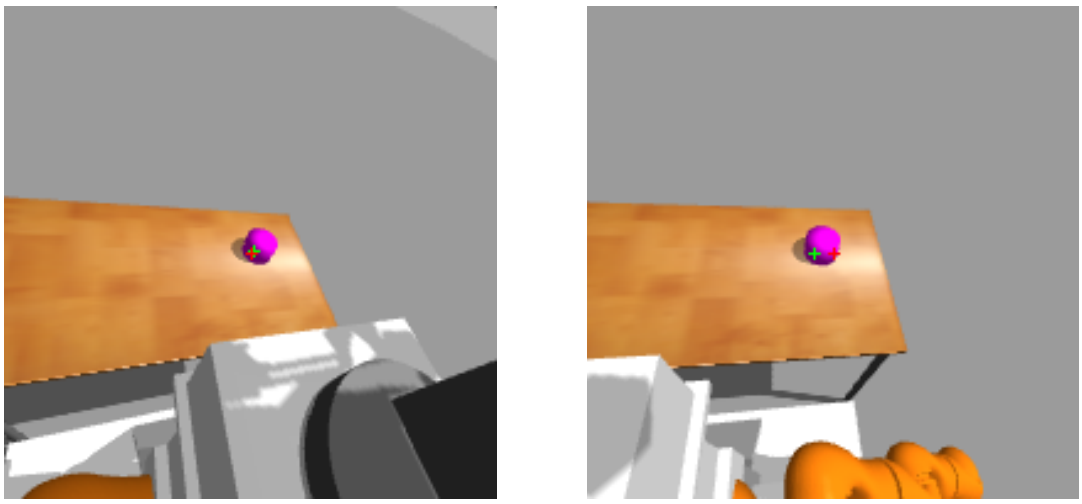


FIGURE 5.1: Binocular fixation achieved on a purple cylinder, the red cross stands for the image center, the green cross is the estimated object pixellic position according to the method described in 5.2.2

as possible. For this purpose, we use a deep reinforcement algorithm (DDPG) to dispense with the use of hand-crafted features for the state representation. In addition, we also design a shaping reward function which requires minimal supervision, i.e. without supervised visual object detection modules or calibration parameters. To

design this reward function, we propose to use a weakly-supervised anomaly localisation mechanism. In our approach, the object is viewed as an anomaly with respect to the agent's knowledge about the environment. The goal is to locate it in the image and then bring it to the image center.

Our object localisation principle is close to the one used in [25] where learning to detect network intrusions and breast cancers is proposed. In this paper, a multilayer perceptron is used as an autoencoder to reconstruct inputs labelled as "with anomalies" or "anomaly free". A positive aspect similar to ours is that it allows to locate the anomaly. However, these applications are rather low-dimensional since no more than 50 neural network inputs have been used in the experiments. [162] uses support vector machine methods to learn how to detect outliers in digit images. For each kind of digit, a support vector machine is trained to model a density. Abnormal images are detected when they are incompatible with the trained density. Similar to [25], the problem is rather low-dimensional since it involves 256 binary dimensions. In our case, the anomaly localisation concerns 2500 raw pixels (50×50 images).

5.2 Methods

5.2.1 Task definition

The problem is modelled as a Markov decision process where:

- $\boldsymbol{S} = (\boldsymbol{I}^{\textbf{left}}, \boldsymbol{I}^{\textbf{right}}, \boldsymbol{q}^{\textbf{camera}})$. The set of states involves the two RGB left and right images ($\boldsymbol{I}^{\textbf{left}}$ and $\boldsymbol{I}^{\textbf{right}}$) and the camera joint angles $\boldsymbol{q}^{\textbf{camera}} \in \mathbb{R}^3$. The camera joint angles $\boldsymbol{q}^{\textbf{camera}} = (\boldsymbol{q}^{\textbf{panLeft}}, \boldsymbol{q}^{\textbf{panRight}}, \boldsymbol{q}^{\textbf{tilt}})$ are represented by two pan angles for each camera and one joint tilt angle. Note that the state space contains only observations at a given time-step t (prior observations are not included).

  With a perfect object detector, the object pixellic position could be used directly as a state instead of images. However, impulse noise (explained in 5.2.3) affecting the object detection makes this impractical. Learning a direct mapping from images to actions solves this issue and avoids the need for any object detector during the exploitation.

- A, the set of actions comprises the three variations of camera coordinates (three continuous scalars)

  $\boldsymbol{A} = \boldsymbol{\Delta q}^{\textbf{camera}} = (\boldsymbol{\Delta q}^{\textbf{panLeft}}, \boldsymbol{\Delta q}^{\textbf{panRight}}, \boldsymbol{\Delta q}^{\textbf{tilt}}) \in \mathbb{R}^3$

The decision process we use is modelled as Markovian because we use a deep reinforcement learning framework theoretically relevant for MDPs. However, it is Markovian under some conditions. Indeed, the Markov property specifies that from a state-action pair, the next state can be predicted. In our case, from current images, camera joint angles and variations of camera joint angles, next images and camera joint angles have to be predictable. It is true for camera joint angles, but not necessarily for images. When cameras move, some environment areas appear and other ones vanish. The latter can be predicted but the former is predictable only if the environment is immobile. We ensure that this condition is valid in our experiments.

5.2.2 Reward computation

The rationale of the reward computation is to estimate first the object localisation in the images (using weak supervision) and then encourage it to be at the image

center. To localize the object, we learn first the environment images without object. Then, the object is localised in the image by considering it as an anomaly with respect to the environment encoding.

Thus, the reward computation involves three steps:

- A pre-training step in which the agent learns to reconstruct the environment without object
- An object detection step
- The reward computation step itself

In the following paragraphs, the superscript cam represents the camera left or right.

Pre-training step

For each camera cam $=$ left or right, 10 000 camera configurations are generated leading to two 10 000-sized image databases D^{left} and D^{right}. The set of camera configurations covers a regular grid of configurations between the joint limits (arbitrarily fixed to keep the table inside the field of view).
The images are converted from RGB 200×200 format to grayscale 50×50 images. Then, the autoencoder A_{μ}^{cam} is trained on D^{cam} with the help of the Adam solver [31].

Object detection

The object detection step takes into account the reconstruction error map (see Figure 5.3 for examples of reconstruction error maps). Objects are badly reconstructed since autoencoders are not trained on them. As previously mentioned, it makes the reconstruction error localized at the object position. This feature allows to estimate the object pixellic position. We choose to estimate it with a Gaussian kernel density estimator.

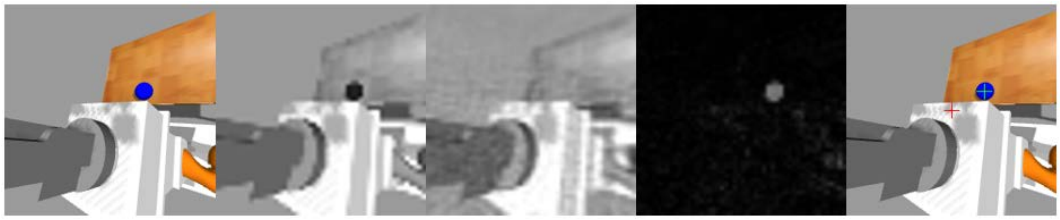


FIGURE 5.2: Object detection computation scheme

Figure 5.2 shows the different steps of the reward computation:

- The image is downsampled and converted into a grayscale one $\boldsymbol{I}^{\text{cam}}_{\text{origin}} \rightarrow \boldsymbol{I}^{\text{cam}}_{\text{gr}}$.
- The image is reconstructed using the learned autoencoder $\boldsymbol{I}^{\text{cam}}_{\text{gr}} \rightarrow \hat{\boldsymbol{I}}^{\text{cam}}_{\text{gr}} = A^{\text{cam}}_{\mu}(\boldsymbol{I}^{\text{cam}}_{\text{gr}})$.
- The error map is computed $|\boldsymbol{I}^{\text{cam}}_{\text{gr}} - \hat{\boldsymbol{I}}^{\text{cam}}_{\text{gr}}|$.
- From the error map, the N points $\{\boldsymbol{x}_i\}_{i\in\{1,\dots,N\}}$ with the highest intensity are extracted. $\{L_i\}_{i\in\{1,\dots,N\}}$ is the set of corresponding luminances.

From these points, a probability distribution $\{p_i\}_{i\in\{1,...,N\}}$ is computed using a kernel density estimator with a Gaussian kernel of zero mean and unit variance:

$$\forall i \in \{1,...,N\}, p_i = \frac{1}{N}\sum_{j=1}^{N} L_j K(\boldsymbol{x_i} - \boldsymbol{x_j}), \tag{5.1}$$

with $K(\boldsymbol{x_i} - \boldsymbol{x_j}) = \frac{1}{2\pi}\exp^{-0.5||\boldsymbol{x}_i - \boldsymbol{x}_j||_2^2}$.

The estimated object pixellic position $\boldsymbol{x}_{\mathbf{obj}}^{\mathbf{cam}}$ is at the maximal density function value:

$$\boldsymbol{x}_{\mathbf{obj}}^{\mathbf{cam}} = \boldsymbol{x}_k, \tag{5.2}$$

where $k = \arg\max_i(p_i)$. N is set to 150 in the experiments.

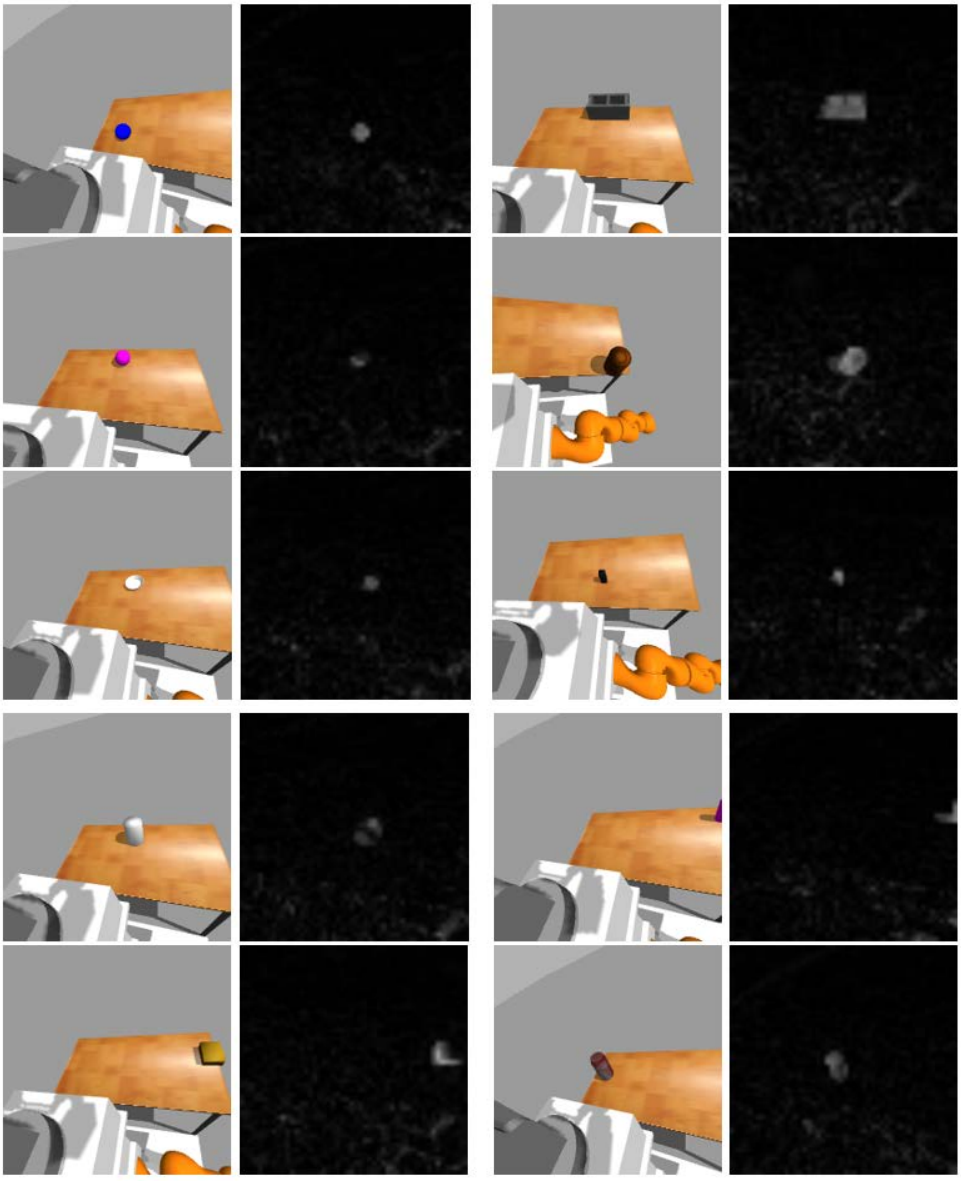


FIGURE 5.3: Examples of autoencoder reconstruction error for different objects of the training set.

Reward computation

The reward is a decreasing function of the Euclidean distance $||\boldsymbol{x}_{\mathbf{obj}}^{\mathbf{cam}} - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{cam}}||_2$. We propose to use an affine function which has a maximal value of 1:

$$r^c = 2 \times \left(\frac{L - ||\boldsymbol{x}_{\mathbf{obj}}^{\mathbf{cam}} - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{cam}}||_2}{L} - \frac{1}{2} \right), \tag{5.3}$$

where L is a constant value (equal to 100.5 in our experiments). Figure 5.4 plots the reward signal in function of $||\boldsymbol{x}_{\mathbf{obj}}^{\mathbf{cam}} - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{cam}}||_2$.

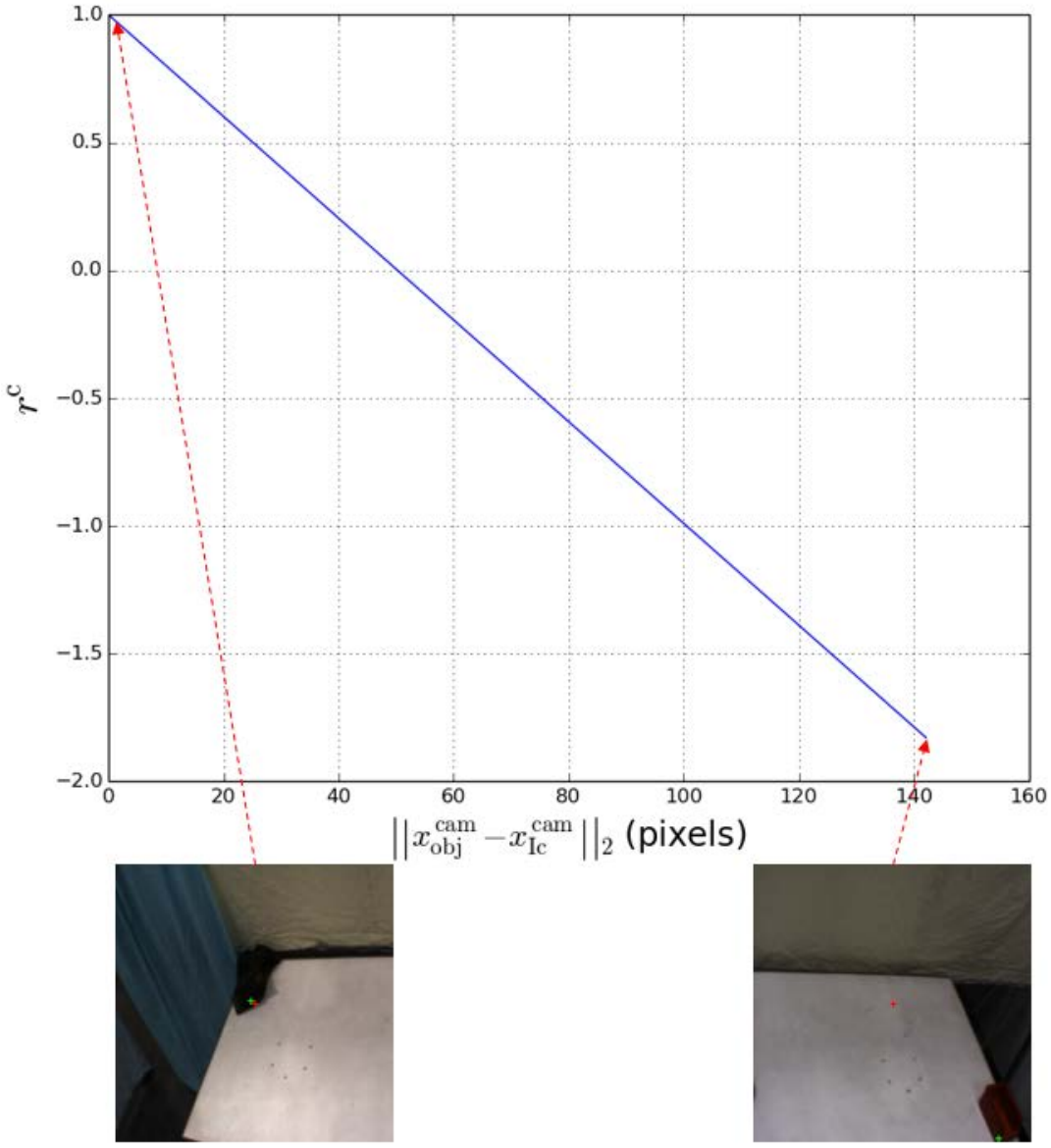


FIGURE 5.4: Plot of the reward signal in function of the distance between the object localization and the image center

The total reward signal is the sum of the left and right reward values:

$$r = r^{\text{left}} + r^{\text{right}}. \tag{5.4}$$

As previously mentioned, this method allows to extract a reward signal in a weakly supervised way. Indeed, the only supervision is the assumption that there is no object in the environment encoding step, and an object in the object fixation step. Moreover, this reward is informative in the sense it can discriminate the values of nearby states.

5.2.3 Mitigating noise

Considered on isolation, the results of our anomaly localisation is affected by impulse noise. The latter is present in the object position estimation because the learned autoencoders do not reconstruct perfectly the images even without object in the scene. Indeed, some high-frequency areas such as table legs are difficult to reconstruct. Consequently, high-frequency areas are sometimes detected, producing impulse noise in the reward function (see Figure 5.5 for an example in a real setting). An impulse noise present in a reward function can damage learning as it will be verified in the experimental part.

Dealing with impulse noise in the reward function is not often considered in deep reinforcement learning papers. To deal with noisy rewards, [163] used a moving average filter on the temporal reward signal with good results at removing impulse and Gaussian noise in a low-dimensional gridworld problem. This method can be applied to high-dimensional RL problems but it would modify noiseless rewards as well. More recently, [164] studied the influence of noise on the state-action value function overestimation. They developed a method to cancel this effect, penalizing unlikely actions in the update given prior knowledge. This method cannot be integrated in our work since we want as little prior information as possible. In our method, we choose to apply a learning approach to remove the reward impulse noise.

We choose to model the function m to detect an abnormal object detection variation in function of the movement amplitude:

$$\Delta \mathrm{d} = m(||\boldsymbol{\Delta q}^{\mathbf{camera}}||_2), \tag{5.5}$$

where Δd is the Euclidean distance in the pixel space between two successive object detections (note that Δd is averaged over the left and right images: $\Delta \mathrm{d} = 0.5(\Delta \mathrm{d}^{\mathrm{left}} + \Delta \mathrm{d}^{\mathrm{right}})$). For example, we want to model that if the cameras move a little, the object

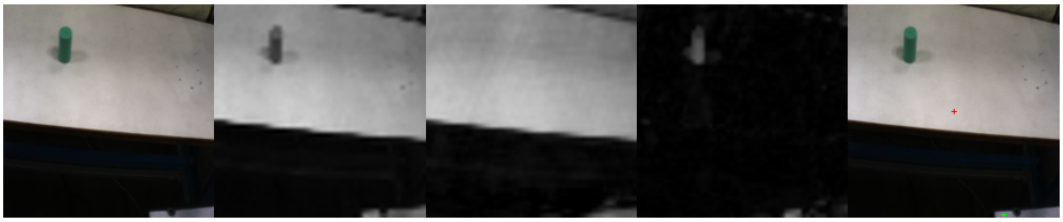


FIGURE 5.5: Example of impulse noise in the object localization process in the real environment. At the bottom right of the original image, a high-frequency area is not well reconstructed and is detected.

detection should move a little as well. Therefore, if in this case the object detection moves a lot in one of the images, this would be considered as abnormal and the RL transition is not added to the memory buffer.

The function m can be approximated by any regression method including neural networks, Gaussian processes or support-vector machines. Here, a Gaussian process with a squared exponential interaction function is used and trained with 1 000 transitions. Using it allows to remove transitions whose difference (in pixels) between the real detection variation and the estimated detection variation is above a threshold.

Note that this method does not theoretically allow to remove all the noise. Indeed, two successive bad detections imply a false positive and a bad detection followed by a good detection produces a false negative. From the observation of successive object localisations, we have noticed that the false positives almost never happen while false negatives are more frequent (though still seldom).

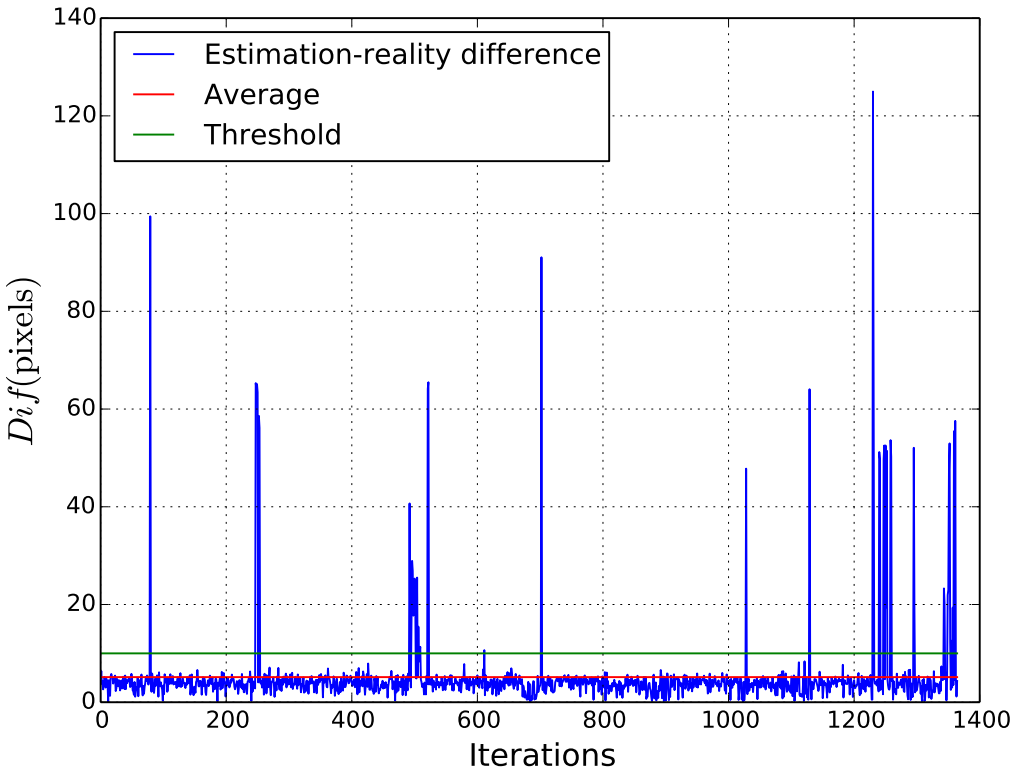


FIGURE 5.6: Evolution of the error in object motion estimation.

Figure 5.6 illustrates the partial noise removal results and reveals the impulse nature of the noise. We display (in blue) the difference Dif between the estimated and the experimented (Δd^{exp}) object detection variations:

$$Dif = |\Delta d^{\text{exp}} - m(||\mathbf{\Delta q}^{\mathbf{camera}}||_2)|. \tag{5.6}$$

All the transitions whose Dif is superior to the threshold ($\nu = 10$) are not added to the memory buffer (3 % of the transitions are removed).

5.3 Experiments

This section presents the experiments made to evaluate our method. They contain three main objectives:

1. We want to evaluate the learning of binocular fixations with our weakly supervised reward and compare it to the learning made with a reward depending on external information like calibration parameters.
2. We want to verify that the impulse noise affects learning and that our method helps to deal with this issue.
3. We want to make a learning run with our weakly-supervised reward on a real setting and evaluate its performances.

5.3.1 Experimental environments

We describe here the environments we use for our experiments.

5.3.1.1 Simulated environment

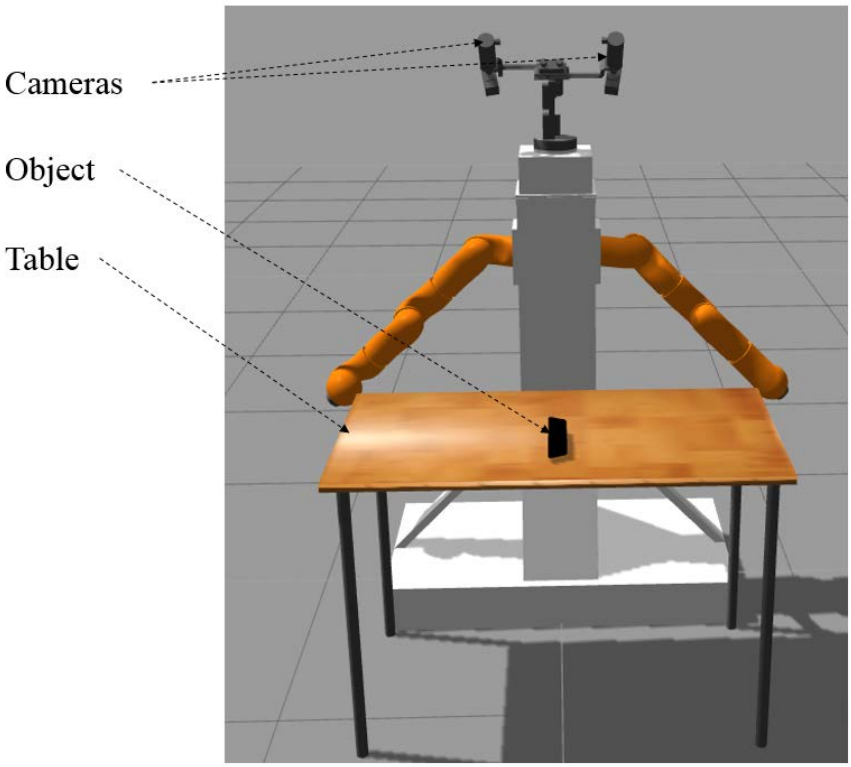


FIGURE 5.7: Simulated robotic platform

We use the Gazebo simulator jointly with the ROS middleware [165]. A two-cameras system is mounted on a robotic platform (cf Figure 5.7). A table from the Gazebo database is placed below the cameras and an object is put on it. The experiments involve two object sets. The training and the test sets can be seen in Figure 5.8. The databases consist of objects from the Gazebo models and hand-designed objects. In order to potentially generalize the fixating skill to new objects, the training set objects were designed with diverse colors and shapes (cylinder, parallelepiped, and sphere). Every test object has a new shape compared with training objects and a new color (turquoise) appears in the test set. The goal is to check if the learned policy can achieve fixations on objects whose color or shape is not present in the training set.

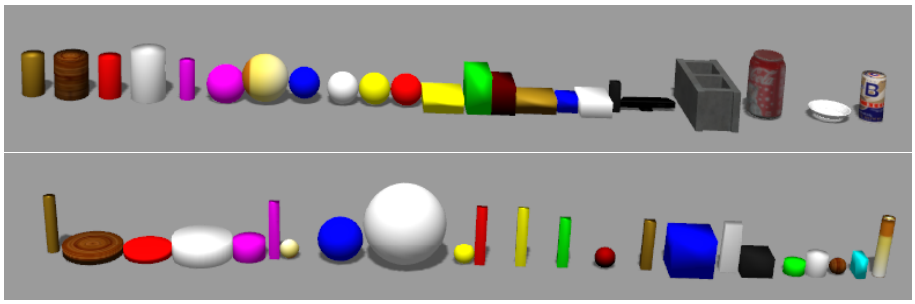


FIGURE 5.8: Training and test sets

5.3.1.2 Real environment

The experiments are conducted on a robotic platform (see Figure 5.9). Even though there are similarities between the environments, the real scene is significantly different. The pan-tilt system has the same robotic architecture as the simulated one. However, the system used in the experiments has a defect. Indeed, there is a an offset on the tilt angle of the right camera, which implies a vertical offset on the left and right images. It is illustrated in Figure 5.10. This offset could be cancelled if

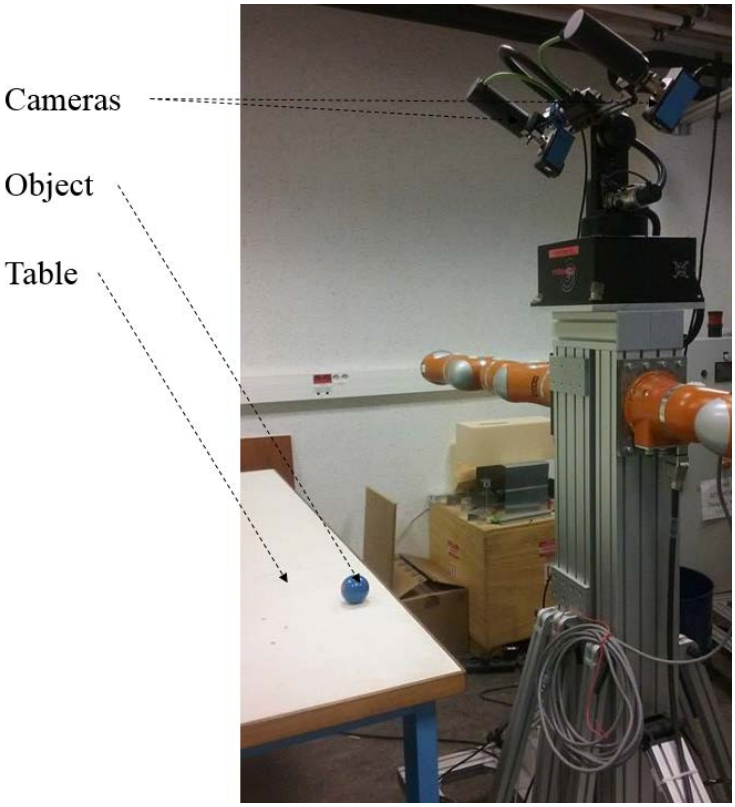


FIGURE 5.9: Real robotic platform and setting

the left image is cropped from the bottom and the right image is resized. However, we decided not to pre-process the images and to feed the neural networks with the raw pixels. Indeed, we made the hypothesis that the camera system would find an intermediate solution itself through learning. In other terms, the reward function can still make the camera system learn to put the object between the red crosses on the vertical axis.

