Deep Reinforcement Learning and Learning from Demonstrations for Robot Manipulators

Jesús Bujalance Martin

To cite this version:

Jesús Bujalance Martin. Deep Reinforcement Learning and Learning from Demonstrations for Robot Manipulators. Robotics [cs.RO]. Université Paris sciences et lettres, 2024. English. ⟨NNT : 2024UPSLM022⟩. ⟨tel-04804036⟩

HAL Id: tel-04804036

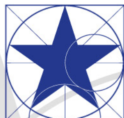
https://pastel.hal.science/tel-04804036v1

Submitted on 26 Nov 2024

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

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

HAL Authorization



THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à MINES Paris-PSL

Deep Reinforcement Learning and Learning from Demonstrations for Robot Manipulators

Apprentissage profond par Renforcement et Démonstrations, pour le comportement de robots manipulateurs

Soutenue par
Jesús Bujalance Martín
Le 24 juin 2024

Dirigée par
Fabien Moutarde

École doctorale n°621
Ingénierie des Systèmes, Matériaux, Mécanique, Energétique

Spécialité
Informatique temps réel, robotique et automatique.

Composition du jury :

David Filliat
Professeur, ENSTA Paris — *Rapporteur*

Philippe Preux
Professeur, Université de Lille — *Rapporteur*

Olivier Sigaud
Professeur, Sorbonne Université — *Examinateur*

Amel Bouzeghoub
Professeur, Telecom SudParis — *Examinateur & Président du jury*

Fabien Moutarde
Professeur, Mines Paris — *Examinateur & Directeur de thèse*

MINES PARIS | PSL

Abstract

Despite having known great success, reinforcement-learning algorithms still need to become more sample-efficient, particularly for robotics where it is much harder to train an agent outside of simulation. As the community leans towards data-driven approaches (offline reinforcement learning, decision transformers, etc.), in this thesis we focus on off-policy reinforcement learning, and explore different ways of incorporating additional data into the algorithms. In particular, we rely on expert demonstrations, which can help with efficiency as well as overall performance. The goal is to design efficient algorithms to solve a range of robotic-manipulation tasks, such as flipping a switch or sliding a cube on a table.

After a thorough review of the reinforcement-learning and imitation-learning frameworks, we first introduce our reward-relabeling method, which can be seen as a form of reward shaping that happens in hindsight, once the entire episode is collected. This approach can easily extend any off-policy algorithm to benefit from both reinforcement and imitation signals. Building on this method, we then introduce a more efficient algorithm that aggregates previous and concurrent works that also address similar concerns.

Finally, we move onto the more realistic setting of vision-based reinforcement learning. To tackle this problem, we design a two-stage training pipeline: first learn a visual representation of the scene by pre-training an encoder from multiple supervised computer-vision objectives, then train a reinforcement-learning agent which can focus solely on solving the task. Despite all the data being collected in simulation, the experiments include one sim-to-real example to show that these techniques can translate to real-world controlled environments.

Résumé en Français

Malgré leur grand succès, les algorithmes d'apprentissage par renforcement doivent encore devenir plus efficaces en termes d'échantillons, en particulier pour la robotique où il est beaucoup plus difficile d'entraîner un agent en dehors d'un environnement de simulation. Alors que la communauté se tourne vers des approches orientées données (apprentissage par renforcement "offline", "decision transformers", etc.), nous nous concentrons dans cette thèse sur l'apprentissage par renforcement "off-policy" et explorons différentes manières d'incorporer des données supplémentaires dans les algorithmes. En particulier, nous nous appuyons sur des démonstrations d'experts, qui peuvent contribuer à l'efficacité ainsi qu'à la performance globale. L'objectif est de concevoir des algorithmes efficaces pour résoudre des tâches de manipulation robotique, comme actionner un interrupteur ou faire glisser un cube sur une table.

Après une étude approfondie de l'apprentissage par renforcement et par imitation, nous présentons tout d'abord notre méthode de ré-étiquetage des récompenses, qui peut être considérée comme une forme de "reward shaping" qui se produit a posteriori, une fois que l'ensemble de l'épisode a été collecté. Cette approche peut s'appliquer à tout algorithme "off-policy" pour bénéficier à la fois des signaux de renforcement et d'imitation. En nous appuyant sur cette méthode, nous présentons ensuite un algorithme plus efficace qui regroupe des travaux antérieurs et concomitants qui traitent également de questions similaires.

Enfin, nous passons au cadre plus réaliste de l'apprentissage par renforcement basé sur la vision. Pour résoudre ce problème, nous concevons un pipeline d'entraînement en deux étapes : d'abord, apprendre une représentation visuelle de la scène en pré-entraînant un encodeur à partir de plusieurs objectifs supervisés de vision, puis entraîner un agent d'apprentissage par renforcement qui peut se concentrer uniquement sur la résolution de la tâche. Bien que toutes les données soient collectées en simulation, les expériences comprennent un exemple de transfert simulation-réalité pour montrer que ces techniques peuvent s'appliquer à des environnements contrôlés du monde réel.

Remerciements

Un grand merci à Fabien pour son encadrement et ses très précieux conseils. Aux membres du jury pour leur disponibilité, leur bienveillance et leurs remarques éclairées. A tous les doctorants et chercheurs des Mines, et notamment mes amis du bureau V026: Arthur, Thomas, Raphaël, Camille et Joseph. Et à ma famille.

Contents

1 Introduction **1**
1.1 Context: Learning for Collaborative Robotics 1
1.2 Publications 3
1.3 Outline 3
1.4 Résumé en français 4

2 State of the art - Deep Reinforcement Learning **6**
2.1 Policy gradient 9
2.1.1 Importance sampling 10
2.1.2 Advanced policy gradient 11
2.1.3 Evolution strategies 12
2.1.4 Summary 13
2.2 Dynamic programming and value function methods 13
2.2.1 Approximate dynamic programming 15
2.2.2 Deep Q-learning 16
2.2.3 Experience replay 18
2.2.4 Off-policy learning 18
2.2.5 Summary 19
2.3 Actor-Critic 19
2.3.1 Summary 21
2.4 Planning and model-based methods 21
2.4.1 Model-based reinforcement learning 23
2.4.2 Local models 25
2.4.3 Global models 25
2.4.4 Model-based policy learning 26
2.4.5 Model-free learning with a model 28
2.4.6 Successor representation 29
2.4.7 Summary 29
2.5 Which algorithm to use? 30
2.6 Exploration 30
2.7 Maximum-entropy reinforcement learning 33

3 State of the art - Imitation Learning **35**
3.1 Behaviour Cloning 36
3.2 Inverse Reinforcement Learning 37
3.2.1 Linear reward 38
3.2.2 Maximum entropy inverse reinforcement learning 39
3.2.3 Deep inverse reinforcement learning 39
3.2.4 Summary 41

3.3 Offline Reinforcement Learning 42

4 Reward Relabeling for combined Reinforcement and Imitation Learning 46
4.1 Introduction 46
4.2 Background - Other related frameworks 47
4.2.1 Self-Imitation Learning 47
4.2.2 Reward bonus 47
4.2.3 Hindsight Experience Replay (HER) 47
4.3 Reward Relabelling 48
4.3.1 Method 48
4.3.2 Optimal Policy: Discussion and Safety Decay 50
4.3.3 Hyper-parameters: Tuning and Scheduled Decay 50
4.4 Experiments 52
4.4.1 Setup 52
4.4.2 Results 54
4.4.3 Impact of hyper-parameters 54
4.4.4 Ablation study 56
4.4.5 Alternative decay 57
4.5 Discussion 58

5 STIR2: Self and Teacher Imitation by Reward Relabeling 60
5.1 Introduction 60
5.2 Background - SAC and DDPG 60
5.3 Related Work 62
5.3.1 Baselines 63
5.4 STIR2: Self and Teacher Imitation by Reward Relabeling 64
5.5 Experiments 66
5.5.1 Setup 66
5.5.2 Results 68
5.5.3 Ablation study 73
5.6 Additional Results 75
5.7 Discussion 78

6 Pre-trained Vision Encoder for Data-Efficient Reinforcement Learning 80
6.1 Introduction 80
6.2 Background - Challenges in Robotics 81
6.3 Background - Simulation to Reality 83
6.4 Related work 84
6.4.1 Baselines 86
6.5 Training the Encoder 86
6.5.1 Architecture 88
6.5.2 Experiments 88
6.6 Training the Agent 93
6.6.1 Experiments 94
6.7 Towards a Shared Encoder across tasks 95
6.8 Towards a Sim-to-Real pipeline 98

6.9 Discussion 100

7 Conclusion **102**

7.1 Summary of Contributions 102

7.2 Perspectives and Future Work 103

Bibliography **104**

List of Figures

1.1 UR3 robot from Universal Robots: robotic arm with 6 degrees of freedom, equipped with a Robotiq gripper and wrist camera. 1
1.2 Real-time gestural control of robot manipulator through Deep Learning human-pose inference 2
1.3 Classic robotic control pipeline (source: Levine, Fa2019) 2

2.1 Reinforcement learning framework (source: Sutton and Barto, 2011) 6
2.2 Graphical model of a MDP (up) and POMDP (down) (source: Levine, Fa2019) 7
2.3 The broad structure of a RL algorithm (source: Levine, Fa2019) 9
2.4 Reinforce algorithm (source: Levine, Fa2019) 9
2.5 Policy gradient (source: John Schulman's slides) 11
2.6 Comparison of backup diagrams for state value functions (source: David Silver's slides) 16
2.7 Fitted Q-iteration algorithm (source: Levine, Fa2019) 17
2.8 General Q-learning algorithm (source: Levine, Fa2019) 17
2.9 Advantage actor-critic algorithm (source: Levine, Fa2019) . . . 20
2.10 DDPG: deep deterministic policy gradient (source: Levine, Fa2019) 20
2.11 Phasic policy gradient (PPG) networks (source: Cobbe et al., 2021) 21
2.12 CEM algorithm (source: Levine, Fa2019) 22
2.13 MCTS general algorithm (source: Levine, Fa2019) 23
2.14 Basic model-based algorithm with MPC (source: Levine, Fa2019) 24
2.15 Basic planning procedure from a model with uncertainty (source: Levine, Fa2019) 24
2.16 LQR-FLM algorithm (source: Levine, Fa2019) 25
2.17 DreamerV2 model architecture (source: Hafner et al., 2020) . . 26
2.18 General guided policy search (source: Levine, Fa2019) 27
2.19 Stochastic (Gaussian) GPS with LQR-FLM (source: Levine, Fa2019) 27
2.20 Imitating MCTS with DAgger (Guo et al., 2014) (source: Levine, Fa2019) 28
2.21 Original Dyna-Q algorithm (source: Sutton and Barto, 2011) . . 28
2.22 Model-based acceleration: generate short simulated rollouts from previously visited states (source: Levine, Fa2019) 29
2.23 RL algorithms by sample efficiency (source: Levine, Fa2019) . . 30
2.24 Exploring with pseudo-counts (Bellemare et al., 2016) (source: Levine, Fa2019) 32

3.1 Learning from demonstrations with supervised learning (source: Levine, Fa2019) . . . 36
3.2 Compounding errors in behaviour cloning (source: Levine, Fa2019) 36
3.3 DAgger: Dataset Aggregation algorithm (source: Levine, Fa2019) 37
3.4 Guided cost learning algorithm (source: Levine, Fa2019) 40
3.5 Offline RL (source: Levine et al., 2020) 42
3.6 QT-OPT fine-tuning (source: Julian et al., 2020) 42
3.7 Over-optimistic Q-values (source: Levine et al., 2020) 43

4.1 Reward Relabeling procedure. 49
4.2 Reward obtained as a function of the agent's success. The more successful the agent, the smaller the bonus, until recovering the fully sparse reward at 100% success rate. We want to avoid the situation on the top, where the highest cumulative reward is obtained for a smaller success rate. 51
4.3 Reaching task in RLBench 53
4.4 Learning curves for the RLBench reaching task. 54
4.5 Impact of hyper-parameters on SAC. For a given hyper-parameter value, we plot the number of training steps required for the average episode success to reach a certain threshold. 55
4.6 Impact of hyper-parameters on DDPG. For a given hyper-parameter value, we plot the number of training steps required for the average episode success to reach a certain threshold. 56
4.7 Learning curves for different ablation studies. 57
4.8 Learning curves for two different types of decay. 58

5.1 Snapshots from each RLBench task: reach target, flip switch, push button, slide block. 66
5.2 Snapshots from each MetaWorld task. 67
5.3 Learning curves for the RLBench reaching task. 69
5.4 Learning curves for three RLBench tasks. 70
5.5 Learning curves for four Meta-World tasks. 72
5.6 Ablation curves on SAC-STIR^2. 73
5.7 Learning curves exploring low-data regimes on the RLBench reaching task. 74
5.8 Learning curves without demonstrations on the RLBench reaching task. 74
5.9 Curves of the isolated SAC-STIR^2 components on SAC-Demo. . 75
5.10 Additional curves (buffer configuration + PER) on SAC-Demo. 76
5.11 Additional curves (pretraining + resetting to demos) on SAC-Demo. 77
5.12 Adding extra components to SAC-STIR^2 doesn't result in further improvements. 79

6.1 Real-world setting with a Universal Robots' UR3 model. The inputs to the agent are the views from two cameras as shown in the left. 80
6.2 Proposed pipeline. 81

6.3 CURL: Contrastive Unsupervised Representations for Reinforcement Learning (source: Srinivas, Laskin, and Abbeel, 2020) . . . 85
6.4 Our four computer-vision objectives to learn an encoded representation of the scene. 86
6.5 Qualitative results with a ViT encoder for the pushing task. The upper two rows show prediction results for the image segmentation task, the lower two rows for the depth prediction task. . . . 89
6.6 State regression: Qualitative results of the pre-trained encoder for the reaching task on some test images. The black square shows the 2D projection of the 3D output for the ball's coordinates. 89
6.7 Depth prediction: Qualitative results of the pre-trained encoder for the pushing task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs. 90
6.8 Image segmentation: Qualitative results of the pre-trained encoder for the reaching task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs. 90
6.9 Auto-encoding: Qualitative results of the pre-trained encoder for the sliding task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs. 91
6.10 Learning curves for the encoder's supervised objectives. 92
6.11 State-encoding procedure. 93
6.12 Snapshots from each task: reach target, push button, slide block. 94
6.13 Learning curves of SAC-PVE on three RLBench tasks. 96
6.14 Learning curves of the shared encoder. 97
6.15 Learning curves of SAC-PVE (shared encoder) on three RLBench tasks. 98
6.16 Full sim-to-real pipeline. Two additional steps with respect to the standard pipeline: texture randomization and data augmentation. 99
6.17 Texture randomization: the texture of the walls, table and target are randomized to bring variety to the dataset. 99
6.18 Data augmentation: during training of the encoder, visual transformations are applied to further diversify the data. 100

List of Tables

2.1 Overview of RL algorithms 31

4.1 SAC-R^2, with and without decay, compared to the baseline SAC-Demo on evaluation episodes. The average success rate and average episode length (of succesful episodes only) are calculated across 3000 evaluation episodes (3 seeds, 1000 per seed). 58

5.1 Summary of the training results (Figures 5.3, 5.4, 5.5). The values correspond to the rounded final training success rate and the amount of iterations needed to converge. The column-wise best results are marked in bold. 69

6.1 Comparison on evaluation episodes. The first value is the average success rate, and the second value (between parentheses) is the average episode length (of succesful episodes only), calculated across 3000 evaluation episodes (3 seeds, 1000 per seed, except for SAC+AE-CURL where we exclude the seed that made zero progress in the reach task). The column-wise best vision results are marked in bold. 95

6.2 SAC-PVE, with and without a shared encoder, compared on evaluation episodes. Same conditions as in Table 6.1. 97

Chapter 1

Introduction

1.1 Context: Learning for Collaborative Robotics

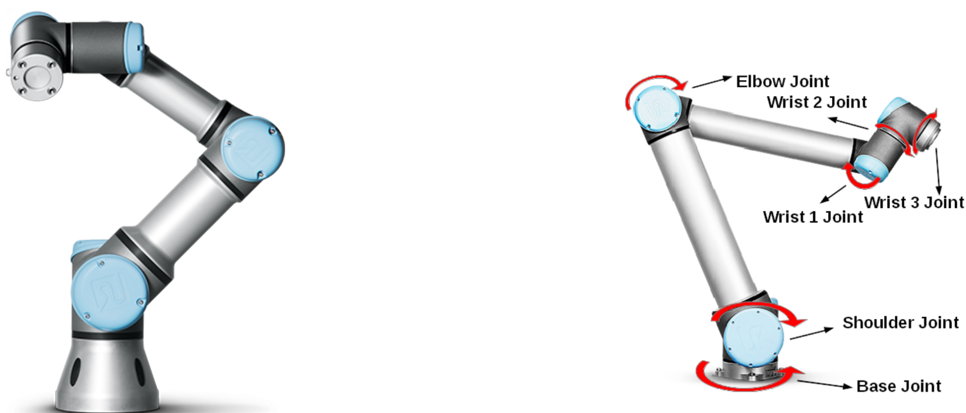


FIGURE 1.1: UR3 robot from Universal Robots: robotic arm with 6 degrees of freedom, equipped with a Robotiq gripper and wrist camera.

The first generation of manufacturing robots were always operating in human-free areas, for safety reasons. During recent years, new types of *collaborative* robots have been designed for deployment in direct contact, and even cooperation, with human workers. One example is shown in Figure 1.1, the UR3 robot from Universal Robots, which assisted us during this thesis. To take advantage from this new opportunity for intelligent human-robot interaction, next-generation robots need to be able to understand dynamic environments and react properly and safely to unexpected changes that might occur. Such behaviour would benefit to a wide range of robot-manipulation applications in factories, but also in hospitals, service industries, and even at home.

When it comes to learning controllers for such robots, recent advances in Deep Learning and computational capabilities allow to solve complex problems in an end-to-end fashion, mapping observations to controls directly, replacing the classic multi-staged control pipeline (Figure 1.3). The main advantage of learning-based controllers is that they produce control policies able to generalize to novel situations, different from the ones encountered during training. These

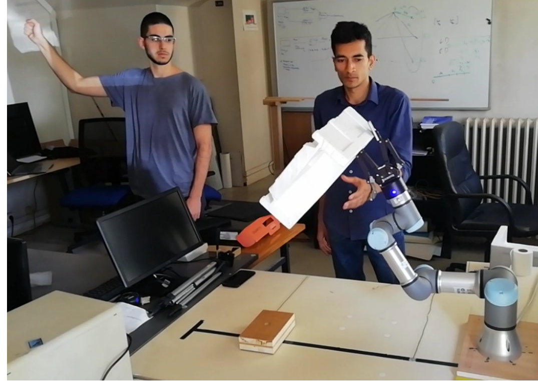


Figure 1.2: Real-time gestural control of robot manipulator through Deep Learning human-pose inference
The robot follows the movements of the operator (left) to pick an object. The robot's camera (not the one taking the shot) is fixed and placed in front of the operator.

kinds of algorithms are mostly derived from the Reinforcement Learning (RL) framework, and aim to learn general agents.

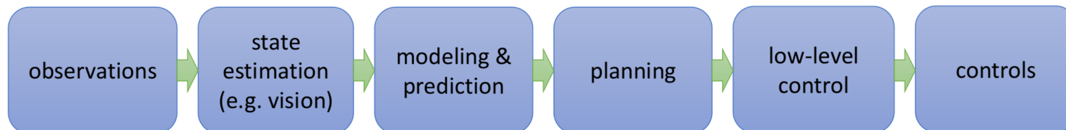


Figure 1.3: Classic robotic control pipeline (source: Levine, Fa2019)

As the RL community leans towards data-driven approaches (offline RL, decision transformers, etc.), in this thesis we focus on off-policy RL, and explore different ways of incorporating additional data into the algorithms. In particular, we rely on expert demonstrations, which can help with efficiency as well as overall performance. The goal is to design efficient algorithms to solve a range of robotic-manipulation tasks, such as flipping a switch or sliding a cube on a table.

Indeed, another effective approach for learning robotic controllers involves imitating expert demonstrations that exhibit desirable behavior, a framework known as Imitation Learning (IL). Until recently, learning from demonstrations was possible through non-visual demonstrations, collected with kinesthetic teaching or tele-operation, which typically required a lot of effort to collect. However, the data-collection process is no longer the bottleneck that it used to be thanks to advances in computer vision and simulation environments.

The context for this PhD is the paradigm shift in industrial automation that will follow from recent advances in robotic hardware and Deep Learning. In particular, it explores how robots are able to learn complex behaviours in a variety of manipulation tasks, by leveraging both imitation and reinforcement signals.

1.2 Publications

This PhD has been conducted in the center for robotics of Mines Paris, PSL University. The main publications and communications of this thesis can be synthesized as follows:

- Jesus Bujalance Martin, Raphaël Chekroun, Fabien Moutarde. **Learning from demonstrations with SACR2: Soft Actor-Critic with Reward Relabeling.** Deep RL Workshop at Conference on Neural Information Processing Systems (NeurIPS), 2021.
- Jesus Bujalance Martin, Tao Yu, Fabien Moutarde. **Pre-trained Image Encoder for Data-Efficient Reinforcement Learning and Sim-to-Real transfer on Robotic-Manipulation tasks.** Workshop on Pre-training Robot Learning at Conference on Robot Learning (CoRL), 2022.
- Jesus Bujalance Martin, Fabien Moutarde. **Reward Relabelling for combined Reinforcement and Imitation Learning on sparse-reward tasks.** Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2023.

The following two additional works fall outside the scope of this thesis, so their contents are not included in this manuscript. In the first one we explored two different ways of enhancing human-robot interaction with the help of existing computer vision technologies: continuous motion control (i.e. 'mirroring' of the human arm in real time, see Figure 1.2), and gesture recognition. In the second one we presented two RL algorithms inspired by Curriculum Learning: one for sampling more relevant experiences from the replay buffer, and one for sampling more relevant initial states.

- Jesus Bujalance Martin, Fabien Moutarde. **Real-time gestural control of robot manipulator through Deep Learning human-pose inference.** International Conference on Computer Vision Systems (ICVS), 2019.
- Dániel Horváth, Jesús Bujalance Martín, Ferenc Gábor Erdős, Zoltán Istenes, Fabien Moutarde. **HiER: Highlight Experience Replay and Easy2Hard Curriculum Learning for Boosting Off-Policy Reinforcement Learning Agents.** Submitted to IEEE Access.

1.3 Outline

This thesis is laid out in seven chapters:

Chapter 1: Introduction. We provide an introduction to Collaborative Robotics and the context surrounding this thesis, a list of contributions, and this outline.