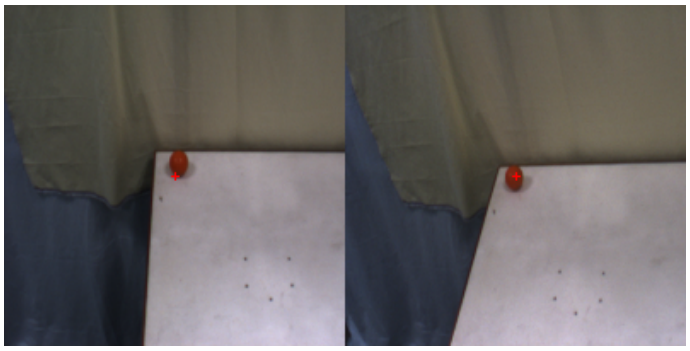


FIGURE 5.10: Two images captured at the same time by the Pan-tilt system. The red cross is at the image center and we observe a vertical offset between the two images.

The training and test sets (see Figure 5.11) are built based on the same principle as for the synthetic platform. The training set is composed of objects of the everyday life (with various colors and shapes) such as a key ring, juggling balls, diverse cards, or battery charger. The test set is composed of objects which are visually very different compared with objects of the training set. There are umbrellas, sun glasses, a plastic bag, a book etc. The goal is not to learn to fixate any object but rather to evaluate how much the policy can generalize with few objects.

FIGURE 5.11: Training (top) and test (bottom) sets

5.3.2 Implementation details

For all the neural network algorithms, we use the caffe library [166]. The neural network structures can be consulted in Appendix A. Batch normalization [167] is used to normalize layer inputs and avoid burdensome pre-processing steps.

Hyperparameter values mentioned in Algorithms 1 and 2 (presented in the next section) are listed in Table B.1 of Appendix B. They are valid for the simulated and the real environments.

5.3.3 Experiments in simulation

We describe the experiments made in simulation. First, we compare training performances of our setting with a supervised one. Second, we evaluate the resulting policies. Finally, we evaluate the object localisation accuracy.

5.3.3.1 Policy training

Algorithm 1 presents the binocular fixation learning procedure which uses an episodic set-up. In order to avoid the problem of correlated data, the object and its position regularly change according to a uniform probability distribution. Note that instead of using the Ornstein-Uhlenbeck process for the exploration as proposed in [95], a constant zero-mean Gaussian noise is used. With an informative reward function, this simple exploration setting is sufficient for learning the fixation task. The number of iterations of the algorithm is set to 200 000 which corresponds to 5 715 episodes.

In order to ensure a fast learning, two mechanisms are implemented in the training procedure:

- We add a term to the reward penalizing both divergence and a too strong convergence. More precisely, if $d_{\text{pan}} = q_{\text{panLeft}} - q_{\text{panRight}}$ is the angular difference between the pan left and right angles, then the additional reward term r_{pan} (see Figure 5.12) is computed as an affine function of d_{pan} when $d_{\text{pan}} > 0$ (eye divergence) or $d_{\text{pan}} < -0.4$ (strong eye convergence):

$$r_{\text{pan}} = \begin{cases} \frac{-d_{\text{pan}}}{3}, & \text{if } d_{\text{pan}} > 0 \\ \frac{d_{\text{pan}}+0.4}{3}, & \text{if } d_{\text{pan}} < -0.4 \\ 0.1, & \text{otherwise.} \end{cases} \tag{5.7}$$

- At each episode end (after N_{eps} reinforcement learning iterations), the camera system is set to the same initial position.

The objective of the experiments on training is twofold. First, we want to show how the impulse noise affects learning. Second, we want to demonstrate that learning with the filtered weakly supervised reward gives similar training performance as with a supervised reward. The supervised reward is obtained by computing the Euclidean distance between the projection of the object center of gravity and the image center $||\boldsymbol{x}_{\mathbf{p}}^{\mathbf{cam}} - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{cam}}||_2$. This function is also affine with the same parameters as for the weakly supervised one:

$$r_{\text{sup}}^{\text{cam}} = 2 \times \left(\frac{d_{\max} - ||\boldsymbol{x}_{\mathbf{p}}^{\mathbf{cam}} - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{cam}}||_2}{d_{\max}} - \frac{1}{2} \right), \tag{5.8}$$

$$r_{\text{sup}} = r_{\text{sup}}^{\text{left}} + r_{\text{sup}}^{\text{right}}. \tag{5.9}$$

It is supervised in the sense it requires to compute the camera extrinsic parameters at each iteration and to do a calibration before learning (to estimate camera intrinsic parameters).

Algorithm 1 Object fixation training procedure

Parameters:

N_{trans}: the size of the circular buffer
γ: the discount factor
$\boldsymbol{q}_{\mathbf{0}}^{\mathbf{camera}}$: the initial position
ϵ: the Gaussian noise variance
N_{tot}: the total number of iterations
N_{eps}: the number of iterations per episode
N_{b}: the number of transitions per batch
ν the removal threshold
N_{gp}: the number of samples required to the train Gaussian process

Inputs:

$\mathcal{D}_{\text{train}}$: the object training set

Outputs: $Q_{\boldsymbol{\lambda}}^{\text{fix}}$, $\pi_{\boldsymbol{\psi}}^{\text{fix}}$

Steps:

Set $t \leftarrow 0$
Initialize $Q_{\boldsymbol{\lambda}}^{\text{fix}}$, $\pi_{\boldsymbol{\psi}}^{\text{fix}}$ and $T_{\text{buf}} = \varnothing$
while $t < N_{\text{tot}}$ **do**
    Choose a random object in $\mathcal{D}_{\text{train}}$ and place it randomly
    Go to the initial position $\boldsymbol{q}_{\mathbf{0}}^{\mathbf{camera}}$
    Set $t_{\text{eps}} \leftarrow 0$
    while $t_{\text{eps}} < N_{\text{eps}}$ **do**
        Apply $\boldsymbol{a_t} = \pi_{\boldsymbol{\psi}}^{\text{fix}}(\boldsymbol{s_t}) + \mathcal{N}(0, \epsilon)$
        Observe $\boldsymbol{s_{t+1}}$ and compute $|\Delta \text{d}_t^{\text{exp}}|$
        Compute r_t using equations (5.4) and (5.7)
        $cond \leftarrow (t < N_{\text{gp}})$ or $((|\,|\Delta \text{d}_t^{\text{exp}}| - m(||\boldsymbol{\Delta q}^{\mathbf{camera}}||_2)| < \nu)$ and $(t > N_{\text{gp}}))$
        if *cond* **then**
            Add $< \boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}} >$ to T_{buf}
        Pick randomly N_{b} transitions from T_{buf}
        Update $Q_{\boldsymbol{\lambda}}^{\text{fix}}$ and $\pi_{\boldsymbol{\psi}}^{\text{fix}}$ using DDPG (equations (2.46) and (2.48))
        if $t = N_{\text{gp}}$ **then** Train $\Delta \text{d} = m(||\boldsymbol{\Delta q}^{\mathbf{camera}}||_2)$
        $t \leftarrow t + 1$
        $t_{\text{eps}} \leftarrow t_{\text{eps}} + 1$
    Withdraw the object

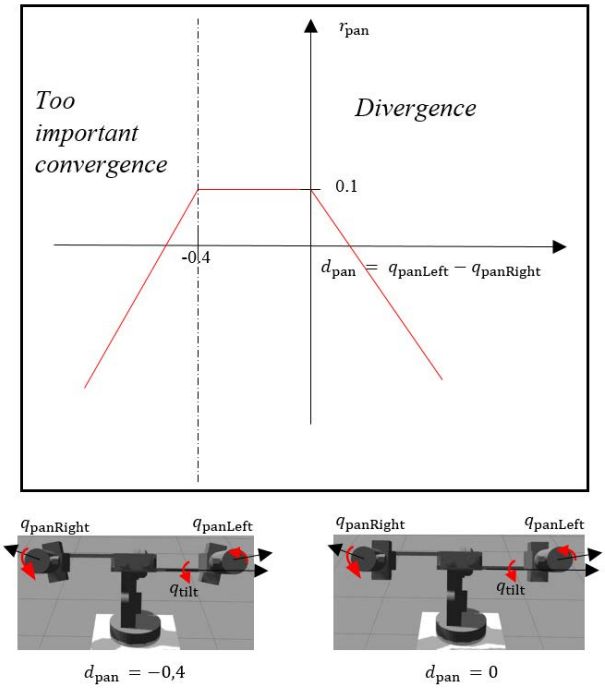


FIGURE 5.12: Visualisation of the r_{pan} computation

To achieve the objectives, learning improvements with the supervised, noisy weakly supervised and filtered weakly supervised rewards are compared. The fixation error is tracked over time:

$$e_{\text{p}}(t) = \frac{||\boldsymbol{x}_{\mathbf{p}}^{\mathbf{left}}(\boldsymbol{t}) - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{left}}||_2 + ||\boldsymbol{x}_{\mathbf{p}}^{\mathbf{right}}(\boldsymbol{t}) - \boldsymbol{x}_{\mathbf{Ic}}^{\mathbf{right}}||_2}{2}. \tag{5.10}$$

$e_{\text{p}}(t)$ represents the average (over the left and right images) Euclidean distance between the image center and the projection of the object center of gravity. We average three experiments for each case.

The results are shown in Figure 5.13. The curves are smoothed using an exponential filter for better readability:

$$e_{\text{p}}^{\text{filtered}}(t) = (1-\omega)e_{\text{p}}^{\text{filtered}}(t) + \omega e_{\text{p}}(t), \tag{5.11}$$

with ω being the smoothing factor (equal to 0.0005 in object fixation experiments).

The vertical axis of the plot represents the $e_{\text{p}}(t)$ variable whereas the horizontal one stands for the time steps. The curve $e_{\text{p}}^{\text{s}}(t)$ represents $e_{\text{p}}(t)$ with the use of the supervised reward, $e_{\text{p}}^{\text{w}}(t)$ with the noisy weakly supervised reward and $e_{\text{p}}^{\text{wf}}(t)$ with the filtered weakly supervised reward.

Given $e_{\text{p}}^{\text{w}}(t)$ presents worse learning performances than $e_{\text{p}}^{\text{wf}}(t)$, we can conclude that the noise affects learning and that our denoising procedure is efficient to attenuate this effect.

The curve $e_{\text{p}}^{\text{s}}(t)$ presents better performances than $e_{\text{p}}^{\text{wf}}(t)$ because the criterion used to evaluate these performances is precisely the one minimised when learning with the supervised reward. It is not the case of our reward which aims at bringing the maximal reconstruction error to the image center.

Figure 5.13 also displays confidence intervals of 95 % probability for the average

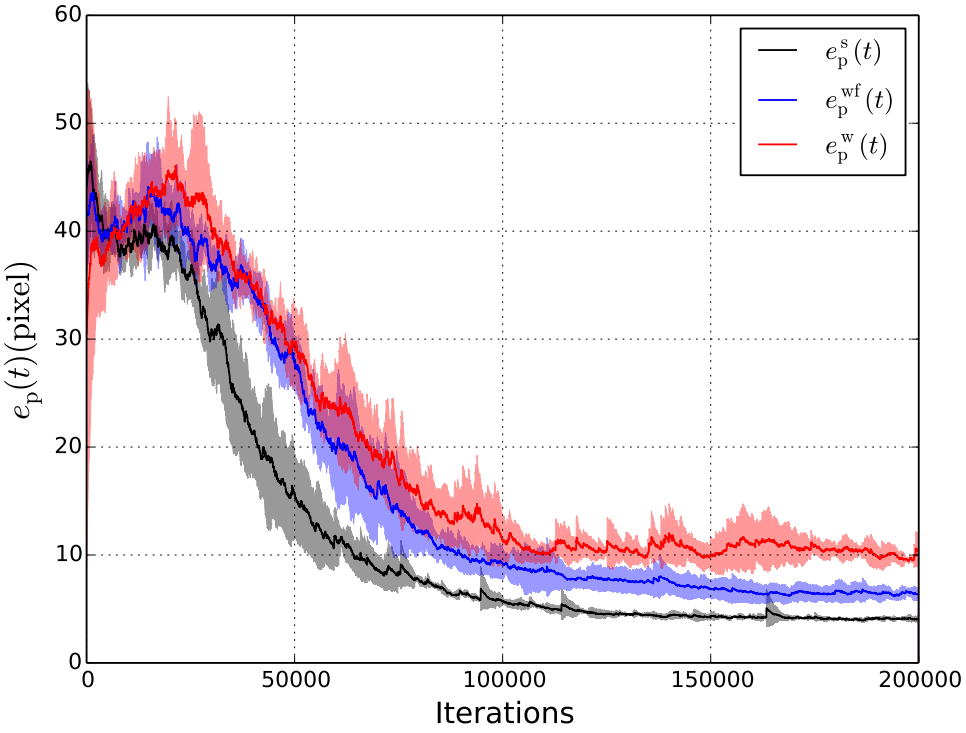


FIGURE 5.13: Binocular fixation position error over time for the supervised reward ($e_p^s(t)$), our reward without filtering ($e_p^w(t)$), our reward with filtering ($e_p^{wf}(t)$)

fixation error estimation[1]. We observe that the confidence intervals rarely intertwine, which means that the average fixation errors are statistically significant.

5.3.3.2 Policy Test

The objective of the policy test can be divided into three sub-goals:

Firstly, we want to compare the performance of the learned policy on the test set with that of the training set. This aims at evaluating how much our learning framework generalizes from few training data. Secondly, we want to compare the policies learned with the noisy reward and with the filtered one. The aim is to show that not only does the noise alter learning, but the performance of the resulting policy is also worse. Thirdly, we want to compare it with the policy learned with an informative and noiseless reward signal. The purpose is to check if the weakly supervised reward allows to learn the same kind of behaviour as the supervised reward.

Besides, other tests will be conducted in section 5.3.4.2 for some comparisons with the policies learned in a real environment. They will be detailed on this occasion.

Algorithm 2 describes the test procedure. We choose to evaluate the fixation error $e_p(t_{eps})$ at the end of each episode. Statistics (mean, median and standard deviations) of these random variables are computed for 6 000 test episodes for each

[1]For each iteration t, the confidence interval is $[e_p(t) - \frac{1.96\sigma(t)}{\sqrt{n}}, e_p(t) + \frac{1.96\sigma(t)}{\sqrt{n}}]$ with $\sigma(t)$ being the standard deviation of the fixation error at time t and n representing the number of trials per reward function (3 for our experiments)

Algorithm 2 Simulated object fixation test procedure

Parameters:

N_{tot}: the total number of episodes
N_{eps}: the number of iterations per episode
$\boldsymbol{q_0^{\text{camera}}}$: the initial position

Inputs:

π_{ψ}^{fix}: the policy
$\mathcal{D} = \mathcal{D}_{\text{test}}$ or $\mathcal{D}_{\text{train}}$: the object set

Output: $\mathcal{D}_{\text{p}}$: the set of final fixation errors

Steps:

Set $t \leftarrow 0$
while $t < N_{\text{tot}}$ **do**
    Choose a random object in $\mathcal{D}$ and place it randomly
    Go to the initial position $\boldsymbol{q_0^{\text{camera}}}$
    Set $t_{\text{eps}} \leftarrow 0$
    while $t_{\text{eps}} < N_{\text{eps}}$ **do**
        Apply $\boldsymbol{a_{t_{\text{eps}}}} = \pi_{\psi}^{\text{fix}}(\boldsymbol{s_{t_{\text{eps}}}})$
        Observe $\boldsymbol{s_{t_{\text{eps}}+1}}$
        $t_{\text{eps}} \leftarrow t_{\text{eps}} + 1$
    $t \leftarrow t + 1$
    Compute the fixation error $e_{\text{p}}(t_{\text{eps}})$ and append it to $\mathcal{D}_{\text{p}}$
    Withdraw the object

reward case. Among these test episodes, 2 000 come from each trained policy. Finally, a cumulative percent curve is presented to more precisely characterize the distribution of $e_{\text{p}}(t_{\text{eps}})$.

The results are summarized in Table 5.1. The columns r_{w}, r_{wf} and r_{s} respectively stand for the case of the noisy weakly supervised, filtered weakly supervised and supervised rewards. Figure 5.14 provides more insight into the distribution of the fixation error. The vertical axis $P(e_{\text{p}}^{D} < e_{\text{p}})$ represents the percentage of tested samples whose fixation error is smaller than the values on the horizontal axis.

Several remarks can be made:

- The fixation error distributions are rather asymmetrical since the median is different from the average in each case. This is due to the presence of outliers (when the robot looks at an area far from the object).
- The fixation accuracy of the policy learned with the noisy weakly supervised reward r_{w} is the worst for both the training and the test sets. In average, the fixation error for r_{w} is 3 and 4 pixels lower than the fixation error for r_{wf} on the objects of the training and the test sets.
- The results of the policies learned with our proposed reward function are worse than with a supervised reward (3 and 4 pixels of difference for the training and the test sets). This is not surprising since with the supervised reward, the optimization criterion is precisely based on the fixation error. Nevertheless, the results are good for our method with filtering.
- The errors are higher for the test set compared with those on the training set (3, 2 and 1 pixels of difference for r_{w}, r_{wf} and r_{s}). However, the difference is not high. This is not surprising since distributions of the training and the test set are

significantly different. Furthermore, the objects are much bigger in the test set. This can explain that even if the camera system correctly stares at the object, the fixation point can be further from the center-of-gravity projection for an object of the test set compared with one of the training set. Nevertheless, the experiments indicate that there is no strong overfitting and that we could adapt to the new distribution through learning.

- Figure 5.14 confirms all these results and enhances the presence of outliers when learning with our method. For instance, we notice some failures for our weakly supervised reward without filtering ($e_p > 150$ pixels).

	Training set			Test set		
Statistics (pixels)	r_w	r_{wf}	r_s	r_w	r_{wf}	r_s
mean(e_p)	8.5 ± 0.2	5.0 ± 0.1	2.0 ± 0.05	11.6 ± 0.4	7.0 ± 0.3	3.0 ± 0.1
median(e_p)	7.3	4.6	1.6	8.6	5.1	2.4
std (e_p)	7.2	2.7	1.8	15.6	9.9	4.9

TABLE 5.1: Performances of the learned policies. The number after the plus or minus sign is the standard error.

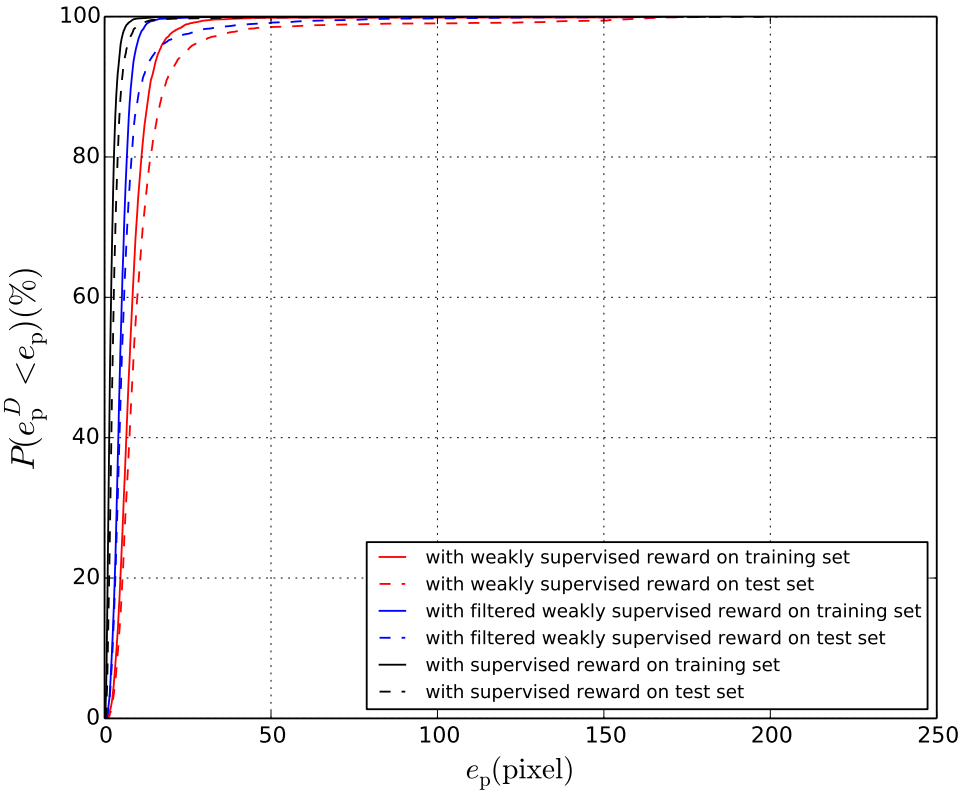


FIGURE 5.14: Cumulative distribution function of the fixation error (in %)

5.3.3.3 3D localization of objects

In this section, since the fixation task is meant to help reaching, we also evaluate the quality of our 3D object localization principle (through the object fixation policy π_{ψ}^{fix}). This analysis for the object localization errors has two objectives:

- First, we want to assess the average accuracy of our localisation system, i.e. how far are the 3D points fixated by the camera system compared with the object center of gravity?
- Second, we want to evaluate the repeatability of the object localisation. To that end, we evaluate the variations of the fixated points around the average fixated point. Indeed, to allow a consistent localization if there is an offset between the object center and the fixated points, these latter should not vary too much around the average fixated point.

Note that the triangulation computations are described in Appendix D.

Let $o = g, v$, with g representing the objective of establishing errors of localization with respect to the ground truth reference frame, and v the objective of establishing variations of computed camera joint angles around the average computed angles.

Technically, for both of the previously mentioned objectives, we want to evaluate at each object position on the table what are absolute uncertainties on 3D fixation $\Delta_{\text{o}}x$, $\Delta_{\text{o}}y$ and $\Delta_{\text{o}}z$ (see Appendix C to visualize the reference frame). They correspond to the standard deviation of the fixated point with respect to the ground-truth object position (equation (5.12)) and the standard deviation of the fixated point (equation (5.13)):

$$\Delta_{\text{g}}x = \sqrt{\frac{\sum_{i=0}^{N_{\text{pt}}-1}(g_x - x_i)^2}{N_{\text{pt}}}}, \tag{5.12}$$

$$\Delta_{\text{v}}x = \sqrt{\frac{\sum_{i=0}^{N_{\text{pt}}-1}(\mu_x - x_i)^2}{N_{\text{pt}}}}, \tag{5.13}$$

with N_{pt} being the number of fixated points for a given object position, μ_x the x coordinate of the average fixated point, g_x the x coordinate of the object center of gravity and x_i the x coordinate of the ith fixated point. The computation for the z and the y coordinates are exactly the same as for x. Furthermore, we compute the average distances (still for each object position on the table) between the fixated points and the ground-truth object position g (equation 5.14) and between the fixated points and the average fixated point μ (equation 5.15):

$$d_{\text{g}} = \frac{1}{N_{\text{pt}}} \sum_{i=0}^{N_{\text{pt}}-1} ||g - X_i||_2, \tag{5.14}$$

$$d_{\text{v}} = \frac{1}{N_{\text{pt}}} \sum_{i=0}^{N_{\text{pt}}-1} ||\mu - X_i||_2. \tag{5.15}$$

At the end of the procedure, we have gathered samples $(x, y, \Delta_{\text{g}}x, \Delta_{\text{g}}y, \Delta_{\text{g}}z, d_{\text{g}}, \Delta_{\text{v}}x, \Delta_{\text{v}}y, \Delta_{\text{v}}z, d_{\text{v}})$. Note that we only use the object which is used in the experiments subsequently: the blue ball. The results with other objects of roughly the same size may be similar whereas increasing the size of the object may increase the localization errors. The experimental procedure is summarized in Algorithm 3 and is repeated for the three learned fixation policies.

Algorithm 3 Evaluation of object localization uncertainties

Parameters:

p_x and p_y the length of the steps in x and y direction

$[m_x, M_x]$ and $[m_y, M_y]$ the intervals of variation of the x and y object coordinates

Inputs:

π_{ψ}^{fix}: the fixation policy

The object (blue ball)

Outputs: $\mathcal{D}$: a set of samples $(x, y, \Delta_{\text{g}}x, \Delta_{\text{g}}y, \Delta_{\text{g}}z, d_{\text{g}}, \Delta_{\text{v}}x, \Delta_{\text{v}}y, \Delta_{\text{v}}z, d_{\text{v}})$

Steps:

set $x \leftarrow m_x$
set $y \leftarrow m_y$
while $x < M_x$ **do**
    while $y < M_y$ **do**
        Move the object to the (x, y) position above the table
        $i \leftarrow 0$
        while i < N_{pt} **do**
            Apply π_{ψ}^{fix} and compute the 3D fixated point using Appendix D
            $i \leftarrow i + 1$
        For $o = g, v$: compute $\Delta_{\text{o}}x$, $\Delta_{\text{o}}y$, $\Delta_{\text{o}}z$, and d_{o} using equations (5.12), (5.13), (5.15) and (5.14)
        Store $(x, y, \Delta_{\text{g}}x, \Delta_{\text{g}}y, \Delta_{\text{g}}z, d_{\text{g}}, \Delta_{\text{v}}x, \Delta_{\text{v}}y, \Delta_{\text{v}}z, d_{\text{v}})$ in $\mathcal{D}$
        $y = y + p_y$
    $x = x + p_y$

For each case, we plot surfaces (x, y, d_{o}). The plots such as $(x, y, \Delta_{\text{o}}x)$ are available in Appendix F.

Figure 5.15 represents d_{g} values for a grid of object positions above the table. We observe that the larger errors are located in the corners of the table. It is explained by the fact the initial position of the camera system for object fixation learning make it fixate the center of the table. Thus, less exploration is conducted in the corner areas. In the latter, the errors can be meaningful since they reach more than 50 cm of uncertainties for some points. It means that our object localization principle is trustworthy for most positions of the objects above the table but can be ineffective for some particular object position areas. Figure 5.16 represents d_{v} values for a grid of object positions above the table. We notice that larger errors still occur at the corners of the table. Furthermore, we observe that the "relative" object localization uncertainty is overall very low. This indicates a good repeatability of the object position estimation implied by the policy.

If we consider that the d values are random variables, their statistics are presented in Table 5.2. We confirm that our object localization principle is not precise with respect to the ground truth object position (around 10 cm of average). However, as the object experimented is a ball with a radius of 8 cm, the localisation error is acceptable. Furthermore, the localization error with respect to the point fixated by the average camera joint angles is very low (around 2 cm). This means that the camera system generally fixates the same point given a precise object position, which is an interesting property even though it is not sufficient for a good localization principle (e.g. a camera system with all the joint angles to zero is also fixating the same point).

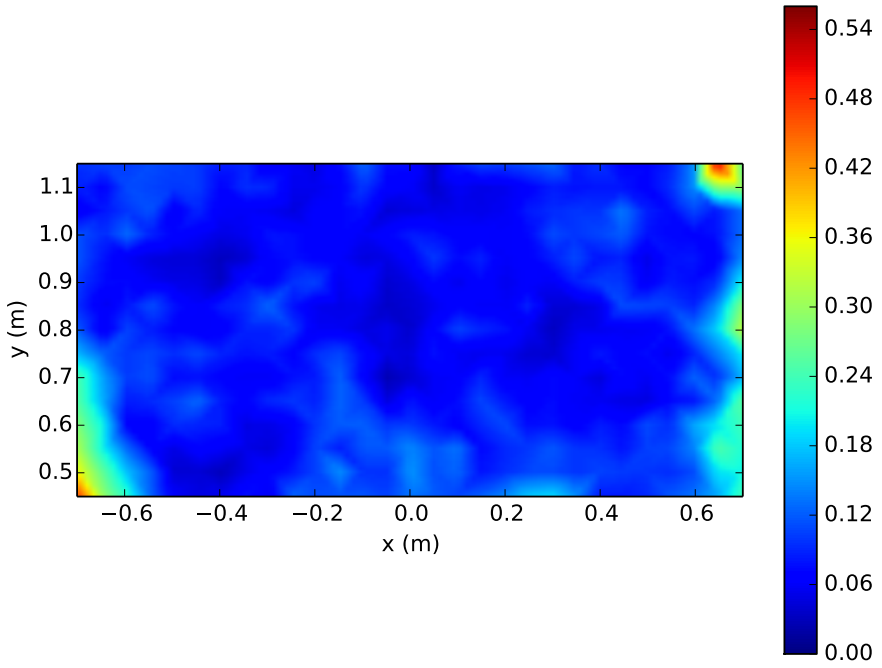


FIGURE 5.15: Scheme representing the standard localization uncertainty with respect to the object center of gravity (d_g)

We also display a visualization of the error using ellipsoids representing the average errors on each axis around the fixated point and the object center of gravity (in Figure 5.17). We notice that for both of the error types, there are larger errors on the z axis and smaller ones on the x axis.

Statistics (m)	d_g	d_v
mean(d_o)	0.095 ± 0.005	0.022 ± 0.003
median(d_o)	0.083	0.012
std (d_o)	0.055	0.034

TABLE 5.2: Statistics on d values for the object

5.3.4 Experiments in a real environment

In the following paragraphs, we present the experiments made on the real platform.

5.3.4.1 Training

The training protocol is similar to the one used in simulation with few exceptions:

- We do not measure the fixation error because the camera system is not calibrated and we do not have access to the 3D Cartesian object coordinates. Instead, we display the reward function evolution during training.

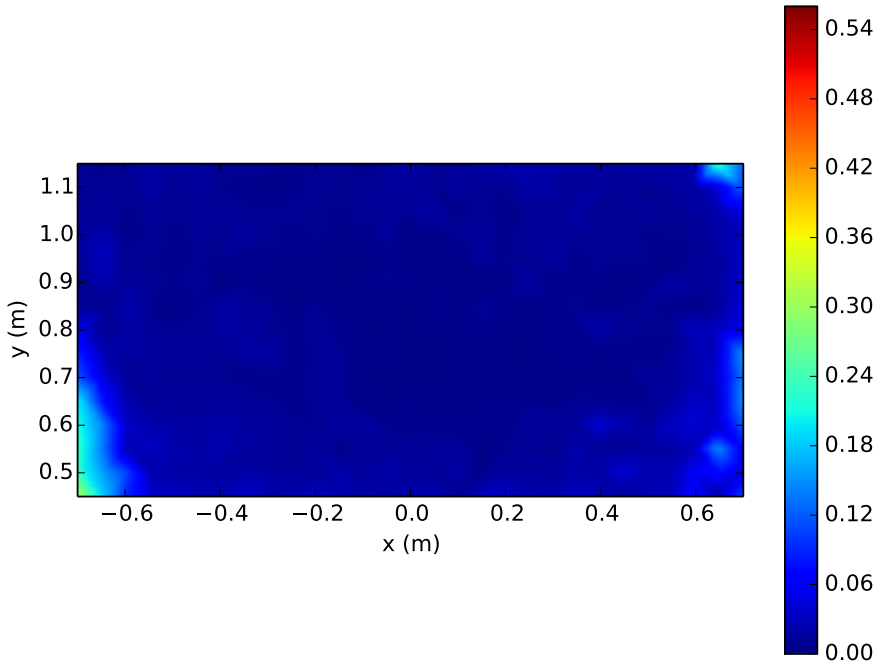


FIGURE 5.16: Scheme representing the standard localization uncertainty with respect to the average fixated point (d_v)

- A human has to change the object and its position at each episode end, which is considerably burdensome and long for the aforementioned human (around 4 600 episodes).
- A single run with the weakly supervised reward with filtering has been launched.
- We do not compute the additional reward term r_{pan} penalizing divergence and a too strong convergence.