Chapter 2: State of the art - Deep Reinforcement Learning. We thoroughly review the RL literature, the foundational algorithms such as policy gradient and dynamic programming, and the different families of deep-RL algorithms, presenting their strengths and shortcomings.

Chapter 3: State of the art - Imitation Learning. We review three different families of IL algorithms: Behavior Cloning, inverse RL, and offline RL.

Chapter 4: Reward Relabeling for combined Reinforcement and Imitation Learning. We introduce our reward-relabeling method, which can be seen as a form of reward shaping that happens in hindsight, once the entire episode is collected. This approach can easily extend any off-policy RL algorithm to benefit from both reinforcement and imitation signals. First, we present a short background on the three most closely-related families of RL algorithms. We then introduce the method, and a theoretical intuition to support and explain the equations governing the hyper-parameters. Finally, we carry a series of experiments to validate the approach and better understand its components via an ablation study.

Chapter 5: STIR2: Self and Teacher Imitation by Reward Relabeling. We introduce a second algorithm STIR^2 that builds on the method presented in the previous chapter, by aggregating previous and concurrent works that also exploit demonstration data and can be applied to any continuous-action off-policy RL algorithm. After introducing these methods individually, we present the full algorithm STIR^2, and finally evaluate its components with a thorough experimental study.

Chapter 6: Pre-trained Vision Encoder for Data-Efficient Reinforcement Learning. We move onto the more realistic setting of vision-based RL. To tackle this problem, we design a two-stage training pipeline: first learn a visual representation of the scene by pre-training an encoder from multiple supervised computer-vision objectives, then train a RL agent which can focus solely on solving the task. After a short background on the challenges that come with this new setting, we introduce both modules of our pipeline, each followed by experimental results. At the end we provide two enhancements: a multi-task encoder, and an alternative pipeline for sim-to-real transfer.

Chapter 7: Conclusion. Finally, we summarize our findings and open potential further directions for research in combining RL and IL for solving more difficult manipulation tasks.

1.4 Résumé en français

Cette thèse contient sept chapitres :

Chapitre 1: Introduction. Nous présentons une introduction à la robotique collaborative et au contexte de cette thèse, une liste des contributions, ainsi que ce plan.

Chapitre 2: État de l'art - Apprentissage par renforcement profond. Nous passons en revue la littérature sur l'apprentissage par renforcement, les algorithmes fondamentaux tels que le policy-gradient et la programmation dynamique, et les différentes familles d'algorithmes d'apprentissage par renforcement profond, en présentant leurs forces et leurs faiblesses.

Chapitre 3: État de l'art - Apprentissage par imitation. Nous passons en revue trois familles différentes d'algorithmes d'apprentissage par imitation : Behavior-Cloning, inverse-RL, et offline-RL.

Chapitre 4: Ré-étiquetage des récompenses pour l'apprentissage par renforcement et imitation.. Nous présentons notre méthode de ré-étiquetage des récompenses, qui peut être considérée comme une forme de reward-shaping qui se produit a posteriori, une fois que l'ensemble de l'épisode a été collecté. Cette approche permet d'étendre facilement tout algorithme RL off-policy pour bénéficier à la fois des signaux de renforcement et d'imitation. Nous présentons tout d'abord un bref historique des trois familles d'algorithmes RL les plus étroitement liées. Nous présentons ensuite la méthode, ainsi qu'une section théorique pour expliquer les équations régissant les hyperparamètres. Enfin, nous menons une série d'expériences et une étude d'ablation pour valider l'approche et mieux comprendre ses composantes.

Chapitre 5: STIR2: Self and Teacher Imitation by Reward Relabeling. Nous introduisons un second algorithme STIR^2 qui s'appuie sur la méthode présentée dans le chapitre précédent, en agrégeant des travaux antérieurs et concurrents qui exploitent également des données de démonstration. Après avoir présenté ces méthodes individuellement, nous présentons l'algorithme complet STIR^2, et nous évaluons ses composants à l'aide d'une étude expérimentale approfondie.

Chapitre 6: Encodeur de vision pré-entraîné pour un apprentissage par renforcement plus efficace. Nous passons ensuite au cadre plus réaliste de l'apprentissage par renforcement basé sur la vision. Pour aborder ce problème, nous concevons un pipeline d'entraînement en deux étapes : d'abord, apprendre une représentation visuelle de la scène en pré-entraînant un encodeur à partir de plusieurs objectifs supervisés de vision, puis entraîner un agent d'apprentissage par renforcement qui peut se concentrer uniquement sur la résolution de la tâche. Après un bref résumé des défis posés par ce nouveau contexte, nous présentons les deux modules de notre pipeline, chacun suivi de résultats expérimentaux. A la fin, nous proposons deux améliorations : un encodeur multi-tâches et un pipeline alternatif pour le transfert au monde réel.

Chapitre 7: Conclusion. Enfin, nous résumons nos résultats et identifions les orientations futures potentielles de notre recherche.

Chapter 2

State of the art - Deep Reinforcement Learning

Reinforcement learning (RL) is the sub-field of machine learning that studies how to use past experience to enhance the future control of an agent in a (potentially unknown) dynamical environment. Imitation learning (IL) is an alternative approach, where rather than past experience the agent relies on a set of demonstrations provided by an expert (typically a human), which it tries to imitate.

In the following two chapters we will define the reinforcement and imitation learning problems and try to capture the main approaches and state-of-the-art algorithms. The main sources are the courses *Reinforcement Learning and Optimal Control* (Bertsekas, 2019) from Dimitri Bertsekas at Arizona State University and *Deep Reinforcement Learning* (Levine, Fa2019) from Sergey Levine at UC Berkeley, as well as the book *Reinforcement Learning: an introduction* (Sutton and Barto, 2011) from Sutton and Barto.

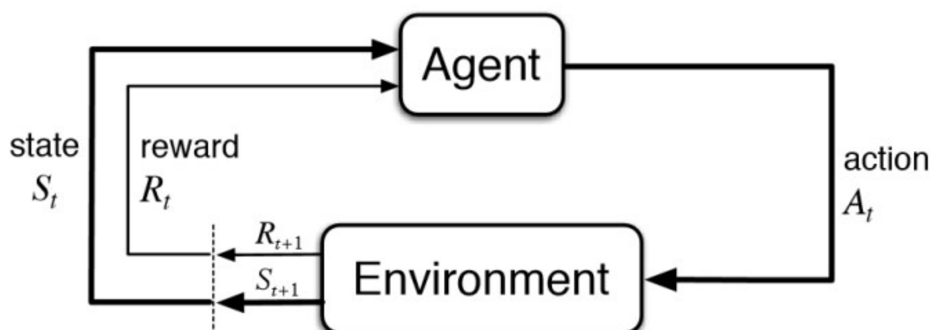


FIGURE 2.1: Reinforcement learning framework (source: Sutton and Barto, 2011)

In RL an agent interacts with an environment by performing an action and observing a feedback signal (reward) and the new state of the environment. A RL problem can be mathematically represented by a MDP (Markov decision process), which is a 5-tuple $(S, A, \mathcal{T}, r, \gamma)$ where:

- S is the space (discrete or continuous) of states s_t (or x_t).

- A is the space (discrete or continuous) of actions a_t (or u_t).
- $\mathcal{T}$ is a set of conditional transition probabilities between states $p(s_{t+1}|s_t, a_t)$, also referred to as the *dynamics* of the system. These transitions aren't necessarily time-invariant but they usually are.
- r is the reward function. $r(s, a) \in \mathbb{R}$ is the reward obtained from doing action a in state s. Also known as cost c, which is the opposite of the reward. The reward function isn't necessarily time-invariant but it usually is. It can sometimes depend on the state only $r(s)$ or on the entire transition $r(s_t, a_t, s_{t+1})$.
- $\gamma \in [0, 1]$ is the discount factor. It has two equivalent interpretations. The most obvious one is that it weighs the rewards depending on when the agent gets them: if $\gamma < 1$ future rewards are smaller. The second interpretation is that, at every time-step, there is a probability $1 - \gamma$ that the agent dies (i.e. goes into an absorbing state with reward zero).

More generally, we can define a POMDP (partially observable MDP) which models an agent that makes decisions based on partial observations of the states (e.g. a robot equipped with a camera observes images of the environment). A POMDP is a 7-tuple $(S, A, \mathcal{T}, r, \gamma, \Omega, \mathcal{O})$ where

- Ω is the space (discrete or continuous) of observations o_t
- $\mathcal{O}$ is the sensor model $p(o_t|s_t)$ or sometimes $p(o_t|s_t, a_{t-1})$

A POMDP can be reframed as an MDP where the states are belief states, i.e. probability distributions over the states of the POMDP.

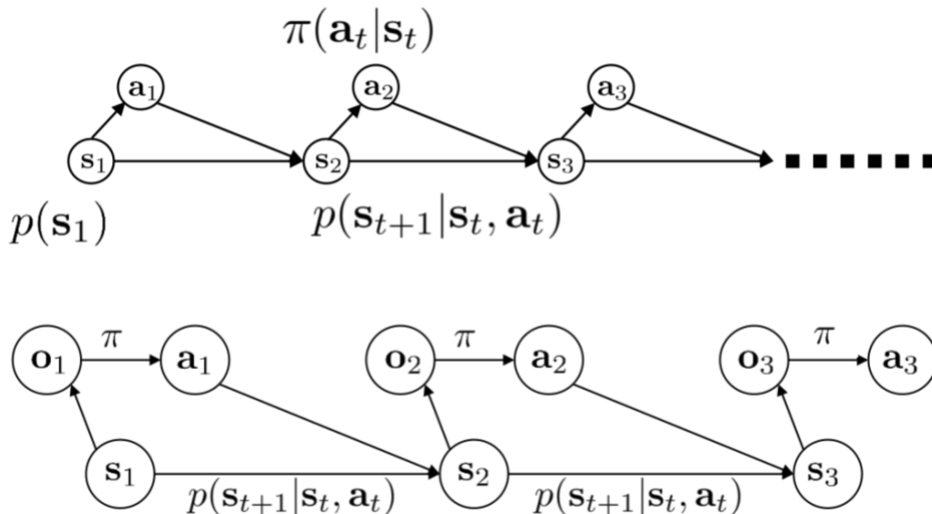


FIGURE 2.2: Graphical model of a MDP (up) and POMDP (down) (source: Levine, Fa2019)

. For an MDP, the state-action distribution $p(s_t, a_t)$ is a Markov chain:

$$p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p(s_{t+1}|s_t, a_t)\pi(a_{t+1}|s_{t+1})$$

The agent's goal is to maximize the cumulative reward it receives in the long run. Let's write down some definitions that will help us define this objective:

- A *policy* (or controller μ in control theory) is a function $\pi : S \to A$ that maps an action to a state. It can be deterministic or stochastic. In fully observed MDPs, we can always find an optimal policy that is deterministic, but stochastic policies have other properties that might be interesting in some cases. Policies aren't necessarily time-invariant, in fact, optimal policies in finite horizon tasks aren't. But we usually consider stationary policies for simplicity.
- A *trajectory* τ is a sequence $(s_1, a_1, s_2, a_2 ... s_T, a_T)$ of the visited states and actions. We have $p_\pi(\tau) = \pi(\tau) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$.
- An *episodic* task is a task in which there is a natural final time step and the interaction breaks naturally into sub-sequences called episodes. In general, each episode has a different *horizon* $T < \infty$ when the episode ended, and we set $\gamma = 1$. When the interaction goes on continually without limit we have a *continuous* task. In this case, we have $T = \infty$ and we set $\gamma < 1$ so that the cumulative reward doesn't go to infinity. In theory, there isn't a big difference between these two settings as γ implicitly defines an expected horizon $T \approx \frac{1}{1-\gamma}$.
- The objective J in RL is to maximize the cumulative reward (discounted for continuous tasks):

$$R_t = \sum_{k=0}^{T} \gamma^k r(s_{t+k}, a_{t+k}) \tag{2.1}$$

$$\pi^* = \arg\max_\pi J(\pi) = \arg\max_\pi \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[R_1 \right] \tag{2.2}$$

Let's define some tools that will help us solve this objective:

- The *Q-function* $Q^\pi(s_t, a_t) = \mathbb{E}_{p_\pi} \left[R_t | s_t, a_t \right] = \sum_{t'=t}^{T} \mathbb{E}_{p_\pi} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t, a_t \right]$ is the reward-to-go from state s_t if we pick the action a_t and then follow π.
- The *value function* $V^\pi(s_t) = \mathbb{E}_{p_\pi} \left[R_t | s_t \right] = \sum_{t'=t}^{T} \mathbb{E}_{p_\pi} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t \right] = \mathbb{E}_{a_t \sim \pi(a_t|s_t)} \left[Q^\pi(s_t, a_t) \right]$ is the reward-to-go from state s_t if we follow the policy π. We have $J(\pi) = \mathbb{E}_{s_1 \sim p(s_1)} \left[V^\pi(s_1) \right]$.
- The *advantage function* $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ represents the value of a particular action a_t with respect to the average action taken by π at state s_t.
- A *model* in RL refers to a model of the environment, i.e. the transition probabilities $p(s_{t+1}|s_t, a_t)$ (and sometimes also the reward $r(s_t, a_t)$). We distinguish *model-based* algorithms from *model-free* algorithms.

  Model-based algorithms, such as policy/value iteration, need explicit access to these probability distributions of next state (and reward). Other model-based algorithms, such as MCTS, only require a sampling model,

i.e. a simulator that takes a state and action as inputs, and returns a single next state and reward with the same probabilities as the target system. Common assumptions are that the simulator generates samples fast enough to plan online, or that it can be reset (in order to collect multiple trajectories).

Model-free algorithms, such as policy gradients, actor-critics or Q-learning, rely on real samples from the environment.

- Another important distinction between algorithms is whether the data must be collected by the current policy (*on-policy*) or not (*off-policy*).

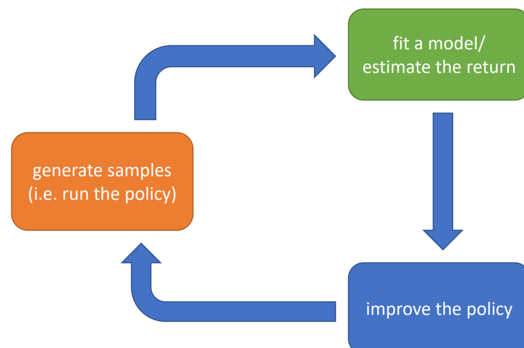


FIGURE 2.3: The broad structure of a RL algorithm (source: Levine, Fa2019)

The main challenges in reinforcement learning are credit assignment (which actions actually led to the reward), exploration (collecting good data) and generalization (to new unseen data). Simpler settings allow to study these problems more independently (exploration and generalization in contextual bandits, exploration and credit assignment in tabular MDPs, generalization and credit assignment in policy improvement).

2.1 Policy gradient

The idea is to do stochastic gradient ascent on the objective $J(\pi)$ to find the best parameterized stochastic policy π_θ. A simple example is the Reinforce algorithm (Williams, 1992).

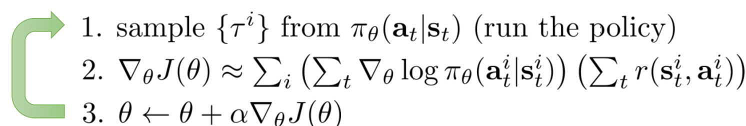


FIGURE 2.4: Reinforce algorithm (source: Levine, Fa2019)

The derivation is straight-forward. The idea is to use a logarithm to get rid of the terms depending on the dynamics: $\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \sum_t \log \pi_\theta(a_t|s_t)$. Step 2 of the algorithm uses samples to approximate an expectation. Step 3 is gradient ascent, making good sampled trajectories more likely, and bad

trajectories less likely.

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[r(\tau)\right] \text{ and } \nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau) r(\tau)\right] \tag{2.3}$$

Causality. We usually replace $\sum_{t=1}^{T} \gamma^{t-1} r(s_t^i, a_t^i)$ with $\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$, i.e. we hold policies accountable for future rewards only, which reduces the variance but might bring a bias when $\gamma < 1$ (Thomas, 2014). Note that γ can be seen as a bias-variance trade-off hyper-parameter, since rewards far away into the future have higher variance.

Reparameterization trick. An alternative to the likelihood-ratio gradient (equation 2.4) used in Reinforce to compute a gradient of the form $\nabla_\theta \mathbb{E}_{z \sim p_\theta(z)}\left[f(z)\right]$, is the reparameterization trick (equation 2.5). It decouples the stochasticity from z, by considering another distribution $\epsilon \sim q(\epsilon)$ independent of θ. Then, $z = g_\theta(\epsilon)$ is recovered from the noise variable. For instance, if $z \sim \mathcal{N}(\mu, \sigma)$, then $z = \mu_\theta + \epsilon \sigma_\theta$ where $\epsilon \sim \mathcal{N}(0, I)$. This alternative only works under some conditions (differentiable model, continuous and reparameterizable distributions), but it has a lower variance due to the implicit modeling of the dependencies. *Stochastic Value Gradients* (SVG) (Heess et al., 2015) is an example of a family of general policy gradient algorithms based on the reparameterization trick.

$$\nabla_\theta \mathbb{E}_{z \sim p_\theta(z)}\left[f(z)\right] = \mathbb{E}_{z \sim p_\theta(z)}\left[f(z) \nabla_\theta \log p_\theta(z)\right] \approx \frac{1}{N} \sum_{i}^{N} f(z_i) \nabla_\theta \log p_\theta(z_i) \tag{2.4}$$

$$\nabla_\theta \mathbb{E}_{z \sim p_\theta(z)}\left[f(z)\right] = \mathbb{E}_{\epsilon \sim q(\epsilon)}\left[\nabla_\theta f(g_\theta(\epsilon))\right] \approx \frac{1}{N} \sum_{i}^{N} \nabla_\theta f(g_\theta(\epsilon_i)) \tag{2.5}$$

2.1.1 Importance sampling

$$J(\theta) = \mathbb{E}_{p_\theta}\left[r(\tau)\right] = \mathbb{E}_q\left[\frac{p_\theta(\tau)}{q(\tau)} r(\tau)\right] \approx \frac{1}{Z(\theta)} \sum_{i=1}^{m} \frac{p_\theta(\tau_i)}{q(\tau_i)} r(\tau_i) \text{ with } \begin{array}{l} \tau_i \sim q \\ \tau_k \sim p_\theta \end{array} \tag{2.6}$$

$$\omega_i = \frac{1}{Z(\theta)} \frac{p_\theta(\tau_i)}{q(\tau_i)} = \frac{1}{Z(\theta)} \prod_{t=1}^{T} \frac{\pi_\theta(a_t^i | s_t^i)}{q(a_t^i | s_t^i)} \text{ and } ESS = m \frac{\mathrm{var}_{p_\theta}\left[\sum_k \frac{1}{m} r(\tau_k)\right]}{\mathrm{var}_q\left[\sum_i \omega_i r(\tau_i)\right]} \tag{2.7}$$

Importance sampling allows us to approximate the objective by sampling from another distribution q, like a previous policy or any off-policy guiding distribution that is somewhat similar to p. The best distribution in terms of minimum variance is $q(\tau) \propto p_\theta(\tau)|r(\tau)|$. We can choose between $Z(\theta) = m$ (unbiased consistent estimator) and $Z(\theta) = \sum_i \frac{p_\theta(\tau_i)}{q(\tau_i)}$ (biased consistent estimator with lower variance, usually preferred). To include samples from multiple distributions, we can use $q(\tau) = \frac{1}{n} \sum_j q_j(\tau)$. Note that the importance weights w_i do not depend on the dynamics because the transition terms cancel out.

The effective sample size ESS is commonly used to measure the quality of an importance sampled estimate. It can be approximated by $\frac{m}{1+\text{var}_q[\omega_i]}$ or $\frac{1}{\sum_i \omega_i^2}$. As the policy search steps away from areas where we have gathered samples, the variance of the estimator increases so we only accept values of θ with high ESS. However, ESS tends to be unreliable with complex policies and long rollouts because very few samples have nonzero weights.

2.1.2 Advanced policy gradient

One big issue with vanilla policy gradient is that small changes in parameter space can lead to big changes in policy space, which makes it difficult to tweak the step-size α and to actually converge. More recent algorithms address this and perform better than vanilla policy gradient. They solve the following optimization problem, which consists on minimizing a surrogate loss derived from importance sampling:

$$L(\theta) = \mathbb{E}_{p_{\theta_{\text{old}}}}\left[\sum_t \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\gamma^t A^{\pi_{\theta_{\text{old}}}}(s_t, a_t)\right] \approx J(\theta) - J(\theta_{\text{old}}) \tag{2.8}$$

$$\theta \leftarrow \arg\max_\theta L(\theta) \quad \text{s.t.} \quad \max_s D_{\text{KL}}(\pi_\theta(\cdot|s)||\pi_{\theta_{\text{old}}}(\cdot|s)) < \epsilon \tag{2.9}$$

For ϵ small enough we are guaranteed to improve our policy because we are optimizing a lower bound of the performance objective (see Figure 2.5). In practice we use the constraint $\mathbb{E}_{s\sim d^{\pi_{\theta_{\text{old}}}}}[D_{\text{KL}}(\pi_\theta(\cdot|s)||\pi_{\theta_{\text{old}}}(\cdot|s))] < \epsilon$ because the max is not tractable. Note that policy gradient can be seen as policy iteration 2.2, in that we evaluate our old policy with a function (the advantage), and then we maximize the function w.r.t. the new policy. The approximation made in equation 2.8 only holds if the distributions p_θ and $p_{\theta_{\text{old}}}$ are close enough, which we translate into the Kullback-Leibler divergence constraint between the policies (the dynamics terms cancel out). If we consider an euclidean constraint on the parameters instead $||\theta - \theta_{\text{old}}||^2 < \epsilon$ we recover the vanilla policy gradient.

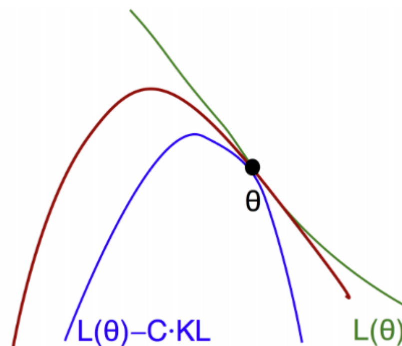


FIGURE 2.5: Policy gradient (source: John Schulman's slides)
The red curve is the performance objective $J(\theta) - J(\theta_{\text{old}})$.

How do we optimize this new objective in practice?