Otherwise, Algorithm 1 and the pre-training step without object are integrally applied and Table B.1 of Appendix B gives the hyperparameter values used in our experiments. Note that the direct transfer of policy and Q networks (initialization of Q and π networks with the ones learned in simulation) from the synthetic environment to the real environment does not work at all. The main reason is that the networks learn environment-specific features in simulation which cannot be transferred to the real world. Indeed, the two environments are too different. Recent studies [168], [169] on simulated-to-real transfer could have been used for our experiments.

Figure 5.18 shows the evolution of the reward signal through time. Ideally, the reward function would reach 2 because it corresponds to the object being exactly at the image center in both images. However, as previously stated, there is a tilt offset which implies that the object fixation point cannot be exactly at the image center. Furthermore, we plot the reward values of all the RL iterations, including those executed at the beginning of an episode, when the object is not at the image center. From this graph, we can conclude that the reward signal converges through time and this shows that our learning framework is efficient on a real setting.

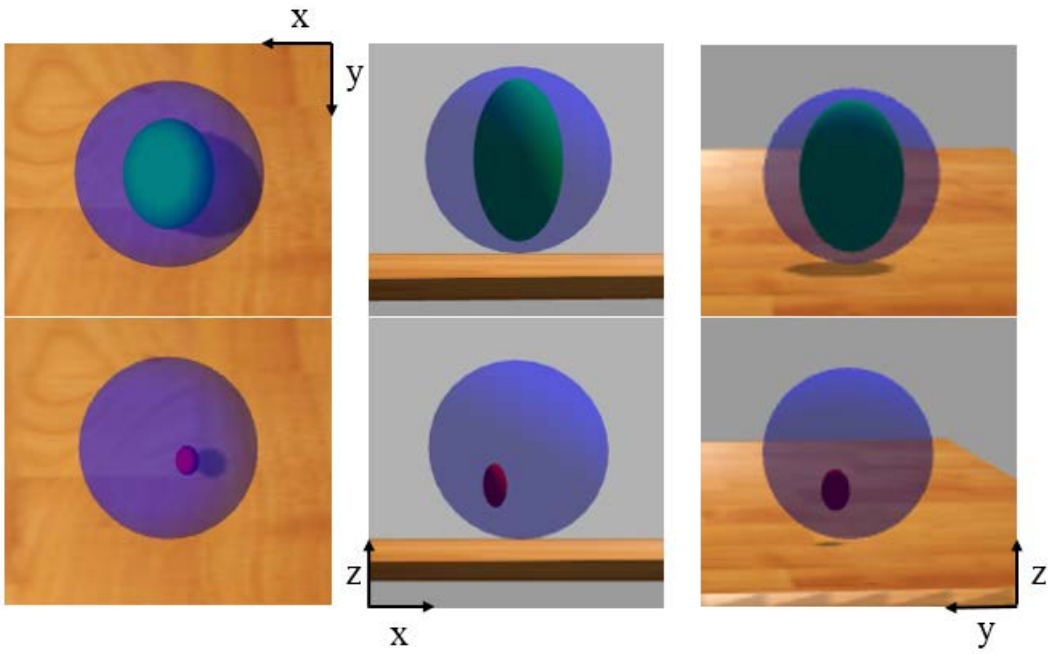


FIGURE 5.17: Average ellipsoid errors for the absolute (top) and relative (bottom) localization uncertainties. The green ellipsoid error (top) stands for the volume where the camera joint angles fixate around the object center of gravity. The red ellipsoid (bottom) represents the volume where the camera joint angles fixate around the average fixated point. The blue ball is the object used in our experiments

5.3.4.2 Test

The tests conducted in the real environment are different from those performed on the synthetic environment. Instead of using the final fixation error at the end of the test episodes, we conduct binary tests: i.e. at the end of an episode, we check if the image centers from left and right images belong to the object. For the tests in the real environment, because of the vertical offset in the images, we check that the average vertical coordinate (over the left and right images) belongs to the object. If the image centers belong to the object, the test episode is successful and otherwise it is a failure. The test procedure is summarized in Algorithm 4. We also compare the results of the policy obtained from this run in the real environment with ones of the policies learned in the simulated environment with the filtered weakly supervised reward.

These results are summarized in Table 5.3. We cannot extract meaningful conclusions in terms of comparison. We can just observe that performances are really good in the training set in both the real and simulated environments. Furthermore, we notice also that there is a loss of performance when policies are applied on the test set. This is not surprising given the gap between training and test object distributions. Indeed, the test objects were mainly designed to trap the learned policies. In addition to these results, we provide in Figure 5.19 examples of succesful and failed test episodes for both the training and the test set. We observe in this figure and in the experiments that for the test set, the robot fails to fixate objects with a colour similar to the background (e.g the white plastic spoon which is difficultly visible for humans).

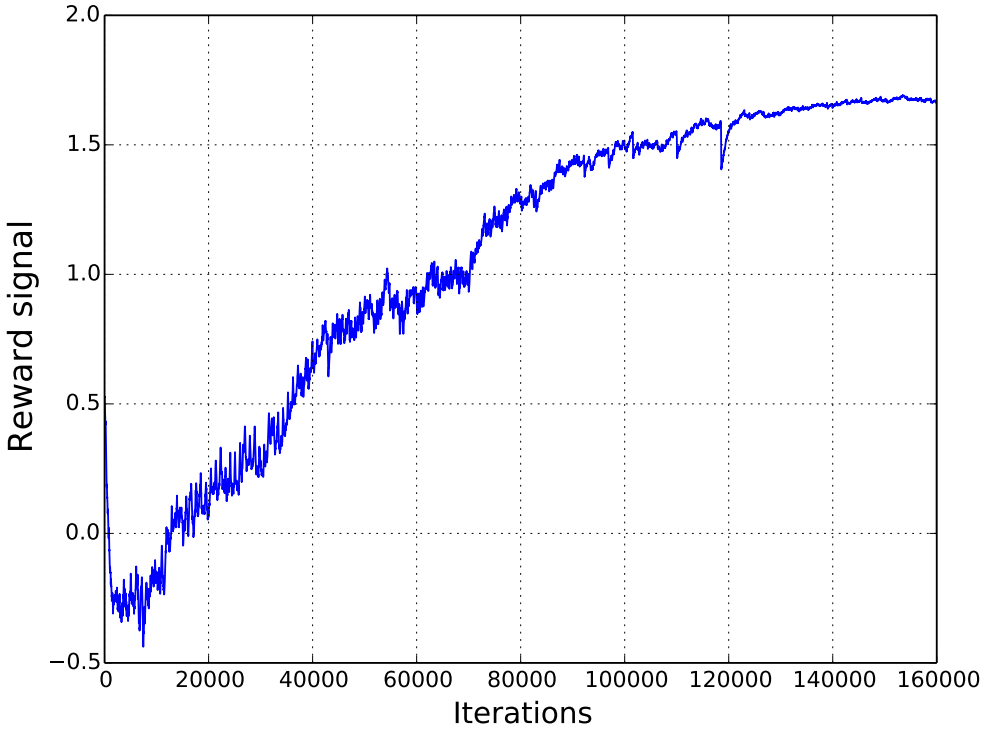


FIGURE 5.18: Figure showing the evolution through time of the reward signal for the real object fixation experiment

	Training set		Test set	
Reward-environment	r_{wf}-real	r_{wf}-simulated	r_{wf}-real	r_{wf}-simulated
Success frequency (%)	93	97.3	79.5	77.1

TABLE 5.3: Performances of the learned fixation policies

5.4 Conclusion

We have designed a shaping reward function for an object binocular fixation task which is informative in that it correctly discriminate values of nearby states. Furthermore, its computation can dispense with calibration parameters and hand-crafted visual modules. The main requirement is that the scene cannot vary during learning. If so, the autoencoder has to be trained again without object. Experiments in the simulated environment show that our reward function can yield policies whose performances approach ones trained with a reward requiring calibration parameters. Experiments in the real environment show first that our learning framework is applicable in a real setting and second that our reward function can learn complex real-world fixation policies. The simulated part of the work on binocular object fixation has been published in IJCNN 2017 [3] (the part on real experiments is unpublished). The next chapter describes how our presented framework was adapted to the learning of end-effector binocular fixations.

Algorithm 4 Real object fixation test procedure

Parameters:

N_{tot}: the total number of episodes
N_{eps}: the number of iterations per episode
$\boldsymbol{q}_{\mathbf{0}}^{\mathbf{camera}}$: the initial camera joint angles

Inputs:

π_{ψ}^{fix}: the object fixation policy
$\mathcal{D} = \mathcal{D}_{\text{test}}$ or $\mathcal{D}_{\text{train}}$: the object set

Outputs: $\mathcal{D}_{\text{p}} \in \{0,1\}^{N_{\text{tot}}}$: the set of results of the test episodes

Steps:

Set $t \leftarrow 0$
while $t < N_{\text{tot}}$ **do**
    Choose a random object in $\mathcal{D}$ and place it randomly
    Go to the initial position $\boldsymbol{q}_{\mathbf{0}}^{\mathbf{camera}}$
    Set $t_{\text{eps}} \leftarrow 0$
    while $t_{\text{eps}} < N_{\text{eps}}$ **do**
        Apply $\boldsymbol{a}_{\boldsymbol{t}_{\text{eps}}} = \pi_{\psi}^{\text{fix}}(\boldsymbol{s}_{\boldsymbol{t}_{\text{eps}}})$
        Observe $\boldsymbol{s}_{\boldsymbol{t}_{\text{eps}}+\mathbf{1}}$
        $t_{\text{eps}} \leftarrow t_{\text{eps}} + 1$
    $t \leftarrow t + 1$
    Observe the result (0 for failure or 1 for success) and append it to $\mathcal{D}_{\text{p}}$
    Withdraw the object

FIGURE 5.19: Images showing successful (on the right) and failed (on the left) test episodes for both the training set (at the top) and the test set (at the bottom)

Chapter 6

Learning hand-eye coordination function

In the previous chapter, we described how the robot learns to fixate objects using a deep reinforcement learning algorithm and a reward function designed with very little supervision. In this chapter, we describe how we adapt the methods of Chapter 5 to learn jointly and still with little supervision how to fixate the end-effector and a hand-eye coordination function.

6.1 Introduction

In Chapter 5, fixating the object is meant to locate it in the 3D world using camera joint angles. In this chapter, the robot learns a skill which make it locate its end-effector in the 3D world given proprioceptive values. Whether this skill is analogue to the proprioceptive skill of humans is still an open question. In our case, the proprioceptive values are represented by robot joint angles. In standard robotics, the end-effector localisation is achieved by using kinematics, i.e. mapping robot joint angles and a Cartesian pose. However, we want to locate the end-effector in the same reference frame as the object, i.e. the space of camera joint angles (instead of Cartesian coordinates). Therefore, from robot joint angles, we learn to locate the end-effector in the camera joint angle space. The resulting mapping is called the hand-eye coordination. The latter is learned in parallel with an end-effector fixation task inspired by the work of Chapter 5. For this, we design a similar reward function based on an end-effector detection in the images.

6.1.1 Hand-eye coordination function

In our approach, the hand-eye coordination function is considered as a way to link arm proprioceptive values with the hand (or end-effector) localization in the 3D world. As previously mentioned, the end-effector is in our case localized using the camera joint angles but can be localized by Cartesian coordinates using traditional methods. In the robotic literature, the hand-eye coordination can refer to two main categories of mapping:

- The first category is the forward mapping [160], [170]. It generally maps robot arm joint angles $\boldsymbol{q}^{\mathbf{robot}}$ to $\boldsymbol{x}$ the localisation of the end-effector $f : S_{q^{\text{robot}}} \rightarrow X$. In our case, $\boldsymbol{x}$ is composed of camera joint angles $\boldsymbol{x} = \boldsymbol{q}^{\mathbf{camera}}$. This function is a forward kinematic model if $\boldsymbol{x}$ is composed of Cartesian coordinates.

- The second category is the inverse mapping [171], [172]. It links end-effector position to arm joint angles: $f^{-1} : X \rightarrow S_{q^{\text{robot}}}$. This is also similar to inverse kinematics if we consider Cartesian coordinates for $\boldsymbol{x}$.

We have chosen to use a forward mapping because the inverse mapping is not a function. Indeed, one input can correspond to several outputs. Then, approximating it using usual function approximators such as feed-forward neural networks is impossible unless only one output is considered. We could have used a special mapping approximator allowing several possible outputs for an input. However, an optimal implementation would have been long to incorporate in our work. In contrast, a forward model is a function and can be simply approximated by any universal function approximator. In addition, straightforward implementations of such approximators are easily incorporated in our work.

6.1.2 End-effector detection

As previously stated, it is important in our study not to use supervision or handcrafted features in this step. Thus, we do not want to use methods using markers and segmentation algorithms [173].

One of the most common methods for the end-effector localization in the image is the use of the end-effector motion and we use this principle for our study. For example, in [174], the end-effector is detected using a pre-defined end-effector movement assuming that the end-effector is the only part moving in the environment. [175] improves the method by correlating the end-effector motion to the resulting optic flow. Then, [176] has used the former to detect the end-effector with a kinect sensor. These methods do not use much supervision with the exception of the end-effector motion specification.

We could have also used as a complement the method of [177] which makes a robotic head directly learn to verge on the end-effector without explicit localization of the end-effector. The method applies the principle of active efficient coding. The image centers of the two cameras are concatenated and an encoding of the resulting vector is learned using a sparse autoencoder. Learning to verge on the end-effector is achieved by moving cameras and arms such that the reconstruction error of the resulting vector is minimized. Indeed, when there is vergence, the two image center areas look alike to each other and present redundancies, which make the reconstruction error minimal.

We do not claim a clear contribution for the end-effector detection mechanism in our method as well as for the hand-eye coordination design. Indeed, state-of-the-art methods have already developed these ideas. However, in this chapter, we show that our framework for the learning of binocular object fixation can extend to end-effector fixation, giving us a unified framework for image area fixation learning.

6.2 Methods

6.2.1 Task definition

In this section, we detail how we adapt the framework of the previous chapter to learn end-effector fixations and a robot hand-eye coordination function.

We model the hand-eye coordination function $f_{\boldsymbol{\eta}} : S_{q^{\text{robot}}} \rightarrow S_{q^{\text{camera}}}$ with a neural network (see Appendix A.1.4 for its structure). This function outputs camera joint

angles which make the camera system fixate the end-effector from arm joint angles:

$$\boldsymbol{q}^{\mathbf{camera}}_{\mathbf{virt}} = f_{\eta}(\boldsymbol{q}^{\mathbf{robot}}). \tag{6.1}$$

To learn it, we need to have a database $\mathcal{D}$ of input-output pairs $(\boldsymbol{q}^{\mathbf{robot}}, \boldsymbol{q}^{\mathbf{camera}})$ where $\boldsymbol{q}^{\mathbf{camera}}$ makes the camera look at the end-effector. To produce such samples, we learn to fixate the end-effector. For this, we use a similar framework as the object fixation task. We use the DDPG algorithm and a reward requiring weak supervision. The Markov Decision Process is the same as for the object fixation with the exception of the reward function (see Section 5.2.1). The latter involves the end-effector detection $\boldsymbol{x}^{\mathbf{cam}}_{\mathbf{eff}}$ instead of $\boldsymbol{x}^{\mathbf{cam}}_{\mathbf{obj}}$:

$$r^{\text{cam}}_{\text{eff}} = 2\frac{\frac{1}{2}L - ||\boldsymbol{x}_{\mathbf{c}} - \boldsymbol{x}^{\mathbf{cam}}_{\mathbf{eff}}||_2}{L} < 1, \tag{6.2}$$

where L takes the same constant value as in section 5.2.1 (100.5). The set-up is also episodic. For each episode, 6 random arm joint coordinates are generated using uniform distributions with fixed limits (the robotic arm has 7 degrees of freedom and we choose not to use the rotation angle of the wrist angle). They are empirically set to provide a large variety of reachable arm configurations. With the latter, we also ensure that the end-effector moving finger is visible.

During learning, training pairs $(\boldsymbol{q}^{\mathbf{robot}}, \boldsymbol{q}^{\mathbf{camera}})$ are added to $\mathcal{D}$ when the reward $r^{\text{cam}}_{\text{eff}}$ is above a fixed threshold R_{th}. When the number of samples in $\mathcal{D}$ is higher than a threshold value N_{upd}, we train f_{η} on random batches of $\mathcal{D}$ each time a new sample is added to $\mathcal{D}$.

6.2.2 End-effector detection

We describe here how we detect the end-effector in the image. Unlike section 5.2.2 which uses an autoencoder to localize the object, the end-effector image position is computed using the simple difference in the image before and after predefined end-effector finger moves. The idea is that the hand is segmented from the rest of the scene because its appearance varies according to finger moves. Then, this end-effector detection method only requires to specify finger moves. In our case, to make the end-effector localization stable, we decided to make only one finger move. Indeed, from one iteration to the other, if all the fingers move, the localization of the end-effector in the images can jump from a pixel to a another relatively distant one. This can cause noise in the reward function and affect learning.

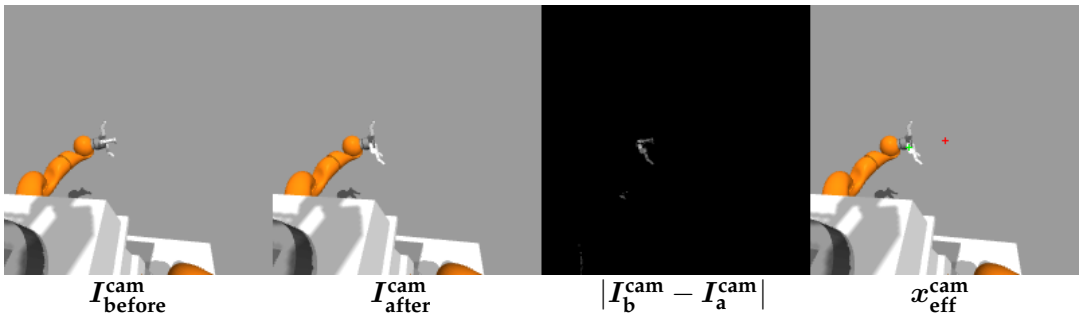


FIGURE 6.1: End-effector detection computation scheme.

Figure 6.1 presents the different steps of the end-effector detection:

- The images before ($\boldsymbol{I}^{\mathbf{cam}}_{\mathbf{before}}$) and after ($\boldsymbol{I}^{\mathbf{cam}}_{\mathbf{after}}$) the end-effector moves are recorded.
- The difference of images is calculated and the end-effector position $\boldsymbol{x}^{\mathbf{cam}}_{\mathbf{eff}}$ is computed using a kernel density estimator (on $N = 150$ pixels) the same way $\boldsymbol{x}^{\mathbf{cam}}_{\mathbf{obj}}$ is computed from the autoencoder reconstruction error image (see section 5.2.2).

6.2.3 Mitigating noise

Note that the end-effector detection is also filtered because of impulse noise. Indeed, as shown by Figure 6.2, the shadow areas of the fingers also move when the finger moves, which can result in end-effector localization error. This produces impulse noise in the reward function and we saw in Chapter 5 that it has a bad effect on learning. We apply exactly the same filtering procedure as in Chapter 5 and with the same threshold value.

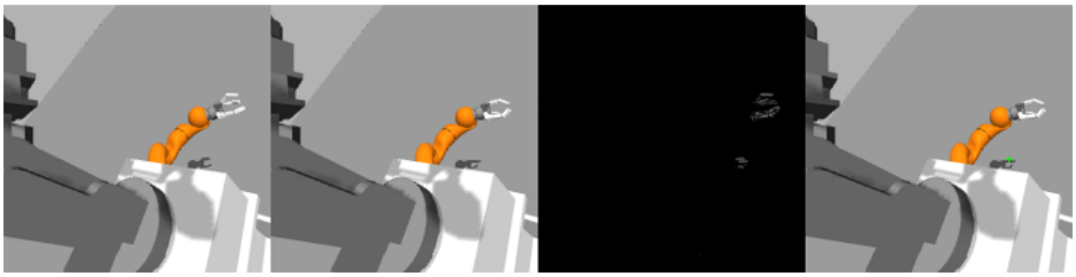


FIGURE 6.2: End-effector detection failure

6.3 Experiments

The experiments for the learning of the hand-eye coordination function are conducted in a synthetic environment. These experiments have two main objectives:

- We want to evaluate when the hand-eye coordination function is properly learned.
- We want to evaluate the quality of the hand-eye coordination function in the estimation of the end-effector position (through camera joint angles) from arm joint angles.

6.3.1 Experimental environment

As for experiments of Chapter 5, we use the Gazebo simulator along with the Ros middleware. The table is withdrawn from the experiments such that the robot arm can move more freely. An end-effector, the Barrett Hand [178] is added to the left arm. Figure 6.3 illustrates the experimental set-up.

6.3.2 Experimental protocol

6.3.2.1 Training

Algorithm 5 summarizes how the robot learns both the hand-eye coordination function and end-effector binocular fixations. As previously mentioned, the framework for the end-effector binocular fixation is very similar to the one used to learn object fixation. The reward function and the initialisation of episodes change, and a

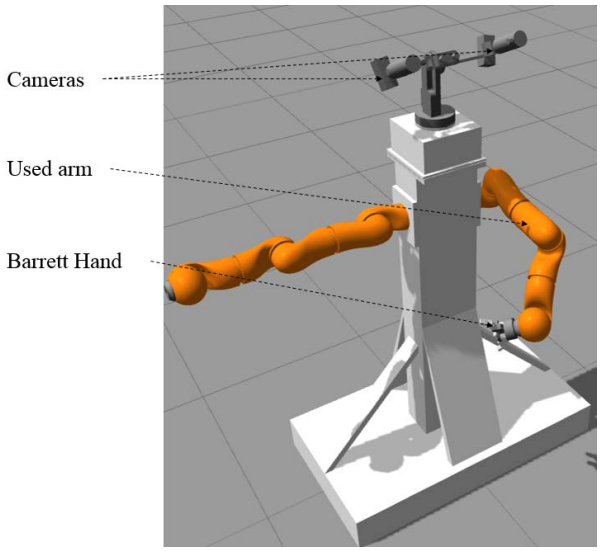


FIGURE 6.3: Experimental environment for the end-effector fixation and hand-eye coordination learning

hand-eye coordination function training process is added. The variety of joint arm angles can be visualized in Figure 6.4, which displays an envelop of 3D end-effector positions in the 3D space. The volume (the one of the convex hull) has been estimated to $0.23m^3$.

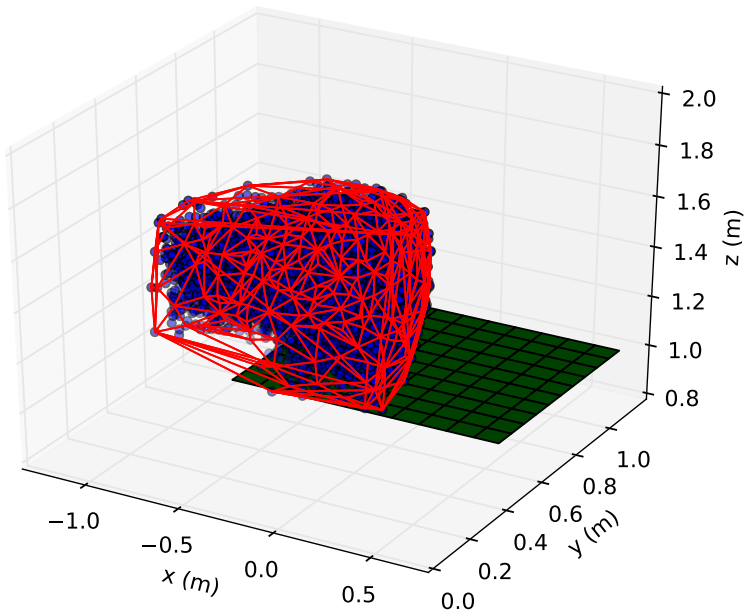


FIGURE 6.4: Representation of the volume of end-effector initial positions in 3D Cartesian coordinates. The positions are in blue and the convex hull in red. To have an idea of the Gazebo scales, the table is visualized in green.

To evaluate the learning of the end-effector policy, we monitor the reward function through time. We choose not to plot the fixation error during training because there is no comparison made between several reward signals here. Indeed, the comparisons with a supervised and noisy rewards have already been conducted in Chapter 5. We assume that doing them again does not introduce something new.

To evaluate the learning of the hand-eye coordination, we plot the loss of the hand-eye coordination function through time and check when it converges.

Algorithm 5 Learning to fixate end-effector and to coordinate eye and hand

Parameters:

N_{trans}: the size of the circular buffer
γ: the discount factor
ϵ: the Gaussian noise variance
N_{tot}: the total number of iterations
N_{eps}: the number of iterations per episode
N_{b}: the number of transitions per batch
th: the removal threshold
N_{gp}: the number of samples required to train Gaussian process
N_{upd}: the required number of samples to start to update $f_{\boldsymbol{\eta}}$
R_{th}: the threshold reward value
$\boldsymbol{q_0^{\text{camera}}}$: the initial position

Inputs:

$S_{q^{\text{arm}}}$: the set of arm initial positions

Outputs: $Q_{\boldsymbol{\sigma}}^{\text{eff}}$, $\pi_{\gamma}^{\text{eff}}$, $f_{\boldsymbol{\eta}}$

Steps:

Set $t \leftarrow 0$
Initialize the transition circular buffer T_{buf}
while $t < N_{\text{tot}}$ **do**
    Choose random arm joint angles $\boldsymbol{q^{\text{robot}}}$ among $S_{q^{\text{arm}}}$ and go to the position
    Go to the initial position $\boldsymbol{q_0^{\text{camera}}}$
    Set $t_{\text{eps}} \leftarrow 0$
    while $t_{\text{eps}} < N_{\text{eps}}$ **do**
        Apply $\boldsymbol{a_t} = \pi_{\gamma}^{\text{eff}}(\boldsymbol{s_t}) + \mathcal{N}(0, \epsilon)$
        Observe $\boldsymbol{s_{t+1}}$
        Compute r_t and $|\Delta \text{d}_t^{\text{exp}}|$
        $cond \leftarrow (t < N_{\text{gp}})$ or $((|\,|\Delta \text{d}_t^{\text{exp}}| - m(||\boldsymbol{\Delta q^{\text{camera}}}||_2)| < th)$ and $(t > N_{\text{gp}}))$
        if $cond$ **then**
            Add $< \boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}} >$ to T_{buf}
        Pick randomly N_{b} transitions from T_{buf}
        Update $Q_{\boldsymbol{\sigma}}^{\text{eff}}$ and $\pi_{\gamma}^{\text{eff}}$ using DDPG (equations (2.46) and (2.48))
        if $r_t > R_{\text{th}}$ **then**
            add the pair $(\boldsymbol{q^{\text{robot}}}(t), \boldsymbol{q^{\text{camera}}}(t+1))$ to the database $\mathcal{D}$
            if $Size(\mathcal{D}) > N_{\text{upd}}$ **then** train $f_{\boldsymbol{\eta}}$ for one step
        if $t = N_{\text{gp}}$ **then** train $\Delta \text{d} = m(||\boldsymbol{\Delta q^{\text{camera}}}||_2)$
        $t \leftarrow t + 1$
        $t_{\text{eps}} \leftarrow t_{\text{eps}} + 1$

6.3.2.2 Test

We evaluate the hand-eye coordination function in terms of final fixation error, i.e. the distance in the image space between the end-effector moving finger and the image center. Note that we do not evaluate the quality of end-effector binocular fixation policies since they are not used to learn reaching skills. We draw the same statistics as in 5.3.3.2. We evaluate the hand-eye coordination functions saved at 100 000 and 200 000 iterations.

Algorithm 6 Hand-eye coordination function test procedure

Parameters:

1: N_{tot}: the total number of episodes

Inputs:

2: $S_{q^{\text{arm}}}$: the set of arm initial positions

3: f_η: the hand-eye coordination function

Output: $\mathcal{D}_{\text{p}}$: the set of final fixation errors

Steps:

1: Set $t \leftarrow 0$

2: **while** $t < N_{\text{tot}}$ **do**

3: Choose random arm joint angles among $S_{q^{\text{arm}}}$ and go to the position

4: Compute $\boldsymbol{q}^{\textbf{camera}}_{\textbf{hand}-\textbf{eye}} = f_{\boldsymbol{\eta}}(\boldsymbol{q}^{\textbf{arm}})$

5: Go to $\boldsymbol{q}^{\textbf{camera}}_{\textbf{hand}-\textbf{eye}}$

6: Compute the fixation error $e_{\text{p}}(t)$ and append it to $\mathcal{D}_{\text{p}}$

7: $t \leftarrow t + 1$

6.3.3 Implementation details

Appendix A displays the neural network structures used for the policy, the Q function and the hand-eye coordination mapping and Table B.2 of Appendix B describes the hyperparameters related to Algorithm 5. For the episode initialization of the end-effector fixation task, the seven arm joint angle distribution amplitudes are 11, 46, 69, 92, 92, 86 and 0° if we consider the ascending order in the kinematic chain i.e. from the base link to the end-effector.

6.3.4 Results

The following paragraphs analyse the results of the experiments. Besides discussing the training runs and the evaluation of the hand-eye coordination mapping, we analyse the 3D localization of the end-effector using the hand-eye coordination.

6.3.4.1 Policy and hand-eye coordination training

The results of the three runs are presented in Figure 6.5 (for better readability, we use the same exponential smoothing defined by equation (5.11) in Chapter 5 with $\omega = 0.001$). We notice that the policy converges. The fact that it does not reach 2 is due to the exploration noise and that the plot takes into account all the reward values of an episode. The error of the eye-hand coordination function reaches minimal values at around 100 000 iterations, which indicates that learning could have been stopped before the end of the experiments. Furthermore, the standard errors and standard deviations of the two plotted signals are small, which is an interesting property for our learning framework.