- *Trust region policy optimization* (Schulman et al., 2015a). TRPO considers a 2nd order approximation of the KL divergence and approximates the Hessian with the Fisher information matrix $F = \mathbb{E}_{p_{\theta_{\text{old}}}} \left[\nabla_\theta \log \pi_{\theta_{\text{old}}}(a|s)^T \nabla_\theta \log \pi_{\theta_{\text{old}}}(a|s) \right]$. We have $\nabla_\theta L(\theta_{\text{old}}) = \nabla_\theta J(\theta_{\text{old}})$, so the update becomes:

$$\theta \leftarrow \theta_{\text{old}} + \alpha F^{-1} \nabla_\theta J(\theta_{\text{old}}) \tag{2.10}$$

  where the term $F^{-1}\nabla_\theta J(\theta_{\text{old}})$ is referred to as natural policy gradient. We have $\alpha = \sqrt{2\epsilon / (\nabla_\theta J(\theta_{\text{old}})^T F \nabla_\theta J(\theta_{\text{old}}))}$ for the initial constrained problem, and $\alpha = \frac{1}{\beta}$ for the penalized problem:

$$\theta \leftarrow \arg\max_\theta \nabla_\theta L(\theta_{\text{old}})^T (\theta - \theta_{\text{old}}) - \frac{\beta}{2} (\theta - \theta_{\text{old}})^T F (\theta - \theta_{\text{old}}) \tag{2.11}$$

  Additionnaly, TRPO does conjugate gradient descent to compute $F^{-1}\nabla_\theta J(\theta_{\text{old}})$ efficiently, and a line search with exponential decay on α to make sure that the KL constraint is respected and performance improves.

- *Proximal policy optimization* (Schulman et al., 2017). PPO is a family of 1st order methods that approximately enforce the KL constraint. One idea is to modify the penalty coefficient β_k between iterations, with dual gradient descent or some heuristic. Another idea is to do stochastic gradient ascent on a clipped objective (where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$):

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_{p_{\theta_{\text{old}}}} \left[\sum_t \gamma^t \min \left[r_t(\theta) A_t^{\pi_{\theta_{\text{old}}}}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) A_t^{\pi_{\theta_{\text{old}}}} \right] \right]$$

2.1.3 Evolution strategies

As pointed out in (Recht, 2019), Reinforce is actually a gradient-free algorithm since the function we care about optimizing (the cumulative reward) is only accessed through function evaluations. Another very simple gradient-free method is random search, which consists in perturbing the parameters of the deterministic policy π_θ randomly (e.g. $\pi_{\theta+w}$ where w is a centered Gaussian) and then update θ based on the received reward at this perturbed value. Thus, instead of sampling from a stochastic policy in step 1, we sample from a deterministic policy with perturbed parameters.
This perturbation approach is closely related to finite differences. If we sample two trajectories τ^+ and τ^- with mirror noise from a centered Gaussian, i.e. $\theta^+ = \theta + \epsilon\sigma$ and $\theta^- = \theta - \epsilon\sigma$ where $\epsilon \sim \mathcal{N}(0,1)$, then Reinforce does approximate gradient ascent with the gradient approximation $\frac{\epsilon}{2\sigma}(r(\tau^+) - r(\tau^-))$.
One advantage of parameter space exploration is its ability to avoid local optima and delayed reward issues better than classic RL algorithms. However, the search space on action parameters is thought to be smaller than the search space on policy parameters (and therefore requires fewer environment interactions to estimate a gradient for).

More generally, Evolution Strategies (ES) is a type of model-agnostic optimization approach, inspired by Darwin's natural selection. Say, we start with a population of random solutions. All of them are capable of interacting with the environment and only candidates with high fitness F scores can survive. A new generation is then created by recombining the settings (gene mutation) of high-fitness survivors. This process is repeated until the population gets good enough. Let's assume the distribution over θ is an isotropic multivariate Gaussian, the gradient of $F(\theta)$ is:

$$\nabla_\theta \mathbb{E}_{\theta\sim\mathcal{N}(\mu,\sigma^2 I)} F(\theta) = \mathbb{E}_{\theta\sim\mathcal{N}(\mu,\sigma^2 I)} \left[F(\theta)\frac{\theta-\mu}{\sigma^2} \right] \tag{2.12}$$

$$\nabla_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\theta+\sigma\epsilon) = \frac{1}{\sigma}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[F(\theta+\sigma\epsilon)\epsilon \right] \tag{2.13}$$

ES don't require any value function approximation, don't use gradient backpropagation, are invariant to delayed or long-term rewards, and are highly parallelizable with very little data communication between workers.

2.1.4 Summary

- Policy gradient converges in theory (although usually to local optima) and works well with continuous actions.
- Policy gradient is entirely on-policy and the variance of the gradient estimator is very big, thus requiring a lot of samples to converge. There are some tricks that can help, like *importance sampling* to use off-policy samples or *baselines* to reduce the variance.

2.2 Dynamic programming and value function methods

The dynamic programming approach to the RL problem is completely different. It actually solves the dual problem of our initial objective 5.1. Let's place ourselves in the finite horizon case with time-dependant policies and value functions. The solution is based on the principle of optimality: if $\pi = (\pi_1, \pi_2 ... \pi_T)$ is an optimal policy, then the optimal strategy starting at state s_t is to follow the policy $(\pi_t, \pi_{t+1} ... \pi_T)$. Let $\pi^* = \arg\max_\pi V^\pi$ an optimal policy and $V^* = V^{\pi^*}$. Dynamic programming lets us recursively find an optimal deterministic policy starting from $V_T^*(s_T) = r(s_T)$ and solving backwards:

$$V_t^*(s_t) = \max_{a_t} \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)} \left[r(s_t,a_t) + V_{t+1}^*(s_{t+1}) \right] \tag{2.14}$$

$$Q_t^*(s_t,a_t) = \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)} \left[r(s_t,a_t) + \max_{a_{t+1}} Q_{t+1}^*(s_{t+1},a_{t+1}) \right] \tag{2.15}$$

The second equation shows the Q-function version of this algorithm. We can keep track of the chosen actions at each time-step to obtain the optimal deterministic policy or we can simply compute the greedy policy with respect to the Q-function:

$$\pi_t^*(s_t) = \arg\max_{a_t} Q_t^*(s_t, a_t) \tag{2.16}$$

In the infinite horizon case where everything is stationary, we obtain the Bellman equations verified by the value function (similar equations hold for the Q-function):

$$V^*(s_t) = \max_{a_t} \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)} \left[r(s_t, a_t) + \gamma V^*(s_{t+1})\right] \tag{2.17}$$

$$V^\pi(s_t) = \mathbb{E}_{a_t\sim\pi(a_t|s_t)} \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)} \left[r(s_t, a_t) + \gamma V^\pi(s_{t+1})\right] \tag{2.18}$$

We obtain three algorithms that have convergence theoretical guarantees based on the Bellman operator being a contraction. However, they require a tabular representation of the value function (or Q-function), i.e. the ability to store in a table the value for all states (or state-action pairs). They also require known dynamics to compute the expectations over the next state:

- Value Iteration (VI) consists on iterating the optimal Bellman operator 2.17 until convergence. In the Q-function version (usually called Q-iteration) we have $\lim_k Q_k(s_t, a_t) = Q^*(s_t, a_t)$ where

  $$Q_{k+1}(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim p(\cdot|s_t,a_t)} \left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_k(s_{t+1}, a_{t+1})\right] \tag{2.19}$$

  Upon convergence, we return the greedy policy with respect to the Q-function. Each iteration is computationally efficient but convergence is only asymptotic.

- Policy Iteration (PI) consists on two steps. In step 1 we evaluate the current deterministic policy π_k with the Q-function (or value function). One way to do so is iterating the Bellman operator 2.18 until convergence starting from a random initial Q-function. We have $\lim_j Q_j(s_t, a_t) = Q^{\pi_k}(s_t, a_t)$ where

  $$Q_{j+1}(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim p(\cdot|s_t,a_t)} \left[r(s_t, a_t) + \gamma \mathbb{E}_{a\sim\pi_k} Q_j(s_{t+1}, a)\right] \tag{2.20}$$

  In step 2 we improve the policy in a greedy manner:

  $$\pi_{k+1}(s_t) = \arg\max_{a_t} Q^{\pi_k}(s_t, a_t) \tag{2.21}$$

  In practice, PI usually converges after a small number of iterations. However, each iteration requires a full policy evaluation that might be expensive.

- Linear Programming. V^* is the solution to the linear program $\min_V \sum_s V(s)$ under the union of the following constraints:

$$V(s) \geq r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s') \quad \forall a \in A, \forall s \in S \tag{2.22}$$

2.2.1 Approximate dynamic programming

In RL we don't assume that we know the model, so we need to build estimators for the value function (or Q-function) based on collected data:

- Monte-Carlo (MC) estimator. Let $r_t^\pi = r(s_t, \pi(s_t))$. By definition, the value function is an expectation of the cumulative reward. We can approximate it by an incremental average upon collecting a new trajectory:

$$\hat{V}^\pi(s_t) \leftarrow (1-\alpha)\hat{V}^\pi(s_t) + \alpha \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^\pi \tag{2.23}$$

  We can use the collected trajectories more than once by reusing the sub-sequences that start from s_t, which gives a biased estimator with smaller variance. If we use more than one trajectory (using sub-sequences or running several trajectories from s_t if our simulator allows it) we can average them $\frac{1}{n}\sum_{i=1}^{n} r(\tau_i(s_t))$.

- Temporal Difference (TD) or bootstrap estimator. Inspired by the Bellman equation 2.18, we can update our estimator like follows:

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha \left(r_t^\pi + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t) \right) \tag{2.24}$$

  TD can work online (we don't have to collect a complete trajectory) and has smaller variance than MC, but is no longer unbiased. The variance of MC targets comes from stochasticity of rewards and dynamics, whereas the bias of TD targets comes from using an imperfect bootstrap to estimate future returns.

- n-step TD: intermediate estimator between TD ($n=1$) and MC ($n=\infty$):

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha \left(\sum_{t'=t}^{t-1+n} \gamma^{t'-t} r_{t'}^\pi + \gamma^n \hat{V}^\pi(s_{t+n}) - \hat{V}^\pi(s_t) \right) \tag{2.25}$$

  TD(λ): weighted average $(1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1}\delta_t^n$ where δ_t^n is the n-step TD error (the term multiplying α in the equation above). $\lambda \in [0,1]$ is a hyper-parameter between 0 (equivalent to TD) and 1 (equivalent to MC).

$$\hat{V}^\pi(s_t) \leftarrow \hat{V}^\pi(s_t) + \alpha \sum_{t'=t}^{T} (\gamma\lambda)^{t'-t} \left(r_{t'}^\pi + \gamma \hat{V}^\pi(s_{t'+1}) - \hat{V}^\pi(s_{t'}) \right) \tag{2.26}$$

  Eligibility traces: computational efficient method to approximate the TD(λ) update online. Instead of updating our value function based on future rewards, we propagate current error information into the states we visited

in the past. The eligibility trace z starts at $z_0(s) = 0$. For all t, we decay the contribution of all states by $\lambda\gamma$ and increase the contribution of the state that was visited.

$$z_t(s) = \lambda\gamma z_{t-1}(s) + 1_{[s_t=s]} \text{ and } \hat{V}^\pi(s) \leftarrow \hat{V}^\pi(s) + \alpha z_t(s)\delta_t^1 \quad \forall s \tag{2.27}$$

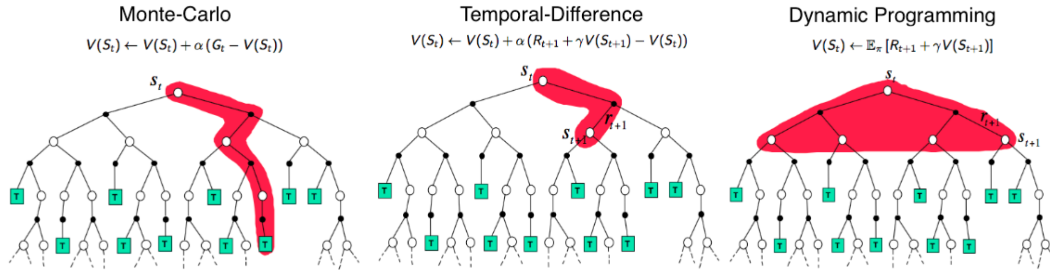


FIGURE 2.6: Comparison of backup diagrams for state value functions (source: David Silver's slides)

Let's look at three algorithms from (Sutton and Barto, 2011) based on bootstrap estimators of the Q-function, i.e. online updates based on the TD error $r_t + \gamma\tilde{Q} - \hat{Q}(s_t, a_t)$. Under some properties (in particular the data needs to be collected by a suitable exploration policy, more on this later) we have convergence of these algorithms in the tabular case.

- *Sarsa* is an on-policy algorithm that estimates Q^π for the current policy π (similar to PI). We have $\tilde{Q} = \hat{Q}(s_{t+1}, \pi(s_{t+1}))$.
- *Q-learning* is an off-policy algorithm that aims directly for Q^* (similar to Q-iteration). We have $\tilde{Q} = \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$. Note that Q-learning is not possible without Q-functions: if we compare 2.14 and 2.15, the order of the *max* operator and the expectation is reversed, which allows to approximate Q^* by sampling without knowing the dynamics.
- *Expected Sarsa* is a generalization. We have $\tilde{Q} = \mathbb{E}_{a_{t+1}\sim\pi}\left[\hat{Q}(s_{t+1}, a_{t+1})\right]$. If π is the greedy policy we recover Q-learning. If π is the current policy we recover an on-policy algorithm similar to Sarsa (more complex computationally but with less variance).

2.2.2 Deep Q-learning

If we step out of the tabular case, we need to approximate the Q-function with some parametric function Q_ϕ. By doing so, we lose all of the convergence guarantees. We obtain our first completely general algorithm called fitted Q-iteration:

Let's cover the two main improvements brought to this algorithm:

- *Replay buffer*. As we interact with the environment sequentially, the samples are strongly correlated which can lead to local over-fitting. Parallel computation can mitigate this issue to some degree, but not completely.

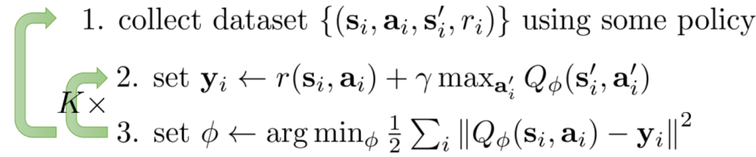


FIGURE 2.7: Fitted Q-iteration algorithm (source: Levine, Fa2019)

The idea is to have a buffer $\mathcal{B}$ where we add the collected data, and then update the Q-function from data sampled uniformly from $\mathcal{B}$.

- *Target network.* We do not run the supervised regression in step 3 until convergence because the optimal estimate from the available data will be very poor in early stages. This means that the targets y are going to change very frequently, which doesn't help with stability. The idea is to do regular backups of the parameters ϕ into a new target network $Q_{\phi'}$ used to compute these targets.

Q-learning with replay buffer and target network: DQN: $M = 1$, $K = 1$

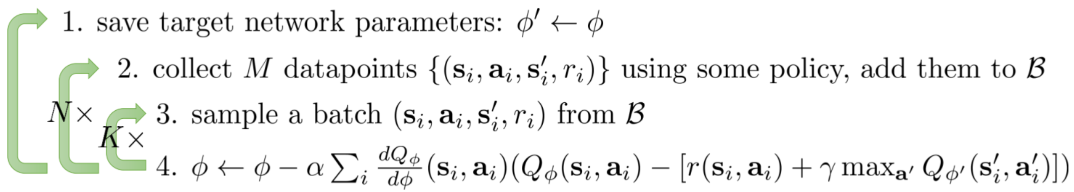


FIGURE 2.8: General Q-learning algorithm (source: Levine, Fa2019)

We obtain 2.8, which can be decomposed into three different processes:

- Data collection (and eviction if $\mathcal{B}$ is finite).
- Target update: update $\phi' \leftarrow \phi$ every several steps, update $\phi' \leftarrow \alpha\phi' + (1 - \alpha)\phi$ every step with a moving average...
- Q-function regression.

A very popular Q-learning algorithm is DQN (Mnih et al., 2015). Since, a lot of further improvements have been proposed, such as n-step DQN, Double DQN (Van Hasselt, Guez, and Silver, 2016), Prioritized Experience Replay (PER) (Schaul et al., 2015), Distributional DRL (Bellemare, Dabney, and Munos, 2017) or Rainbow (Hessel et al., 2018).

What about continuous actions? Computing the targets y and the greedy policy involves taking a max over actions, which is a problem. One idea is to use stochastic optimization methods (e.g. cross-entropy method - CEM), which consider $\max_a Q(s, a) = \max [Q(s, a_1)...Q(s, a_n)]$ where $(a_1...a_n)$ are sampled from some distribution. Another idea is to use a Q-function representation that is convex on the actions, providing an analytical solution (e.g. NAF: Gu et al., 2016b). A third idea is to learn an approximate maximizer $\mu_\theta(s) = \arg\max_a Q(s, a)$ like in DDPG (Lillicrap et al., 2015).

2.2.3 Experience replay

The replay buffer $\mathcal{B}$ is typically implemented as a circular buffer, where the oldest transition in the buffer is removed to make room for a transition that was just collected. The most basic strategy to sample transitions from the buffer is uniform sampling. Other strategies include PER (Schaul et al., 2015), that favors transitions with a high temporal difference (TD) error.

Three properties are affected when modifying the buffer size.

1. The *replay capacity* is the total number of transitions stored in the buffer. A larger replay capacity will typically result in a larger state-action coverage.

2. The *age of the oldest policy*, i.e. the age of the policy that collected the oldest transition in the buffer. It is a proxy of the degree of off-policyness: the older a policy, the more likely it differs from the current policy.

3. The *replay ratio* is the number of gradient updates per environment transition. It can be viewed as a measure of the relative frequency the agent is learning on existing data versus acquiring new experience.

Whenever the replay buffer size is increased, both the replay capacity and the age of the oldest policy increase. The replay ratio stays constant, but will change if one of the other two factors is independently modulated. Works like (Fedus et al., 2020) discuss in more detail the impact of these parameters.

2.2.4 Off-policy learning

In off-policy algorithms we do not assume that the policy π that is collecting the data, called *behaviour policy*, is the same than the policy that we are learning, called *target policy*. Why is Q-learning off-policy? Given s', the value $\max_{a'} Q_\phi(s', a')$ is independent of π because it computes the Q-function for another policy (the greedy policy). Given (s, a), the transition to s' and the reward $r(s, a)$ are given by the environment only. Given s, the action a does depend on π, but we don't actually care because we need to estimate the Q-function for all state-action pairs anyway.

What about n-step returns? If we consider more than one transition in our TD error we can no longer claim that Q-learning is off-policy. The n-step Tree Backup Algorithm (Sutton and Barto, 2011) solves this problem: "because we have no sample data for the unselected actions [by π during the n transitions], we bootstrap and use the estimates of their values in forming the target for the update". The algorithm is designed for any target policy (similar to Expected Sarsa). In practice, for Q-learning, the algorithm simply considers the regular n-step return, which gets truncated as soon as there is disagreement between π and the greedy target policy.

One caveat with off-policy learning is that there might be a distribution mismatch between our training set (collected by π) and the actual data that we are going to encounter at test-time (induced by our target policy). To mitigate

this, we need π to cover the regions that our final policy will visit in order to get good estimates of the Q-function where it matters. If we cannot use an ϵ-greedy policy (greedy w.r.t. the current Q-function with probability $1 - \epsilon$ and random with probability ϵ), a good idea is to use a mix of policies with different behaviours.

More generally, function approximation of Q-values, bootstrapping, and off-policy learning have been identified as the deadly triad of properties that, when combined, can negatively affect learning or even cause divergence. For instance, (Kumar, Gupta, and Levine, 2020) points out an absence of corrective feedback during training: errors in Q-values happen at some states due to their low frequency and aliasing ($\sim$ function approximation) with other states, and these values propagate ($\sim$ bootstrapping) and corrupt other states, even if the Bellman error is fully minimized at those states. The key is to better understand the data generating distribution and its properties: degree of on-policyness (how close is the data-generating distribution to the current policy being evaluated?), state-space coverage, correlation between transitions, cardinality of distribution support. Practically, these aspects may be difficult to control independently, and the typical algorithmic adjustments we can make affect several of these simultaneously.

2.2.5 Summary

- Q-learning methods are off-policy, so they tend to be much more sample efficient than direct policy optimization methods.
- They can handle continuous actions spaces with approximate methods, but are initially intended for discrete action spaces only.
- The main issue with value function methods is that, although they are designed to find a global optimal solution, they lose the convergence guarantees under function approximation. The learning process tends to be unstable as small updates to Q may significantly change the policy and the data distribution.

2.3 Actor-Critic

Actor-critic methods lie between policy gradients and value function methods, in that they use function approximation for both the policy and the value function. They are based on generalised policy iteration, interleaving policy evaluation (a critic Q_ϕ estimates the action-value function) with policy improvement (an actor adjusts the parameters of the policy π_θ).

As briefly mentioned in the policy gradient section, we can use a baseline b to reduce the variance of the gradient estimator while remaining unbiased (as long as the baseline doesn't depend on action in logprob(action)):

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) - b \right) \tag{2.28}$$

We are actually using the MC estimator of the Q-function with a single sample $\hat{Q}^\pi(s_t, a_t) = \sum_{t'=t}^{T} \gamma^{t'-t} r(s^i_{t'}, a^i_{t'})$, hence the high variance. We can use a bootstrap estimator instead $\hat{Q}^\pi(s_t, a_t) = r(s_t, a_t) + \gamma \hat{V}^\pi(s_{t+1})$ to reduce the variance. The next idea is to use a state-dependant baseline that approximates the value function $b = \hat{V}^\pi(s_t)$, giving us an approximation of the advantage function $\hat{A}^\pi(s_t, a_t)$. As usual, we approximate the function of interest with a parametric function V_ϕ.

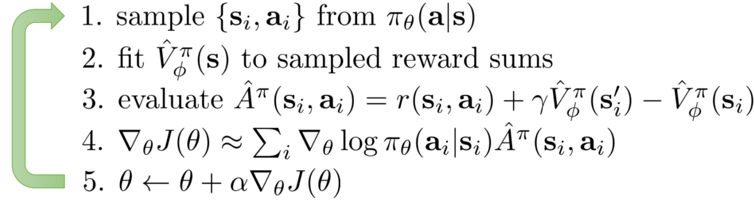


FIGURE 2.9: Advantage actor-critic algorithm (source: Levine, Fa2019)

The parallel asynchronous version of this algorithm, A3C (Mnih et al., 2016), is very popular.

- In step 2, we define targets y_i and minimize the supervised regression loss $\frac{1}{2} \sum_i ||V^\pi_\phi(s^i_t) - y_i||^2$. As usual, we have a choice for the targets between bootstrap $y_i = r(s^i_t, a^i_t) + \gamma V^\pi_\phi(s^i_{t+1})$ and MC $y_i = \sum_{t'} \gamma^{t'-t} r(s^i_{t'}, a^i_{t'})$.
- Some extensions use action-dependant baselines (i.e. $b = Q^\pi_\phi(s_t, a_t)$) or intermediate estimators for the advantage function between the MC estimator $\hat{A}^\pi(s^i_t, a^i_t) = \sum_{t'} \gamma^{t'-t} r(s^i_{t'}, a^i_{t'}) - V^\pi_\phi(s^i_t)$ and the critic estimator $\hat{A}^\pi(s^i_t, a^i_t) = r(s^i_t, a^i_t) + \gamma V^\pi_\phi(s^i_{t+1}) - V^\pi_\phi(s^i_t)$, like we did for n-step TD 2.2.1 (e.g. GAE: Schulman et al., 2015b).

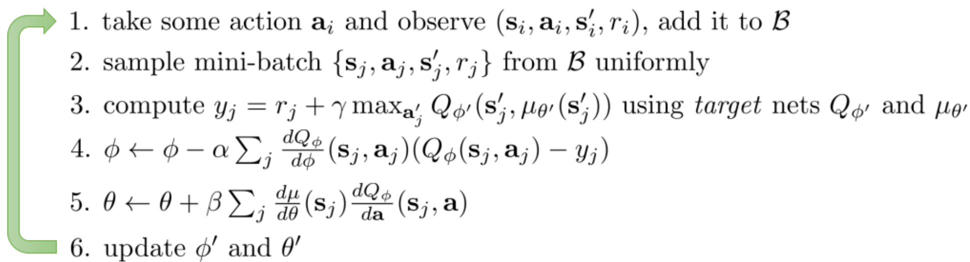


FIGURE 2.10: DDPG: deep deterministic policy gradient (source: Levine, Fa2019)

DDPG (Lillicrap et al., 2015) (deep DPG: Silver et al., 2014) is another popular actor-critic algorithm. It is an interesting example because it was actually introduced as an alternative to policy gradient for deterministic policies, and can also be seen as an alternative to Q-learning for continuous actions. A key feature is that it works off-policy: importance sampling in the actor is not necessary because the deterministic policy gradient removes the integral over actions, and importance sampling in the critic is not necessary because it uses Q-learning. Other popular actor-critic algorithms include TD3 (Fujimoto, Van

Hoof, and Meger, 2018), which brings some improvements to DDPG; D4PG (Barth-Maron et al., 2018), wich considers DDPG with a distributional critic and other improvements; Soft Actor-Critic (Haarnoja et al., 2018b), which is also off-policy and looks for a maximum-entropy stochastic policy; ACKTR (Wu et al., 2017), which expands A2C by taking steps in the natural gradient direction; Q-Prop (Gu et al., 2016a), which uses an action-dependant baseline Q_ϕ as a critic which can be updated off-policy, while the actor is updated on-policy; MPO (Abdolmaleki et al., 2018), which is off-policy and performs an expectation maximization form of policy iteration.

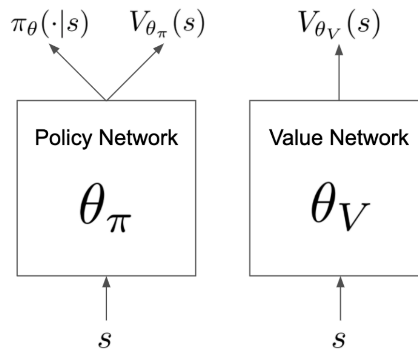


FIGURE 2.11: Phasic policy gradient (PPG) networks (source: Cobbe et al., 2021)

PPG (Cobbe et al., 2021) is a more recent on-policy actor-critic framework that achieves feature sharing between the policy and value function while decoupling their training, by operating in two alternating phases: the first phase trains the policy, and the second phase distills useful features from the value function. Also, by optimizing the training of each network with the appropriate level of sample reuse (higher for V), PPG significantly improves sample efficiency compared to PPO.

2.3.1 Summary

- Actor-critic algorithms have smaller variance than policy gradients and still work well with continuous actions. Some algorithms are off-policy.
- They have twice as many networks (although we can design a shared architecture) and the learning process suffers from instability like in Q-learning.

2.4 Planning and model-based methods

In control theory with known dynamics and finite horizon, planning refers to finding the best sequence of actions $u_1 ... u_T$ without learning a policy. If the dynamics are deterministic, i.e. $x_{t+1} = f(x_t, u_t)$ instead of $x_{t+1} \sim p(x_{t+1}|x_t, u_t)$, we can find the optimal sequence of actions with open-loop control (no temporal feedback). We usually refer to this problem as trajectory optimization or

optimal control:

$$
\begin{aligned}
&\text{shooting} && \min_{u_1 \dots u_T} \sum_{t=1}^{T} c(x_t, u_t) \quad \text{s.t.} \quad x_{t+1} = f(x_t, u_t) \\
&\text{collocation} && \min_{x_1, u_1 \dots x_T, u_T} \sum_{t=1}^{T} c(x_t, u_t) \quad \text{s.t.} \quad x_{t+1} = f(x_t, u_t) \\
&\text{direct collocation} && \min_{x_1 \dots x_T} \sum_{t=1}^{T} c(x_t, u_t) \quad \text{s.t.} \quad u_t = f^{-1}(x_t, x_{t+1})
\end{aligned}
\tag{2.29}
$$

Note that the *shooting* approach is actually an unconstrained problem. The main con is that it is very sensible to early actions, so it is poorly conditioned. We could instead optimize over the whole trajectory (states and actions), called *collocation*, which allows for local corrections but becomes a constrained problem. There is a trade-off between the implicit hard constraint of shooting and the soft constraint of collocation: a shooting algorithm provides a feasible set of controls during the entire optimization process, which ensures that the final output will be feasible but limits the exploration space and is more prone to local optima. If we have access to an inverse dynamics model we can opt for direct collocation instead. Let's look at some simple shooting algorithms:

- *Linear Quadratic Regulator* (LQR). If the dynamics are linear and deterministic and the cost is quadratic, dynamic programming provides a simple linear controller $u_t = K_t x_t + k_t$. If the dynamics are linear Gaussian, the same controller is still optimal. There is a version of this algorithm called iLQR (iterative LQR), which is very similar to another one known as DDP (differential dynamic programming), that works for general dynamics and cost function through local linear and quadratic approximations.
- *Cross-Entropy Method* (CEM). Under stochastic dynamics, we can use an open-loop stochastic optimization algorithm such as CEM. More recent extensions include CMAES, which stores information about previous updates, or PI^2-CMA (Stulp and Sigaud, 2012), a combination with PI^2 (Theodorou, Buchli, and Schaal, 2010) suited for parameterized policies.

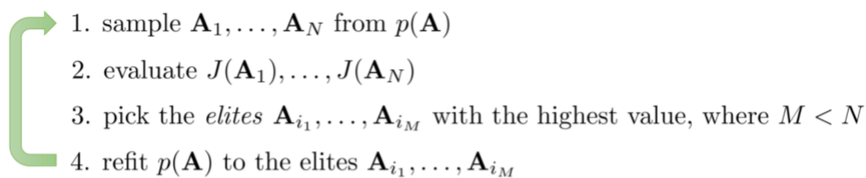


FIGURE 2.12: CEM algorithm (source: Levine, Fa2019)
$p(A)$ is usually chosen to be Gaussian.

- AICO (Toussaint, 2009), RRTs (Rapidly-exploring random trees), PRM (Probabilistic roadmap), etc.

If we have a discrete action space and a simulator fast enough to plan online, we can use re-planning algorithms like:

- *Rollout.* Run a base policy π on the simulator until the end of the episode (or until a particular horizon and then approximate the terminal cost) to obtain an approximation of the cost-to-go. Then, take the greedy action:

$$u_t = \arg\min_{u_t} \mathbb{E}_{x_{t+1} \sim p(x_{t+1}|x_t,u_t)} \left[c(x_t, u_t) + \hat{V}^{\pi}(x_{t+1}) \right] \tag{2.30}$$

  Monte-Carlo Tree Search (MCTS) is an algorithm based on rollouts which has known great success (e.g. AlphaGo Silver et al., 2016). It is an adaptive simulation algorithm that builds a tree over all possible controls and expands only the leafs that seem promising based on some pre-defined rules (e.g. upper confidence bound - UCB).

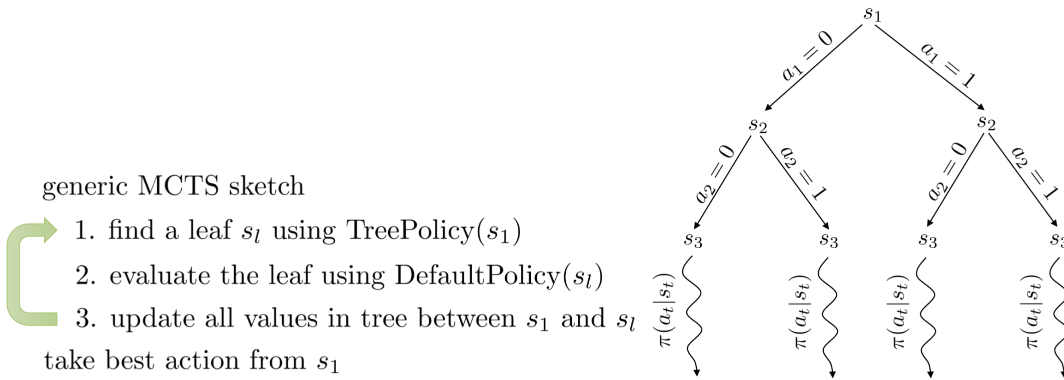


FIGURE 2.13: MCTS general algorithm (source: Levine, Fa2019)

- *Model Predictive Control (MPC).* At each time step, compute controls by solving an open-loop optimization problem for the prediction horizon l. Apply the first value of the computed control sequence and repeat.

$$u_t \in \arg\min_{u_t, \mu_{t+1} \cdots \mu_{t+l-1}} \mathbb{E}_{p(x'|x,u)} \left[c(x_t, u_t) + \sum_{t'=t+1}^{t+l-1} c(x_{t'}, \mu_{t'}(x_{t'})) \right] \tag{2.31}$$

  If we need to satisfy a set of constraints, we can just add them to the optimization above. We can also use rollouts to approximate the terminal cost $\hat{V}^{\pi}(x_{t+l})$.

2.4.1 Model-based reinforcement learning

We have seen some methods that exploit the model of the environment to find the optimal controls. Model-based RL aims at learning such a model so that we can apply similar solutions. What is the right representation for our model?

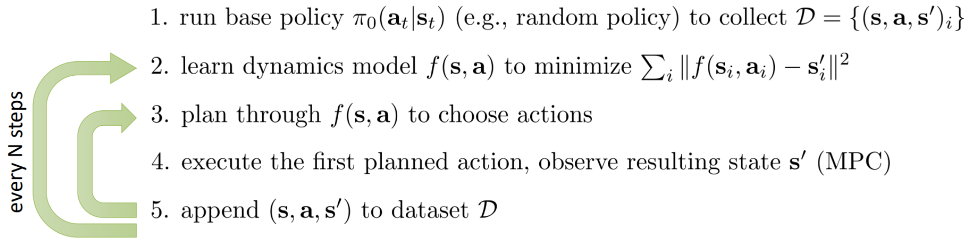


FIGURE 2.14: Basic model-based algorithm with MPC (source: Levine, Fa2019)

Two popular choices for models are Gaussian Processes (GP) and Neural Networks (NN). NNs are more expressive and can model better non-smooth dynamics if a lot of data is available. However, high-capacity models tend to over-fit because the planner will exploit the mistakes of the model, i.e. it will seek out regions where the model is erroneously optimistic. GPs mitigate this issue because they provide an estimate of the uncertainty on the model: the planner will make less mistakes because the expected reward of an erroneously optimistic region under high variance will now be smaller (e.g. PILCO: Deisenroth and Rasmussen, 2011). Note that we can also obtain an estimate of the uncertainty with NNs via Bayesian NNs or bootstrap ensembles (e.g. PETS: Chua et al., 2018; ME-TRPO Kurutach et al., 2018).

Model uncertainty. The action optimization process will seek out-of-distribution states where the output of the model is erroneously optimistic. This means that we don't care about the uncertainty that the model has on the data (e.g. the variance value of a NN that outputs a Gaussian distribution for the next state) because this uncertainty will also be flawed for out-of-distribution inputs. This type of uncertainty is called aleatoric or statistical. What we care about is epistemic or model uncertainty, i.e. the entropy of $p(\theta|\mathcal{D})$ where θ are the parameters of our model.
With bootstrap ensembles we have $p(\theta|\mathcal{D}) \approx \frac{1}{N}\sum_i \delta(\theta_i)$ so that $\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|\mathcal{D}) \approx \frac{1}{N}\sum_i p(s_{t+1}|s_t, a_t, \theta_i)$. Note that, in principle, we need to generate independent datasets to get independent models, by training each θ_i on $\mathcal{D}_i$ sampled with replacement from $\mathcal{D}$ (or by naively splitting $\mathcal{D}$ into smaller datasets). In practice for NNs, SGD optimization and random initialisation already provide independent models.

In general, for candidate action sequence $\mathbf{a}_1, \ldots, \mathbf{a}_H$:

Step 1: sample $\theta \sim p(\theta|\mathcal{D})$

Step 2: at each time step t, sample $\mathbf{s}_{t+1} \sim p_\theta(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$

Step 3: calculate $R = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$

Step 4: repeat steps 1 to 3 and accumulate the average reward

FIGURE 2.15: Basic planning procedure from a model with uncertainty (source: Levine, Fa2019)

2.4.2 Local models

Sometimes we might only need to estimate the dynamics on a local region around our current policy. Let's look at one such example: LQR with fitted linear models (LQR-FLM) (Levine and Abbeel, 2014), which attempts to learn local time-varying linear Gaussian (TVLG) dynamics, and a linear Gaussian controller with iLQR.

To fit the dynamics, the authors propose linear regression on the collected trajectories at each time-step, so that $p(x_{t+1}|x_t, u_t) = \mathcal{N}(f(x_t, u_t), N_t)$ where $f(x_t, u_t) = A_t x_t + B_t u_t + C_t$. A_t and B_t play the roles of the derivatives $\frac{df}{dx_t}$ and $\frac{df}{du_t}$ respectively, which are the quantities that we are missing (since we don't know the model) in order to run LQR directly.

Then, running iLQR gives the actions $u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t$, which becomes the controller $p(u_t|x_t) = \mathcal{N}(K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t, \Sigma_t)$ so that the trajectories that will be collected aren't too close to each other. A good choice for the variance is $\Sigma_t = Q^{-1}_{u_t,u_t}$, which will be high when the actions don't have a big impact on the reward-to-go (i.e. many different actions might lead to good rewards).

In order to avoid regions where the fitted dynamics are not a good estimate (since they are only valid locally) the new controller must stay close to the old one. They do so by imposing the constraint $D_{\text{KL}}(p(\tau)||\bar{p}(\tau)) < \epsilon$. If the trajectories induced by the new controller $p(u_t|x_t)$ and the old controller $\bar{p}(u_t|x_t)$ are close, then the dynamics will be close too.

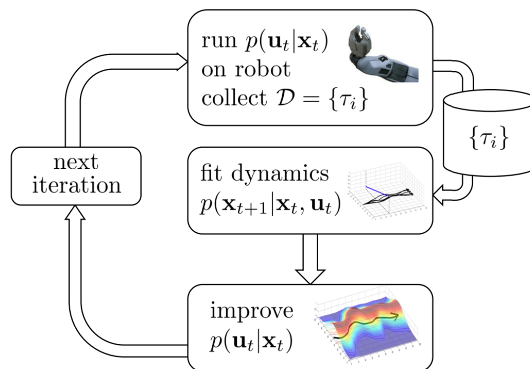


FIGURE 2.16: LQR-FLM algorithm (source: Levine, Fa2019)

2.4.3 Global models

Some examples include MuZero (Schrittwieser et al., 2020), Imagination-based Planner (Pascanu et al., 2017), and DreamerV2 (Hafner et al., 2020).

DreamerV2 and the newer version DreamerV3 (Hafner et al., 2023) learn a model over latent variables. An encoder turns each image into 32 distributions over 32 classes each, the meanings of which are determined automatically as the world model is learning. The one-hot vectors sampled from these distributions are concatenated to a sparse representation that is passed on to the recurrent

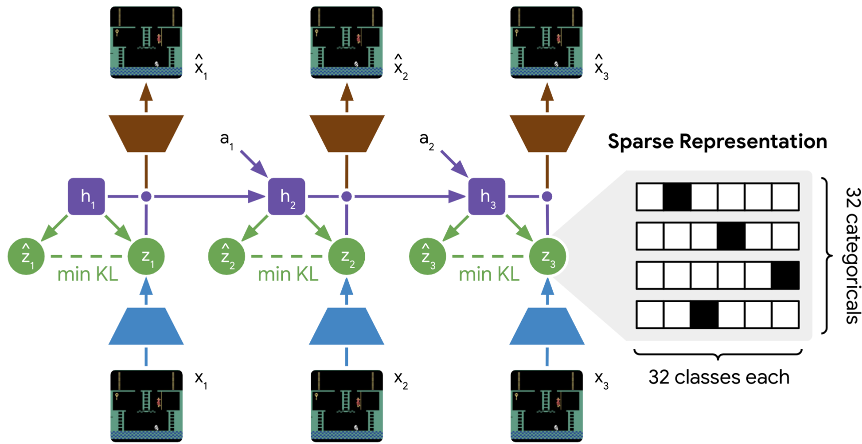


FIGURE 2.17: DreamerV2 model architecture (source: Hafner et al., 2020)

state. The world model loss function is the ELBO or variational free energy of a hidden Markov model that is conditioned on the action sequence. It encourages accurate reconstructions while keeping the stochastic representations (posteriors) close to their predictions (priors) to regularize the amount of information extracted from each image and facilitate generalization.

2.4.4 Model-based policy learning

We might want to adopt a closed-loop setting and learn a policy together with the model. Policies are much faster to execute at test-time and are better at generalization. Here are some ideas to do so:

- Back-propagate into the policy (e.g. PILCO: Deisenroth and Rasmussen, 2011):

$$\min_{\theta,\phi} \sum_{t=1}^{T} c(x_t, u_t) \quad \text{s.t.} \quad x_{t+1} = f_\phi(x_t, u_t) \quad \text{s.t.} \quad u_t = \pi_\theta(x_t) \tag{2.32}$$

For deterministic policy and dynamics, we can simply back-propagate through the dynamics and cost function to update the policy. We alternate solving for dynamics parameters (standard supervised regression) and solving for policy parameters (back-propagation through time). It can also work for stochastic policy and dynamics with a re-parameterization trick.

This problem is very ill-conditioned: the Hessian has some very large eigen-values (corresponding to parameters that affect actions early on) and some very small eigen-values (corresponding to parameters that affect actions towards the end). This translates into similar parameter sensitivity problems as shooting methods, but we can no longer use dynamic programming methods like LQR because θ couples all the time-steps. This is analagous to the vanishing/exploding gradients problem in RNNs, which is usually adressed by designing an architecture that simplifies the dynamics of the network, which we cannot do in

RL because the dynamics are imposed to us by the real data. This method is therefore only recommended for simple dynamics like linear dynamics.

- Guided policy search (GPS) (Levine et al., 2016):

$$\min_{\tau,\theta} c(\tau) \quad \text{s.t.} \quad u_t = \pi_\theta(x_t) \tag{2.33}$$

In order to apply the constraint $u_t = \pi_\theta(x_t)$ the authors use dual gradient descent, so the surrogate $\tilde{c}(\tau)$ is given by the Lagrangian or any other function that can incorporate such constraints. Step 1 (Figure 2.18) is optimal control, which we can solve however we want (shooting vs collocation, local vs global model). Step 2 is supervised learning, which can be interpreted as imitation of the optimal control "expert". As they iterate this process, the "expert" adapts to the learner by the constraint $u_t = \pi_\theta(x_t)$, avoiding actions that the learner can't mimic.

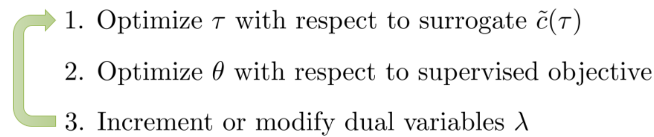
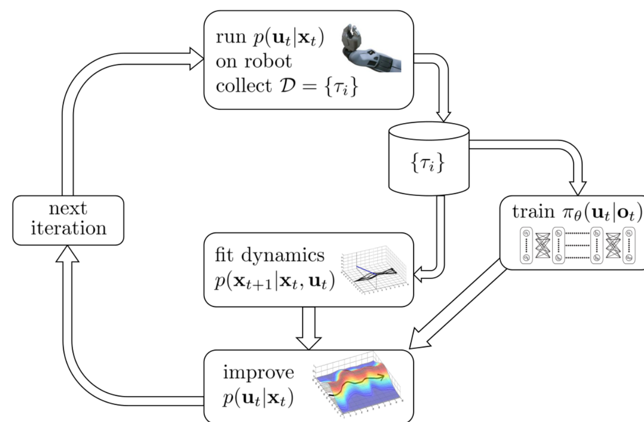


FIGURE 2.18: General guided policy search (source: Levine, Fa2019)

One concrete example is presented in (Levine and Koltun, 2013), where step 1 provides a set of controllers $\pi_{LQR,i}(u_t|x_t)$ with LQR-FLM. The guiding controllers are only valid around a single trajectory, so GPS can be seen as distilling a collection of trajectories into a general policy.



1. optimize each local policy $\pi_{\text{LQR},i}(\mathbf{u}_t|\mathbf{x}_t)$ on initial state $\mathbf{x}_{0,i}$ w.r.t. $\tilde{c}_{k,i}(\mathbf{x}_t, \mathbf{u}_t)$
2. use samples from step (1) to train $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ to mimic each $\pi_{\text{LQR},i}(\mathbf{u}_t|\mathbf{x}_t)$
3. update cost function $\tilde{c}_{k+1,i}(\mathbf{x}_t, \mathbf{u}_t) = c(\mathbf{x}_t, \mathbf{u}_t) + \lambda_{k+1,i} \log \pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$

FIGURE 2.19: Stochastic (Gaussian) GPS with LQR-FLM (source: Levine, Fa2019)

- DAgger (Ross, Gordon, and Bagnell, 2010) is an imitation learning algorithm. Instead of a human expert, we can imitate an optimal control planner to

learn our policy (e.g. PLATO: Kahn et al., 2017). DAgger doesn't require an expert that can adapt to the learner, but it assumes that it's possible to match the expert's behavior up to a bounded loss.