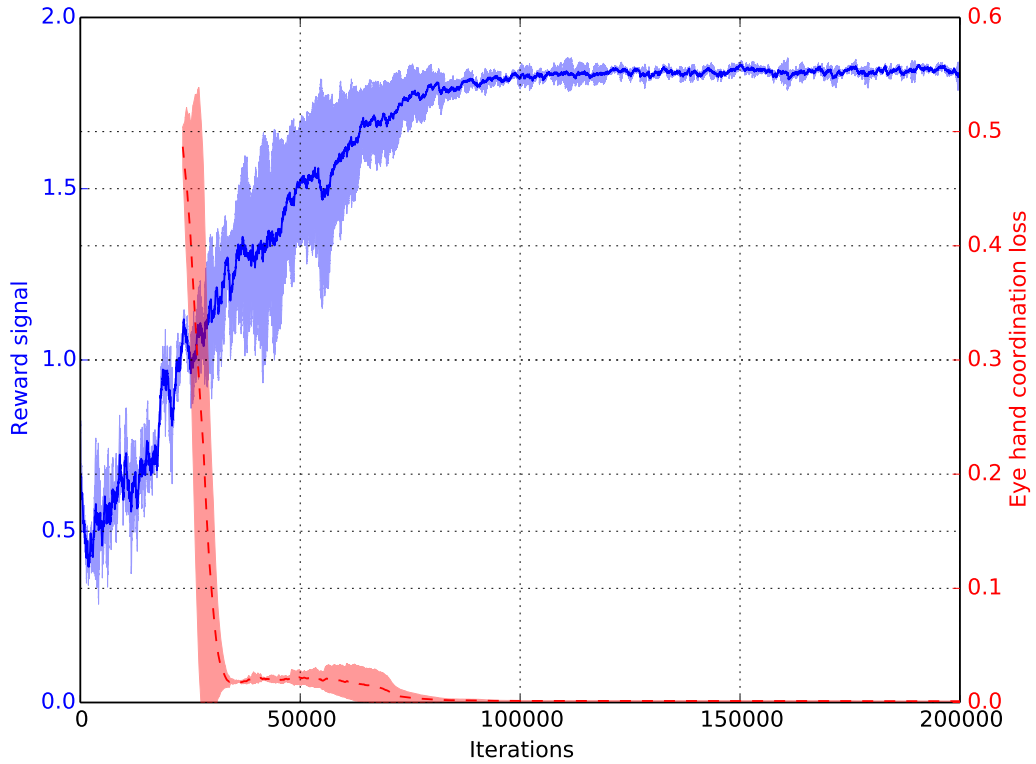


FIGURE 6.5: Evolution of the reward signal and the eye hand coordination loss through time during training

6.3.4.2 Evaluation of the hand-eye coordination mapping

The results of the evaluation of the hand-eye mappings are presented in Table 6.1. We notice several facts:

- The fixation error is rather important with respect to the fixation error for the object fixation task (8 pixels against 5). This can be explained by the fact that the camera system learns to fixate what has changed in the image but not necessarily the initial position of the end-effector finger.
- The standard deviation is rather low. This means that the learning of the hand-eye coordination function does not present outliers for specific arm joint angle configurations.
- The results are roughly the same for the mappings at 100 000 and 200 000 iterations though results are better after 200 000 iterations. Therefore, we confirm that there is no need to learn the mapping for more than 100 000 iterations.

Statistics (pixels)	at 100 000 iterations	at 200 000 iterations
mean(e_p)	7.98 ± 0.06	7.84 ± 0.05
median(e_p)	7.71	7.63
std (e_p)	2.52	2.10

TABLE 6.1: Performances of the hand-eye coordination mappings in terms of fixation error

6.3.4.3 3D localization of the end-effector

As we studied the quality of the 3D localization for the object, we also want to do the same for the end-effector. Our reaching process presented in the next chapter uses the hand-eye coordination f_η to locate the end-effector finger. From 7 arm joint angles, f_η outputs 3 camera joint angles which make the camera system look at the end-effector. The camera joint angles encode the end-effector localisation. As the latter is parametrized by the 7 arm joint angles, we cannot use the same visualization process (moving object in the x and y axes above the table) as for the object localization evaluation. At each iteration of the evaluation procedure, we generate random arm joint angles using uniform distributions. We apply the hand-eye coordination function and calculate the 3D fixated point X of coordinates (x, y, z) with triangulation (using Appendix D). Then, we compute the Euclidean distance d between the fixated point and the moving finger centre g:

$$d = ||X - g||_2. \tag{6.3}$$

After repeating the process for $N = 2000$ steps (for each of the three hand-eye coordination mappings), we compute statistics about localization uncertainties. Note that we only evaluate end-effector localization uncertainties with respect to the moving finger localization, i.e. the accuracy of the hand-eye coordination mapping. In other terms, the standard uncertainties are computed based on the difference between the fixated point using the hand-eye coordination function and the moving finger center. As the hand-eye coordination function always outputs the same results, there is no reason to compute uncertainties with respect to the average fixated point, i.e. perfect repeatability is ensured. The experimental procedure is summarized in Algorithm 7.

Algorithm 7 Evaluation of End-effector localization uncertainties

Parameters:
1: U: a set of uniform distribution for each axis
2: N: the number of evaluations
Inputs:
3: f_η: the hand-eye coordination function
Outputs: $\mathcal{D}$, a set of 3D fixation errors d
Steps:
1: set $t \leftarrow 0$
2: **while** t < N **do**
3: Generate random arm joint angles $\boldsymbol{q}^{\mathbf{robot}}$ using U
4: Move the arm to $\boldsymbol{q}^{\mathbf{robot}}$
5: Estimate d using Appendix D
6: Store d in $\mathcal{D}$
7: $t = t + 1$
8: Draw statistics on $\mathcal{D}$

The results are presented in Table 6.2. We observe a mean difference close to 10 cm, which can seem high. However, the fixated point is generally just in front of the palm as illustrated by two examples of Figure 6.6. These results show that the hand-eye coordination mapping encodes a location close to the end-effector, but it does not encode the location of a precise part of the end-effector. Indeed, the camera system has just learned to fixate the area with the highest luminance in the difference

of images (before and after the end-effector finger move) rather than a specific end-effector part projection.

Statistics (meters)	Value
mean	0.097
median	0.09
std	0.033

TABLE 6.2: Statistics on d values for the end-effector in meters

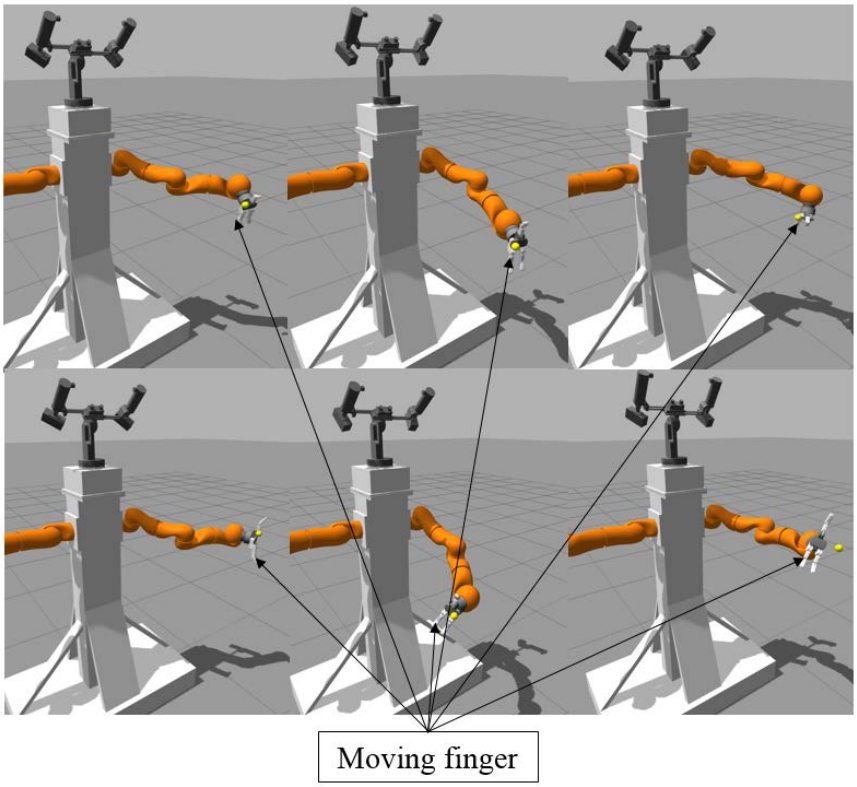


FIGURE 6.6: Six examples of the fixated point localization (yellow ball) using the hand-eye coordination function. The bottom right image presents one of the worst cases.

To conclude with, we observe that the hand-eye coordination network allows to extract the 3D approximative position of the end-effector. The interesting property of the localization is that the variance of the fixation error is rather low.

6.4 Conclusion

We have presented how the learning framework for object fixation was adapted to end-effector fixation by modifying the reward function. The computation of the latter assumes that the background is immobile. This issue is addressed in several state-of-the-art methods which can be incorporated in our work. Furthermore, we have also shown how the hand-eye coordination function was learned with supervised learning (but with self-supervised labels). Our results show first a reliable learning of both the end-effector policies and the hand-eye coordination function in 100 000 iterations. Second, the analyses of test results and 3D localization indicate

that the hand-eye coordination does not yield a perfect fixation of the end-effector moving finger because it learns instead to fixate the optical flow generated by the finger motion. However, the variance of the fixation error is low, which indicates that the hand-eye coordination mapping is reliable for an approximative localization of the end-effector.

Chapter 7

Learning arm motor skills based on binocular object fixation and hand-eye coordination

In this chapter, we describe how the previously learned skills can be used to learn both to reach objects and to learn the object reachability.

7.1 Learning to reach with the palm

This section concerns the reaching task itself. We describe in details how we use the object fixation policy and the hand-eye coordination function to learn touching policies and provide experiments with different initial settings.

7.1.1 Task definition

The task consists in touching with the end-effector palm the object on the table at different reachable positions from a large set of initial arm positions. Figure 7.2 display 8 randomly generated initial configurations and Figure 7.1 represents a plot of possible initial end-effector positions in the 3D space. This volume differs from the one of Chapter 6. The latter volume resulted in too many collisions between the robot and the table in the initialization step (see Appendix H). Thus, we choose a different set of initial arm positions which makes most initial end-effector positions above the table. This set still produces a large volume of possible initial end-effector positions (the volume of its convex hull is $0.27m^3$).

The goal of learning to reach both at different target positions and from a large set of initial robot joint angles is mainly motivated by the fact it allows to learn policies that are more robust to perturbations in the joint space according to [179]. In addition, reaching from different initial joint angles allows to reach from positions with a badly oriented end-effector which is a challenging task. Indeed, as it will be explained in section 7.1.2, the only term rewarding a good orientation of the end-effector is the sparse reward, which means that the robot needs to solve the credit assignment problem (see section 2.2.6.2) for the orientation. To enhance the complexity of learning for both different initial arm and object positions, we provide in this chapter experiments for the initial episode configurations summarized in Table 7.1.

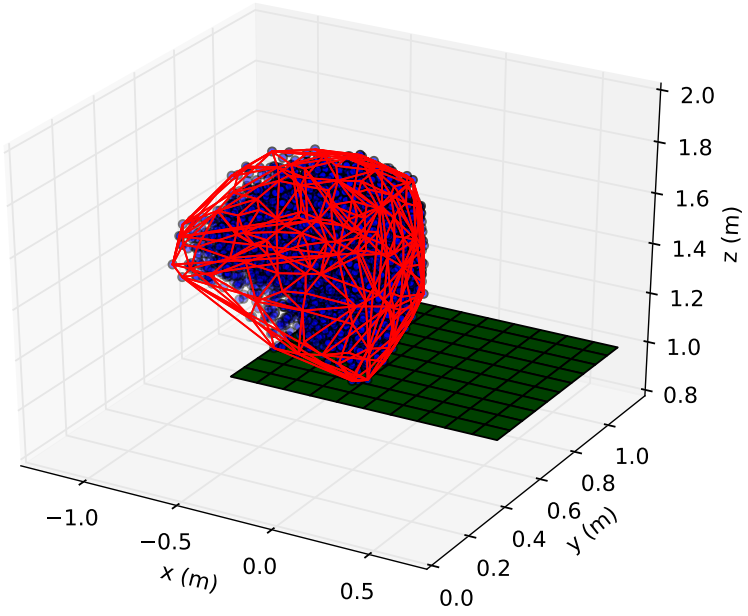


FIGURE 7.1: Representation of the volume of possible end-effector initial positions in 3D coordinates. The 3D points are in blue and the convex hull is in red. The table is visualized in green.

Name	A	B	C	D
Initial object positions	single	random	single	random
Initial arm joint angles	single	single	random	random

TABLE 7.1: Names for the different initial episode conditions

The objective of the task is defined by a sparse reward term r_{sparse} which indicates if there is palm-touching or not:

$$r_{\text{sparse}} = \begin{cases} 1, & \text{if success,} \\ p_{\text{time}} \in \mathbb{R}^-, & \text{otherwise.} \end{cases} \tag{7.1}$$

Note that the negative term p_{time} ensures that the robot looks for the quickest path to the goal. The state space S is composed of the arm and camera joint angles $\boldsymbol{q}$ as well as eight binary tactile sensors $\boldsymbol{c_b}$ attached to the fingers of the Barrett Hand. Images are not required here because we use a single object and consider that the camera joint angles give sufficient information about the 3D object position. However, they would be necessary if objects with different shapes were used in the experiments. The actions are variations of the robot joint angles: $\boldsymbol{a} = \boldsymbol{\Delta q^{robot}}$ which are seven real-valued scalars.

Specifically to the reaching learning process, the actions are made bounded to avoid too violent collisions between the arm and its environment. To this end, we apply a correction [180] to the gradient $\frac{\partial Q}{\partial a}$ (gradient used for the policy update in the DDPG) allowing to invert the sign of the gradient components if the action overpasses its limit. Furthermore, this procedure has usually the effect of downscaling

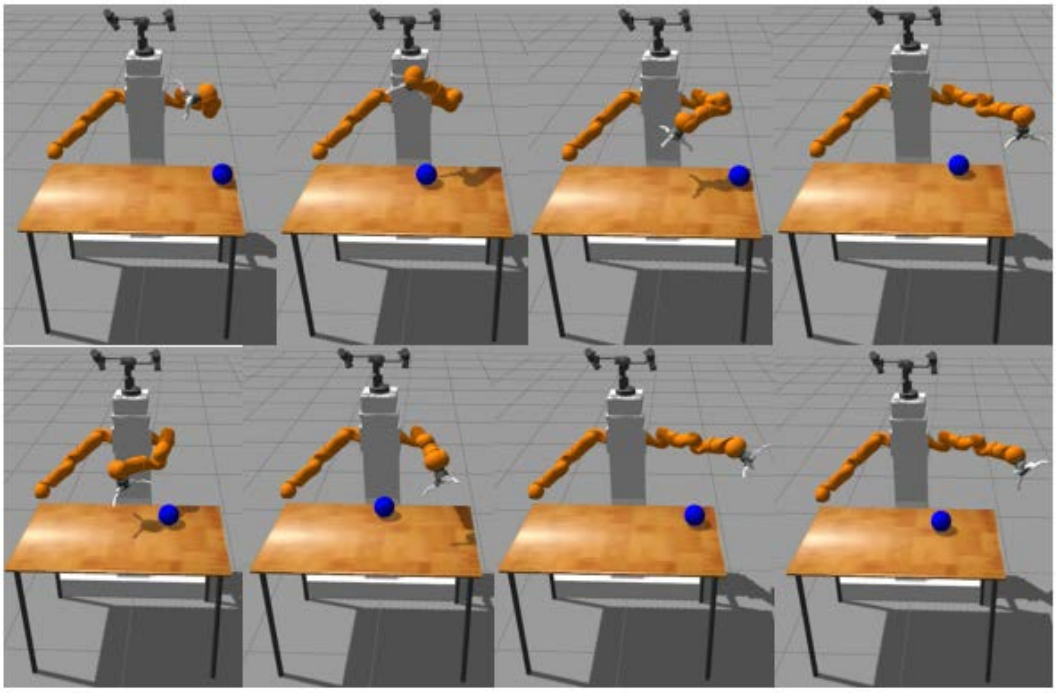


FIGURE 7.2: Random examples of initial configurations for the D problem

the gradient amplitude though this effect is completely minimized in our case by the use of the ADAM solver (ADAM update is invariant to gradient scale).

7.1.2 Reward computation

To compute the touching reward function, we use the object binocular fixation policy and the hand-eye coordination function. After the execution of an object fixation step (using the object fixation policy $\boldsymbol{\pi_\psi}$), we get the fixation camera angles $\boldsymbol{q}_{\mathbf{fix}}^{\mathbf{camera}}$ which implicitly encode the object 3D position. After that, using equation (6.1) at each time-step, the hand-eye coordination function $f_{\boldsymbol{\eta}}$ gives us $\boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}}$ which implicitly encodes the end-effector 3D position. Then, a reward shaping term r_{shCam} can be computed:

$$r_{\text{shCam}} = \begin{cases} 0, & \text{if success,} \\ c_{\text{cam}} ||\boldsymbol{q}_{\mathbf{fix}}^{\mathbf{camera}} - \boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}}||_2 - p_{\text{time}}, & \text{otherwise.} \end{cases} \tag{7.2}$$

with $c_{\text{cam}} \in \mathbb{R}^-$. r_{shCam} represents an informative term which depends on the distance between the virtual camera coordinates and the camera coordinates which make the camera system fixate the object. Thus, it encourages the end-effector to be close to the object. Note that the slope c_{cam} is chosen to ensure shaping rewards are small compared with the higher sparse reward value. Furthermore, $c_{\text{cam}} ||\boldsymbol{q}_{\mathbf{fix}}^{\mathbf{camera}} - \boldsymbol{q}_{\mathbf{virt}}^{\mathbf{camera}}||$ is negative and already penalizes the time-steps. Thus, we subtract p_{time} because this term is not useful anymore if we apply a sum of r_{shCam} and r_{sparse}.

Using these sole terms yields decent performances. However, as will be shown in section 7.1.4, the robot is badly guided in many arm positions when it is close to the table. Indeed, fewer moves are physically plausible and the algorithm can take time to learn them. To accelerate this selection, we propose a new tactile reward term $r_{\text{penContact}}$ to the reward function penalizing states where the end-effector is in

contact with the table:

$$r_{\text{penContact}} = \begin{cases} p_{\text{contact}} \in \mathbb{R}^-, & \text{if contact between the end-effector and the table,} \\ 0, & \text{otherwise.} \end{cases} \tag{7.3}$$

Indeed, by applying penalties, we hope that the robot explores areas where it is not in contact with the table, i.e. where it can move without too many constraints to the goal. To compute this term, we make the assumption that the robot knows from its tactile sensors whether it is touching the table.

Finally, our proposed reward function is built from the three previous terms:

$$r_{\text{proposedPen}} = r_{\text{sparse}} + r_{\text{penContact}} + r_{\text{shCam}} \tag{7.4}$$

The relative effect of each of these terms is evaluated in the experiments.

7.1.3 Experiments

In this section, we describe the experimental protocol. The objective of experiments is to evaluate learning performances using the proposed reward function and other ones because they allow to evaluate the relative impacts of each reward term. Besides, we wish to evaluate whether our weakly supervised reward can reach same performances as with a supervised counterpart.

7.1.3.1 Different reward functions

The reward functions which will be used in our experiments are listed below:

- $r_{\text{sparse}} = \begin{cases} 1, & \text{if success,} \\ p_{\text{time}}, & \text{otherwise.} \end{cases}$
  This reward is described by equation (7.1) and rewards the robot only when the palm touches the object. Besides, it penalizes each unsuccessful movement to encourage the robot to quickly touch the object. Note that using such a sparse reward means that we only dispense with the hand-eye coordination information. Indeed, information brought by the object fixation (the camera joint angles) is still present in the state space.

- $r_{\text{proposedPen}} = r_{\text{sparse}} + r_{\text{penContact}} + r_{\text{shCam}}$
  This is the proposed reward function (described in equation (7.4)).

- $r_{\text{proposed}} = r_{\text{sparse}} + r_{\text{shCam}}$
  This is the proposed reward function without the penalties for the contact between the end-effector and the table. This is used to show the influence of the penalty in the learning procedure.

- $r_{\text{sparsePen}} = r_{\text{sparse}} + r_{\text{penContact}}$
  We add to r_{sparse} a term penalizing contacts of the end-effector with the table.

- $r_{\text{supervisedPen}} = r_{\text{sparse}} + r_{\text{penContact}} + \begin{cases} 0, & \text{if success,} \\ c_{\text{cart}}||\boldsymbol{p} - \boldsymbol{p}_{\textbf{target}}||_2 - p_{\text{time}}, & \text{otherwise,} \end{cases}$

with $c_{\text{cart}} < 0$. To build this reward, we give a 3-dimensional end-effector target Cartesian pose $\boldsymbol{p}_{\textbf{target}}$ for the shaping part and we add a sparse reward as well as a term penalizing end-effector contacts with the table. This reward is the closest to the proposed $r_{\text{proposedPen}}$ but its shaping term requires forward kinematics and 3D object pose information. Finally, the slope c_{cart} is chosen to make the shaping term take about the same values as r_{shCam}.

We summarize all the reward functions and their attributes in Table 7.2. Note that we choose not to compare our reward function with a Cartesian shaping reward without a sparse term. Indeed, for such a reward function, a success would be to touch with the palm from a specific orientation and position. In our case, a success can be to touch with the palm in any position. The tasks are then too different to be compared in terms of touching improvement.

Reward	$r_{\text{proposedPen}}$	r_{proposed}	$r_{\text{supervisedPen}}$	$r_{\text{sparsePen}}$	r_{sparse}
Sparse term	yes	yes	yes	yes	yes
Contact penalties	yes	no	yes	yes	no
Shaping term	ours	ours	supervised	none	none

TABLE 7.2: Names for the different initial episode conditions

7.1.3.2 Material

We describe here the material we use for our experiments.

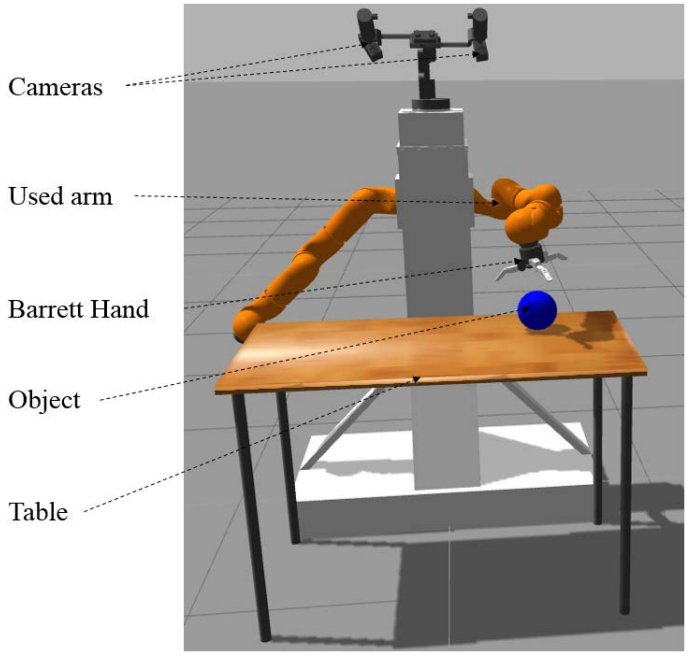


FIGURE 7.3: Scheme of the setting

Figure 7.3 presents the setting we use for our experiments. The simulations still use the Gazebo simulator with the ROS [165] middleware and we use the same robotic setting as in Chapter 6. We add to this setting a table (the same as in Chapter 5) and a blue ball put on it.

The neural network structures used for the policy and the Q functions are described in Appendix A. Tables B.3 and B.5 of Appendix B provide values for the parameters used in the experiments.

7.1.3.3 Experimental protocol

We describe how we compare the policies learned with different reward signals for the touching task. The protocol contains a training and a test phase.

Training phase

For training, we use the DDPG algorithm [95] and the previously defined reward functions. Learning happens on N_{tot} bounded-length episodes of maximal size N_{max} (the episode ends after N_{max} reinforcement learning iterations or when the robot touches the object with its palm). Each episode has an initial arm position and an object position. For the episode initialization of problems C and D, the initial robot joint angles are sampled from a set of uniform distributions (each one corresponding to a robot joint angle) and the uniform distribution limits are 23, 57, 80, 91, 103, 80 and 11° (see Figure 7.1 and 7.2 for the illustrations). For the problems B and D, the object position is uniformly chosen from a rectangular area of reachable object positions on the table (this area is displayed in black in Figure 7.10). When an initial configuration leads to a collision between the arm and its environment, the initial position is re-set until a collision-free position is sampled.

For the exploration, we use the Ornstein-Uhlenbeck process. The latter correlates the noise $\epsilon_j(t)$ for a joint at time t with the noise $\epsilon_j(t-1)$ of the same joint at time $t-1$ with the equation:

$$\epsilon_j(t) = \theta_j \mu_j + (1-\theta_j)\epsilon_j(t-1) + \left(\xi_j(t) \sim \mathcal{N}\left(0, \sigma_j\right)\right). \tag{7.5}$$

θ_j is a factor trading-off the correlation with the previous noise and the correlation with the equilibrium value μ_j and σ_j is the standard deviation of the Gaussian distribution. This exploration procedure is particularly interesting in problems in which the same action applied during several time-steps can be the optimal behaviour. For example, let us say that the optimal sequence of moves for a manipulation robotic task is to move the arm downwards with the same vector of actions during 5 time-steps. This sequence of moves is more likely to be experimented with an Ornstein-Uhlenbeck process (because of correlation) rather than with a Gaussian exploration.

As the task requires a precise orientation of the end-effector, the robot frequently blocks itself close (see Figure 7.4) to a reaching position.

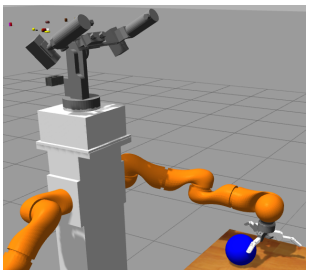


FIGURE 7.4: Example of an unsuccessful collision between the end-effector and the object

For instance, the robot can touch the object with its fingers without touching it with the palm. And, if the actions computed by the policy make the end-effector move downwards, the robot can be blocked by the table despite exploration. Thus, to avoid these situations, we handle the times when the robot is blocked without succeeding in reaching. More precisely, when the robot is blocked a backward action is taken, i.e the robot goes back to a previous contact-less position. This allows

to more correctly discriminate actions in the contact areas in the sense the robot is provided with other chances of success. Furthermore, when the robot is blocked, we recompute the actions that the robot really experiments by reading how much the arm angle encoder values have varied. The rationale is to keep the successful experimented actions as small as possible for a better simulation quality. However, we have not experimented if this was determinant for the success of our experiments.

Through the training experiments, we wish to compare our reward requiring little supervision with other ones. Consequently, for all the reward signals, in order to monitor the learning progress, we specifically plot the reaching frequency ν^{reward} over the episodes,"reward" referring to a specific reward function. The number of experiments per reward functions varies according to the difficulty of the problem and is specified later for each problem (for better readability of the curves, we use the same exponential smoothing defined by equation (5.11) in Chapter 5 with $\omega = 0.003$). Furthermore, we record N_1 the number of episodes it took to get a first reaching success, N_{90} the number of RL iterations it took to reach and remain above reaching performance of 90 %.

Algorithm 8 summarizes the experimental protocol for the learning of the reaching skills.

Test phase

To evaluate the learned policies, we apply them without any exploration noise on N_{tot} random episodes and we compute the touching frequency $\nu_{\text{test}}^{\text{reward}}$. Note that we do not apply the systematic backward motion used in the training phase to deal with blocked situations. Instead, when the robot is blocked, it just keeps following the learned policy.

7.1.4 Results

This section presents the experimental results for the 4 initial configurations presented in Table 7.1.

7.1.4.1 Problem A

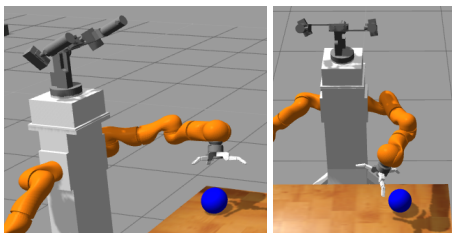


FIGURE 7.5: Two views of the initial configuration for the case A

The results of the A case (single initial arm joint angles, single object position, see Figure 7.5) are presented in Figure 7.6 and Table 7.3. Figure 7.6 displays the average (four runs for each reward function) reaching frequencies of the previously described reward functions.