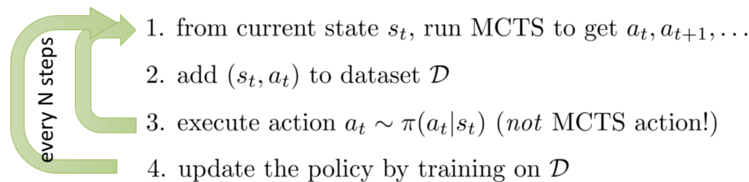
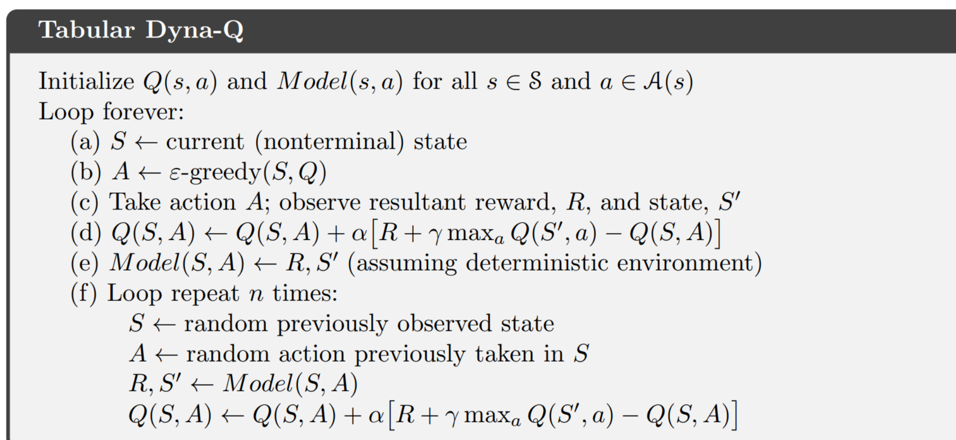


FIGURE 2.20: Imitating MCTS with DAgger (Guo et al., 2014) (source: Levine, Fa2019)

2.4.5 Model-free learning with a model

Another approach involves using any model-free algorithm while jointly learning a model. Some examples include Imagination-Augmented Agents (I2As) (Racanière et al., 2017) and PILQR (Chebotar et al., 2017).



Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S, Q)
(c) Take action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Loop repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

FIGURE 2.21: Original Dyna-Q algorithm (source: Sutton and Barto, 2011)

Dyna (Sutton, 1990). This algorithm has inspired many others (e.g. MBA: Gu et al., 2016b; MVE: Feinberg et al., 2018; MBPO: Janner et al., 2019) that follow the same idea: learn a model to simulate rollouts that are used as additional data to a model-free algorithm. The advantage over regular model-free algorithms is that we are getting additional samples by generating many short rollouts starting from random previously visited states. The advantage over fully model-based algorithms, which consider long rollouts from the model to plan ahead, is that we avoid potential compounding errors (the model is imperfect so the error accumulates). This is usually referred to as *model-based acceleration*.

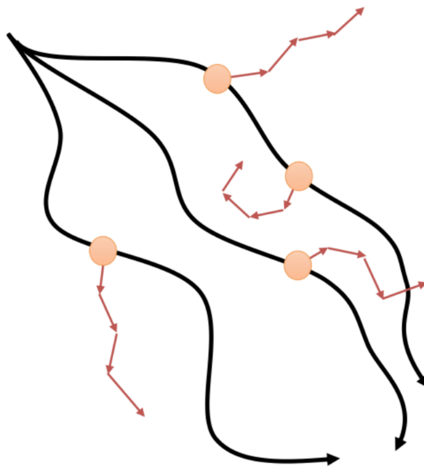


FIGURE 2.22: Model-based acceleration: generate short simulated rollouts from previously visited states (source: Levine, Fa2019)

2.4.6 Successor representation

In model-based reinforcement learning, models predict immediate next states, are trained on a prediction problem with a horizon of one, deal with the long-horizon nature of the prediction task during testing, and face the challenge of compounding model prediction errors at test-time. On the flip side, in model-free reinforcement learning, value functions predict long-term sums of rewards, are trained with either Monte Carlo estimates of expected cumulative reward or with dynamic programming, deal with the long-horizon nature of the prediction task during training, and face the challenge of bootstrap error accumulation during training.

Successor representation (Dayan, 1993) is a hybrid approach that learns a model able to make long-horizon state predictions by virtue of training-time amortization. Instead of approximating $p(s_{t+1}|s_t, a_t)$, we wish to approximate the discounted occupancy $d_\gamma^\pi(s|s_t, a_t) = (1-\gamma)\sum_{\Delta t=1}^{\infty} \gamma^{\Delta t-1} p(s_{t+\Delta t} = s|\pi, s_t, a_t)$.

In the discrete state space case, we can learn such a model M with a Bellman-like vector equation (where $\mathbf{1}$ is a one-hot indicator vector):

$$M(s_t, a_t) \leftarrow \mathbb{E}_{s_{t+1}}[\mathbf{1}(s_{t+1}) + \gamma M(s_{t+1})] \tag{2.34}$$

More recent works exist for the continuous-state space, such as (Janner, Mordatch, and Levine, 2020).

2.4.7 Summary

- Learning a model can be a good idea if the interactions with the real environment are expensive or we want to generalize to other tasks in the same environment.

- Learning a model can be harder than learning a policy for some tasks and environments. Also, the right representation isn't always obvious depending on the problem and the algorithm that we want to use.

2.5 Which algorithm to use?

There are many factors that might guide the choice of an algorithm: discrete/continuous actions, ability to generalize, safety concerns, etc. In chapter 4 we will see in more detail which algorithms have been successfully implemented for robotics.

Sample efficiency. How many samples, i.e. interactions with the environment, do we need to converge? Note that sample efficiency might differ from wall-clock speed depending on our hardware (sample inefficient methods like gradient-free algorithms usually scale very well), our data collection procedure (in a fast simulator vs in a slow real system), the training speed of our algorithm, etc.

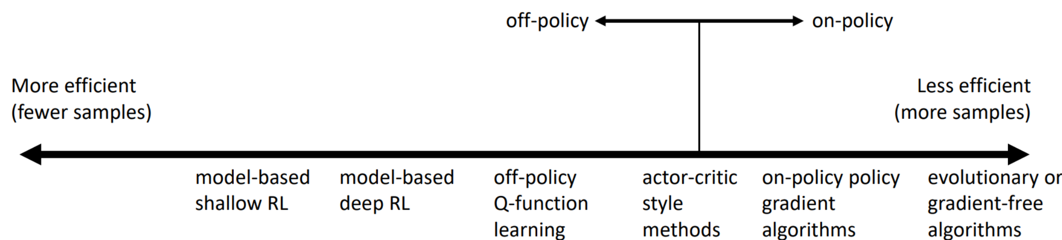


Figure 2.23: RL algorithms by sample efficiency (source: Levine, Fa2019)

2.6 Exploration

A key concept in RL is the idea of exploration. Do we look for new strategies to solve the task or do we stick to the best one found so far? Off-policy methods can use any explorative strategy to collect data. In on-policy methods, exploration must be built into the policy we are learning and determines the rate of the policy improvements.

In some cases a simple strategy can provide good enough exploration. The ϵ-greedy policy (left) is greedy w.r.t Q-function with probability $1-\epsilon$ and random otherwise. The Boltzmann policy (right) is similar: it approaches the greedy policy as the temperature α decreases.

$$\pi(a|s) = \begin{cases} 1-\epsilon & \text{if } \ a = \arg\max_a Q^\pi(s,a) \\ \epsilon/(|A|-1) & \text{otherwise} \end{cases} \qquad \pi(a|s) \propto \exp(\frac{1}{\alpha}Q^\pi(s,a))$$

For continuous-action spaces, the simplest exploration method is to use action noise, which adds small random perturbations to the policy's actions. Most algorithms use white noise (explicitly for deterministic policies like in DDPG or TD3, or implicitly via sampling actions from the stochastic policy like in SAC

		On/Off policy	Action space	Neural Network	known Dynamics
Dynamic Programming	VI PI	-	discrete	-	yes
approximate Dynamic Programming	SARSA	On	discrete	-	no
	Q-learning	Off			
	Dyna-Q	Off			
deep Q-learning	DQN	Off	discrete	Q	no
Policy gradient	REINFORCE TRPO PPO	On	both	π	no
Actor-Critic	A3C	On	both	Q π	no
	DDPG	Off	continuous	Q π	
	SAC	Off	continuous	Q V π	
Optimal Control	iLQR CEM	-	continuous	-	yes
	MPC MCTS		discrete		
Model-based reinforcement learning	PETS	-	continuous	p	no
	PILCO	On	continuous	p π	
	GPS	Off	continuous	p π	

TABLE 2.1: Overview of RL algorithms
Note that for PILCO the model is a GP, and for GPS it is a TVLG model, not NNs.

or MPO), which is uncorrelated over time, or OrnsteinUhlenbeck (OU) noise, which is strongly correlated. OU noise produces better exploration, but it also produces strongly off-policy trajectories (see distribution mismatch discussion in 2.2.4), so (Eberhard et al., 2022) explore noises with intermediate temporal correlation (pink noise) that handle this trade-off better.

Many other different approaches have been proposed over the years. For instance, NoisyNet (Fortunato et al., 2017) adds parametric noise to the weights of the Q-function and/or policy networks, and AGAC (Flet-Berliac et al., 2021) incentivizes the policy to differentiate itself from an adversary that is trying to mimic it. Let's look at three different families in more detail.

Multi-arm bandits

Exploration vs exploitation is a well studied problem in the field of multi-arm bandits. Here are some exploration strategies derived from bandits:

- Optimistic exploration: we assume the unknown to be good. A simple idea is to consider count-based exploration bonuses to create a new reward $r^+(s,a) = r(s,a) + B(N(s))$, where B is some bonus decreasing function and $N(s)$ counts the visits to state s. A very popular bandit bonus is UCB: $B(N) = \sqrt{\frac{2 \ln n}{N}}$, where $n = \sum_s N(s)$. If the space is too large or continuous we can learn a density model instead $p_\theta(s) \approx N(s)/n$. For

instance, $p_\theta(s) \propto N(f_\theta(s))$ where f_θ embeds the states into a smaller space (Tang et al., 2017).

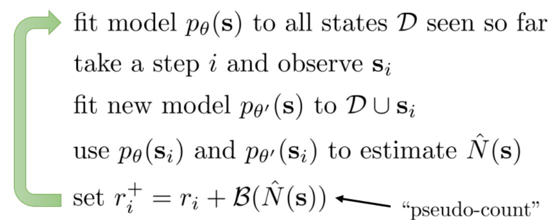


FIGURE 2.24: Exploring with pseudo-counts (Bellemare et al., 2016) (source: Levine, Fa2019)

- Posterior sampling. Inspired from bandit's Thompson sampling, we sample from a prior distribution over Q-functions or policies and act according to the sample for one episode, then update the prior. Exploring with randomized actions like ϵ-greedy can oscillate back and forth, but exploring with a randomized Q-function can reveal more sophisticated strategies because the exploration during the entire episode is consistent.
- Approximate information gain $IG(z, y) = D_{\text{KL}}(p(z|y)||p(z))$ methods try to measure how much we learn about z from the quantity y. Let $z = \theta$. How much we learn about the dynamics $p_\theta(s'|s, a)$ given the new transition $y = (s, a, s')$? In VIME (Houthooft et al., 2016), the authors consider that a transition is more informative if it causes belief over θ to change. How much we learn about the state density $p_\theta(s)$ given the new state $y = s'$? If the density changed a lot (e.g. $\log p_\theta(s) - \log p_{\theta_{\text{old}}}(s)$ big), the state was novel.

Prediction errors

A more broad idea is to use any type of prediction error, i.e. train a model to make some type of prediction and build an exploration bonus based on the model error. The prediction error can quantify the novelty of new experience because it should be high where few similar examples have been seen so far.

For instance, (Pathak et al., 2017) encode image observations s_t into $\phi(s_t)$ using an auto-encoder, and train another network to predict $\phi(s_{t+1})$ from $(\phi(s_t), a_t)$. The error w.r.t. the encoding of the actual new state s_{t+1} is used as an exploration bonus.

In Random Network Distillation (RND) (Burda et al., 2018), the authors realize that the prediction error might also be high for reasons that do not characterize novelty, like stochasticity of the target function (trying to predict the ouptut of a stochastic function such as $p(s'|s, a)$ might provide the agent with endless novelty if the function is "too random") or model misspecification (prediction error is high because the predictor's model can't fit the complexity of the target). Their solution is to predict the output of a fixed randomly initialized neural network on the current observation: the target network can be chosen to be deterministic and inside the model-class of the predictor network.

Auxiliary losses

These exploration bonuses are addressing a deeper problem in RL: the reward function is not enough/good supervision. (Shelhamer et al., 2016) presents a variety of loss functions that mine further supervision from the same data available to existing RL methods: the transitions (s, a, r, s'). These self-supervised losses are simple proxies of the reward (e.g. predict the sign of the reward), the dynamics (or inverse dynamics), or based on reconstruction errors (e.g. autoencoders), and can reduce the burden of exploration. The challenge of these approaches, just as any exploration bonus, is that the separate sources of supervision must not conflict with each other. UNREAL (Jaderberg et al., 2016) and MERL (Flet-Berliac and Preux, 2019) also introduce auxiliary training objectives.

2.7 Maximum-entropy reinforcement learning

Exploration is closely related to a more general concept: stochasticity. Most algorithms covered so far, like Q-learning, try to find the optimal policy which is always deterministic under full observability. But stochastic policies have interesting properties: better exploration and robustness (due to wider coverage of states). Also, stochastic policies try to learn all ways of performing the task (not just the better one), which can help model human behaviour (see section 3.2.2) and provides better policies to finetune from for specific environments. Another interesting property is shown in (Haarnoja et al., 2018a), where they obtain a decent policy for a complex task (e.g. move to the target and avoid the obstacle) by simply adding the Q-functions of the stochastic policies trained on the simpler tasks (e.g. two policies here, one that moves to the target and one that avoids obstacles).

How to implement entropy regularization? For instance, MICARL (Della Vecchia et al., 2022) provides a closed form for an entropy-regularized policy-iteration update, provided one can keep track of all previously estimated Q-functions. Let's focus on a family of methods that are derived from the following objective, designed to promote stochasticity by maximizing the entropy of the policy (we omit γ here for simplicity):

$$\pi^* = \arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim p_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right] \tag{2.35}$$

We can define new soft value functions. They are referred to as "soft" because they use a LogSumExp softmax instead of the max operator of typical value functions. Note that if π is equal to $\tilde{\pi}$, where $\tilde{\pi}(a|s) \propto \exp(\frac{1}{\alpha} Q^{\pi}_{\text{soft}}(s, a))$ is an energy-based policy, then $\frac{1}{\alpha} V^{\pi}_{\text{soft}}(s)$ is the log-partition function of the negative

energy function $\frac{1}{\alpha}Q^{\pi}_{\text{soft}}(s,a)$.

$$\begin{aligned}
Q^{\pi}_{\text{soft}}(s_t,a_t) &= r_t + \mathbb{E}_{\tau\sim p_\pi}\left[\sum_{l=1}^{T}\gamma^l(r_{t+l} - \alpha\log\pi(a_{t+l}|s_{t+l}))\right] \\
&= r_t + \gamma\mathbb{E}_{s_{t+1}\sim p(\cdot|s_t,a_t)}\left[V^{\pi}_{\text{soft}}(s_{t+1})\right] \\
V^{\pi}_{\text{soft}}(s) &= \mathbb{E}_{a\sim\pi(\cdot|s)}\left[Q^{\pi}_{\text{soft}}(s,a) - \alpha\log\pi(a|s)\right] \\
&= \alpha\log\int\exp(\frac{1}{\alpha}Q^{\pi}_{\text{soft}}(s,a))da - \alpha D_{\text{KL}}(\pi(\cdot|s)||\tilde{\pi}(\cdot|s))
\end{aligned} \tag{2.36}$$

Let's look at two Q-learning algorithms that use these soft value functions:

- *Soft policy iteration.* We evaluate the policy with Q^{π}_{soft} by iterating the soft Bellman equation until convergence:

  $$Q^{\pi}_{\text{soft}}(s,a) \leftarrow r + \gamma\mathbb{E}_{s'\sim p(\cdot|s,a),a'\sim\pi(\cdot|s')}\left[Q^{\pi}_{\text{soft}}(s',a') - \alpha\log\pi(a'|s')\right] \tag{2.37}$$

  The policy improvement step consists on $\pi_{\text{new}} = \arg\min_\pi D_{\text{KL}}(\pi||\tilde{\pi}_{\text{old}})$.

  Soft Actor-Critic (SAC) (Haarnoja et al., 2018b) translates this idea into an actor-critic architecture with function approximations for policy and soft value functions.

- *Soft Q-iteration.* We aim for $\pi^*(a|s) \propto \exp(\frac{1}{\alpha}(Q^*_{\text{soft}}(s,a)) = \exp(\frac{1}{\alpha}(Q^*_{\text{soft}}(s,a) - V^*_{\text{soft}}(s)))$ by iterating the optimal soft Bellman equation until convergence:

  $$Q^*_{\text{soft}}(s,a) \leftarrow r + \gamma\mathbb{E}_{s'\sim p(\cdot|s,a)}\left[\alpha\log\int\exp(\frac{1}{\alpha}Q^*_{\text{soft}}(s',a'))da'\right] \tag{2.38}$$

  Soft Q-learning (Haarnoja et al., 2017) builds upon this idea. One problem when dealing with energy-based policies is that they are intractable because of the partition function which involves an integral over the action space. Since soft Q-learning requires us to sample from such a policy (to take on-policy actions and to estimate V^*_{soft}), the authors implement a stochastic sampling network trained to output approximate samples from the target distribution. Like SAC, it can handle both discrete and continuous action spaces.

Note that although the Boltzmann explorative policy looks similar, it only greedily maximizes entropy at the current time step, and doesn't maximize the entropy of the entire trajectory distribution like the methods shown in this section.

Chapter 3

State of the art - Imitation Learning

One of the main practical challenge in RL is defining the reward function. In controlled environments like games, the reward usually comes up very naturally. But what about real systems like robots? A sparse reward is easy to design but very sample inefficient, even if we supervise the agent with other signals (e.g. exploration bonuses). Manually designing a more complex reward will work better, but *reward shaping* is far from easy and very task-specific. In *imitation learning* we attempt to leverage expert demonstrations as a source of useful supervision for the agent. Let's look at two examples that are similar to the algorithms presented in the previous chapter, but rely on demonstrations to compensate for the lack of feedback provided by the reward signal.

Soft Q Imitation Learning (SQIL) (Reddy, Dragan, and Levine, 2019) is derived from soft Q-learning. The replay buffer is initially filled with demonstrations where the rewards are always $r = 1$. New experiences collected by the agent are added with reward $r = 0$. For learning, the sampled experiences are balanced (50% each). Intuitively, these modifications create a simple reward structure that gives the agent an incentive to imitate the expert in demonstrated states, and to take actions that lead it back to demonstrated states when it strays from the demonstrations. The paper shows theoretical connections between SQIL and regularized behaviour cloning.

Recursive Classification of Examples (RCE) (Eysenbach, Levine, and Salakhutdinov, 2021) presents a framework similar to goal-conditioned RL, but the aim is to learn a policy for solving one task, rather than a goal-conditioned policy. It only requires success images, not full demonstrations. Let a random variable e_t indicate whether the task is solved at time t. Given a policy, the (discounted) probability of solving the task at a future step is $p_\pi(e_{t+}|s_t, a_t)$. The objective is equivalent to the standard RL objective with the unkown reward function $r(s_t, a_t) = p(e_t = 1|s_t)$. The challenge is to learn a classifier $C_\theta^\pi(s_t, a_t)$ in a positive-unlabeled setting. To do so, the idea is to use the classifier's own predictions (at the next time step) as labels for the classifier.

Other than RL-inspired algorithms, there is a wide scope of imitation-learning algorithms. For instance, (Muelling et al., 2012) use demonstrations to

create a library of table tennis movements from kinesthetic data. In this chapter we will look in more detail at three major families. In behaviour cloning we directly look for a policy that acts like the expert by solving a supervised-learning problem. In inverse reinforcement learning (IRL) we try to infer the reward function that the expert was most likely trying to maximize, while optionally jointly learning a policy. In offline RL, agents are trained from datasets collected beforehand, without additional online data collection.

Note that demonstrations can also be used alongside a reward signal. Indeed, demonstrations can be used in RL to design an additional reward, guide exploration, augment the training data, initialize policies, etc. We will focus more on these hybrid approaches in Chapter 5.

3.1 Behaviour Cloning

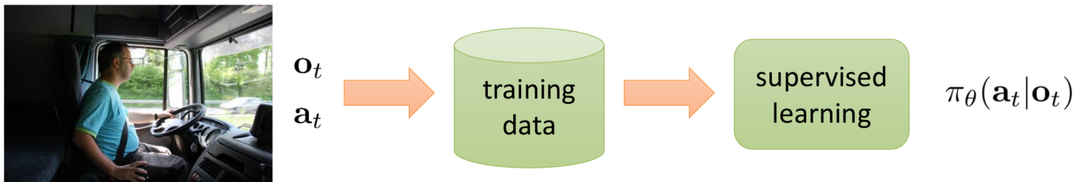


FIGURE 3.1: Learning from demonstrations with supervised learning (source: Levine, Fa2019)

Imitation learning as supervised learning: we train a network to match the actions of the expert (assumed i.i.d.) conditioned on the corresponding observation $\pi_\theta(a_t|o_t)$.

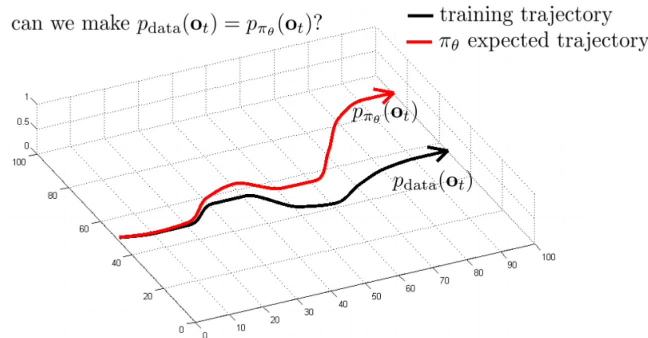


FIGURE 3.2: Compounding errors in behaviour cloning (source: Levine, Fa2019)

The main challenge of behaviour cloning are compounding errors. When the policy makes a small mistake, it deviates from the state distribution seen during training, making it more likely to make a mistake again. Eventually, the agent is no longer able to recover. A simple idea to correct this is to present the agent with data from unwanted states labelled with the actions to correct them (e.g. turn inwards if we are about to exit the road: Bojarski et al., 2016). Another example is presented in (Zhou et al., 2023), where they sample perturbations

around expert robotic poses, calculate the corrective actions that would stabilize the trajectories, and render observations for the perturbed poses with a neural radiance field (NeRF).