For these results, we make the following observations:

- For the experiments not using shaping reward terms, the robot never touches the object. This means that given the exploration strategy, the initial policy and the initial configuration of Figure 7.5, the probability to touch the object is very low using

Algorithm 8 Algorithm describing the reaching learning procedure

Parameters:

N_{trans}: the size of the circular buffer
γ: the discount factor
Γ: the set of parameters related to the Ornstein-Uhlenbeck process
N_{tot}: the total number of episodes
N_{max}: the maximal number of iterations per episode
N_{b}: the number of transitions per batch

Inputs:

π_{ψ}^{fix}: the fixation policy
f_{η}: the hand-eye coordination mapping

Outputs: Q_{ϕ}, π_{θ}

Steps:

Initialize the transition circular buffer T_{buf}, Q_{ϕ} and $\pi_{\boldsymbol{\theta}}$
Set $n_{\text{eps}} \leftarrow 0$
while $n_{\text{eps}} < N_{\text{tot}}$ **do**
    Set $t \leftarrow 0$
    Reset arm joint angles and place the object according to the type of problem (A, B, C or D)
    Fixate the object using π_{ψ}^{fix}
    Go to arm joint angles according to the type of problem (A, B, C or D)
    while !*cond* **do**
        if blocked **then**
            Apply a backward action $\boldsymbol{a_t}$ to go to the 7th former arm joint angles.
        else
            Update $\boldsymbol{\epsilon}^{\Gamma}(t)$ using equation (7.5)
            Apply $\boldsymbol{a_t} = \pi_{\boldsymbol{\theta}}(\boldsymbol{s_t}) + \boldsymbol{\epsilon}^{\Gamma}(t)$
        Detect if the robot is blocked ($\boldsymbol{a_t}$ not completed)
        if blocked **then** adjust $\boldsymbol{a_t}$
        Compute the reward signal r_t and sense $\boldsymbol{s_{t+1}}$
        Add $< \boldsymbol{s_t}, \boldsymbol{a_t}, r_t, \boldsymbol{s_{t+1}} >$ to T_{buf}
        Pick randomly N_{b} transitions from T_{buf}
        Update Q_{ϕ} and π_{θ} using DDPG (equations (2.46) and (2.48))
        $t = t + 1$
        $cond = (Success)$ or $(t == N_{\text{max}})$
    $n_{\text{eps}} = n_{\text{eps}} + 1$

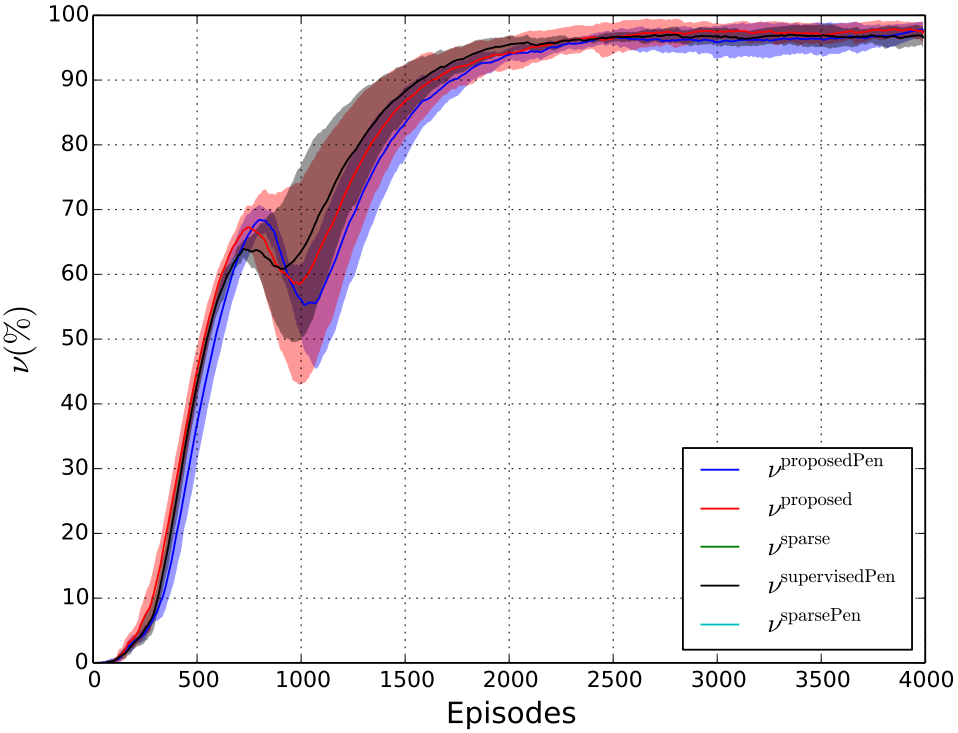


FIGURE 7.6: Evolution of the average reaching frequency during training for the different reward functions for the problem A

Reward	$r_{\text{proposedPen}}$	r_{proposed}	$r_{\text{supervisedPen}}$	$r_{\text{sparsePen}}$	r_{sparse}
N_1	100 ± 38	96 ± 10	83 ± 21	N/A	N/A
N_{90}	$(1.0 \pm 0.1) \times 10^5$	$(0.9 \pm 0.2) \times 10^5$	$(0.9 \pm 0.1) \times 10^5$	N/A	N/A
$\nu_{\text{test}}^{\text{reward}}$ (%)	100	100	100	0	0
σ^{N_1}	38	11	21	N/A	N/A
$\sigma^{N_{90}}$	0.1×10^5	0.2×10^5	0.1×10^5	N/A	N/A
$\sigma^{\nu_{\text{test}}^{\text{reward}}}$ (%)	0	0	0	0	0

TABLE 7.3: Values featuring learning velocity (N_1 and N_{90}), final reaching frequency ($\nu_{\text{test}}^{\text{reward}}$) and associated standard deviations in the problem A

only sparse rewards. Yet, the initial configuration of the problem should not make the problem too complex since the robot has just (almost) to move downwards the arm to touch the object.

- For the experiments using shaping reward terms, we exhibit the same perfect final reaching performances (see the $\nu_{\text{test}}^{\text{reward}}$ values in Table 7.3) and the confidence intervals of Figure 7.6 are not separate enough to draw conclusions about a hierarchy of settings. The variance is zero for the final reaching performances (in test conditions i.e. without exploration noise) because the episodes have exactly the same initial configurations in terms of camera and arm joint angles. Furthermore, we observe that the N_1 and N_{90} values are a bit lower for $r_{\text{supervisedPen}}$. Therefore, the learning velocity and the initial touching probability are slightly higher than with our proposed reward functions. Nevertheless, our proposed reward functions yield similar results and the confidence intervals of N_1 and N_{90} do not allow to draw definitive conclusions.

- Besides, for the experiments using shaping reward terms, we observe a drop of performances starting before 1 000 episodes. This is explained by the fact the end-effector initial position is not perfectly aligned with the initial object position. When learning, at the first iterations, the gradients of actions $\left(\frac{\partial Q}{\partial a_1}, ..., \frac{\partial Q}{\partial a_i}, ..., \frac{\partial Q}{\partial a_7}\right)$ of the memory buffer are of the same direction. These gradients are applied to optimize the policy and at some point, the robot starts to overshoot the object position. This requires some time for the Q function to take into account these bad performances and to update itself accordingly. In order to reduce this delay, a prioritized experience replay strategy [181], [182] could be used. This would consist in updating transitions with the highest temporal-difference values in the memory buffer (for the DDPG update) instead of updating random transitions. With such a strategy, the transitions which occur when the robot overshoot the object position would be temporarily updated more frequently.

- Finally, for this setting, the use of penalties to discourage the robot to touch the table does not affect learning. Indeed, we do not observe a meaningful difference between respectively $r_{\text{sparsePen}}$ and r_{sparse}, and $r_{\text{proposedPen}}$ and r_{proposed}. This is explained by the fact in the first cases the robot never touches the object and in the second cases, the problem is too easy and the robot almost never touches the table.

7.1.4.2 Problem B

The results of the B case (single initial arm joint angles, random object positions) are presented in Figure 7.7 and Table 7.4. Figure 7.7 displays the average (two runs for each reward function) reaching frequencies of the experimented reward functions.

Reward	$r_{\text{proposedPen}}$	r_{proposed}	$r_{\text{supervisedPen}}$	$r_{\text{sparsePen}}$	r_{sparse}
N_1	136±20	126±32	121±4	N/A	N/A
N_{90}	(8±2)×10^5	(4.4±0.1)×10^5	(7±1)×10^5	N/A	N/A
$\nu_{\text{test}}^{\text{reward}}$ (%)	99±2	99±1	99.9±0.1	0	0
σ^{N_1}	14	23	3	N/A	N/A
$\sigma^{N_{90}}$	1×10^5	0.8×10^5	0.8×10^5	N/A	N/A
$\sigma^{\nu_{\text{test}}^{\text{reward}}}$ (%)	1	0.7	0.01	0	0

TABLE 7.4: Values featuring learning velocity (N_1 and N_{90}), final reaching frequency ($\nu_{\text{test}}^{\text{reward}}$) and associated standard deviations in the problem B

We notice several important facts:

- Learning the reaching task for several object positions can work very well because the robot always reaches very good touching performances with the reward functions using shaping terms. This shows that the camera joint angles integrated in the state space encode sufficiently well the object position. Therefore, we confirm that the object localization is well encoded through the camera joint angles.

- Learning with sparse rewards using several initial object positions and still a unique initial arm position does not improve learning. Indeed, we do not observe a single success, which indicates that using several initial object positions does not increase the probability of touching enough.

- Using shaping rewards efficiently increases the probability of success and in all the settings, we get very good policies.

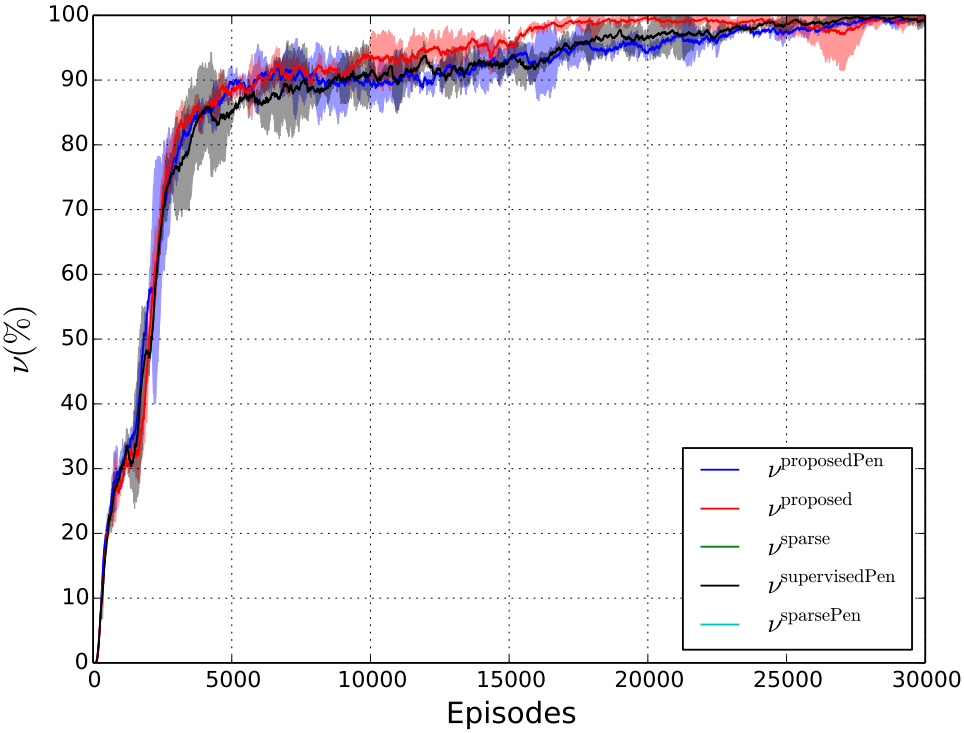


FIGURE 7.7: Evolution of the average reaching frequency during training for the different reward functions for the problem B

- It seems that learning with contact penalties is slower than without in this context. This means that in this context, it is useful to explore around areas where there is contact between the table and the end-effector.
- We do not observe statistically significant differences between the supervised setting and our proposed one, which confirms the results of the setting A.
- We do not observe the sudden drop of performances that is present in the problem A. It is explained by the fact the robot tries to reach several object positions, so that the gradients of actions $(\frac{\partial Q}{\partial a_1}, ..., \frac{\partial Q}{\partial a_i}, ..., \frac{\partial Q}{\partial a_7})$ are more diverse, and the reaching policy is not trained to do a single kind of move.

7.1.4.3 Problem C

The results of the C case (random initial arm joint angles, single object position) are presented in Figure 7.8 and Table 7.5. Figure 7.8 displays the average (four runs for each reward function) reaching frequencies of the experimented reward functions.

From the results displayed in Table 7.5 and in Figure 7.8, we draw the following observations:

- With several initial arm positions, learning with sparse rewards is much more tractable than with a single initial position (problem A). It can be surprising at first because we can assume that by increasing the number of initial positions, the problem becomes more and more complex. However, by increasing the number of initial arm positions, the robot is also more likely to touch the object. Indeed,

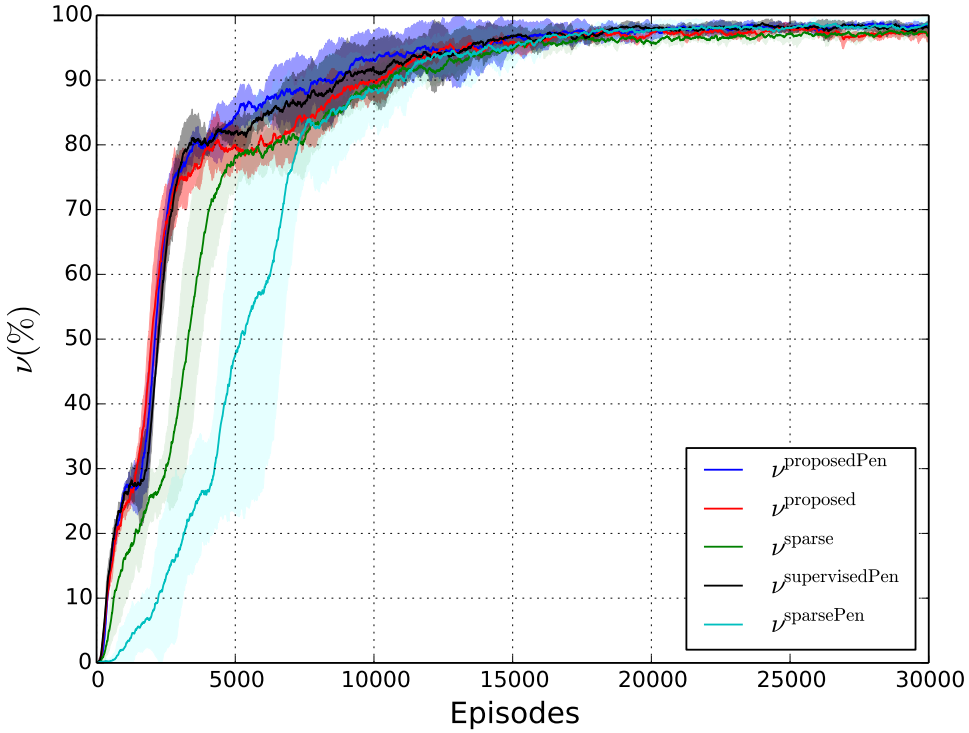


FIGURE 7.8: Evolution of the average reaching frequency during training for the different reward functions for the problem C

Reward	$r_{\text{proposedPen}}$	r_{proposed}	$r_{\text{supervisedPen}}$	$r_{\text{sparsePen}}$	r_{sparse}
N_1	85±50	102±47	81±37	726±1237	50±27
N_{90}	(6±2) $\times 10^5$	(6.2±0.7) $\times 10^5$	(6±2) $\times 10^5$	(8 ±2) $\times 10^5$	(7±1) $\times 10^5$
$\nu_{\text{test}}^{\text{reward}}$ (%)	97.9±0.8	96±1	97.9±0.9	97.7±0.7	96.8±0.8
σ^{N_1}	51	48	38	1262	27
$\sigma^{N_{90}}$	2 $\times 10^5$	0.7 $\times 10^5$	2 $\times 10^5$	2$\times 10^5$	1$\times 10^5$
$\sigma^{\nu_{\text{test}}^{\text{reward}}}$ (%)	0.8	1	1	0.8	0.8

TABLE 7.5: Values featuring learning velocity (N_1 and N_{90}), final reaching frequency ($\nu_{\text{test}}^{\text{reward}}$) and associated standard deviations in the problem C

some initial arm positions bring the robot end-effector very close to the object and the arm has to move only a little to touch the object (for some positions, the robot touches the object in three time-steps). Therefore, with this initial setting and using only sparse rewards, the robot can be initialized close to the object. Then, it can touch with a high probability the object and somehow learns from easy missions to reach from all the initial positions.

- Learning is still faster with the use of shaping reward terms, because the robot takes less time to touch the object.
- The penalty term seems to influence the learning velocity. Indeed, learning with $r^{\text{proposedPen}}$ is faster than learning with r^{proposed}. This is explained by the fact some initial arm positions result (through exploration) in blocked situations in which the end-effector is in contact with the table. Penalizing such situations helps in

this case to accelerate learning. However, we observe a reverse inequality when we do not use shaping reward terms: learning with r^{sparse} is faster than learning with $r^{\text{sparsePen}}$. Indeed, when using shaping rewards, the end-effector is attracted by the object such that the penalty term cannot make the policy choose upward arm moves. When we do not use shaping rewards, the penalty term becomes more important and sometimes this makes the policy choose upward moves to avoid to touch the table.

- At the end of experiments, we get very good and comparable results for $r^{\text{proposedPen}}$, $r^{\text{supervisedPen}}$ and $r^{\text{sparsePen}}$. The results for r^{proposed} and r^{sparse} are slightly worse than with the other reward signals. This indicates that the penalty term plays a significant role for the learning performances if the episode starts with "complex initial arm positions", i.e. arm positions which require substantial end-effector orientation movements to touch the object.

- Here again, we do not observe the sudden drop of performances as in the problem A. It is explained by the fact the robot explores from many initial arm configurations, so that the gradients of actions $(\frac{\partial Q}{\partial a_1}, ..., \frac{\partial Q}{\partial a_i}, ..., \frac{\partial Q}{\partial a_7})$ are more diverse, and the reaching policy is not trained to do the same kind of move.

7.1.4.4 Problem D

Finally, we present results for the setting with multiple initial arm positions and object positions (6 runs per reward function were conducted).

Table 7.6 presents the final performances of the different policies as well as the number of episodes it takes to get a first reaching success and the number of RL iterations it takes to reach (and remain above) an average reaching performance of 90 %.

Reward	$r_{\text{proposedPen}}$	r_{proposed}	$r_{\text{supervisedPen}}$	$r_{\text{sparsePen}}$	r_{sparse}
N_1	95±16	110±19	105±29	7505±6918	8214±4145
N_{90}	(1.7±0.2) $\times 10^6$	(2.3±0.1) $\times 10^6$	(1.5±0.1) $\times 10^6$	N/A	N/A
$\nu_{\text{test}}^{\text{reward}}$ (%)	95±1	91±3	97.2±0.7	60±40	82±17
σ^{N_1}	20	23	36	8646	5180
$\sigma^{N_{90}}$	0.3$\times 10^6$	0.2$\times 10^6$	0.2 $\times 10^6$	N/A	N/A
$\sigma^{\nu_{\text{test}}^{\text{reward}}}$ (%)	2	4	0.8	47	22

TABLE 7.6: Values featuring learning velocity (N_1 and N_{90}), final reaching frequency ($\nu_{\text{test}}^{\text{reward}}$) and associated standard deviations in the problem D

Figure 7.9 shows the experimental training curves as well as associated confidence intervals. We can notice several important facts:

- With the use of sparse-only rewards, the probability of getting the first success is low. It takes a lot of episodes to reach a first success (from 7 505 episodes for the N_1 values). Furthermore, we cannot have a precise idea about the time when the first success occurs because the standard deviations are very high. Moreover, N_{90} values are not available for these two reward settings because some of the runs were not successful at all. Finally, as shown in Figure 7.9, the confidence intervals for the average reaching frequency are very large, which means that the average estimation is not precise at all for the sparse reward settings. The only fact we can notice for these settings is that it can work for a run and totally fails for another one. Then, these reward functions do not ensure a reliable learning for the most complex initial configuration setting.

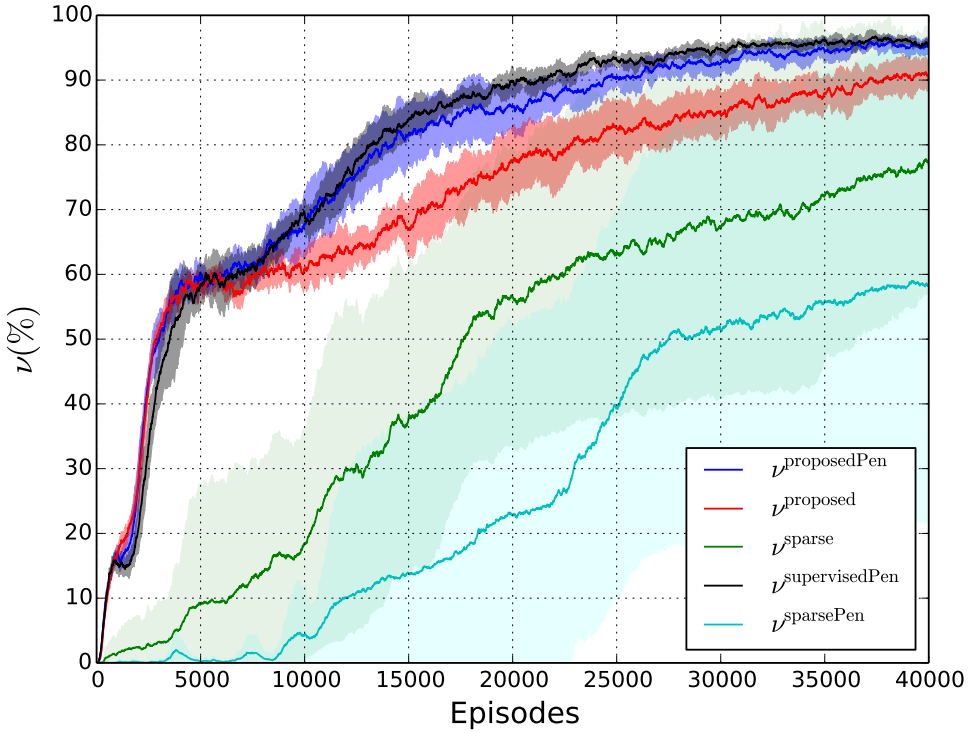


FIGURE 7.9: Evolution of the average reaching frequency during training for the different reward functions for the problem D

- With a shaping term, the probability of having first successes is much higher. The different N_1 values for $r_{\text{proposedPen}}$, r_{proposed}, $r_{\text{supervisedPen}}$ are of the same order of magnitude, are small, and exhibit low standard deviations. Importantly, our weakly-supervised setting allows to approach similar reaching performances compared with its supervised counterpart even if the final reaching frequency is slightly lower (95 % with little supervision vs 97 % with supervision). In addition, Figure 7.9 shows that the confidence intervals of $\nu^{\text{supervisedPen}}$ and $\nu^{\text{proposedPen}}$ intertwine even if the bounds of $\nu^{\text{supervisedPen}}$ are generally higher. This shows that even if $\nu^{\text{supervisedPen}}$ is higher than $\nu^{\text{proposedPen}}$ most of the time, results are close. Furthermore, we notice that three phases can be distinguished. From 0 to about 5 000 episodes, the reward curves increase with the same velocity. It corresponds to a phase in which some initial positions are mastered without substantial end-effector orientation changes. Indeed, for some initial positions, the robot has to change only a little its end-effector orientation to reach a grasping posture. After 5 000 episodes, there is a period of slow increase for the three settings and from about 7 000 episodes, "harder" initial positions are more and more mastered. We observe that the three settings start to distinguish from each other and the term penalizing contacts seems to be a decisive factor.

- Indeed, as in the problem C, jointly using a shaping signal and a term penalizing contacts between the end-effector and the table makes learning faster. To show this, we can compare $r_{\text{proposedPen}}$ and r_{proposed}: N_{90} is lower for $r_{\text{proposedPen}}$, $\nu_{\text{test}}^{\text{proposedPen}}$ is higher than $\nu_{\text{test}}^{\text{proposed}}$, and Figure 7.9 shows that $\nu^{\text{proposedPen}}$ is always superior to ν^{proposed} after 10 000 episodes. Furthermore, in Figure 7.9 we notice that the upper bound M^{proposed} is generally inferior to the lower bound

$m^{\text{proposedPen}}$. All of these observations show the supremacy of $r^{\text{proposedPen}}$ over r^{proposed}. It shows that penalizing contacts between the end-effector and the table has an important influence on learning performances. The reason is that it is easier to experiment "good" moves in contact-less areas given that the robot can easily be blocked when it touches the table. It can seem in contradiction with conclusions of the problem B (one initial arm position, several object positions) in which using r^{proposed} yields better results than with $r^{\text{proposedPen}}$. The reason is that for the specific initial arm position, it is easier to touch while also being in contact with the table. This is not the case in most arm positions of the problem D. Indeed, in many cases, the end-effector is stuck to the table and cannot reach the object. Thus, the penalties play an important role to deal with this issue. Finally, we observe as in the problem C that if we do not use shaping terms, the penalty term becomes too important and makes the policy choose upward arm moves (as the table acts as a repellent), which considerably slows down the learning (compare the learnings for r^{sparse} and $r^{\text{sparsePen}}$ in Figure 7.9).

- From Figure 7.9, we can notice that some runs could have been extended because the convergence is not totally achieved. However as they can take a very long time, they were not continued. We choose to stop when our reward function with penalty terms yields good performances.

7.1.5 Conclusion

We have shown that a hand-eye coordination mapping and an object fixation error were sufficient ingredients to learn reaching skills and that the resulting performances approach those with policies learn with supervised reward signals. Moreover, we have demonstrated that penalizing contacts between the end-effector and the object improves the learning speed for the most complex initial setting and can have a negative impact if the robot can easily reach successful positions from positions in which it is in contact with the table. The experiments in the settings A and C have been published in [5] with some modifications of hyperparameters (the sizes of the memory buffer and the mini-batch are lower in the paper), less types of reward functions (we did not present experiments with contact penalties in the paper) and without the tactile states. The work presented in the setting D has been published in [6]. In the next section, we show how the robot can learn both reaching skills and the ability to assess about the object reachability.

7.2 Learning object reachability

In this section, we describe how the robot learns about its reachability while learning to reach using exactly the same priors as before. It is interesting to make the robot learn about its reachability area itself because without it, humans have either to place the object in reachable areas so that the robot is always sure to be able to reach (like in section 7.1), or they have to compute the reachability area themselves. Furthermore, it is interesting that the robot knows about its reachability area because it can be a powerful tool for a mobile manipulator. Indeed, the latter might learn using its reachability prediction to move such that an object becomes reachable.

7.2.1 Task definition

We briefly describe here some state-of-the-art work on the workspace reachability estimation in manipulation robotics. These methods use different representations for the reachable workspace.

[183] represents the reachable workspace of a humanoid robot as a set of 3D Cartesian points. It samples random 3D positions and check (1) if inverse kinematics provide a solutions and (2) if the center of mass of the robot at the computed joint configuration is within an area. If so, the point is added to the database of reachable points. Goal babbling [184] can also be considered as a way to learn reachable areas in terms of 3D points while learning inverse kinematics (provided forward kinematics is known). This method presents a way to learn inverse kinematics with a random exploration of 3D goals. [185], by assuming the knowledge of both inverse and forward kinematics, develops an interesting visualization algorithm for the reachability of the workspace and the number of joint configurations which can be used for a given 3D position.

All of these methods assume the knowledge of inverse or/and forward kinematics. In contrast, [186] and [187] can dispense with it, because a reachable point is defined by camera joint angles $\boldsymbol{q}^{\mathbf{camera}}$. Indeed, they can feature a 3D position and the previously mentioned papers use it to build a reachability mapping: $R = Re(\boldsymbol{q}^{\mathbf{camera}})$ through exploration of different $\boldsymbol{q}^{\mathbf{camera}}$ positions. In this work, Re is approximated by a LWPR neural network and the generated targets are $R_{\mathrm{T}} = \frac{1-er}{2}$ (er is a scaled distance in the camera joint angle space between the end-effector and the fixated point) if the target is not reached and $R_{\mathrm{T}} = \frac{1+optarm}{2}$ if the target is reached, $optarm$ being an indicator value of the optimality of the arm joint configuration.

We inspire from the latter method in the sense (1) a reachable point is defined by camera joint angles, (2) a mapping $R = Re(\boldsymbol{q}^{\mathbf{camera}})$ is learned. However, we exhibit some differences. First, the way to check whether the robot reaches the point is achieved by a tactile sensor instead of measuring distances between the end-effector and the fixated point (in terms of camera joint angles). Second, we do not need markers and hand-crafted segmentation algorithms in our framework.

As regards the learning process, reaching is learned exactly the same way as in section 7.1, using the same Markov decision process (same state space, same action space, and same reward function). For the reachability, we approximate it by a feed-forward neural network: $R = Re_{\alpha}(\boldsymbol{q}^{\mathbf{camera}})$ of parameter vector α. The targets for the supervised learning of the camera are generated based on the tactile sensor, when the robot exhibits good reaching performances. For a given episode, we record the pair $(\boldsymbol{q}^{\mathbf{camera}}, R_{\mathrm{T}})$, with R_{T} computed as follows:

$$R_{\mathrm{T}} = \begin{cases} 1, & \text{if success,} \\ 0, & \text{otherwise.} \end{cases} \tag{7.6}$$

Another important aspect of our reachability application is that we only learn object reachability when the object is put on the table. We do not make the robot learn if it can reach a random point in the 3D space.

7.2.2 Ground-truth object reachability estimation

To evaluate the quality of our reachability network, we need a way to estimate the ground-truth workspace reachability (see Appendix E for its computation). Figure 7.10 displays a representation of the latter. The reachable map is delimited by

two circular arcs and the left and bottom borders of the table. The black area corresponds to the initial object position distribution for prior experiments (problems B and D).

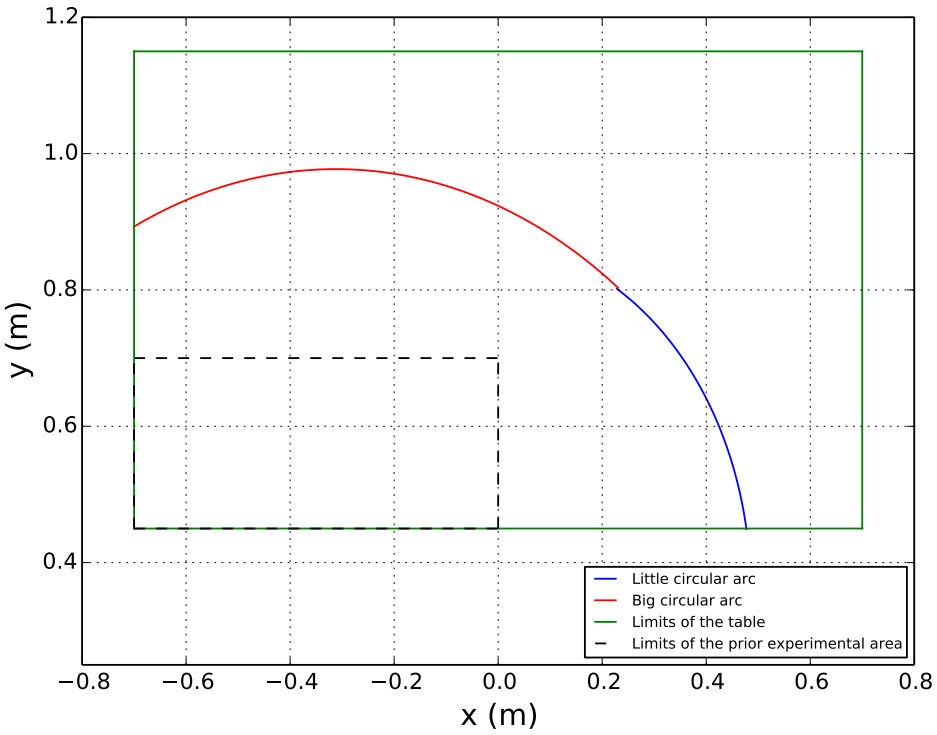


FIGURE 7.10: Scheme of the reachable workspace

7.2.3 Experiments

The experiments have two main objectives, evaluating our learned reachability predictor, and checking whether reaching is still correctly learned.

7.2.3.1 Reachability

We want to evaluate how accurate is the learned reachability mapping compared to the estimated ground-truth reachability. In that end, we first plot through the episodes the prediction error, which is the difference between the prediction of the learned mapping and the ground truth prediction. Second, we evaluate the reachability map on a grid covering all the object positions on the table by comparing and evaluate the results with respect to ground-truth predictions.

Figure 7.11 represents the evolution of the reachability prediction accuracy over the episodes with confidence intervals. We observe that the reachability prediction seems to reach around 95 % of performances, which is interesting. However, this curve does not provide any spatial clues, i.e. we do no know where the reachability predictor is wrong.

Figure 7.12 gives us a spatial view on object reachability prediction. For this figure, we cover a regular grid of object positions on all the table. At each object position, the cameras fixate the object using π_{ψ}^{fix} and we get camera joint angles $\boldsymbol{q}_{\textbf{fix}}^{\textbf{camera}}$ which make the camera system fixates the object. Then, we output the prediction R

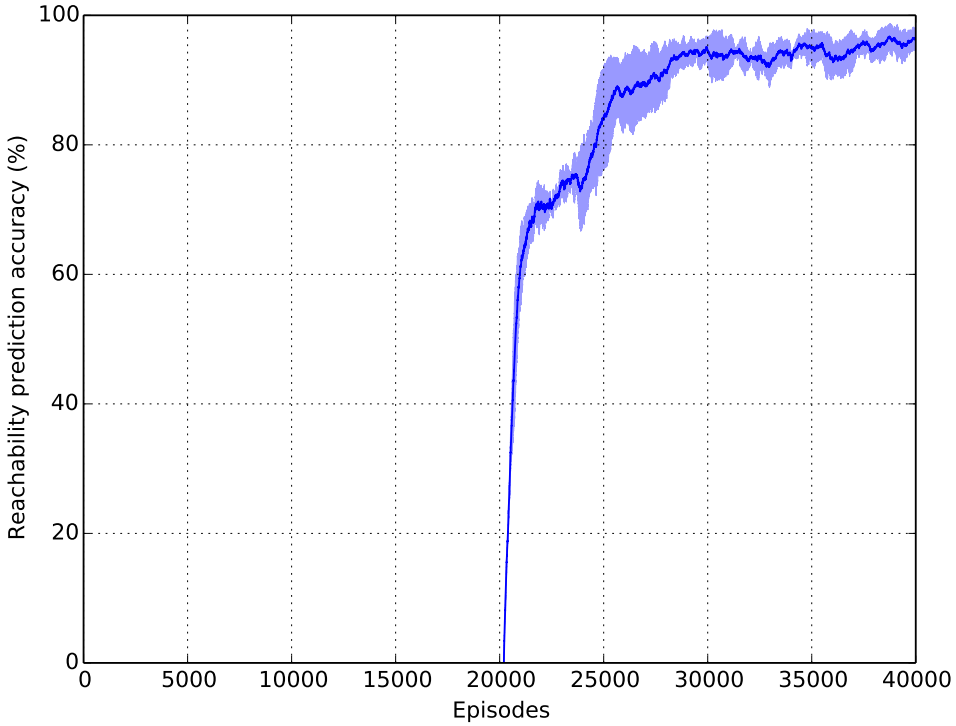


FIGURE 7.11: Evolution of the accuracy of the reachability predictor with respect to the theoretical estimation

based on the reachability predictor and these camera joint angles: $R = Re_{\alpha}(\boldsymbol{q}_{\mathbf{fix}}^{\mathbf{camera}})$. The first sub-figure presents the unmodified predictions of the reachability prediction network, which is a real value between 0 and 1. The other sub-figure presents a way to do prediction with the learned reachability predictor. A binary prediction is applied, i.e. if R is superior to 0.5, the prediction is set to 1 and 0 otherwise.

We do the following observations:

- We observe that for positions far from the "reachability border", the prediction is always good.
- For the positions close to the border, the prediction is sometimes wrong, particularly for the border built with the second circular arc.

These mistakes close to the workspace border can be due to several factors. First, it can be due to the localization inaccuracies. Indeed, we saw in Chapter 5 that the object localization principle was not very precise. While this can explain some errors for the border delimited by the first circular arc, it cannot explain ones for the border delimited by the second circular arc. Indeed, as shown by Figure 5.16 of Chapter 5, the difference of localization precision between the two borders is not large. Furthermore, in all the sub-figures, there is a reachable area close to the second border that the predictor classifies as not reachable. The reason is that the robot has not completely learned to reach all its reachable positions. More precisely, we have observed that the hand-eye coordination function does not produce accurate localization when the end-effector approaches this border. Therefore, expanding this hand-eye coordination mapping to additional arm joint angles might help to solve the issue.

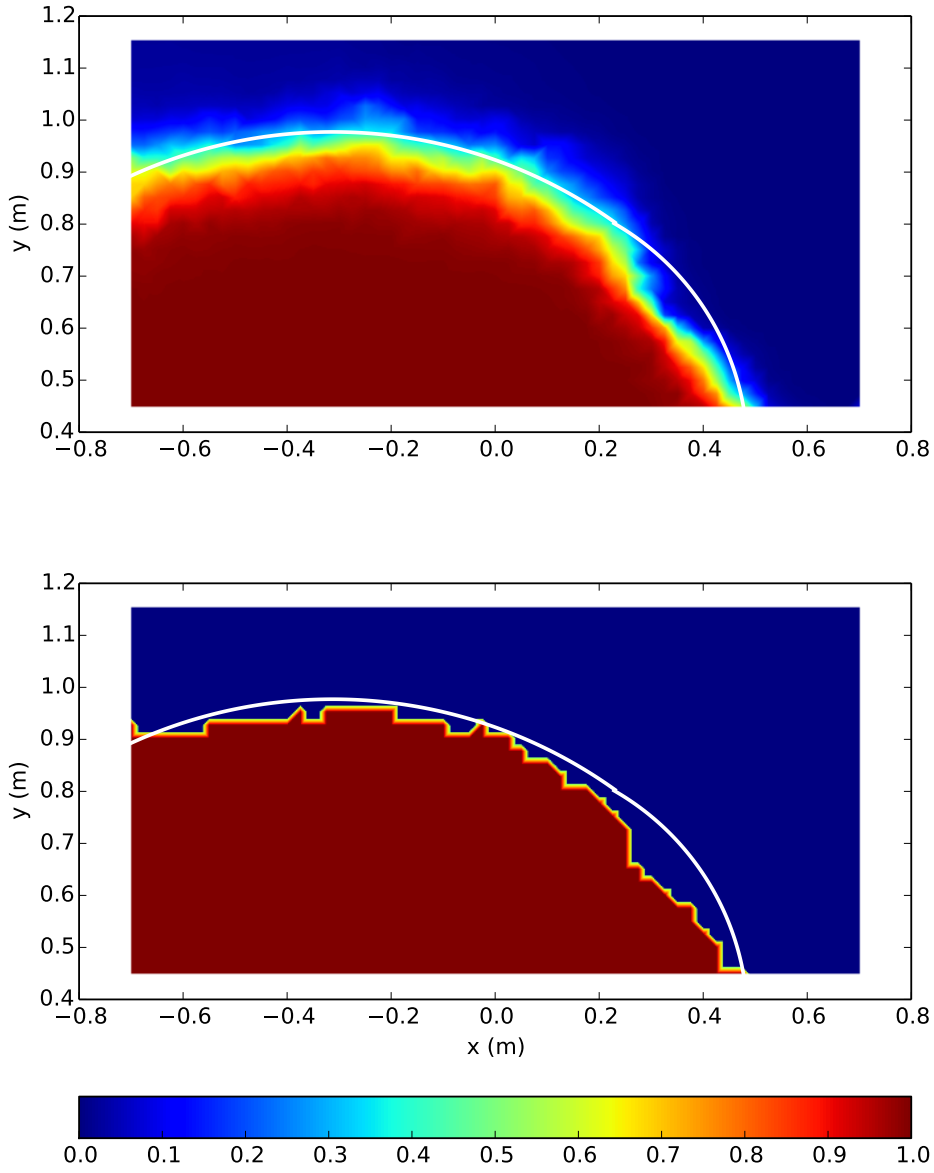


FIGURE 7.12: Evaluation of the reachability predictor accuracy on the table. In all the figures, the white curve stands for the border of the estimated object reachability area. The figure at the top plots the output of the reachability predictor (real value between 0 and 1), the one at the bottom represents the binarized reachability predictor (0 or 1).

These experiments show that even though our object localization principle is not very precise, a decent reachability predictor can be learned. The latter can accurately classifies object positions which are far from the borders. To improve the predictor quality for areas close to the borders, the first hint is to improve the object localization precision. Second, the hand-eye coordination can be expanded to a more diverse set of arm joint angles. And, third, the experiments can run longer. As the

latter solution can take a large amount of time, dedicated exploration strategies can be applied. For instance, the robot can explore only object positions close to the border, or areas which have prediction values between 0.4 and 0.6. This solution is left for future work.

7.2.3.2 Reaching performances

We want to confirm that we can learn to reach an object with minimal supervision but within the whole reachable area. To that end, we evaluate through episodes the success of the robot with respect to its ground-truth object reachability $S(\text{eps})$.

$$S(\text{eps}) = \begin{cases} 1, & \text{if ((success) and (reachable)) or((failure) and(unreachable)),} \\ 0, & \text{otherwise.} \end{cases} \tag{7.7}$$

We plot through the episodes the frequency of 1 values in S_{eps} values, which resembles the reaching frequency of section 7.1. We average the results of three trials and Algorithm 9 summarizes the procedure.

In Figure 7.13, we evaluate whether the reaching policy can match results of the theoretical reachability prediction results. In other terms, if the theoretically estimated reachability prediction tells that an object position is reachable, we evaluate if the robot reaches it. In contrast, if the robot is not able to reach it, we always consider this situation as a success. We notice that the reaching performances converge to decent performances (around 95%). This shows that our principle allows to learn to reach in most object reachable areas.

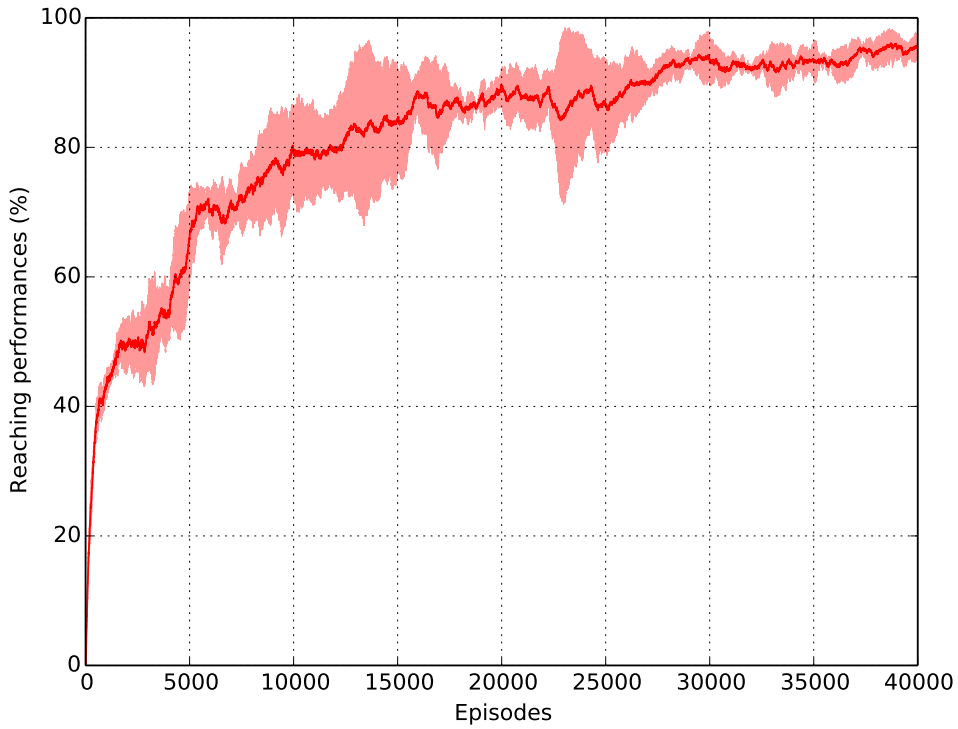


FIGURE 7.13: Evolution of the reaching performances for the learned policies

Algorithm 9 Algorithm describing the joint learning of reaching and predicting reachability

Parameters:

N_{trans}: the size of the circular buffer
γ: the discount factor
Γ: the set of parameters related to the Ornstein-Uhlenbeck process
N_{tot}: the total number of episodes
N_{max}: the maximal number of iterations per episode
N_{b}: the number of transitions per batch
N_{upd}: the number of samples in the reachability database required to start the learning of reachability
N_{start}: the number of episodes required to start the learning of the reachability

Inputs:

$\boldsymbol{q_0^{\text{robot}}}$: initial arm joint angles
π_{ψ}^{fix}: the fixation policy
$f_{\boldsymbol{\eta}}$: the hand-eye coordination mapping

Outputs: Q_{ϕ}, $\pi_{\boldsymbol{\theta}}$, $Re_{\boldsymbol{\alpha}}$

Steps:

Initialize T_{buf}, Q_{ϕ}, $\pi_{\boldsymbol{\theta}}$ and $Re_{\boldsymbol{\alpha}}$
Set $n_{\text{eps}} \leftarrow 0$
while $n_{\text{eps}} < N_{\text{tot}}$ **do**
    $t \leftarrow 0$
    Reset arm joint angles and place the object randomly on the table
    Fixate the object using π_{ψ}^{fix}
    Go to the position $\boldsymbol{q_0^{\text{robot}}}$
    while !$cond1$ **do**
        if blocked **then**
            Apply a backward action $\boldsymbol{a_t}$ to go to the 7th former arm joint angles.
        else
            Update $\boldsymbol{\epsilon}^{\Gamma}(t)$ using equation (7.5)
            Apply $\boldsymbol{a_t} = \pi_{\boldsymbol{\theta}}(\boldsymbol{s_t}) + \boldsymbol{\epsilon}^{\Gamma}(t)$
        Detect if the robot is blocked ($\boldsymbol{a_t}$ not completed)
        if blocked **then** adjust $\boldsymbol{a_t}$
        Compute the reward signal r_t and sense $\boldsymbol{s_{t+1}}$
        Add $< \boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}} >$ to T_{buf}
        Pick N_{b} random transitions from T_{buf}
        Update Q_{ϕ} and $\pi_{\boldsymbol{\theta}}$ using the DDPG
        if $n_{\text{eps}} \geq N_{\text{start}}$ **then**
            Append $(\boldsymbol{q^{\text{camera}}}, Success)$ to $\mathcal{D}$
        $cond2 = (t \geq N_{\text{start}})$ and $(size(\mathcal{D}) \geq N_{\text{upd}})$
        if $cond2$ **then**
            Update Re_{α}
        $t = t + 1$
        $cond1 = (Success)$ or $(t == N_{\text{max}})$
    $n_{\text{eps}} = n_{\text{eps}} + 1$

7.3 Conclusion

In this chapter, we have shown several important results:

- First, using learned object fixation policies and hand-eye coordination functions, the robot reliably learns to reach an object with a variety of initial conditions. This is achieved without kinematics, calibration parameters, pre-processing modules or expert demonstrations and we claim these specific features as the main contributions of our work.
- Second, we have enhanced the combined influence of reward terms and initial settings. We summarize the most striking facts. When there is a single initial arm position (which is yet favourable: the end-effector is well-oriented), learning without shaping rewards does not work. However, using different initial arm positions significantly increases the probability of success and makes learning reliable when the object position is fixed. This is because some initial arm positions are made very close to successful positions. Shaping rewards make learning practical in all the settings. Penalty terms are interesting when contacts between the table and the end-effector prevent the robot from being blocked against the table. When it is not the case, penalty terms can have a negative impact. Importantly, in all the cases, our proposed reward function produces results similar to ones obtained with a supervised reward.
- Finally, we have shown that by still using the same level of supervision, our system can get a decent level of reachability prediction accuracy (95 %) while still learning to reach objects in all the reachable area. As mentioned above, this is a promising result for a mobile manipulator.

Chapter 8

Conclusion

This chapter ends the manuscript. First, our contributions are summarized. Second, we enhance what are the limitations of our work. Then, we describe some guidelines potentially solving the limitations of our work. Finally, we give preliminary ideas on how our work can be expanded to a multi-object setting, other manipulation tasks or to a real world environment.

8.1 Contributions

The goal of this thesis was to learn a complex robotics task with very few priors about the models of the robot and the environment. We managed to make the robot learn how to reach an object with its palm, and to predict if an object can be reached using camera joint angles as input. These tasks have been learned with very little information, i.e. we did not use camera calibration parameters, kinematics, hand-designed vision modules and expert knowledge. To do that, we took inspiration from the human behaviour. First, before reaching an object, humans generally fixate it before reaching it. Second, humans are provided with a kind of kinematic model, where the objects are encoded in a eye-centred frame. Third, humans can generally tell if an object is reachable using visual inputs. From these observations, we divided the whole task into three sub-tasks, and we detail the contributions separately for each task.

8.1.1 Object fixation

The object fixation task consists for the robot in making its stereo system fixate the object put on the table by considering raw visual and proprioceptive values. To dispense as much as possible with external human information, we decided to make autoencoders reconstruct the images (taken by the camera system) of the environment without object. The only assumption of this learning framework is that no object is in the scene, apart from that, the framework is unsupervised. After this step, when learning binocular object fixations, the object is detected and localized as being an artefact in the image reconstructions (from the autoencoders) and is used to compute an informative reward signal. The way of computing the reward signal without neither calibration parameters nor hand-designed visual modules is our first contribution.

The second contribution related to the object fixation task is linked to the denoising procedure in the learning runs. Indeed, the computed reward signal suffers from impulse noise due to some high-frequency areas which are not reconstructed very well. We have shown that this leads to poor performance, and have implemented a method which solves the issue still without using supervision. In that aim, we have

learned a model of variation of object localization in the images given camera joint angle moves. When the difference between the reality and the model is too large, it means that the object localization has abnormally moved and that the reward signal may be wrong. Thus, we remove transitions with such anomalies.

These two contributions were successfully applied in a simulated environment and a real one. To summarize, we have computed a reward signal for the object fixation task which does not require calibration parameters, camera system kinematics and visual modules. Moreover, we add an unsupervised noise removal procedure to remove transitions whose reward is affected by impulse noise.

8.1.2 End-effector fixation and hand-eye coordination

This task consists in learning a kind of forward kinematics model that we call "hand-eye coordination". The latter outputs a 3D position (through camera joint angles) from robot joint angles. The robot learns this function using supervised learning techniques with self-supervised labels. It builds its database thanks to an end-effector fixation task which is run in parallel.

We do not claim a strong contribution for this task since the reward signal computation itself is not new. Indeed, the reward signal is computed from finger moves, which, though requiring only minimal supervision (finger moves have to be specified) does not differ a lot from state-of-the-art work. What is interesting is that the full setting of object fixation task is applied to the one of end-effector fixation task and produces good results. In other terms, the structure of the reward function is similar, the state and action spaces are the same and the denoising procedure is exactly the same. This task was implemented and validated in a synthetic environment.

8.1.3 Reaching skills

The reaching skills consists first of the reaching task itself and second of the reachability prediction as a function of the robot gaze.

The first joint contribution to both tasks is the fact that they are jointly learned using the object fixation policy and the hand-eye coordination function in an original and functioning reinforcement learning framework. More precisely, the object fixation policy acts as an object locator, and the hand-eye coordination function as an end-effector locator. They are used to compute an informative shaping reward which encourages the end-effector to be close to the object. Consequently, they make the contact between the end-effector palm and the object more likely. As the requirements of these skills are mainly the requirements of the fixation tasks, these skills do not require a lot of supervision, which constitutes the second shared contribution.

Specifically to the reaching task, we conducted experiments to evaluate the combined influence of initial conditions and some reward terms. The possible initial conditions are related to the uniqueness (or not) of the initial object positions and initial arm joint angles. Besides, the reward terms that we have studied are a term penalizing contacts between the table and the end-effector, a sparse reward term and two shaping reward terms (one is our main contribution, the other is a shaping term depending on forward kinematics and a perfect knowledge on the Cartesian object position). We made the following observations:

- If we only use sparse reward terms, learning does not work at all when there is only one initial arm position at the beginning of learning episodes. Indeed,

the probability that the end-effector palm makes contact with the object is really small, which confirms one of the issues mentioned in Chapter 2. We noticed that by using several initial arm positions, we increased the probability of touching the object (even when the initial object position can vary). While it yields decent performances when the object position is fixed, it does not lead to a reliable learning otherwise.

- For all the initial episode configurations, using shaping terms speeds up learning.
- In most arm positions in which the end-effector is in contact with the table, using terms penalizing contacts can allow to guide more the exploration towards contact-less areas, which accelerates learning. This is the case if we also use shaping terms. However, if we use only sparse reward and contact penalty terms, the latter can act as a repellent. Indeed, when we do not use a shaping reward term, the probability of success is lower, and as the robot has to move the end-effector downwards to touch the object, the robot generally receives more penalties than "success" terms. Thus, the robot can be discouraged to make downward moves which are necessary to reach the object.
- Learning with our shaping reward function approaches learning with the fully-supervised reward function even if it is a bit slower.

The two reaching skills were implemented and validated in a synthetic environment.

8.2 Limitations

The limitations of our approach are all related to the assumptions we made during the learning process. In the following, we detail them task by task.

8.2.1 Object fixation

For this task, we do not need calibration parameters, kinematics, hand-designed visual modules, or expert knowledge. We only assume that the object is not present when the autoencoder learns to encode the environment, and is present when learning occurs. These assumptions can seem rather light at first sight. However, they make our application not robust to environment variations. Indeed, if the environment is changing, the autoencoder is not valid anymore and has to be trained again. To do that, the object has to be withdrawn first. This procedure is not convenient at all inasmuch as it has to be repeated each time the environment changes.

8.2.2 End-effector fixation and hand-eye coordination

This task dispenses as well with calibration parameters, kinematics, or hand-designed visual modules. The assumption we made to learn both the end-effector fixation task and the hand-eye coordination is that nothing besides the robot fingers moves in the environment. It means that our end-effector detection does not work if other motions occur in the environment.

8.2.3 Reaching skills

The limitations of these tasks are mainly the limitations of the previous tasks because they depend on them. The additional limitation that we add concerns the penalty term that we use for a better exploration and the sparse reward term. They assume that the robot can tell from its tactile sensors whether it is touching the table or the object.

8.3 Perspectives

We give here some hints that might help to overcome our limitations and we describe how we can adapt our work to a multi-object setting, to other manipulation tasks and a real environment.

8.3.1 Alleviating limitations

The main perspectives for our work consist in addressing the limitations mentioned above. The latter will be addressed globally because we believe the main issue is not related to each limitation in the individual tasks. It is rather related to the fact the tasks are learned sequentially one by one. Indeed, in this thesis, we took inspiration from the human behaviour. However, we did not inspire from the way these tasks are learned. For newborns and even for children, the learnings of these tasks overlap and some of them drive each other. For example, the object fixation task is driven by the object reaching task. Therefore, we propose an untested learning framework when the learnings of several skills overlap:

- An area fixation task:
  This task consists in fixating an area of an image with the camera system.
- A curious mechanism to select curious areas to be fixated:
  For this, we would use a function (represented by a neural network) which outputs from an image (left or right) a heat-map representing the image areas to be explored. This function should be adapted through time to output unexplored areas.
- An object detector network:
  It is a function, which outputs from an image a heat map in which high-value areas represent objects. This detector is learned through time using reaching successes, i.e. when the fixated area has been touched (we assume that the robot knows whether it is touching an object), it is considered as an object. In this approach, we consider the object as something which can be manipulated whereas in the thesis, we consider it as an anomaly in the environment.
- A hand-eye coordination function:
  This is the same function as in the presented work and would be learned using the area fixation task and finger moves. The issue about the confusion between the latter and other moves in the environment can be addressed by methods in the literature using correlation [175]. Note that we can also think about adding tactile feedback to the hand-eye coordination function task. Indeed, when the robot reaches an object, we can add the pair $(\boldsymbol{q}^{\mathbf{robot}}, \boldsymbol{q}^{\mathbf{camera}})$ to the hand-eye coordination database.

- A reachability prediction function:
  This is the same function as in the presented work and would be learned in a similar way.
- A reaching skill:
  This is the same function as in the presented work and would be learned in a similar way.

Figure 8.1 displays the framework developed in the thesis and the potentially more autonomous one. Note that in the latter, we do not provide the robot with the knowledge of the correct order of the task. The goal would be that the robot finds it itself using hierarchical reinforcement learning algorithms ([143] and [137] could be used). Note that this framework is nothing but a guideline to go further towards autonomy and is still untested.

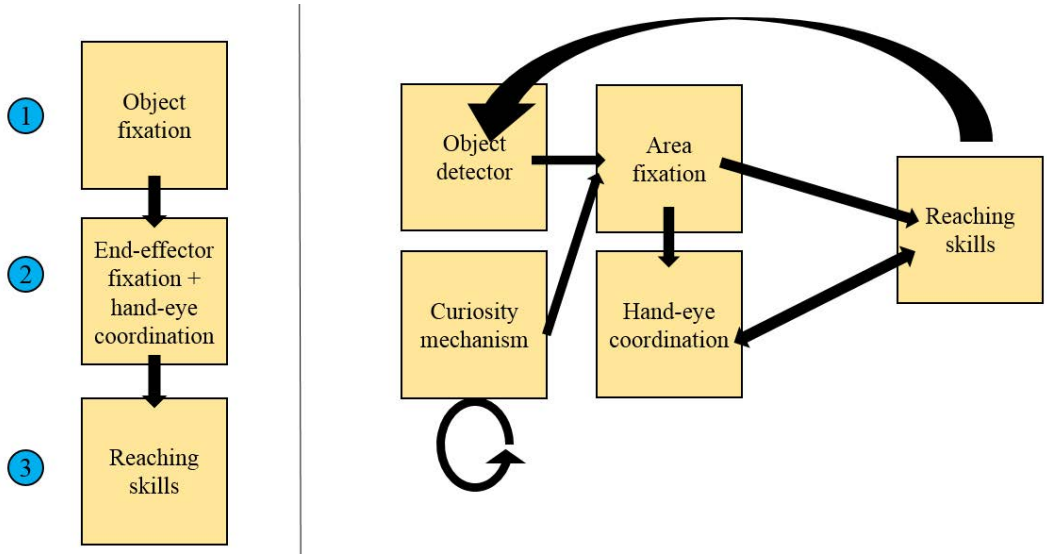


FIGURE 8.1: Comparison of our framework (left) with a potentially more autonomous framework (right)

8.3.2 Extending current work

8.3.2.1 Improving the reaching task

Multi-object setting

Our reaching skill state space is composed of "only" camera joint angles to indicate the object localization and arm joint angles. If the objects involved in the experiments have different shapes, the camera joint angles cannot give the correct indication where the robot has to reach, i.e what are the correct arm joint angles. Indeed, information about the shape of the object is required. To that end, in the same vein as in our approach, i.e. without using pre-computed features as states, raw pixels from the left and right cameras can be added to the state space, because they can encode together the shape of the object. The problem which arises from the use of this state space is that we get a non Markovian decision process. Indeed, the object might be occluded by the arm, and then, the images are not predictable from previous images and actions. Two solutions can be used. On the one hand,

we might include in the state space observations taken at different time-steps (recurrent neural networks are generally used to ensure that the correlation between some time-steps is correctly done) so that the object is visible in at least one observation. This kind of method suffers from several drawbacks. First, taking several pairs of images in the state space makes the application very resource-demanding and memory-consuming. Second, we cannot ensure that in the K considered steps, the object is visible and that the decision process is Markovian. On the other hand, we can simply include the left and right images of the environment after the object fixation step, when the object is visible (it is ensured). The latter solution would make the decision process Markovian and solve the issue without too many resource requirements.

Improve the exploration using curiosity

In all our tasks, exploration is rather basic (Gaussian noise or Ornstein-Uhlenbeck process) and might not be optimal. It could be interesting to add to our setting a curiosity-based exploration such as in [188]. Since the latter associates intrinsic rewards signals with model learning improvements, it can be very helpful for the reaching task. Indeed, there are non-linearities in robot dynamics when the end-effector touches the table or the object. Thus, the robot might be even more attracted by successful positions.

8.3.2.2 Learning other manipulation tasks using the same principle

We have made a robot learn a reaching task. It would be interesting to check if our approach can make the robot learn other object manipulation tasks such as grasping objects or clear the table. Theoretically, switching to another task just requires to switch to another sparse reward. If the success of the new task is more likely to occur than our task, it should not be a problem to complete learning with our framework. However, if the success of the new task is very unlikely to happen (even with our framework), e.g. peg-in-hole task with a random initial setting, there is no guarantee that our framework can achieve its learning though we hope it would work.

8.3.2.3 Real-world implementation

Among the learned skills in this thesis, we have only implemented the object fixation in the real world. We draw here some indications about the difficulties which might arise and potential progresses. There are two main issues.

The first one is that it is very time-consuming to learn our tasks in the real world. Indeed, the object fixation task took one week and a half in tedious conditions (a human operator has to change the object types and positions at every episode). The end-effector fixation task should take longer because it implies end-effector finger moves at each reinforcement learning iteration and the reaching task would be much longer. Thus, as it stands, our whole task is hardly implementable in the real-world. To solve this issue, the first solution consists in using every ways to make reinforcement learning faster, i.e. a more careful choice of hyperparameter values, lighter neural network structures, more data-efficient reinforcement learning algorithms (model-based algorithms), synthetic-to-real transfers and initialization using expert demonstrations. A second solution would consist in accepting this waste of time and using reliable robots which can act in a repeatable way during extended periods of time.

The second main issue is the safety. Our current framework for the reaching task involves variations of arm joint angles as actions. In practice, from these variations of angles, some target joint angles are computed and fed to PID controllers. If the arm touches the table and has a target which makes it move downwards, the robot risks to be damaged. To deal with it and using variations of joint angles as actions, we can prevent the robot from exploring some areas of the state space (where the end-effector touches the table or itself), but that requires forward kinematics, which is not in line with our objectives. Another solution is to use soft controllable robots which are not damaged when they touch solid environment parts.

Appendix A

Neural networks structures

We present here the neural network structures used for the experiments as well as associated hyper parameters for the solver algorithms. We do not claim that these structures are optimal, because in some cases (autoencoder and structures for the fixation experiments), lighter structures would be preferable. Furthermore, the related hyperparameters were not found using a rigorous method but by trial and error, i.e. we stopped looking for better structures or better hyperparameters when learning started to give interesting results. Thus, the following elements are described only for the purpose of reproducibility and are by no means optimal.

A.1 Neural networks for the fixation experiments

A.1.1 Autoencoder

The autoencoder structure has an encoder consisting of convolutional layers and fully connected layers and a fully connected decoder. The weights linked to the central feature vector h are transposed to reduce the number of parameters. We decided not to compress too much the inputs, because the autoencoder is used to represent as faithfully as possible the environment images without object. This structure may not be optimal because of its too large number of parameters.

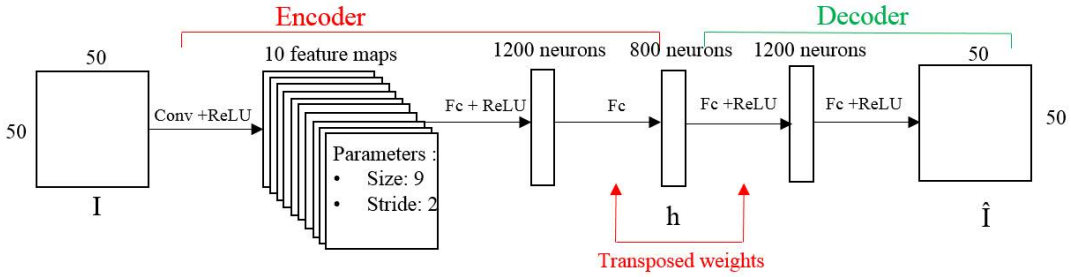


FIGURE A.1: Autoencoder structure

The autoencoders are learned using the ADAM solver using the parameters presented in table A.1.