A more robust idea is presented in DAgger (Ross, Gordon, and Bagnell, 2010): have an interactive expert tell us what we did wrong. Supervised learning works under the assumption that training and test data distributions match. Therefore, the goal is to collect data from our current policy π_θ to reduce the distribution mismatch between p_{π_θ} and p_{data}.

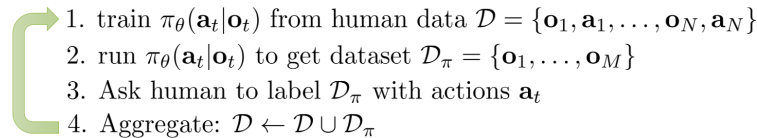


FIGURE 3.3: DAgger: Dataset Aggregation algorithm (source: Levine, Fa2019)

Of course, DAgger requires human feedback during training which isn't very practical. What if we had a model so good that it didn't actually drift in practice? These are the two main challenges and some ideas to overcome them:

- *Multi-modal behaviour.* Human behaviour is often multi-modal. For instance, we can avoid obstacles on the road by going left or right. If our policy model does not have enough capacity to represent the human distribution it will learn a distribution which "covers" all of the modes, which will also cover unwanted parts of the space corresponding to sub-optimal behaviour. With discrete actions, a softmax output and cross-entropy loss can represent any complex multi-modal distribution. What about continuous actions? We could use an autoregressive discretization network, which discretizes one dimension of the action space at a time (to avoid the curse of dimensionality). We could also learn a latent variable model (e.g. VAE), which are more expressive but are tricky to train. A simpler idea if we know the number of modes is to output a mixture of Gaussians.
- *Non-Markovian behavior.* Human behaviour is often non-Markovian, depending on several previous observations. Recurrent neural networks (RNNs) allow to condition the policy on all previous inputs $\pi_\theta(a_t|o_1...o_t)$, but present additional training challenges.

3.2 Inverse Reinforcement Learning

In behaviour cloning we attempt to directly mimic the expert. But what if the expert has different capabilities? Can we reason about what the expert is trying to achieve instead? In the IRL setting we assume that the reward function is more succinct than the optimal policy. We consider an MDP with unknown reward, which we need to recover from a dataset of expert demonstrations. We assume that these samples come from an optimal policy π_E. Our goal is to find

the reward r^* such that:

$$E_{\tau\sim\pi_E}\left[r^*(\tau)\right] \geq \mathbb{E}_{\tau\sim\pi}\left[r^*(\tau)\right] \quad \forall\pi \tag{3.1}$$

3.2.1 Linear reward

Let's consider a reward r_ψ which is represented as a linear combination of features f:

$$\mathbb{E}_{\tau\sim\pi}\left[r_\psi(\tau)\right] = \mathbb{E}_{\tau\sim\pi}\left[\sum_t \gamma^t \psi^T f(s_t, a_t)\right] = \psi^T \mathbb{E}_{\tau\sim\pi}\left[\sum_t \gamma^t f(s_t, a_t)\right] = \psi^T f(\pi)$$

We assume that the expert is solving an MDP with unknown reward $r^*(s,a) = {\psi^*}^T f(s,a)$. The advantage of the linear representation is that we can approximate the left-hand term in 3.1 with samples to build an estimator $\hat{f}(\pi_E)$. We want to find ψ such that the expert policy outperforms other policies:

$$\psi^T f(\pi_E) \geq \psi^T f(\pi) \quad \forall\pi \tag{3.2}$$

Maximum margin. In order to reduce the ambiguity of the reward (e.g. $\psi = 0$ is a solution) we can solve a maximum margin problem: pick the reward for which the expert is better than anybody else by as big of a gap as possible (equation 3.3). The margin can depend on the distance of the policy to the expert (e.g. the number of states where π_E was observed and disagrees with π). We could also relax the assumption of expert optimality by introducing slack variables like in SVMs to account for mistakes (Ratliff, Bagnell, and Zinkevich, 2006).

$$\min_\psi \frac{1}{2}||\psi||^2 \quad \text{s.t.} \quad \psi^T f(\pi_E) \geq \max_\pi \psi^T f(\pi) + d(\pi, \pi_E) \tag{3.3}$$

Feature matching. The Apprenticeship Learning (Abbeel and Ng, 2004) algorithm doesn't guarantee to recover ψ^*, but finds a policy π^{ψ^*} optimal with respect to the unknown reward r_{ψ^*}. To do so, the authors prove that it is both necessary and sufficient to match the feature expectation of the expert's policy, i.e. find ψ such that $||f(\pi_E) - f(\pi^\psi)||^2 \leq \epsilon$. For instance, if we are driving a car we would like our policy to have same average "speed" and average "number of collisions" than the expert. The algorithm starts with π_0 and iterates over i the following two steps:

1. "Guess" the reward function: find ψ_i such that the teacher outperforms all previous policies $(\pi_k)_{0\leq k<i}$ by the maximum margin m.
2. Find optimal control policy $\pi_i = \pi^{\psi_i}$ for the current guess of the reward function ψ_i. If $m \leq \frac{\epsilon}{2}$ exit the algorithm.

If we have enough expert samples so that $||\hat{f}(\pi_E) - f(\pi_E)||^2 \leq \frac{\epsilon}{2}$, the last returned policy matches the expert's feature expectation up to ϵ. We can account for expert sub-optimality errors by solving an optimization problem on the returned policies (see paper) or by manually testing them and choosing the best

one.
This method is ambiguous: there is an infinite number of reward functions with the same optimal policy, and an infinite number of stochastic policies that can match feature expectations.

Unfortunately, these methods are still ambiguous and don't handle sub-optimal behaviour in a principled way.

3.2.2 Maximum entropy inverse reinforcement learning

How can we model sub-optimal behaviour? One idea is to use a probabilistic model of the expert. (Ziebart et al., 2008) propose to follow the principle of maximum entropy, which resolves the ambiguity by choosing among the candidates the distribution that does not exhibit any additional preference. It turns out that this is equivalent to choosing the following path distribution:

$$p(\tau|\psi) \propto p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) \exp(r_\psi(\tau)) \tag{3.4}$$

We can adopt this model of the expert for any reward r_ψ, linear or not. Under this model, the expert takes paths with equivalent rewards with same probability, and prefers (exponentially) paths with higher rewards. Let π^ψ the corresponding policy, which is the optimal maximum-entropy policy under reward r_ψ.

We look for ψ^* by maximizing the likelihood $L(\psi)$ of the expert data $(\tau_i)_i$:

$$L(\psi) = \frac{1}{N} \sum_i \log p(\tau_i|\psi) = \frac{1}{N} \sum_i r_\psi(\tau_i) - \log Z(\psi) \tag{3.5}$$

$$Z(\psi) = \int p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) e^{r_\psi(s_t,a_t)} d\tau \tag{3.6}$$

$$\nabla_\psi L(\psi) = \frac{1}{N} \sum_i \nabla_\psi r_\psi(\tau_i) - \mathbb{E}_{\tau \sim p(\tau|\psi)} \left[\nabla_\psi r_\psi(\tau)\right] \tag{3.7}$$

In the tabular case and under known dynamics, the second term in 3.7 is tractable and can be computed with a dynamic programming algorithm that recovers the state-action probability of π^ψ to compute $\sum_s \sum_a \sum_t p(s_t = s, a_t = a|\psi) \nabla_\psi r_\psi(s, a)$. See (Ziebart et al., 2008) for the details in the linear reward case. See (Wulfmeier, Ondruska, and Posner, 2015) for a similar method with a neural network reward representation that works for continuous spaces (but still requires known dynamics).

3.2.3 Deep inverse reinforcement learning

We want algorithms that can handle large and continuous state and action spaces and unknown dynamics.

Let's write down a general problem formulation that sums up what we have seen so far and that is adopted by many recent papers:

$$\arg\max_c -\phi(c) + (\min_\pi -\mathcal{H}(\pi) + \mathbb{E}_\pi [c(s,a)]) - \mathbb{E}_{\pi_E} [c(s,a)] \tag{3.8}$$

We recall that the cost c is the opposite of the reward, and ϕ here refers to a convex cost function regularizer (e.g. l_2 norm of the cost parameters). The optimal policy for the corresponding cost is given by:

$$\arg\min_\pi -\mathcal{H}(\pi) + \phi^*(\rho^\pi - \rho^{\pi_E}) \tag{3.9}$$

where ϕ^* is the convex conjugate of ϕ and $\rho^\pi(s,a)$ is the occupancy measure, i.e. the action-state stationary distribution $\rho^\pi(s,a) = \pi(a|s)d^\pi(s)$ where $d^\pi(s) = \lim_t p(s_t = s|\pi)$ measures the fraction of time the agent spends in state s. Note that a constant regularizer leads to an imitation learning algorithm that exactly matches occupancy measures (similar idea to the feature matching algorithm seen above).

Guided cost learning (GCL) (Finn, Levine, and Abbeel, 2016). Let's go back to the objective in 3.5. One idea is to learn π^ψ by using any max-ent RL algorithm (see section 2.7), then approximate the second term of the gradient $\nabla_\psi L(\psi)$ with samples. In practice, they don't learn π^ψ until convergence because it would be too expensive and ψ changes with every gradient update. Instead, they consider a policy π_θ that they improve a little (towards π^ψ) in each iteration. They use importance sampling to account for this approximation:

$$\nabla_\psi L(\psi) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j \omega_j}\sum_{j=1}^{M} \omega_j \nabla_\psi r_\psi(\tau_j) \quad \text{with} \quad \omega_j = \frac{\exp r_\psi(\tau_j)}{\pi_\theta(\tau_j)} \tag{3.10}$$

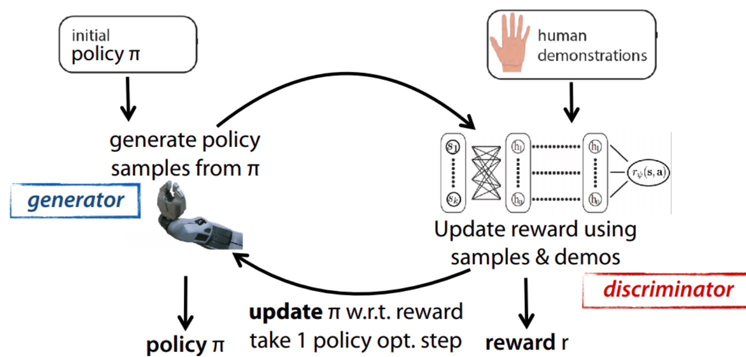


FIGURE 3.4: Guided cost learning algorithm (source: Levine, Fa2019)

As shown in figure 3.4, GCL presents an adversarial mechanism similar to GANs: the discriminator/reward is updated to make the demos more likely than the samples, and the generator/policy is updated to produce samples that are harder to distinguish from the demos. The authors follow this idea with AIRL (Finn et al., 2016) and consider a discriminator $D_\psi(\tau)$ that outputs the

probability that τ comes from the expert π_E. The goal is to minimize the following loss (both terms can be approximated with samples):

$$L_{\text{discriminator}}(\psi) = -\mathbb{E}_{\tau\sim\pi_E}\left[\log D_\psi(\tau)\right] - \mathbb{E}_{\tau\sim\pi_\theta}\left[\log(1 - D_\psi(\tau))\right] \tag{3.11}$$

The optimal discriminator for any GAN is given by $D^*(\tau) = \frac{\pi_E(\tau)}{\pi_\theta(\tau)+\pi_E(\tau)}$. Of course, we don't know $\pi_E = \pi^{\psi^*}$, but the authors replace it with the current π^ψ that the generator π_θ is trying to approach. This choice for the discriminator is one of the main differences with a normal GAN. The other one is that we can't back-propagate through the discriminator into the generator (we could under known dynamics), so they improve the policy as before with any max-ent policy optimization algorithm, giving $D_\psi(\tau) = \frac{\frac{1}{Z}\exp(r_\psi(\tau))}{\prod_t \pi_\theta(a_t|s_t)+\frac{1}{Z}\exp(r_\psi(\tau))}$.

Generative Adversarial Imitation Learning (GAIL) (Ho and Ermon, 2016). Similar idea to adversarial GCL: find a policy that makes it impossible for a discriminator to distinguish between samples from the expert and samples from the imitator agent. The main difference is that the reward is not explicitly represented so the discriminator D can be any binary classifier, which makes the optimization process simpler but doesn't recover the optimal reward. The generator/policy is optimized with a max-entropy TRPO step under the current reward $r(s,a) = -\log D(s,a)$, which will guide the policy towards expert-like regions (as classified by D). See the paper for the theoretical derivation of their algorithm, which starts with a particular choice for the regularizer ϕ that yields the final objective:

$$\min_\pi \max_D \mathbb{E}_\pi\left[\log D(s,a)\right] + \mathbb{E}_{\pi_E}\left[\log(1 - D(s,a))\right] - \lambda\mathcal{H}(\pi) \tag{3.12}$$

There have been a number of improvements brought to this algorithm. For instance, InfoGAIL (Li, Song, and Ermon, 2017) (based on InfoGAN: Chen et al., 2016) assumes that the expert policy is a mixture of experts and considers a discrete latent variable c that selects one of these policies, and AGAIL (Sun and Ma, 2019) can handle demonstrations with incomplete action sequences. PU-GAIL (positive unlabeled) (Xu and Denil, 2019) addresses the problem of over-fitting (GAIL manages to distinguish between agent and expert based on non-important features like lightning) by learning classifiers from positive (expert) data and unlabeled (agent) data, and also addresses the more general problem of delusion in reward supervised learning (wrongfully over-optimistic reward predictions are exploited by the agent).

3.2.4 Summary

- IRL requires fewer demonstrations than behavioral cloning and avoids supervised learning issues such as compounding errors.
- Current methods that can handle complex behaviours in real systems are based on adversarial optimization, which is hard to train. So far, only successfully tested on first-person non-visual demos (kinesthetic teaching, teleoperation).

3.3 Offline Reinforcement Learning

The main breakthroughs in machine learning have come from supervised learning, where large pools of data are used to make predictions. What if RL algorithms could effectively use large datasets as well? Offline RL (Levine et al., 2020), also known as data-driven RL, proposes a paradigm where the training leverages datasets collected beforehand, without additional online data collection. It provides a way to learn from any previously collected data D: reinforcement learning experiments, expert demonstrations, random interactions, hand-crafted policies...

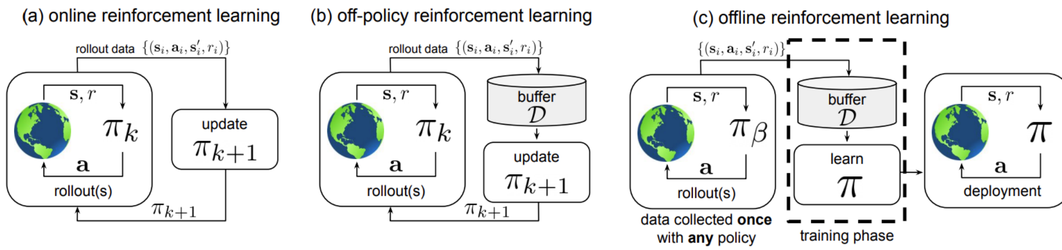


Figure 3.5: Offline RL (source: Levine et al., 2020)
In contrast with classic RL, the dataset $\mathcal{D}$ is collected once by some (potentially unknown) behavior policy π_β, and is not altered during training.

Note that learning from a fixed dataset without online environment interaction is difficult even if provided expert actions. For behaviour cloning we have $l(\pi) = \mathbb{E}_\pi \sum_{t=0}^{T} [\delta(a_t \neq a_t^*)] \leq C + T^2\epsilon$ where ϵ is the generalization error on demo data. For DAgger (online interaction allowed) the bound becomes $l(\pi) \leq C + T\epsilon$.

Note that exploration is no longer an algorithmic problem in offline RL since there is no way of collecting new data, so we must assume that D adequately covers the space of high-reward transitions to make learning feasible. However, although the idea is to not rely on online data, it is still possible and beneficial to fine-tune the offline policy on a smaller task-specific dataset. For instance, Figure 3.6 shows results of fine-tuning the algorithm QT-OPT for robotic grasping.

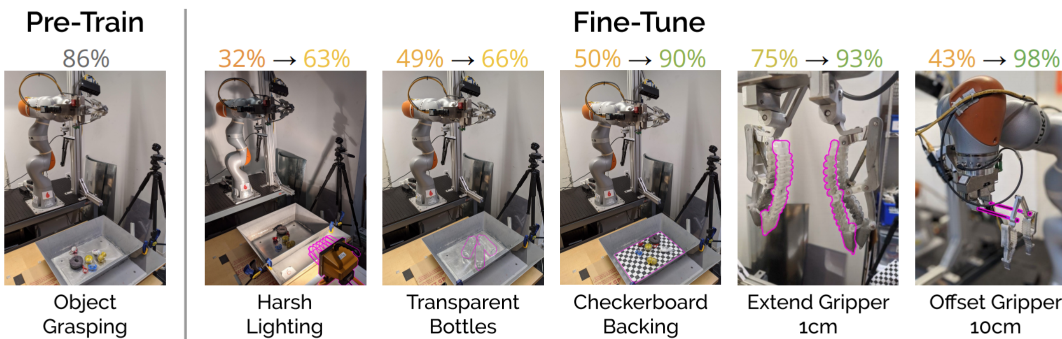


Figure 3.6: QT-OPT fine-tuning (source: Julian et al., 2020)

The main challenge is distributional shift, which affects offline RL at test time (for all algorithms) and at training time (for dynamic programming and

model-based algorithms). The distributional shift problem at test-time is inevitable and more pronounced than in general supervised learning. First, the data is not i.i.d. but sequential, so errors can cumulate. Second, the goal is to learn a policy that does something differently (better) than the behaviour policy π_β that collected the data D. Therefore, the state visitation frequency $d^\pi(s)$ will differ systematically from $d^{\pi_\beta}(s)$ at test time. Constraining $\pi(a|s)$ such that $D_{\mathrm{KL}}(\pi(a|s)||\pi_\beta(a|s)) \leq \epsilon$ can bound $D_{\mathrm{KL}}(d^{\pi(s)}||d^{\pi_\beta(s)})$ by $\delta \sim O(\epsilon/(1-\gamma)^2)$.

Dynamic programming methods, such as Q-learning algorithms, in principle can offer a more attractive option for offline RL as compared to pure policy gradients. However, they face an action distribution shift during training: when $\pi(a|s)$ differs substantially from $\pi_\beta(a|s)$, the Q-function regression targets won't be accurate. Since $\pi(a|s)$ is explicitly optimized to maximize reward, the policy will seek out-of-distribution actions that lead to wrong high returns. This means that, when an approximation error in a Q-value is an overestimation, it gets propagated (explicitly via the argmax of the Q-learning greedy policy update, or implicitly in actor-critic algorithms). In practice, this problem has a similar effect as over-fitting (it "unlearns" as the Q-values become more and more erroneous), although it can't simply be solved with a larger dataset. Note that this problem happens also with world models (the policy is still trying to exploit the model in order to maximize rewards). In standard reinforcement learning, this issue is somewhat corrected naturally when the policy runs in the environment to collect more data for the next iteration: it will attempt the over-optimistic transitions, and observe that in fact they are not.

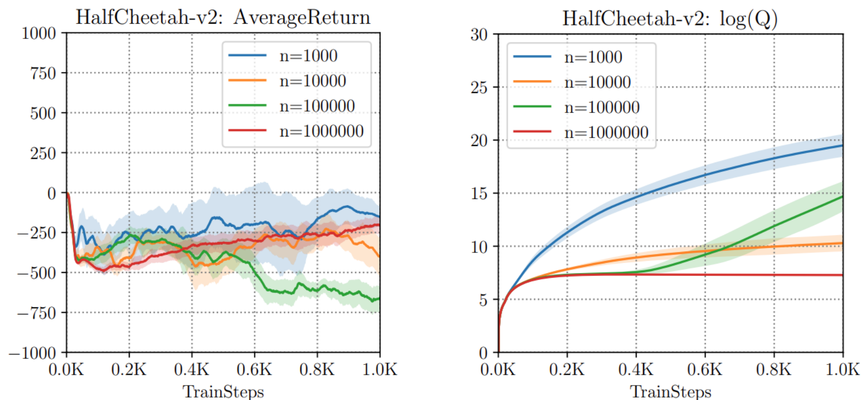


FIGURE 3.7: Over-optimistic Q-values (source: Levine et al., 2020)
Offline performance of SAC, showing return as a function of gradient steps (left) and average learned Q-values on a log scale (right), for different n (size of training set).

Importance sampling

Importance sampling, typically used for policy gradient algorithms, seems like a bad candidate since it already suffers from high variance. Recall that the importance weights at successive time steps are multiplied together, resulting in exponential blowup, which is not manageable when both policies are too

different from each other like in offline RL. Self-normalizing the importance weights results in a biased weighted importance sampling estimator, but with much lower variance.

Policy constraints

The goal is to ensure that $\pi(a|s)$ is close to the behavior distribution $\pi_\beta(a|s)$, while still deviating a little in order to improve. The constraints can be enforced either as direct policy constraints on the actor update (equation 3.14), or via a policy penalty added to the target Q-value or reward function, such as $\bar{r}(s,a) = r(s,a) - \alpha D(\pi(\cdot|s), \pi_\beta(\cdot|s))$.

$$\hat{Q}^\pi_{k+1} \leftarrow \arg\min_Q \mathbb{E}_{(s,a,s')\sim D}\left[\left(Q(s,a) - \left(r(s,a) + \gamma \mathbb{E}_{a'\sim\pi_k(\cdot|s')}\hat{Q}^\pi_k(s',a')\right)\right)^2\right] \tag{3.13}$$

$$\pi_{k+1} \leftarrow \arg\max_\pi \mathbb{E}_{s\sim D}\left[\mathbb{E}_{a\sim\pi(\cdot|s)}\hat{Q}^\pi_{k+1}(s,a)\right] \quad \text{s.t.} \quad D(\pi,\pi_\beta) \leq \epsilon \tag{3.14}$$

A common f-divergence constraint is the KL-divergence $D_{\mathrm{KL}}(\pi,\pi_\beta) = \mathbb{E}_{a\sim\pi(\cdot|s)}[\log\pi(a|s) - \log\pi_\beta(a|s)]$. However, explicit policy constraints require explicit estimation of the behavior policy, which means that the performance of these algorithms is limited by the accuracy of this estimation.

Other algorithms such as *Advantage Weighted Actor Critic* (AWAC) (Nair et al., 2020) opt for implicit f-divergence constraints. These methods first solve for the optimal next policy iterate under the KL-divergence constraint, non-parameterically, and then project it onto the policy function class via supervised regression.

Uncertainty estimation

The epistemic uncertainty (i.e. the entropy of $p(\phi|D)$, not the variance of the output of Q_ϕ) of the Q-function should be substantially larger for out-of-distribution actions. The goal is to learn a distribution $\mathcal{P}_D(Q^\pi)$ over possible Q-functions, and use it to update the policy as follows:

$$\pi_{k+1} \leftarrow \arg\max_\pi \mathbb{E}_{s\sim D, a\sim\pi(a|s)}\left[\mathbb{E}_{Q^\pi_{k+1}\sim\mathcal{P}_D(Q^\pi)}[Q^\pi_{k+1}(s,a)] - \alpha\mathrm{Unc}(\mathcal{P}_D(Q^\pi))\right]$$

Ways to compute uncertainty estimates include Gaussian processes, Bayesian neural networks, Bootstrap ensembles... Note that uncertainty methods have been used in the past for exploration, but in an optimistic way (for pessimistic purposes the uncertainty estimation needs to be much more precise).

Conservative Q-learning

The idea behind *Conservative Q-learning* (CQL) (Kumar et al., 2020) is to regularize the value function or Q-function directly to avoid overestimation for

out-of-distribution actions. Let the Bellman error:

$$\mathcal{E}(D,Q) = \mathbb{E}_{(s,a,s')\sim D}\left[\left(Q(s,a) - \left(r(s,a) + \gamma \mathbb{E}_{a'\sim\pi(\cdot|s')} Q(s',a')\right)\right)^2\right]$$

A first conservative objective is introduced. It provides a lower bound $\hat{Q}^\pi(s,a) \leq Q^\pi(s,a)\ \forall(s,a) \in D$, but risks underestimation. It should mostly push down on Q-values for out-of-distribution actions because the Bellman error anchors in-distribution actions.

$$\hat{Q} \leftarrow \arg\min_Q \max_\mu \alpha \mathbb{E}_{s\sim D, a\sim\mu(\cdot|s)}\left[Q(s,a)\right] + \mathcal{E}(D,Q)$$

A second less conservative objective ($\mathbb{E}_{\pi(a|s)}\hat{Q}^\pi(s,a) \leq \mathbb{E}_{\pi(a|s)}Q^\pi(s,a)\ \forall s \in D$) is also presented. When μ is equal to π_β, the penalty is zero, so high Q-values are only assigned to in-distribution actions.

$$\hat{Q} \leftarrow \arg\min_Q \alpha \mathbb{E}_{s\sim D}\left[\log \sum_a \exp Q(s,a) - \mathbb{E}_{a\sim D} Q(s,a)\right] + \mathcal{E}(D,Q)$$

COG (Singh et al., 2020) and (Bharadhwaj et al., 2020) are two of the many algorithms that follow similar principles. MoREL (Kidambi et al., 2020) and MOPO (Yu et al., 2020) are conservative model-based algorithms. In model-based offline RL, the model suffers from distribution shift both in states and actions, since the policy (or action sequence, in the case of planning) is optimized to obtain the highest possible expected reward under the current model.

Chapter 4

Reward Relabeling for combined Reinforcement and Imitation Learning

4.1 Introduction

Two of the main remaining challenges in RL are reward shaping and sample efficiency. Reward shaping can be defined as the process of engineering the right reward function for the problem at hand. Since the reward function is the only supervision signal that the agent will receive, reward shaping is a crucial component of the training process. However, having to design a new reward function for every task makes it very difficult to translate results between tasks, and often relies on the intuition of the designer rather than a robust methodology. On the other hand, sample-efficient algorithms are required to obtain faster and more reliable results, particularly in robotics where it is much harder to train an algorithm outside of simulation. The recent trend towards more data-driven algorithms could be a solution to both of these problems. Indeed, additional data can alleviate the need for online data collection, and demonstration data can act as additional supervision, guiding the agent to good behaviours alongside a simpler task-agnostic reward function.

With these objectives in mind, we designed a method called Reward Relabelling, able to leverage both demonstrations and episodes collected online in any sparse-reward environment with any off-policy algorithm. Before introducing our method, we present a short background on the three most closely-related families of RL algorithms. This chapter focuses mainly on the theoretical side and inner workings of our reward-relabeling method, while the following chapter will present an extension of the algorithm and more detailed experimental results.

4.2 Background - Other related frameworks

4.2.1 Self-Imitation Learning

Having an expert to learn from can be very helpful, but such a luxury is not always available. Instead, we can apply IL methods to the experiences collected by the agent in the environment, to further supervise its behaviour. (Oh et al., 2018) introduced Self-Imitation Learning (SIL), where an additional loss function pushes the agent to imitate its own decisions in the past only when they resulted in larger returns than expected.

Further works such as Self-Imitation Advantage Learning (SAIL) (Ferret, Pietquin, and Geist, 2020) followed. One of the main improvements over SIL is that SAIL mitigates the problem of stale returns by choosing the most optimistic return estimate between the observed return and the current action-value for self-imitation. Indeed, relying just on the observed return (like in SIL) is not ideal because it quickly becomes outdated as the policy improves.

4.2.2 Reward bonus

One major challenge in RL is exploration, and one common approach to encourage it is to augment the environment reward with an additional bonus. Many previous works rely on some sort of optimistic exploration: assume the unknown to be good. A simple idea is counting state occurrences, even if the space is too large or continuous via pseudo-counts (Bellemare et al., 2016) or hashing (Tang et al., 2017). Another idea is to use any type of prediction error, which can quantify the novelty of new experience. For instance, in (Pathak et al., 2017) they learn a forward model to predict future states, and in RND (Burda et al., 2018) they predict the output of a fixed randomly initialized neural network on the current observation. More generally, reward bonuses are used to provide further supervision to the agent. For instance, (Shelhamer et al., 2016) presents a variety of self-supervised losses such as simple proxies of the reward (e.g. predict the sign), the dynamics (or inverse dynamics), or based on reconstruction errors (e.g. auto-encoders).

4.2.3 Hindsight Experience Replay (HER)

Since off-policy RL algorithms can theoretically use data coming from any policy, a natural idea was to share data between tasks in a multi-task setting. An even better idea came in HER (Andrychowicz et al., 2017), where the authors pointed out that if we accidentally solve one task when trying to perform another task, that experience is still optimal if we relabel the goal that was initially intended. Indeed, RL can be viewed as a joint optimization problem over both the policy and the data, alternating between finding good data (generally using relabeled data) and training a model on that data.

Similar and more general works followed, such as GCSL (Ghosh et al., 2019), Generalized Hindsight (Li, Pinto, and Abbeel, 2020), and HIPI (Eysenbach et al., 2020), which reframes the relabeling problem as inverse RL. RCP (Kumar,

Peng, and Levine, 2019) extended the idea to the single-task setting, by learning a policy conditioned on the trajectory return. Indeed, non-expert trajectories collected from sub-optimal policies can be viewed as optimal supervision, not for maximizing the reward, but for matching the reward of the given trajectory. By then conditioning the policy on the numerical value of the reward, they obtain a policy that generalizes to larger returns.

4.3 Reward Relabelling

Off-policy algorithms are more suited to benefit from demonstration data than on-policy ones, so we only focus on the former. Indeed, one can simply add any off-policy data into the replay buffer and the algorithm should be able to make good use of it. The replay buffer then contains both both demonstrations and episodes collected online. As discussed in the introduction of the chapter, we also want to focus on sparse-reward environments. The method that we present here can therefore be applied to any sparse-reward environment with any off-policy algorithm, and is able to accelerate the training process.

Our method is based on a reward bonus given to demonstrations and successful episodes (via relabeling), encouraging expert imitation and self-imitation. In this work we focus on off-policy RL, and present a way to leverage offline data in the form of expert demonstrations. Our method is based on the observation that, in hindsight, a successful episode of collected experience is in fact a demonstration, so it should receive the same treatment. In particular, we propose to add a reward bonus to transitions coming from both demonstrations and successful episodes. Our approach provides a simple way of tying positive rewards and desired behaviour, without any task-specific reward shaping.

We instantiate our approach with Soft Actor-Critic (SAC) (Haarnoja et al., 2018b) and Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016). In this section we introduce a new method that consists in giving a reward bonus to demonstrations and relabeling successful episodes as demonstrations. We thoroughly test it in a reaching task, and show that it greatly improves upon the base algorithm.

4.3.1 Method

We propose SAC-R^2 and DDPG-R^2, acronyms for SAC with Reward Relabeling and DDPG with Reward Relabeling respectively, based on a straight-forward method that could be implemented to any other off-policy RL algorithm with sparse rewards. Let R be the sparse reward from the environment, b the reward bonus of our method, and L the amount of transitions that will receive the bonus. First, we add demonstration data to the buffer: the last transition of each expert trajectory is given the sparse reward R, and the last non-final L transitions are given a reward equal to b. Then, as the agent explores the environment during training, every new successful episode is relabeled the same way: the last L transitions leading to the sparse reward are assigned a reward equal to b. Intuitively, our method helps propagate the signal of the sparse

reward by explicitly turning zero rewards into positive rewards, rather than entirely relying on bootstrapping rare future rewards.

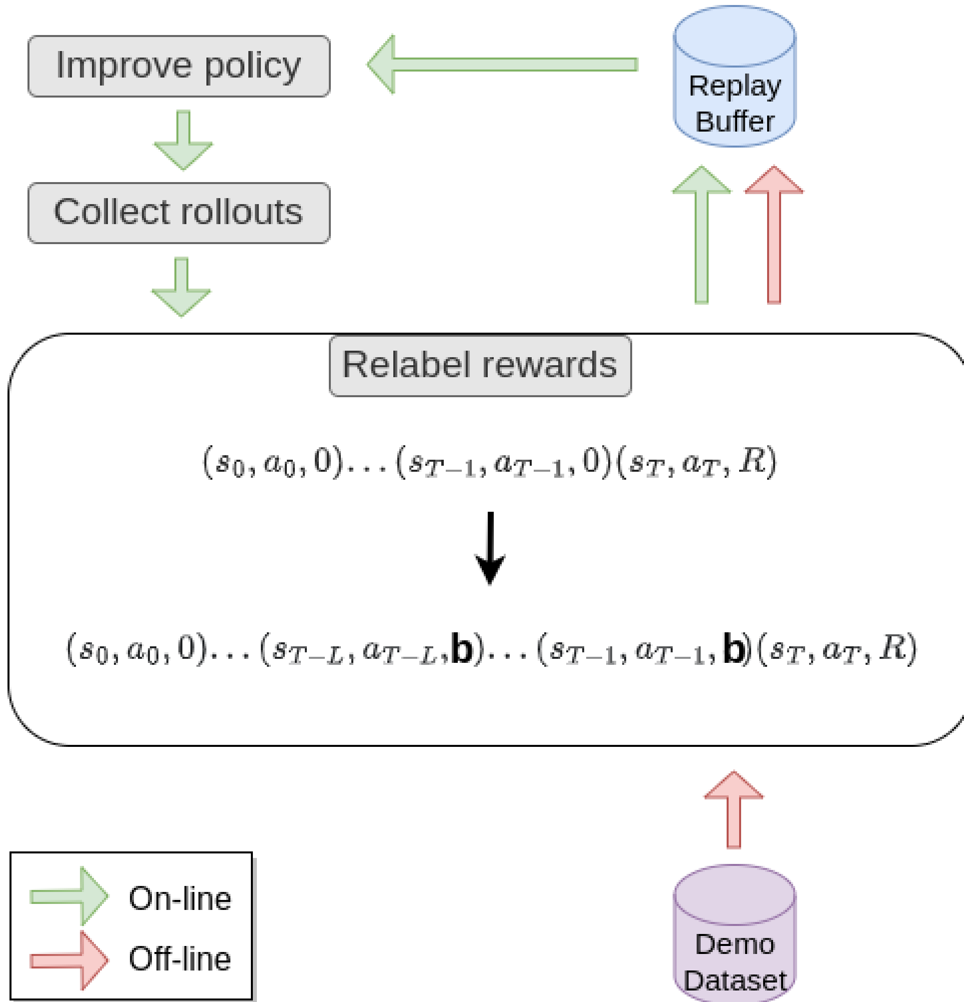


FIGURE 4.1: Reward Relabeling procedure.

Reward bonus to demonstrations. The first part of our algorithm is most similar to SQIL (Reddy, Dragan, and Levine, 2020). Intuitively, it gives the agent an incentive to imitate the expert. Their paper shows theoretical connections between SQIL and regularized behaviour cloning. One important difference is that SQIL is a pure IL algorithm, while our method learns from both the reward bonuses and the reward from the environment. Also, SQIL gives an incentive to avoid states that were not in the demonstration data, which could potentially be harmful if those states led to successful behaviour.

Reward bonus to successful episodes. The relabeling part of our algorithm tries to mitigate this issue and is most similar to SIL (Oh et al., 2018) or SAIL (Ferret, Pietquin, and Geist, 2020). In our method, the self-imitation is achieved by effectively treating successful episodes as if they were demonstrations. In order to give a reward bonus to successful episodes, we need to wait for the episodes to end first. We follow the idea presented in HER (Andrychowicz

et al., 2017) to relabel past experiences and modify the transitions' rewards. To the best of our knowledge, HIPI (Eysenbach et al., 2020) is the only existing method to also modify the rewards this way. While their method relies on inverse RL to tackle the more general multi-task setting, ours is more straightforward for the simpler single-task setting with sparse rewards.

4.3.2 Optimal Policy: Discussion and Safety Decay

One issue with our method so far, is that it might change the optimal policy of the problem. Intuitively, the bonus encourages a policy that requires at least L steps to reach the goal, which might not take the shortest path available. We want our method to converge to the optimal policy of the sparse reward problem.

As stated in (Ng, Harada, and Russell, 1999), in order to still converge to the optimal policy of the original problem one can only add a reward term such as, for a given transition between two states, the term is expressible as in the difference in value of an arbitrary potential function applied to those states. Our reward bonus cannot be expressed as such since it depends on the time-step of the states. In fact, our reward violates the Markov assumption of the MDP, its future evolution is no longer independent of its history.

To ensure that our method converges to the optimal policy, we decide to use a decay that eventually causes the bonus to completely disappear. By doing so, our method operates in two steps: An initial RL training with reward bonuses that might not converge to the optimal policy, followed by a second RL training without any reward bonuses which should converge to the optimal policy. In practice, on top of not adding any future bonuses during data collection, we also have to remove the reward bonuses from the transitions stored in the buffer. Since the environment's rewards are fully sparse, this simply requires to ignore any reward from any non-final transition sampled from the buffer.

For our experiments, we use a simple linear decay that quickly turns the bonus to 0 when the success rate stops improving. We refer to this decay as **safety decay**. In the following section we will define a second decay, called **scheduled decay**, that controls the bonuses in a more organic way throughout the entire training process.

4.3.3 Hyper-parameters: Tuning and Scheduled Decay

Our method has two hyper-parameters to tune, b and L. L represents how far the sparse reward should be propagated into the past, and is similar to other parameters in RL such as the n of the n-step look-ahead in Q-learning. Why not relabel all time-steps? We do not wish to reward the early transitions of overly long episodes since they probably did not help to the completion of the task. Also, the cumulative rewards would greatly vary between episodes.

How about b? We propose to adjust the value of b during training based on the agent's recent success. This also acts as a more natural decay during

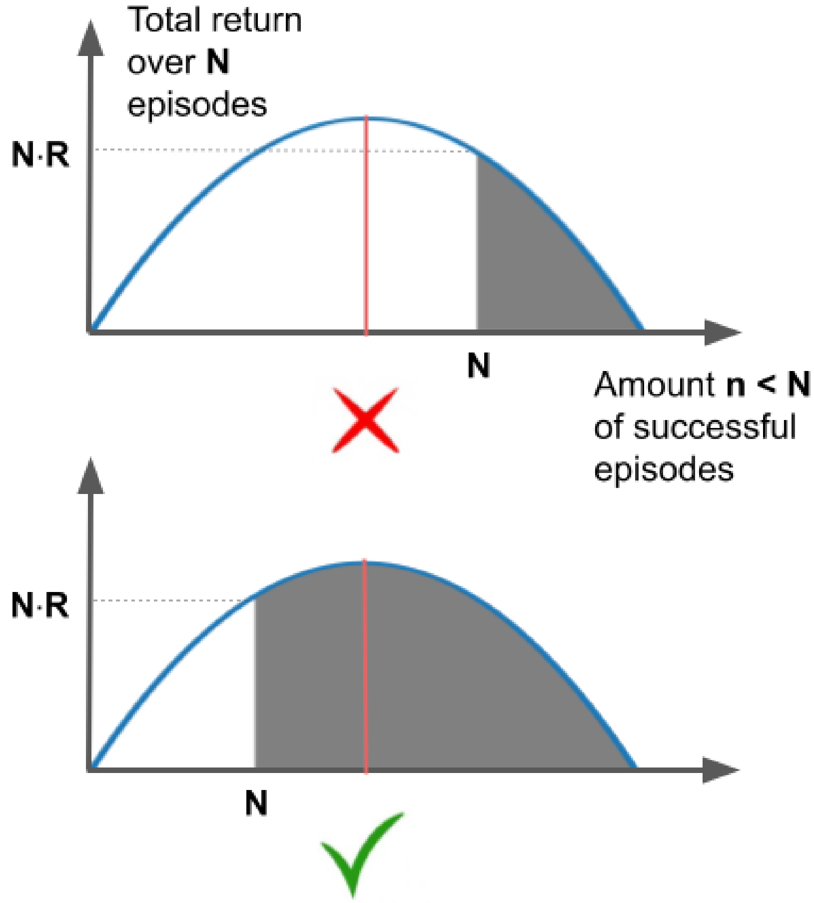


FIGURE 4.2: Reward obtained as a function of the agent's success. The more successful the agent, the smaller the bonus, until recovering the fully sparse reward at 100% success rate. We want to avoid the situation on the top, where the highest cumulative reward is obtained for a smaller success rate.

the entire training process. Let's place ourselves in the episodic undiscounted scenario. Let ζ be the current average success rate (in practice, we use the fraction of successful episodes in the last 100-episodes window), and r the total reward obtained over a successful episode τ. We wish to have the following reward:

$$r(\tau) = R + L \cdot b \cdot (1 - \zeta) \tag{4.1}$$

Intuitively, the more the agent struggles, the larger the bonus to help guide it. We can study the behaviour of this reward function in the case where the agent reaches a constant final success rate. Let f the cumulative reward obtained over N runs, and n the number of successful episodes.

$$f(n) = n(R + L \cdot b \cdot (1 - \frac{n}{N})) = n(R + b \cdot L) - n^2 \frac{b \cdot L}{N}$$

The cumulative reward f is shown in Figure 4.2. If not careful, the optimal

behaviour for this reward can correspond to a policy that has not actually reached a success rate of 100%. There is a simple condition to avoid this: $N \leq \arg\max f$ if and only if $b \cdot L \leq R$. We are free to chose b and L as we want, as long as their product, corresponding to the total reward coming from bonuses in an episode, is smaller than the final sparse reward. A similar condition holds for the discounted case.

$$\begin{aligned} \text{undiscounted} \quad & b \cdot L \leq R \\ \text{discounted} \quad & b\left(\sum_{i=0}^{L-1} \gamma^i\right) \leq R \cdot \gamma^L \end{aligned} \tag{4.2}$$

Therefore, having a reward as defined in 4.1 yields an intuitive condition: R should prevail over the bonuses. Since R is always the main source of supervision during the entire training process, this condition should also help to ensure that the policy's behaviour never deviates too much from the optimal policy of the original MDP problem, which would facilitate that the policy can recover to it once the bonuses have disappeared from the training.

The most straight-forward implementation for such a reward is to use the same bonus for all transitions within an episode τ_t, but which decays over time t as the success rate improves. We refer to this decay as **scheduled decay**:

$$b_k \leftarrow b \cdot (1 - \zeta_k) \tag{4.3}$$

Although we always use both the safety and scheduled decays, in practice our experiments showed that the scheduled decay alone is enough to avoid the undesirable side-effects of our method.

4.4 Experiments

4.4.1 Setup

In this chapter we focus on a single task from RLBench (James et al., 2020), we choose one of the simplest and most fundamental tasks in robotics: reaching a target. We evaluate our method on a 6-degrees-of-freedom robot manipulator. The target ball appears randomly at the beginning of each episode within the reach of the robot (anywhere in the 3D scene). The state has 22 dimensions, 19 from the robot proprioceptive state (joint angles, joint speeds, gripper pose) and 3 from the task-related information (3D coordinates of the ball). The robot receives 6-dimensional joint speed commands.

The expert demonstrations are provided by the environment and rely on OMPL (Sucan, Moll, and Kavraki, 2012) for motion planning. An episode ends once the robot has completed the task, or after expiration (100 time-steps). The reward is fully sparse, and is equal to +100 if the robot solves the task and 0 otherwise.

```
Algorithm 1: R^2
Require: L amount of steps to relabel.
Set b according to equation 4.2. Initialize buffer with demonstrations:
    set reward r = b for the last L non-final transitions.
Initialize empty episode.
while not converged do
    sample a batch B from the buffer
    if b > 0 then
        b ← decay_rule(b) according to equation 4.3
    else
        set r = 0 for all non-final transitions in B
    end
    do off-policy RL update
    if len(episode) == 0 then
        collect one episode
        if episode is successful then
            set r = b for the last L non-final transitions
    pop a transition (s, a, s', r) from the episode and add it to the buffer
end
```

We compare our methods SAC-R^2 to a simple baseline SAC-Demo, which we define as SAC with demonstrations in the buffer. For SAC-R^2, we use the scheduled decay presented in equation 4.3. If the success rate doesn't improve over a period of 5000 iterations, the safety decay is applied to completely erase the bonus. We always choose the largest bonus allowed by the condition 4.2.

All the results are smoothed with a rolling window of 100 episodes, and the standard error is computed on three random seeds.