Parameters	α	β_1	β_2
Values	0.0001	0.9	0.999

TABLE A.1: ADAM parameters for the autoencoder updates

A.1.2 Policy

The policy structure involves convolution layers which are used to deal with spatially coherent inputs such as images. This structure may not be optimal because of its large number of parameters.

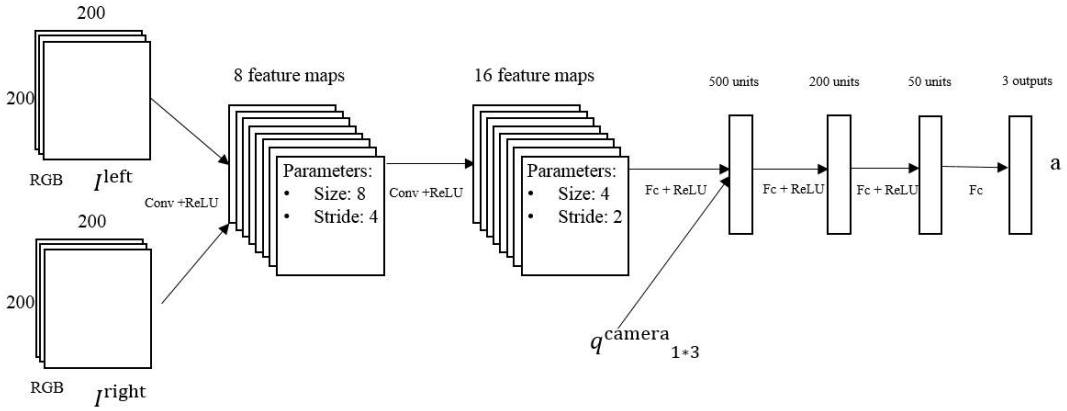


FIGURE A.2: Policy structure for the fixation experiments

The policy is updated using the ADAM algorithm and the hyperparameters of Table A.2. At first sight, the value of α can be considered as aberrant because it is very low. However, we found useful to update it with this value to ensure that the learning process is stable enough (same reasoning as many algorithms: the policy should not be updated with too large updates). Indeed, it is considered in the literature [189] that the DDPG is very sensitive to the learning rate of the actor and the critic (in the Half-cheetah problem for example). Furthermore, these rates depend on the nature of the tasks and the environment considered. We choose a very low value because the resulting learning was reliable. However, we do not guarantee that this value is optimal.

Parameters	α	β_1	β_2
Values	0.0000001	0.9	0.999

TABLE A.2: ADAM parameters for the policy updates

A.1.3 Q function

We do the same observations as for the policy structure. The Q function is updated using the ADAM algorithm using the parameters of table A.3.

Parameters	α	β_1	β_2
Values	0.0005	0.9	0.999

TABLE A.3: ADAM parameters for the policy updates

A.1.4 Hand-eye mapping

The hand-eye mapping us updated using the ADAM algorithm and the following hyperparameters of Table A.4.

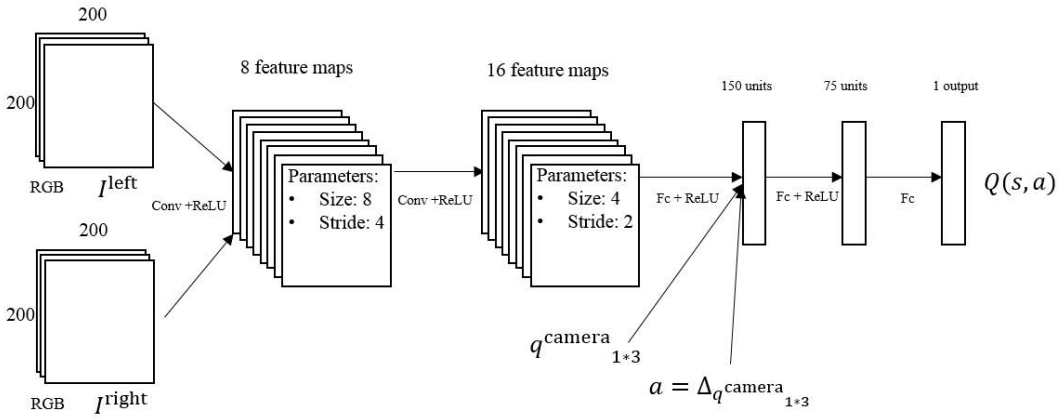


FIGURE A.3: Q function structure for the fixation experiments

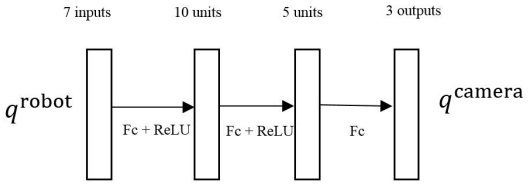


FIGURE A.4: Hand eye mapping structure

Parameters	α	β_1	β_2
Values	0.0001	0.9	0.999

TABLE A.4: ADAM parameters for the policy updates

A.2 Neural networks for the reaching experiments

A.2.1 Policy

As the state space does not involve any images, we did not use convolution layers. The policy is updated using the ADAM algorithm and the parameters of Table A.5. Concerning the low learning rate value of the policy, the observations of Section A.1.2 are valid here.

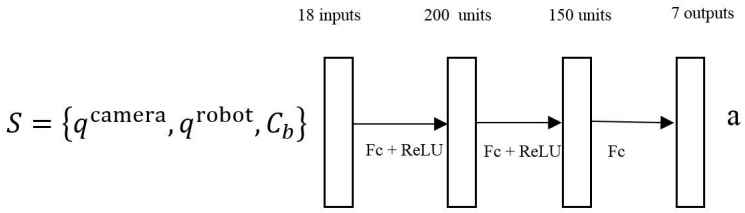


FIGURE A.5: policy structure for the reaching experiments

Parameters	α	β_1	β_2
Values	0.0000001	0.9	0.999

TABLE A.5: ADAM parameters for the policy updates

A.2.2 Q function

The same observations are applied for the Q function and the parameters for the ADAM algorithm are in A.6.

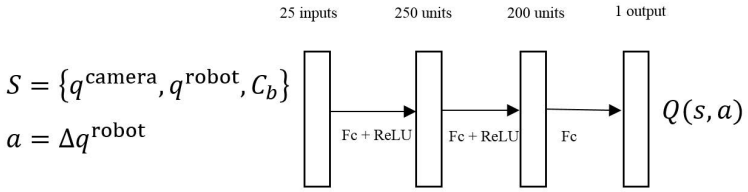


FIGURE A.6: Q function structure for the reaching experiments

Parameters	α	β_1	β_2
Values	0.0001	0.9	0.999

TABLE A.6: ADAM parameters for the Q updates

A.2.3 Reachability prediction network

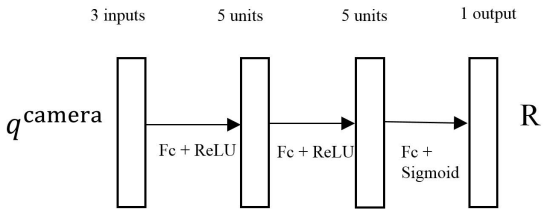


FIGURE A.7: Reachability structure

The reachability prediction is updated using the ADAM algorithm and with the hyperparameters of Table A.7.

Parameters	α	β_1	β_2
Values	0.0001	0.9	0.999

TABLE A.7: ADAM parameters for the reachability prediction updates

Appendix B

Hyperparameters for the reinforcement learning applications

B.1 Object fixation learning

The parameter values of the reinforcement learning procedure of the object fixation algorithm are presented in Table B.1.

Parameters	N_{trans}	γ	ϵ	N_{tot}	N_{eps}	N_{b}	ν	N_{gp}
Training (simulation)	20 000	0.2	0.02	200 000	35	16	10	1 000
Test (simulation)				2 000	35			
Training (real	20 000	0.2	0.02	160 000	35	16	10	1 000
Test (real)				200	35			

TABLE B.1: Parameters for the object fixation algorithm

We remind here of the hyperparameters meanings:

- N_{trans} the maximum number of transitions in the memory buffer
- γ is the discount factor
- ϵ is the variance of the noise applied on actions computed by the policy
- N_{tot} is the total number of RL iterations used in the experiments
- N_{eps} is the number of RL iterations by episode
- N_{b} is the batch size
- ν is the threshold value used to decide whether to remove or add a transition in the memory buffer
- N_{gp} is the required number of samples to build the denoising model with a Gaussian process.

B.2 End-effector fixation and hand-eye coordination learning

The parameters of the reinforcement learning procedure of the end-effector fixation and hand-eye coordination learning algorithm are presented in Table B.2.

Parameters	N_{trans}	γ	ϵ	N_{tot}	N_{eps}	N_{b}	ν	N_{gp}	N_{upd}	R_{th}
Training	20 000	0.2	0.02	200 000	35	16	10	1000	200	1.85

TABLE B.2: Parameters for the algorithm used to learn the hand-eye coordination mapping and end-effector fixations

The meaning of hyperparameters remain the same as before, the new ones are the following:

- N_{upd} is the number of required samples in the hand-eye mapping database to start the learning of the hand-eye coordination.
- R_{th} is a threshold value. If the reward signal is above, a sample is added to the database and the hand-eye mapping is updated.

B.3 Reaching and reachability learning

The parameters of the reinforcement learning procedure of the reaching and reachability fixation algorithm are presented in Table B.3, B.4 and B.5.

Parameters	γ	N_{max}	N_{b}	N_{trans}	N_{tot} (training)	N_{tot} (test)
Values	0.99	100	256	60 000	40 000	1 000

TABLE B.3: Parameter values

Parameters	c_{cam}	c_{cart}	p_{contact}	p_{time}
Values	$-\frac{1}{30}$	$-\frac{1}{40}$	-0.01	-0.0125

TABLE B.4: Parameter values for the used reward functions

Parameters	$\theta_j, j \in \{1,...,7\}$	$\mu_j, j \in \{1,...,7\}$	$\sigma_j, j \in \{1,...,4\}$	$\sigma_j, j \in \{5,...,7\}$
Values	0.8	0	0.01	0.04

TABLE B.5: Ornstein-Uhlenbeck process parameters

The meanings of the new hyperparameters are:

- N_{max} is the maximal number of RL iterations per episode. However, the number of episodes can be inferior to N_{max} if the robot reaches the object before completing N_{max} iterations in an episode.
- The parameters θ_j, μ_j, σ_j are the parameters for the Ornstein-Uhlenbeck process used to apply noise in the reaching experiments.

Appendix C

Reference frame of the simulated environment

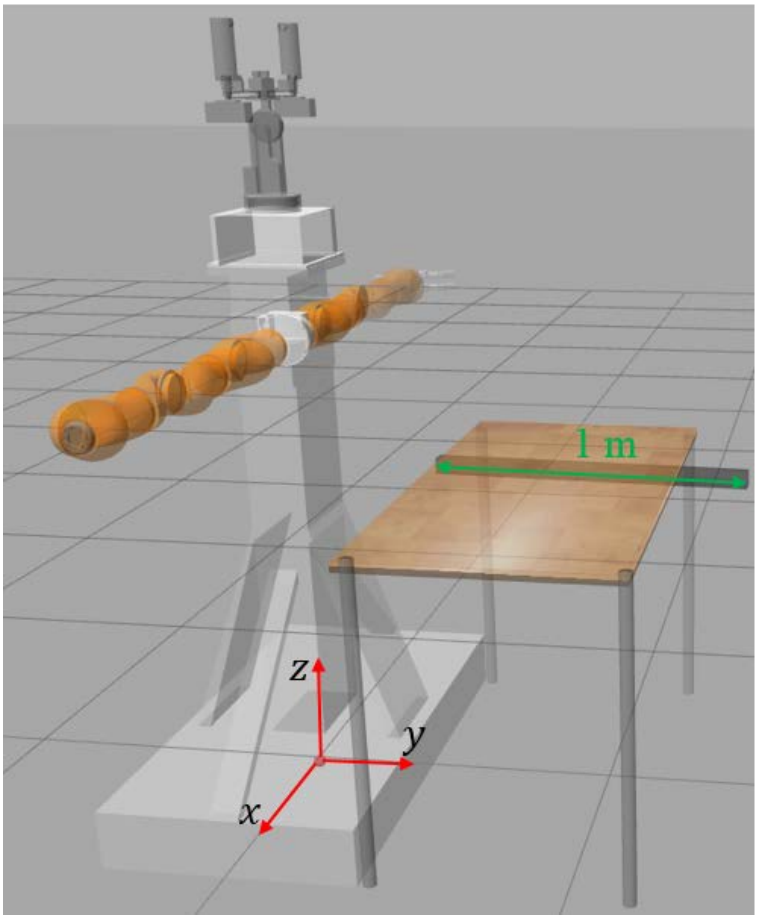


FIGURE C.1: Reference frame of the simulated environment

In Figure C.1, we show the reference frame used in our experiments.

Appendix D

Triangulation

This appendix proposes to establish the Cartesian coordinates x, y and z of a 3D point X with respect to the camera joint angles q^{tilt}, q^{panLeft} and q^{panRight} which make the camera system fixate X (see Figure D.1 for a visualization of links between aforementioned variables). Let $C^{\text{left}} = (C_x^{\text{left}}, C_y^{\text{left}}, C_z^{\text{left}})$ and $C^{\text{right}} = (C_x^{\text{right}}, C_y^{\text{right}}, C_z^{\text{right}})$ be the Cartesian coordinates of the left and right cameras. For a simplification pur-

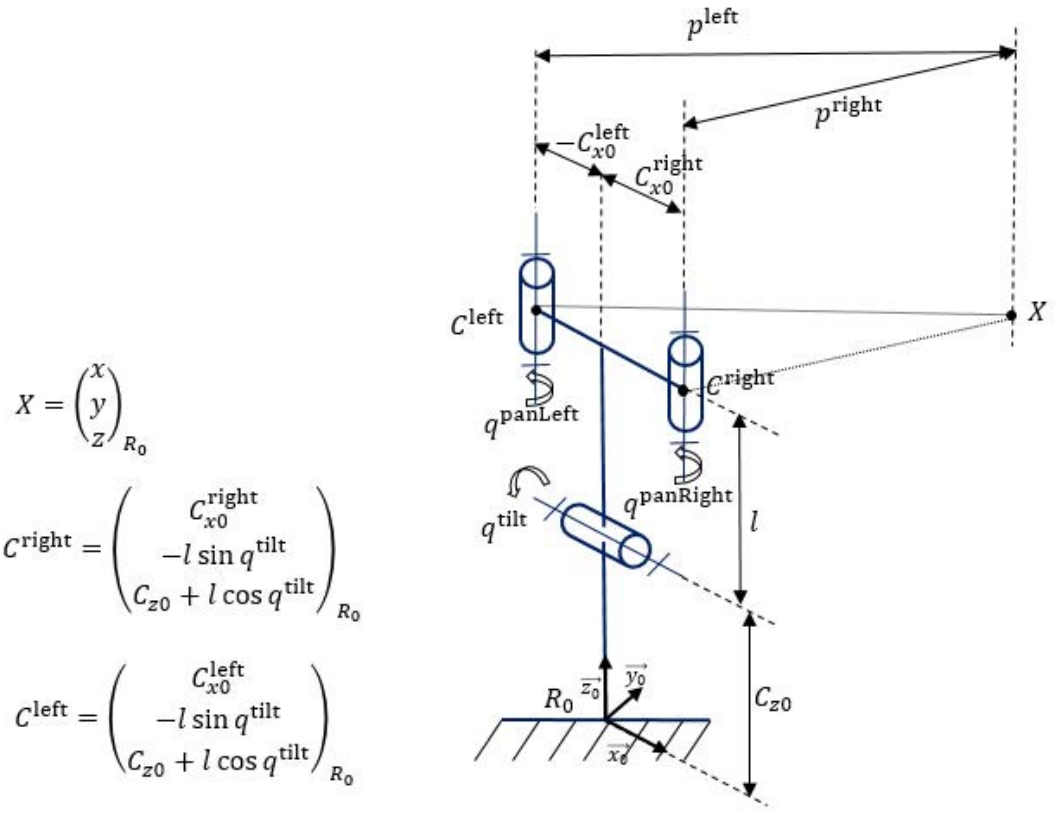


FIGURE D.1: Scheme showing geometrical links between camera joint angles and the fixated 3D point

pose, we note $C_y = C_y^{\text{left}} = C_y^{\text{right}}$ and $C_z = C_z^{\text{left}} = C_z^{\text{right}}$ (these relationships are true only for the perfect simulated model). The following relationships can be found geometrically:

$$x = C_x^{\text{left}} - p^{\text{left}} \sin(q^{\text{panLeft}}) \tag{D.1}$$

$$y = C_y + p^{\text{left}} \cos(q^{\text{tilt}}) \cos(q^{\text{panleft}}) \tag{D.2}$$

$$z = C_z + p^{\text{left}} \sin(q^{\text{tilt}}) \cos(q^{\text{panleft}}) \tag{D.3}$$

$$x = C_x^{\text{right}} - p^{\text{right}} \sin(q^{\text{panRight}}) \tag{D.4}$$

$$y = C_y + p^{\text{right}} \cos(q^{\text{tilt}}) \cos(q^{\text{panRight}}) \tag{D.5}$$

$$z = C_z + p^{\text{right}} \sin(q^{\text{tilt}}) \cos(q^{\text{panRight}}) \tag{D.6}$$

$$C_x^{\text{c}} = C_{x0}^{\text{c}} \tag{D.7}$$

$$Cy = -l \sin(q^{\text{tilt}}) \tag{D.8}$$

$$C_z = C_{z0} + l \cos(q^{\text{tilt}}) \tag{D.9}$$

l, C_{x0}^{c} and C_{z0}^{c} are constant numbers. By equalizing equations D.1 and D.4 on the one hand, and D.2 and D.5 on the other hand, we derive the following equations:

$$C_x^{\text{right}} - C_x^{\text{left}} = p^{\text{right}} \sin(q^{\text{panRight}}) - p^{\text{left}} \sin(q^{\text{panLeft}}) \tag{D.10}$$

$$p^{\text{left}} \cos(q^{\text{panLeft}}) = p^{\text{right}} \cos(q^{\text{panRight}}) \tag{D.11}$$

Considering the unknown variables p^{left} and p^{right}, we solve the system of equations D.10 and D.11, provided that $q^{\text{panLeft}} \not\equiv q^{\text{panRight}}[\pi]$, $q^{\text{panRight}} \not\equiv \frac{\pi}{2}[\pi]$ and $q^{\text{panLeft}} \not\equiv \frac{\pi}{2}[\pi]$:

$$p^{\text{right}} = \frac{C_x^{\text{right}} - C_x^{\text{left}}}{(\tan(q^{\text{panRight}}) - \tan(q^{\text{panLeft}})) \cos(q^{\text{panRight}})} \tag{D.12}$$

$$p^{\text{left}} = \frac{C_x^{\text{right}} - C_x^{\text{left}}}{(\tan(q^{\text{panRight}}) - \tan(q^{\text{panLeft}})) \cos(q^{\text{panLeft}})} \tag{D.13}$$

Consequently, we can derive x, y and z expressions by replacing p^{right} in equations D.4, D.5 and D.6 using equation D.12.

$$x = C_{x0}^{\text{right}} - \frac{(C_{x0}^{\text{right}} - C_{x0}^{\text{left}}) \tan(q^{\text{panRight}})}{(\tan(q^{\text{panRight}}) - \tan(q^{\text{panLeft}}))} \tag{D.14}$$

$$y = -l \sin(q^{\text{tilt}}) + \frac{(C_{x0}^{\text{right}} - C_{x0}^{\text{left}}) \cos(q^{\text{tilt}})}{(\tan(q^{\text{panRight}}) - \tan(q^{\text{panLeft}}))} \tag{D.15}$$

$$z = C_{z0} + l \cos(q^{\text{tilt}}) + \frac{(C_{x0}^{\text{right}} - C_{x0}^{\text{left}}) \sin(q^{\text{tilt}})}{(\tan(q^{\text{panRight}}) - \tan(q^{\text{panLeft}}))} \tag{D.16}$$

In our work, the constant values are (in meters):

- $C_{z0} = 2.095$
- $C_{x0}^{\text{left}} = -0.22$
- $C_{x0}^{\text{right}} = 0.24$
- $l = 0.045$

Appendix E

Ground-truth object reachability estimation

To compute the ground-truth object reachability estimation, we build on the knowledge of the robot structure and experiments:

- We know from experiments that the left arm can reach an object if it is put at the bottom left of the table. Indeed, in Section 7.1.4.2, the robot learns to reach an object at different positions at the bottom left area of the table.
- We know from the geometry that the furthest points that the robot can reach follow two circular arcs of different radii. Indeed, in Figure E.1, we can see the ways to reach the furthest points. In the first configuration, the arm is outstretched and the second revolute joint can make the arm move while keeping the arm outstretched. The second configuration corresponds to the position where the robot cannot move around its second revolute joint because it would be hitting its structure. In this configuration, the arm can reach the furthest points only by moving around its fourth revolute joint and forcing the end-effector to conserve a given orientation.

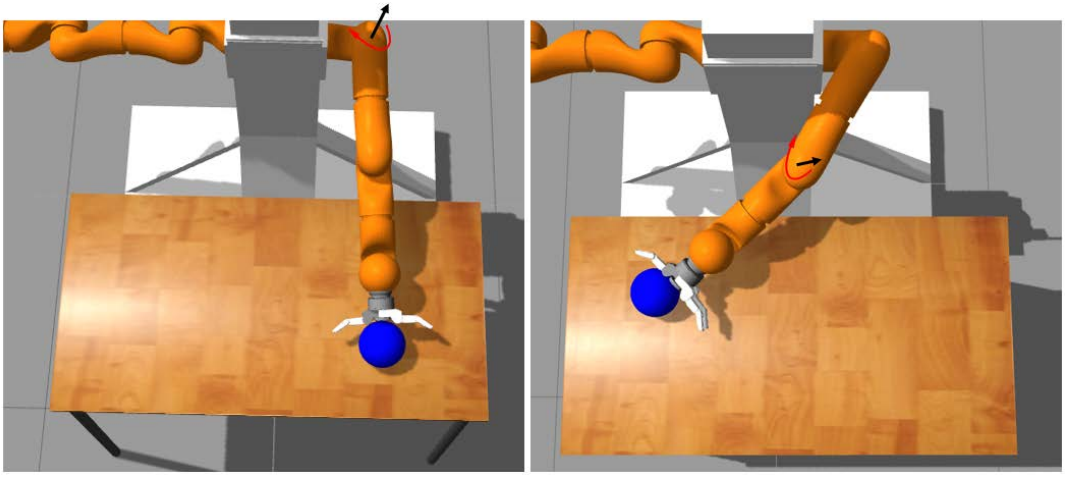


FIGURE E.1: Top views of the left arm showing the ways for the robot to reach the furthest points. The left figure represents the way to obtain the second circular arc: by making the elbow joint of the robot vary. The right figure represents the way to obtain the first circular arc: by making the shoulder joint of the robot vary.

Using these facts, the workspace limits are some borders of the table and the mentioned circular arcs. Then, to know the object reachability, we need to estimate the

parameters of the two circular arcs $(x_0^{\mathrm{l}}, y_0^{\mathrm{l}}, r^{\mathrm{l}})$ $(x_0^{\mathrm{b}}, y_0^{\mathrm{b}}, r^{\mathrm{b}})$, l and b denoting little and big.

To do so, we measure object positions which belong to these circular arcs. We get two sets of points: $S^{\mathrm{l}} = \{(x_1^{\mathrm{l}}, y_1^{\mathrm{l}}), ..., (x_{N_{\mathrm{l}}}^{\mathrm{l}}, y_{N_{\mathrm{l}}}^{\mathrm{l}})\}$ and $S^{\mathrm{r}} = \{(x_1^{\mathrm{r}}, y_1^{\mathrm{r}}), ..., (x_{N_{\mathrm{r}}}^{\mathrm{r}}, y_{N_{\mathrm{r}}}^{\mathrm{r}})\}$. We find the parameters using a least mean square optimization by considering the following linear problem for each circular arc:

$$Ax = b \implies x \simeq (A^T A)^{-1} A^T b, \tag{E.1}$$

$$A = \begin{pmatrix} 2x_1 & 2y_1 & 1 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 2x_N & 2y_N & 1 \end{pmatrix}, \tag{E.2}$$

$$b = \begin{pmatrix} x_1^2 + y_1^2 \\ \cdot \\ \cdot \\ x_N^2 + y_N^2 \end{pmatrix}, \tag{E.3}$$

$$x = \begin{pmatrix} x_0 \\ y_0 \\ r^2 \end{pmatrix}. \tag{E.4}$$

We obtain reasonable solutions for the two circular arcs. We estimated (using Monte-Carlo sampling) the surface of the reachable area as 0.506 m^2, which represents 51.6% of the table area (we use the Gazebo scales). In the experiments, this estimation of reachability produces only 0.04 % of false positives (the ground-truth estimation says it is not reachable, but the robot manages to reach it). This confirms that this estimation can act as a ground-truth for a comparison with our reachability prediction.

Appendix F

Detailed results for uncertainties estimation of the x, y and z coordinates

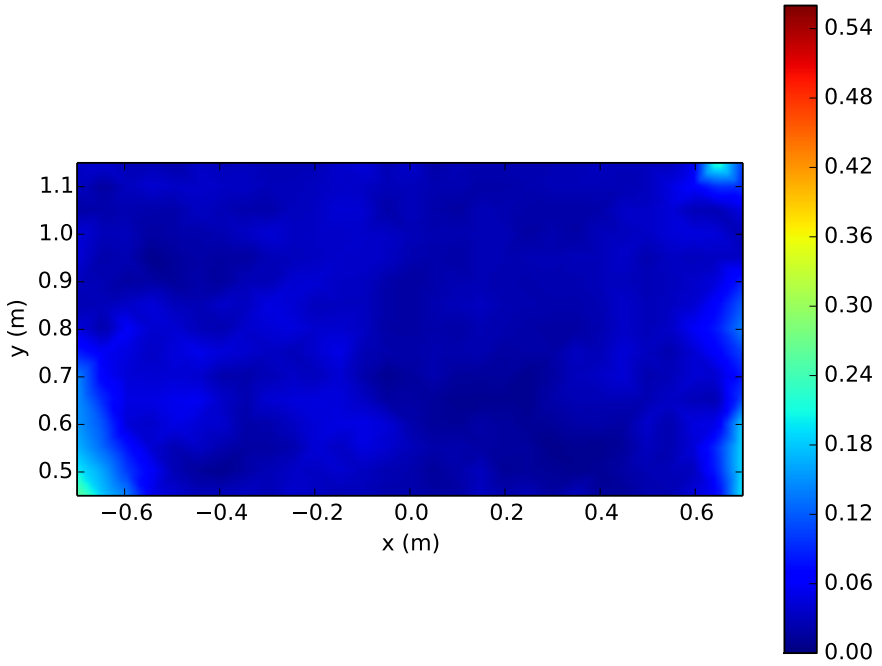


FIGURE F.1: Absolute uncertainties on the x axis in function of x and y coordinates of the object on the table

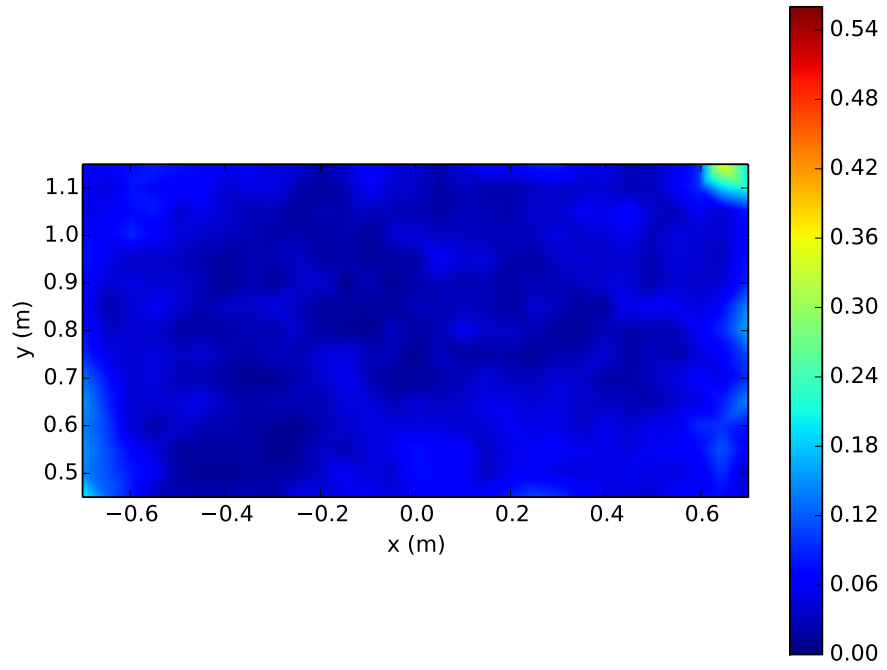


FIGURE F.2: Absolute uncertainties on the y axis in function of x and y coordinates of the object on the table

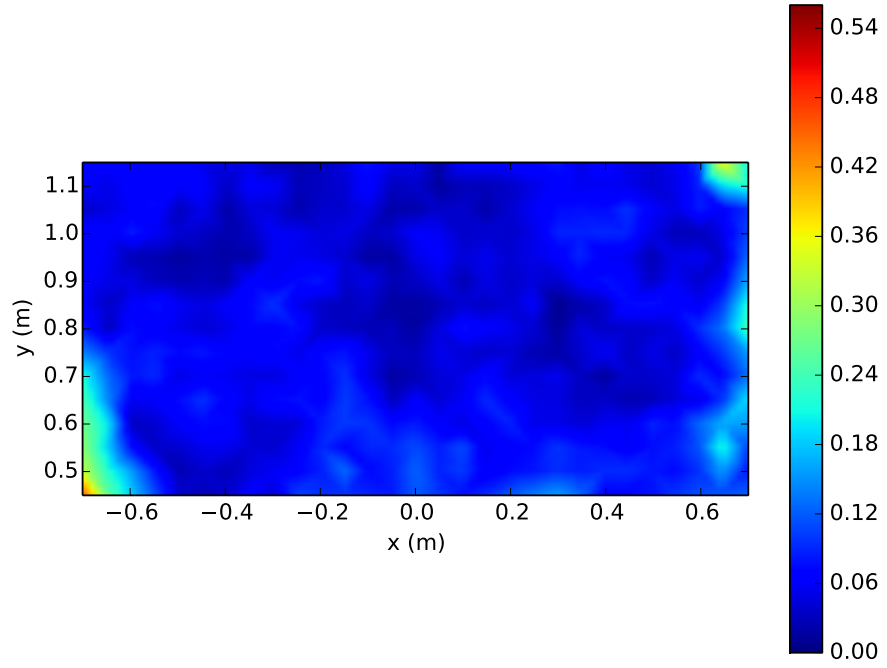


FIGURE F.3: Absolute uncertainties on the z axis in function of x and y coordinates of the object on the table

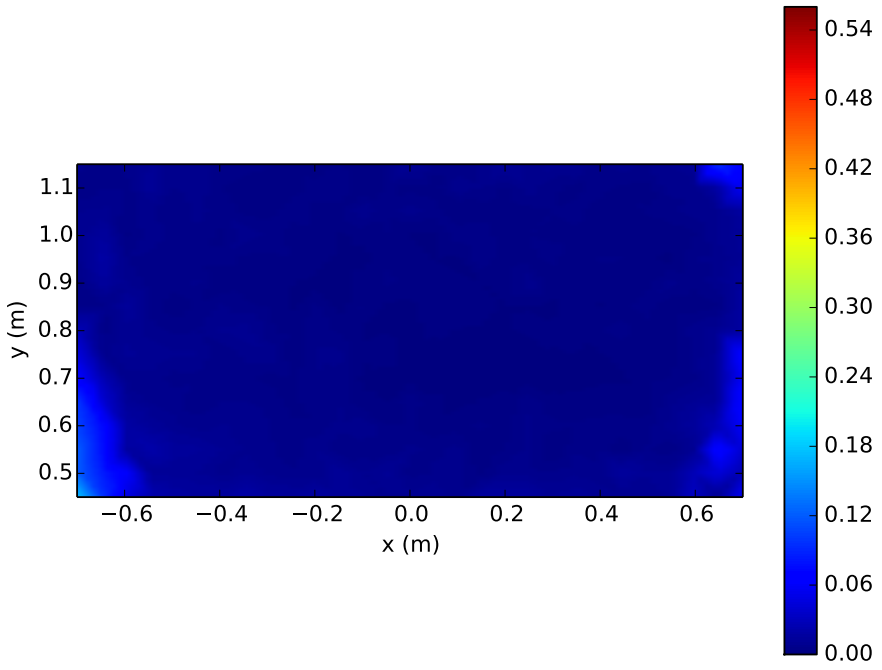


FIGURE F.4: Relative uncertainties on the x axis in function of x and y coordinates of the object on the table

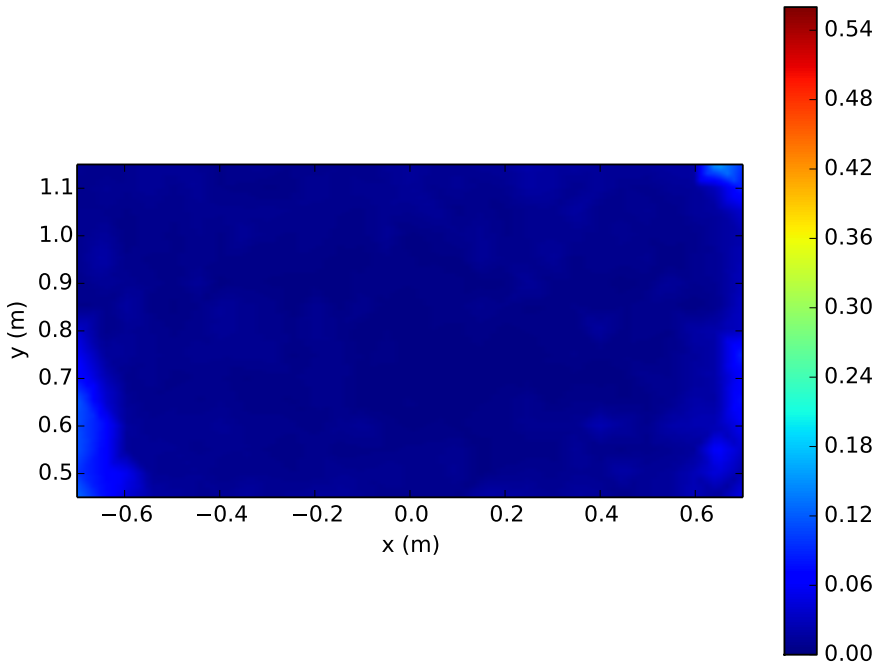


FIGURE F.5: Relative uncertainties on the y axis in function of x and y coordinates of the object on the table

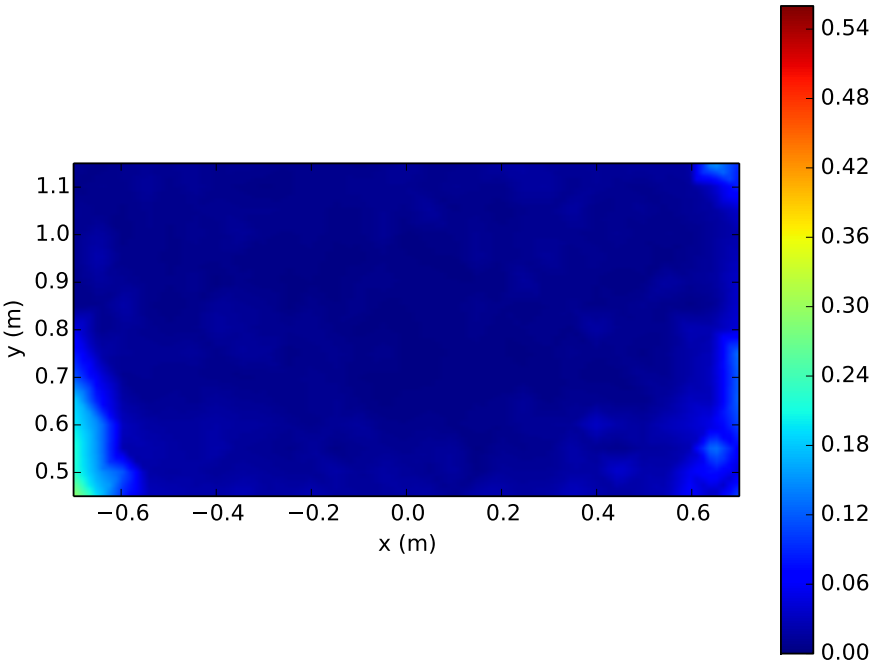


FIGURE F.6: Relative uncertainties on the z axis in function of x and y coordinates of the object on the table

Appendix G

Material

G.1 Middleware

We have used ros [165] as the middleware.

G.2 Software

In our work, we used the following informatic tools:

- Gazebo with the ODE engine as a simulator
- Caffe [166] the library for neural network learning algorithms
- C++(11) as the programming language
- C++ stl, boost, cuda, opencv, ros, gazebo as main dependencies
- Python as the language for plotting curves and drawing graphs
- FreeCad, one of the only free CAD softwares with which we can draw fully parametrized ellipsoids

G.3 Hardware

The learning runs were launched using different computers:

- One with a Nvidia GTX TitanX as GPU
- Several ones with Nvidia GTX 1080 as GPUS

Appendix H

Planar views of 3D clouds and convex hulls for the possible initial end-effector positions in the reaching and hand-eye coordination learnings

This appendix groups planar views of 3D point clouds and associated convex hulls for initial end-effector 3D Cartesian coordinates for both the reaching task and the hand-eye coordination learning.

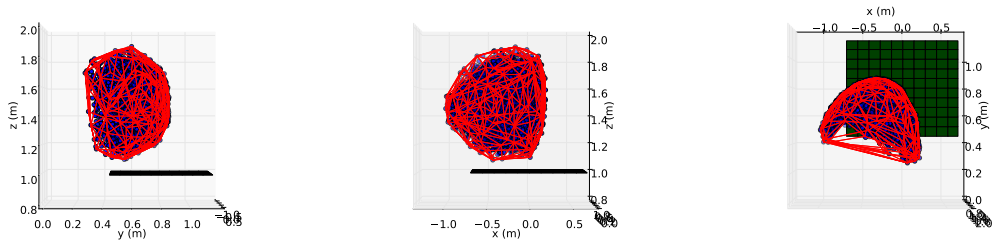


FIGURE H.1: Possible end-effector initial positions for the reaching task

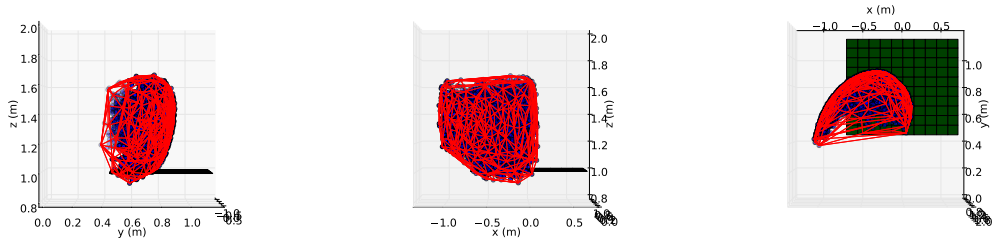


FIGURE H.2: Possible end-effector initial positions for the hand-eye coordination learning.

Figures H.1 and H.2 represent some views of the volume of possible end-effector initial positions for respectively the reaching task and the hand-eye coordination

learning. More precisely, the figures represent YZ (left), XZ (middle) and XY (right) planar views on the 3D point cloud (blue), and convex hull (red) of possible end-effector initial positions. The table is visualized in green.

Bibliography

[1] S. Levine et al. “End-to-end Training of Deep Visuomotor Policies”. In: *J. Mach. Learn. Res.* (2016).

[2] A. Ghadirzadeh et al. “Deep predictive policy training using reinforcement learning”. In: *IROS*. 2017.

[3] F. de La Bourdonnaye et al. “Learning of binocular fixations using anomaly detection with deep reinforcement learning.” In: *IJCNN*. 2017.

[4] F. de La Bourdonnaye et al. “Apprentissage par renforcement profond de la fixation binoculaire en utilisant de la détection d’anomalies”. In: *ORASIS 2017*. Colleville-sur-Mer, France, 2017.

[5] F. de La Bourdonnaye et al. “Learning to touch objects through stage-wise deep reinforcement learning”. In: *IROS* (2018).

[6] F. de La Bourdonnaye et al. “Stage-wise learning of reaching using little prior knowledge”. In: *Frontiers in Robotics and AI* (2018).

[7] R. A. Fisher. “The Use of Multiple Measurements in Taxonomic Problems”. In: *Annals of Eugenics* (1936).

[8] A. M. Legendre. “Nouvelles méthodes pour la détermination des orbites des comètes”. In: *Courcier* (1805).

[9] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* (2014).

[10] A. Krogh and J. A. Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *NIPS*. 1992.

[11] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[12] C. Cortes and V. Vapnik. “Support-Vector Networks”. In: *Mach. Learn.* (1995).

[13] W. S. Mcculloch and W. H. Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* (1943).

[14] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* (1958).

[15] M. E.Hoff and B. Widrow. “Associative Storage and Retrieval of Digital Information in Networks of Adaptive “Neurons””. In: *Second Annual Bionics Symposium sponsored by Cornell University and the General Electric Company, Advanced Electronics Center*. 1961.

[16] S. Grossberg. “Some Networks That Can Learn, Remember, and Reproduce any Number of Complicated Space-Time Patterns, I”. In: *Indiana University Mathematics Journal* (1969).

[17] T. Kohonen. “Correlation Matrix Memories”. In: *IEEE Transactions on Computers* (1972).

[18] G. Palm. “On associative memory”. In: *Biological Cybernetics* (1980).

[19] J. J. Hopfield. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". In: *Proceedings of the National Academy of Sciences of the United States of America* (1982).

[20] J. Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* (2015).

[21] P. Smolensky. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1". In: MIT Press, 1986. Chap. Information Processing in Dynamical Systems: Foundations of Harmony Theory.

[22] G. E. Hinton and T. J. Sejnowski. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1". In: MIT Press, 1986. Chap. Learning and Relearning in Boltzmann Machines.

[23] D. H. Ballard. "Modular Learning in Neural Networks." In: *AAAI*. 1987.

[24] G. E. Hinton and R. R. Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *Science* (2006).

[25] S. Hawkins et al. "Outlier Detection Using Replicator Neural Networks." In: *DaWaK*. Lecture Notes in Computer Science. 2002.

[26] P. J. Werbos. "Applications of Advances in Nonlinear Sensitivity Analysis". In: *IFIP*. 1981.

[27] D.B. Parker. *Learning-logic: Casting the Cortex of the Human Brain in Silicon*. 1985.

[28] Y. LeCun. *A Theoretical Framework for Back-Propagation*. 1988.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1". In: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation.

[30] A-L. Cauchy. "Méthode générale pour la résolution des systèmes d'équations simultanées". In: *Compte Rendu des S'eances de L'Acad'emie des Sciences XXV* (1847).

[31] D. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR*. 2015.

[32] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *ICML*. 2010.

[33] K.He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *ICCV*. 2015.

[34] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* (1989).

[35] K. Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* (1991).

[36] O. Delalleau and Y. Bengio. "Shallow vs. Deep Sum-Product Networks". In: *NIPS*. 2011.

[37] T. Poggio et al. "Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review". In: *International Journal of Automation and Computing* (2017).

[38] R. Salakhutdinov and G. E. Hinton. "Deep Boltzmann Machines." In: *Journal of Machine Learning Research - Proceedings Track* (2009).

[39] S. Hochreither and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* (1997).

[40] K. Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *EMNLP* (2014).

[41] Athanasios Voulodimos et al. "Deep Learning for Computer Vision: A Brief Review". In: *Computational Intelligence and Neuroscience* (2018).

[42] K. Fukushima. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* (1980).

[43] Y. Lecun et al. "Back-Propagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* (1989).

[44] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* (1998).

[45] Y. Lecun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *NIPS*. 1990.

[46] P. Baldi and Y. Chauvin. "Neural Networks for Fingerprint Recognition". In: *Neural Computation* (1993).

[47] S. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München*. 1991.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *NIPS*. 2012.

[49] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks." In: *ECCV (1)*. 2014.

[50] C. Szegedy et al. "Going deeper with convolutions." In: *CVPR*. 2015.

[51] K. J. Piczak. "Environmental sound classification with convolutional neural networks". In: *MLSP*. 2015.

[52] P. Swietojanski, A. Ghoshal, and S. Renals. "Convolutional neural networks for distant speech recognition". In: *IEEE Signal Processing Letters* (2014).

[53] A. Bhandare et al. "Applications of Convolutional Neural Networks". In: *IJCSIT* (2016).

[54] M. van Otterlo and M. Wiering. "Reinforcement Learning and Markov Decision Processes". In: *Reinforcement Learning: State-of-the-Art*. Springer Berlin Heidelberg, 2012.

[55] J. Schaeffer and A. Plaat. "Kasparov versus Deep Blue: The Rematch". In: *ICGA Journal* (1997).

[56] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[57] R.E. Bellman. *Dynamic programming*. 1957.

[58] Manuel Watter et al. "Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images". In: *NIPS*. 2015.

[59] C. Florensa et al. "Reverse Curriculum Generation for Reinforcement Learning". In: *CoRL*. 2017.

[60] M. P. Deisenroth, C. E. Rasmussen, and D. Fox. "Learning to Control a Low-Cost Manipulator using Data-Efficient Reinforcement Learning." In: *Robotics: Science and Systems*. 2011.

[61] R. E. Bellman. *Adaptive Control Processes: A Guided Tour*. MIT Press, 1961.

[62] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* (2015).

[63] S. Levine, N. Wagener, and P. Abbeel. "Learning contact-rich manipulation skills with guided policy search". In: *ICRA*. 2015.

[64] R.S. Sutton and A.G. Barto. *Reinforcement learning: an introduction*. MIT Press, 1998.

[65] C. J. C. H. Watkins. "Learning from Delayed Rewards". PhD thesis. 1989.

[66] C. J. C. H. Watkins and P. Dayan. "Q-Learning". In: *Machine Learning* (1992).

[67] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist sytems*. Tech. rep. 1994.

[68] G. A. Rummery. "Problem Solving with Reinforcement Learning". PhD thesis. 1995.

[69] R. S. Sutton. "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding." In: *NIPS*. MIT Press, 1995.

[70] J. Peng and R. J. Williams. "Incremental Multi-Step Q-Learning." In: *Machine Learning* (1996).

[71] M. A. Wiering and J. Schmidhuber. *Fast online Q(λ)*. Tech. rep. 1997.

[72] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* (2015).

[73] S. Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *Int. J. of Rob. Res.* (2017).

[74] M. P. Deisenroth, G. Neumann, and J. Peters. "A Survey on Policy Search for Robotics." In: *Foundations and Trends in Robotics* (2013).

[75] N. Kohl and P. Stone. "Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion." In: *ICRA*. 2004.

[76] J. Peters and S. Schaal. "Policy gradient methods for robotics". In: *IROS*. 2006.

[77] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* (1992).

[78] J. Baxter, P. Bartlett, and L. Weaver. "Experiments with Infinite-Horizon, Policy-Gradient Estimation". In: *Journal of Artificial Intelligence Research* (2001).

[79] R. S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *NIPS*. 2000.

[80] J. Peters and S. Schaal. "Natural Actor-Critic." In: *Neurocomputing* (2008).

[81] J. Peters and S. Schaal. "Reinforcement learning of motor skills with policy gradients." In: *Neural Networks* (2008).

[82] J. Peters, K. Mülling, and Y. Altun. "Relative Entropy Policy Search." In: *AAAI*. 2010.

[83] J. Schulman et al. "Trust Region Policy Optimization." In: *ICML*. 2015.

[84] S. Levine and P. Abbeel. "Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics". In: *NIPS*. 2014.

[85] J. Kober and J. Peters. "Policy Search for Motor Primitives in Robotics." In: *NIPS*. 2008.

[86] V. Heidrich-Meisner and C. Igel. "Neuroevolution strategies for episodic reinforcement learning." In: *J. Algorithms* (2009).

[87] E. Theodorou, J. Buchli, and S. Schaal. "A Generalized Path Integral Control Approach to Reinforcement Learning." In: *Journal of Machine Learning Research* (2010).

[88] A. J. Ijspeert, J. Nakanishi, and S. Schaal. "Learning Attractor Landscapes for Learning Motor Primitives." In: *NIPS*. 2002.

[89] G. Tesauro. "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play". In: *Neural Computation* (1994).

[90] C. W. Anderson. "Learning to Control an Inverted Pendulum Using Neural Networks". In: *IEEE Control Systems Magazine* (1989).

[91] C. F. Touzet. "Neural reinforcement learning for behaviour synthesis." In: *Robotics and Autononous Systems* (1997).

[92] R. Coulom. "Reinforcement Learning Using Neural Networks, with Applications to Motor Control. (Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur)." PhD thesis. 2002.

[93] R. H. Crites and A. G. Barto. "Improving Elevator Performance Using Reinforcement Learning". In: *NIPS*. 1996.

[94] S. Lange and M. Riedmiller. "Deep auto-encoder neural networks in reinforcement learning". In: *IJCNN*. 2010.

[95] T. P. Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. 2016.

[96] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning." In: *ICML*. 2016.

[97] D.Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* (2017).

[98] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *ICML*. Beijing, China, 2014.

[99] L. Baird. "Residual Algorithms: Reinforcement Learning with Function Approximation". In: *Machine Learning Proceedings 1995*. 1995.

[100] J. N. Tsitsiklis and B. Van Roy. "An Analysis of Temporal-Difference Learning with Function Approximation". In: *IEEE Transactions of Automatic Control* (1997).

[101] L. J. Lin. "Reinforcement Learning for Robots Using Neural Networks". PhD thesis. 1993.

[102] J. Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation." In: *CoRR* (2015).

[103] J. Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* (2017).

[104] S. Gu et al. "Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic." In: *CoRR* (2016).

[105] S. Levine and V. Koltun. "Guided Policy Search." In: *ICML (3)*. 2013.

[106] S. Levine and V. Koltun. "Variational Policy Search via Trajectory Optimization." In: *NIPS*. 2013.

[107] L. Peshkin and C. R. Shelton. "Learning from Scarce Experience." In: *ICML*. 2002.

[108] G. Neumann. "Variational Inference for Policy Search in changing situations." In: *ICML*. 2011.

[109] S. Levine and V. Koltun. "Learning Complex Neural Network Policies with Trajectory Optimization". In: *ICML*. 2014.

[110] Y. Chebotar et al. *Path Integral Guided Policy Search*. 2016.

[111] V. Gullapalli, J. A. Franklin, and H. Benbrahim. "Acquiring robot skills via reinforcement learning". In: *IEEE Control Systems* (1994).

[112] S. Schaal and C. G. Atkeson. "Robot juggling: implementation of memory-based learning". In: *IEEE Control Systems* (1994).

[113] M. T. Rosenstein and A. G. Barto. "Reinforcement learning with supervision by a stable controller". In: *American Control Conference*. 2004.

[114] B. Wang, J. w. Li, and H. Liu. "A Heuristic Reinforcement Learning for Robot Approaching Objects". In: *2006 IEEE Conference on Robotics, Automation and Mechatronics*. 2006.

[115] J. Kober and J. Peters. "Learning motor primitives for robotics". In: *ICRA*. 2009.

[116] P. Kormushev, S. Calinon, and D. G. Caldwell. "Robot motor skill coordination with EM-based Reinforcement Learning". In: *IROS*. 2010.

[117] M. Tamosiunaite et al. "Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives." In: *Robotics and Autonomous Systems* (2011).

[118] J. Buchli et al. "Learning variable impedance control." In: *I. J. Robotic Res.* (2011).

[119] M. Kalakrishnan et al. "Learning force control policies for compliant manipulation". In: *IROS*. 2011.

[120] T. Lampe and M. Riedmiller. "Acquiring visual servoing reaching and grasping skills using neural reinforcement learning". In: *IJCNN*. 2013.

[121] Y. Zhu et al. "Reinforcement and Imitation Learning for Diverse Visuomotor Skills". In: *CoRR* (2018).

[122] Y. Duan et al. "One-Shot Imitation Learning". In: *NIPS*. 2017.

[123] A. Nair et al. "Overcoming Exploration in Reinforcement Learning with Demonstrations." In: *CoRR* (2017).

[124] M. Večerík et al. "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards". In: *CoRR* (2017).

[125] C. Finn, S. Levine, and P. Abbeel. "Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization". In: *ICML*. 2016.

[126] P. Sermanet, K. Xu, and S. Levine. "Unsupervised Perceptual Rewards for Imitation Learning." In: *Robotics: Science and Systems*. 2017.

[127] P. Englert and M. Toussaint. "Learning manipulation skills from a single demonstration". In: *The International Journal of Robotics Research* (2018).

[128] P. Englert, N. A. Vien, and M. Toussaint. "Inverse KKT: Learning cost functions of manipulation tasks from demonstrations". In: *The International Journal of Robotics Research* (2017).

[129] K. Hausman et al. "Multi-Modal Imitation Learning from Unstructured Demonstrations using Generative Adversarial Nets". In: *NIPS*. 2017.

[130] F. Zhang et al. "Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control." In: *CoRR* (2015).

[131] S. James and E. Johns. "3D Simulation for Robot Arm Control with Deep Q-Learning". In: *CoRR* (2016).

[132] V. Kumar, E. Todorov, and S. Levine. "Optimal control with learned local models: Application to dexterous manipulation." In: *ICRA*. 2016.

[133] C. Finn et al. "Deep spatial autoencoders for visuomotor learning". In: *ICRA*. 2016.

[134] S. Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates." In: *ICRA*. 2017.

[135] Y. Tsurumine et al. "Deep dynamic policy programming for robot control with raw images". In: *IROS*. 2017.

[136] I. Popov et al. "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation." In: *CoRR* (2017).

[137] A. Gudimella et al. "Deep Reinforcement Learning for Dexterous Manipulation with Concept Networks". In: *CoRR* (2017).

[138] A. Boularias, J. A. Bagnell, and A. Stentz. "Learning to Manipulate Unknown Objects in Clutter by Reinforcement". In: *AAAI*. 2015.

[139] M. Asada et al. "Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning." In: *Machine Learning* (1996).

[140] C. Finn and S. Levine. "Deep visual foresight for planning robot motion". In: *ICRA*. 2017.

[141] M. Gualtieri, A. Pas, and R. Platt. "Category Level Pick and Place Using Deep Reinforcement Learning". In: *CoRR* (2017).

[142] M. Andrychowicz et al. "Hindsight Experience Replay". In: *NIPS*. 2017.

[143] M. A. Riedmiller et al. "Learning by Playing - Solving Sparse Reward Tasks from Scratch". In: *CoRR* (2018).

[144] Lanzilotto M. et al. "Neuronal Encoding of Self and Others' Head Rotation in the Macaque Dorsal Prefrontal Cortex". In: *Scientific Reports* (2017).

[145] R. Huys and P. J. Beek. "The coupling between point-of-gaze and ballmovements in three-ball cascade juggling: the effects of expertise, pattern and tempo". In: *Journal of Sports Sciences* (2002).

[146] A.P Basptista, C.A Snyder, and R.A Andersen. "Reach plans in eye-centered coordinates". In: *Science* (1999).

[147] J. Konczak. "On the notion of motor primitives in humans and robots". In: *Workshop on Epigenetic Robotics* (2005).

[148] F. Vassella and B. Karlsson. "Asymmetric Tonic Neck Reflex". In: *Developmental Medicine & Child Neurology* 4 (1962).

[149] B. L. White, P. Castle, and R. Held. "Observations on the Development of Visually-Directed Reaching". In: *Child Development* (1964).

[150] J. Law et al. "From Saccades to Grasping: A Model of Coordinated Reaching Through Simulated Development on a Humanoid Robot". In: *IEEE Trans. on Autonomous Mental Development* (2014).

[151] N. Alahyane et al. "Development and learning of saccadic eye movements in 7- to 42-month-old children". In: *Journal of Vision* (2016).

[152] R.K Clifton et al. “Is visually guided reaching in early infancy a myth?” In: *Child Development* (1993).

[153] A. Mathew and M. Cook. “The Control of Reaching Movements by Young Infants”. In: *Child Development* (1990).

[154] R. W. White. “Motivation Reconsidered: The Concept Of Competence”. In: *Psychological review* (1959).

[155] D.E. Berlyne. *Conflict, Arousal and Curiosity*. McGraw-Hill, 1960.

[156] J. Schmidhuber. “Formal Theory of Creativity, Fun, and Intrinsic Motivation (1990-2010)”. In: *IEEE Trans. on Auton. Ment. Dev.* (2010).

[157] W. Schenck, H. Hoffmann, and R. Möller. “Learning Internal Models for Eye-Hand Coordination in Reaching and Grasping”. In: *European Cognitive Science Conference*. 2003.

[158] H. Hoffmann, W. Schenck, and R. Möller. “Learning visuomotor transformations for gaze-control and grasping.” In: *Biological Cybernetics* (2005).

[159] F. Nori et al. “Autonomous learning of 3D reaching in a humanoid robot.” In: *IROS*. 2007.

[160] L. Jamone et al. “Autonomous Online Learning of Reaching Behavior in a humanoid Robot.” In: *I. J. Humanoid Robotics* (2012).

[161] E. Chinellato et al. “Implicit Sensorimotor Mapping of the Peripersonal Space by Gazing and Reaching”. In: *IEEE Trans. on Autonomous Mental Development* (2011).

[162] B. Schölkopf et al. “Support Vector Method for Novelty Detection.” In: *NIPS*. 1999.

[163] A. Moreno et al. “Noisy Reinforcements in reinforcement learning: some case studies based on gridworlds”. In: *WSEAS*. 2006, pp. 296–300.

[164] R. Fox, A. Pakman, and N. Tishby. “Taming the Noise in Reinforcement Learning via Soft Updates”. In: *UAI*. 2016.

[165] M.Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software*. 2009.

[166] Y. Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *ACM*. 2014.

[167] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *ICML*. 2015.

[168] J. Tobin et al. “Domain randomization for transferring deep neural networks from simulation to the real world”. In: *IROS*. 2017.

[169] A. A. Rusu et al. “Sim-to-Real Robot Learning from Pixels with Progressive Nets”. In: *CoRL* (2017).

[170] C. Salaün, V. Padois, and O. Sigaud. “Learning Forward Models for the Operational Space Control of Redundant Robots”. In: *From Motor Learning to Interaction Learning in Robots*. 2010.

[171] S. Vijayakumar et al. “Statistical Learning for Humanoid Robots”. In: *Autonomous Robots* (2002).

[172] J. T. Lapreste et al. “An efficient method to compute the inverse Jacobian matrix in visual servoing”. In: *ICRA*. 2004.

[173] J. Sturm, C. Plagemann, and W. Burgard. "Body schema learning for robotic manipulators from visual self-perception". In: *Journal of Physiology-Paris* (2009).

[174] M. Marjanovic, B. Scassellati, and M. Williamson. "Self-Taught Visually-Guided Pointing for a Humanoid Robot". In: *From Animals to Animats: Proceedings of 1996 Society of Adaptive Behavior*. MIT Press, 1996.

[175] G. Metta and P. Fitzpatrick. "Early Integration of Vision and Manipulation". In: *IJCNN* (2003).

[176] A.Broun et al. "Bootstrapping a robot's kinematic model". In: *Robotics and Autonomous Systems* (2014).

[177] L. P. Wijesinghe et al. "Learning multisensory neural controllers for robot arm tracking". In: *IJCNN*. 2017.

[178] *Barrett Tech*. URL: `https://www.barrett.com/about-barrethand/`.

[179] A.Rajeswaran et al. "Towards Generalization and Simplicity in Continuous Control". In: *NIPS*. 2017.

[180] M.J.Hausknecht and P.Stone. "Deep Reinforcement Learning in Parameterized Action Space." In: *ICLR*. 2016.

[181] A. W. Moore and C. G. Atkeson. "Prioritized sweeping: Reinforcement learning with less data and less time". In: *Machine Learning* (1993).

[182] T. Schaul et al. "Prioritized Experience Replay". In: *ICLR*. 2016.

[183] Y. Guan and K. Yokoi. "Reachable Space Generation of A Humanoid Robot Using The Monte Carlo Method". In: *IROS*. 2006.

[184] M. Rolf. "Goal babbling with unknown ranges: A direction-sampling approach". In: *ICDL*. 2013.

[185] F. Zacharias, C. Borst, and G. Hirzinger. "Capturing robot workspace structure: representing robot capabilities". In: *IROS*. 2007.

[186] L. Jamone et al. "Interactive online learning of the kinematic workspace of a humanoid robot". In: *IROS*. 2012.

[187] L. Jamone et al. "Autonomous online generation of a motor representation of the workspace for intelligent whole-body reaching". In: *Robotics and Autonomous Systems* (2014).

[188] V. R. Kompella et al. "Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots". In: *Artificial Intelligence* (2017).

[189] R. Islam et al. "Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control". In: *CoRR* (2017).