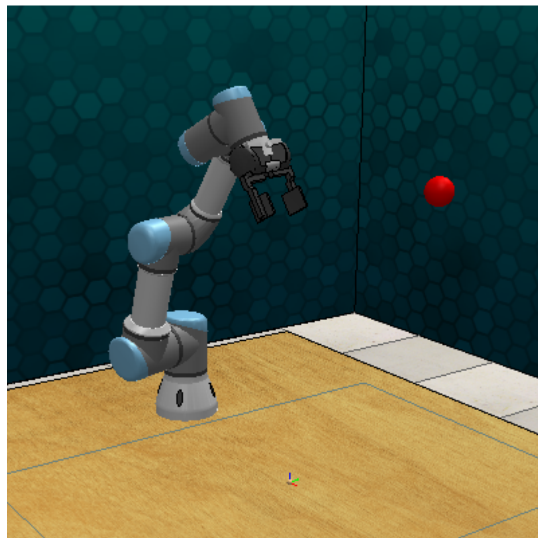


FIGURE 4.3: Reaching task in RLBench

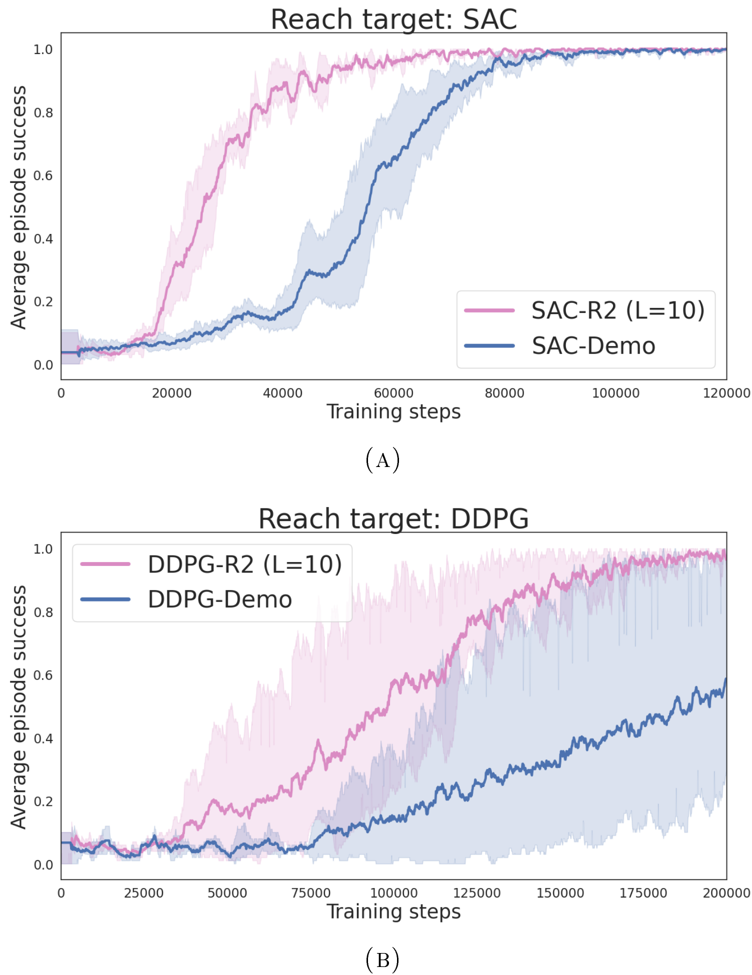


FIGURE 4.4: Learning curves for the RLBench reaching task.

4.4.2 Results

Figures 4.4a and 4.4b show that SAC-R^2 and DDPG-R^2 are significantly faster than the baselines on the *reach target* task. Most of the speed increase happens in the beginning of the training, so R^2 seems to provide a more efficient exploration, guiding the agent faster towards the environment rewards when these are hard to come by. More importantly, both R^2 versions consistently solve the task, whereas the baselines show a much larger variance, and even fail on some seeds in the case of DDPG.

4.4.3 Impact of hyper-parameters

We remind that there are two hyper-parameters, but in practice if we set b as the largest bonus allowed by the condition 4.2, then only L needs to be chosen. As we see in Figures 4.5 and 4.6, there appears to be an optimal value for this particular task, which is consistent with the choice of the largest b allowed. Interestingly, setting an overly large b or L doesn't hinder the performance too much, which means that the algorithm is robust to the hyper-parameter values. However this robustness is more apparent during the early and middle stages

FIGURE 4.5: Impact of hyper-parameters on SAC. For a given hyper-parameter value, we plot the number of training steps required for the average episode success to reach a certain threshold.

of the training process, whereas near convergence the drop in performance is larger (especially for b).

Impact of hyper-parameter L. Figure 4.5 (bottom) shows the impact of different values of L. Having a longer window hurts the performance, as adding a bonus to transitions closer to the goal is probably more important. Also, if L is bigger than the length of an episode some bonuses will not be assigned. For this task, there seems to exist an optimal value somewhere between 5 and 10.

Impact of hyper-parameter b. What happens if we ignore the condition found in Section 4.3.2 and choose b as an independent hyper-parameter? Figure 4.5 (top) shows the impact of different values of b for a fixed $L = 10$. Surprisingly, violating the condition ($b > 8$) does not hinder the performance that much, which means that as long as there is some sort of decay the value of b is not that important. The drop in performance only becomes noticeable with very large values. As expected, choosing a smaller bonus ($b < 8$) reduces the performance as the algorithm becomes closer to SAC-Demo (which can be seen as SAC-R^2 with $b = 0$).

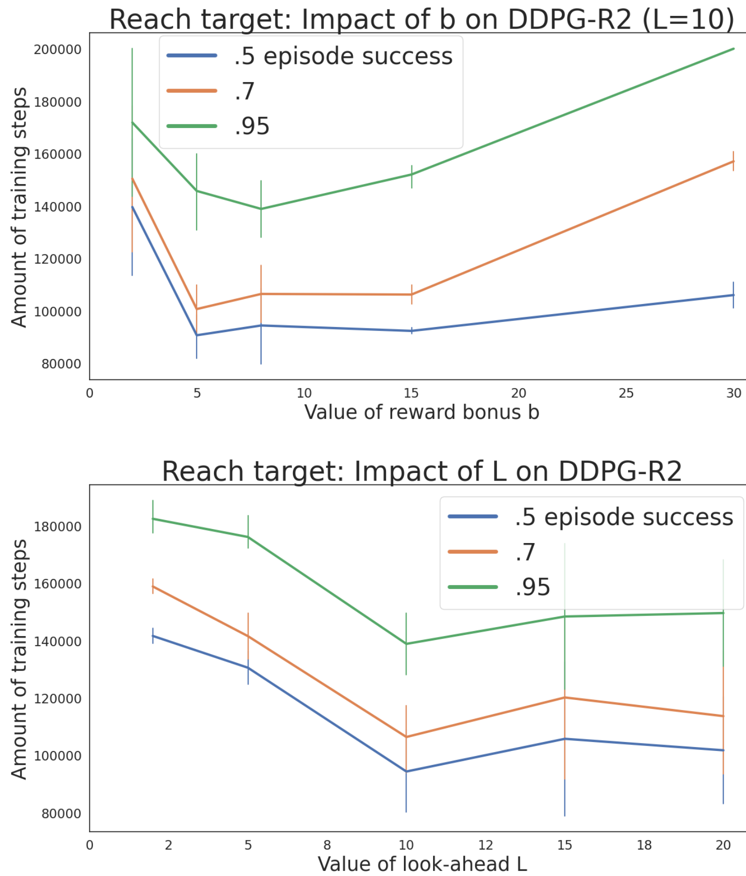


FIGURE 4.6: Impact of hyper-parameters on DDPG. For a given hyper-parameter value, we plot the number of training steps required for the average episode success to reach a certain threshold.

4.4.4 Ablation study

Figure 4.7 shows the impact of online relabelling and decay. What happens if we skip the former? We still give a reward bonus to the transitions coming from demonstrations, but we no longer relabel successful episodes. The red curve in Figure 4.7b shows that, without relabeling, the learning process seems more unstable, probably due to the lack of consistency in the rewards once the agent has collected enough successful episodes. However, even without relabeling we can notice a significant boost with respect to the baseline.

The red curve in Figure 4.7a shows the impact of not decreasing the bonus during training. Both curves (pink and red, with and without decay) are very similar, but the agent without decay does not actually reach a 100% success rate, and additional results show that the larger the bonus, the smaller the final performance. As discussed in Section 4.3.2, SAC-R^2 does not seem to converge to the optimal policy without a decay.

Table 4.1 confirms these results. Not only does it show that the average success rate across evaluation episodes is smaller if the decay is absent, but the

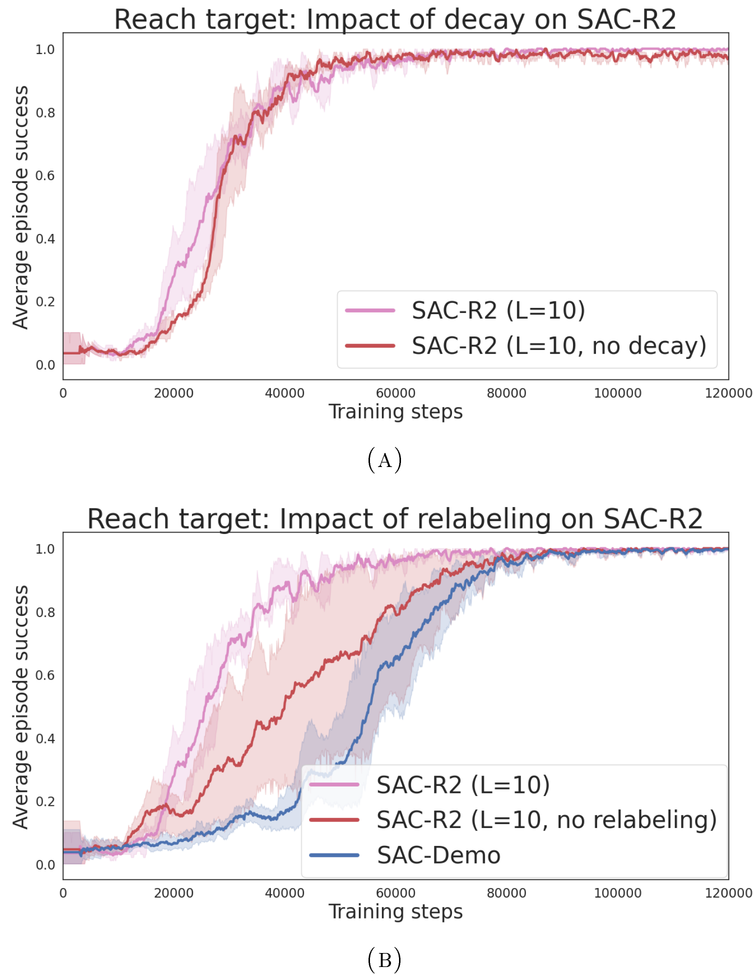


FIGURE 4.7: Learning curves for different ablation studies.

average length of the succesful episodes is also longer. This last point shows the drift from the optimal policy of the MDP, which should try to find the goal as quickly as possible without detours. When the decay is present, the average episode length is consistent with that of the baseline, which shows that our method appears to be able to recover to the optimal policy.

4.4.5 Alternative decay

The scheduled decay defined in equation 4.3 is not the only way to obtain the desired reward 4.1. We recall that the goal is to ultimately drive the sum of bonuses to 0 when they are no longer needed. Since L and b play symmetrical roles in that equation, we can apply the decay on L rather than b, as shown in Figure 4.8. This alternative reward was actually our initial implementation, because it provides a more intuitive interpretation from the lense of curriculum learning.

Our method operates on two steps: first learn to reach a vicinity of the goal (be within L steps of the goal), then learn to reach the goal. A decay on L is consistent with this interpretation: the more succesful the agent, the closer

	Success rate	Episode length (succesful episodes)
SAC-Demo	96.1	12.92
SAC-R2 (L=10)	**99.1**	**12.36**
SAC-R2 (L=10, no decay)	90.1	16.49

TABLE 4.1: SAC-R^2, with and without decay, compared to the baseline SAC-Demo on evaluation episodes. The average success rate and average episode length (of succesful episodes only) are calculated across 3000 evaluation episodes (3 seeds, 1000 per seed).

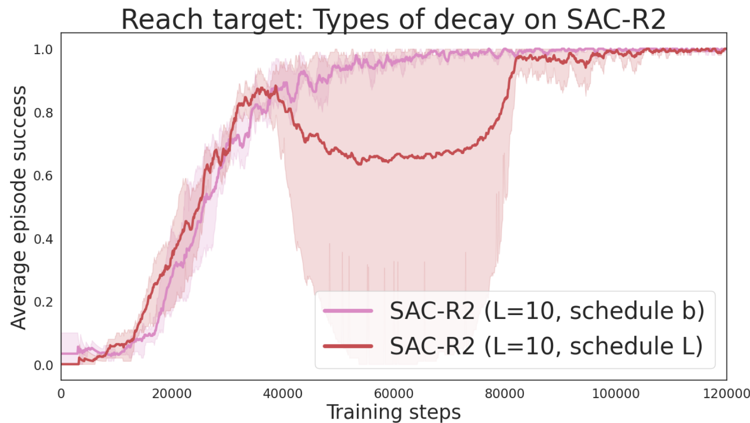


FIGURE 4.8: Learning curves for two different types of decay.

to the goal it needs to go in order to see rewards. However, as shown in the figure, the results are not as good as with a decay on b, as the training is more unstable.

Rather than a constant bonus (within the transitions of an episode), we also tried other alternatives, such as having a vanishing bonus (in both directions, larger towards the beginning or the end of the L-step segment), but the results didn't report any noticeable differences compared to the simpler constant bonus.

4.5 Discussion

We propose Reward Relabeling (R^2), a generic method that can be applied to any off-policy RL algorithm in any sparse-reward environment. It encourages two behaviours: imitate the expert demonstrations (if available), and imitate the past successful trajectories. We have seen that it can greatly improve upon the base algorithm on a simple reaching task, and the next chapter will focus on an algorithm extension evaluated on a larger pool of tasks.

However, one obvious limitation of our method is that it is only suited for a specific kind of task: those defined by a final discrete goal. For tasks defined as continuous problems, like balancing an object, our method is not the best choice. What about tasks in which beginning actions are crucial to the performance of the agent? One way to handle those tasks is to simply decompose them into

a sequence of sparse-reward tasks. For instance, if the scene of the reaching task was divided by a wall, we could simply give an intermediate reward after choosing the right fork (the one leading to the side of the scene containing the target). For those sequential tasks, our method could still be applied as is. For more complex tasks where credit assignment is especially tricky, simply increasing L to the entire episode could work to some extent. But we believe that a different relabeling strategy should be used for those tasks, such as sampling the set of transitions to relabel according to some heuristic or some neural network trained for that purpose.

Chapter 5

STIR2: Self and Teacher Imitation by Reward Relabeling

5.1 Introduction

In the previous chapter we introduced Reward Relabeling (R^2), which is able to leverage a set of demonstrations added to the replay buffer, alongside episodes collected online in any sparse-reward environment with any off-policy RL algorithm. Our method is based on a reward bonus given to demonstrations and successful episodes (via relabeling), encouraging expert imitation and self-imitation.

In the past, other ideas have been proposed to make good use of the demonstrations added to the replay buffer, such as pretraining on demonstrations only or minimizing additional cost functions. After presenting a brief background on some of these ideas, in this chapter we present a new algorithm, an extension of R^2, which we call Self and Teacher Imitation by Reward Relabeling ($STIR^2$). Similarly to other algorithms such as Rainbow (Hessel et al., 2018), $STIR^2$ integrates into our method multiple improvements from other works, and show that it outperforms all baselines. Our experiments focus on several robotic-manipulation tasks across two different simulation environments.

5.2 Background - SAC and DDPG

In this section, we briefly recall the equations behind Soft Actor-Critic (SAC) (Haarnoja et al., 2018b) and Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016), two of the most widespread algorithms for continuous-action spaces in RL.

The usual RL framework is the following: an agent interacts with an environment by performing an action and observing a feedback signal (reward r) and the new state of the environment. The goal is to find the policy π (function mapping states s to actions a) that maximizes the discounted cumulative

reward (the parameter $\gamma \leq 1$ makes future rewards smaller).

$$\pi^* = \arg\max_\pi \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{k=0}^{T} \gamma^k r(s_{1+k}, a_{1+k}) \right]$$

The *Q-function* $Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{p_\pi} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t, a_t \right]$ is defined as the reward-to-go from the state s_t if we pick the action a_t and then follow π. This function obeys the Bellman equation:

$$Q^\pi(s, a) = r + \gamma \mathbb{E}_{s' \sim p(\cdot|s,a), a' \sim \pi(\cdot|s')} \left[Q^\pi(s', a') \right]$$

Deep Deterministic Policy Gradient. The goal is to learn a deterministic policy μ_θ. To train the critic, we can approximate the right-hand expectation of the Bellman equation with samples, set it equal to y, and minimize the MSBE loss on a parameterized Q_ϕ (in practice, two additional *target* approximators $Q_{\phi'}$ and $\mu_{\theta'}$ are used to compute the targets y):

$$\mathcal{L}(Q_\phi) = \mathop{\mathbb{E}}_{(s,a,r,s',d) \sim \mathcal{D}} \left[(Q_\phi(s, a) - y)^2 \right]$$

To train the actor, simple gradient descent w.r.t. θ on the following loss:

$$\mathcal{L}(\mu_\theta) = - \mathop{\mathbb{E}}_{s \sim \mathcal{D}} \left[Q_\phi(s, \mu_\theta(s)) \right]$$

Soft Actor-Critic. In classic RL, the optimal policy is always deterministic under full observability, but stochastic policies have interesting properties: better exploration and robustness (due to wider coverage of states), and multi-modality. This new objective promotes stochasticity by maximizing the entropy $\mathcal{H}$ of the policy (α is a hyper-parameter, and we omit γ for simplicity):

$$\pi^* = \arg\max_\pi \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right]$$

A new Q-function (slightly different) is derived and follows the *soft* Bellman equation:

$$Q^\pi(s, a) = r + \gamma \mathbb{E}_{s' \sim p(\cdot|s,a), a' \sim \pi(\cdot|s')} \left[Q^\pi(s', a') - \alpha \log \pi(a'|s') \right]$$

Similar to DDPG, to train the critic we can approximate the right-hand expectation with samples, set it equal to y, and minimize the MSBE loss on a parameterized Q_ϕ (in practice, two approximators Q_{ϕ_1} and Q_{ϕ_2} are trained, with their respective *target* versions $Q_{\phi'_1}$ and $Q_{\phi'_2}$).

To train the actor π_θ, the actor loss is derived from the reparameterization trick to compute samples $\tilde{a}_\theta(s, \epsilon) = \mu_\theta(s) + \sigma_\theta(s)\epsilon$, where ϵ is some random

noise:

$$\mathcal{L}(\pi_\theta) = -\mathop{\mathbb{E}}_{s\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\epsilon)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\epsilon)|s)\right]$$

5.3 Related Work

Demonstrations can be used to design the reward, guide exploration, augment the training data, initialize policies, etc. Let's look at some previous works that also combine demonstrations and RL. For example, (Zhu et al., 2018) introduce a hybrid reward that combines a simple (multi-stage sparse) manually designed task reward with an imitation-based reward.

DQfD (Hester et al., 2017) uses demonstrations to populate the replay buffer of DQN as a preliminary step to learn an initial policy, and as additional off-policy data during training.
During the pre-training phase four losses are applied: the 1-step double Q-learning loss, an n-step double Q-learning loss, a supervised large margin classification loss, and an L2 regularization loss on the network weights and biases. The Q-learning loss ensures that the network satisfies the Bellman equation and can be used as a starting point for TD learning. The supervised loss pushes the value of the demonstrator's actions above the other action values, thus helping to ground all state-actions pairs (including those not seen in the demonstrations) to realistic values.
During TD learning, the agent never over-writes the demonstration data, and the supervised loss is not applied to self-generated data. Proportional prioritized sampling is used to control the relative sampling of demonstration versus agent data (demonstration data is given a bonus to be sampled more often).
DDPGfD (Vecerik et al., 2017) adds similar improvements to DDPG: transitions from demonstrations are added to the replay buffer, prioritized replay is used for sampling transitions (demonstration data is slightly boosted), a mix of 1-step and n-step return losses are used, and L2 regularization losses on the weights of the critic and the actor are used.

In (Nair et al., 2018a) the authors address the problem of stacking 6 blocks from a dataset of 100 demonstrations collected via teleoperation. They use DDPG + HER in a goal-conditioned setting with sparse rewards, conditioning both the actor and the critic on the desired final positions of the blocks.
Their first idea is to consider an additional replay buffer containing demonstration data. Their second idea is to introduce an auxiliary behaviour cloning loss applied to samples from this buffer, which prevents the policy from deviating too much from the demonstrations, and accounts for sub-optimimality of the demonstrations by filtering out updates where the critic under-performs. The third idea, if the simulator allows it, is to reset some episodes to a state sampled uniformly from a demonstration to overcome the challenge of sparse rewards. As in HER, the final state in that demonstration is the goal for the episode.

Normalized Actor-Critic (NAC) (Gao et al., 2018) is based on soft Q-learning and uses the demonstrations as the only training data during the first iterations

of the algorithm. The maximum entropy framework provides a natural normalization of the Q-function so that it is initialized everywhere: high values for the expert's action and low values for alternative actions.

Demonstration Actor-Critic (DAC) (Liu et al., 2021) introduce a novel objective with policy-dependent shaping reward, which can provide both generalized supervision for all states as well as direct supervision signal over these demonstrated states. The augmented reward equals zero if the state is unseen in the demonstrations, otherwise is a positive number, the larger the closer the policy to the expert policy.

$$J(\pi) = \mathbb{E}_{(s,a)\sim\rho_\pi}\left[r(s,a) + \mathbf{1}_{s\in\text{supp}\rho_{\pi_E}(s)}\left(M - D_{\text{KL}}(\pi(\cdot|s), \pi_E(\cdot|s)\right)\right] \quad (5.1)$$

The full derivation of the algorithm includes a discriminator $D_\omega(s,a)$ which has to tell if a comes from π or the expert.

Demo-Augmented Policy Gradient (DAPG) (Rajeswaran et al., 2017) To combat distribution drift, a small amount of noise (uniform random $[-0.1, 0.1]$ radians) is added to the actuators per timestep in the demonstrations. The demonstrations are used twice: to pretrain with behaviour cloning, and to augment the policy gradient $g = \sum_{(s,a)\in\rho_\pi}[\nabla_\theta \log \pi_\theta(a|s) A^\pi(s,a)] + \sum_{(s,a)\in\rho_D}[\nabla_\theta \log \pi_\theta(a|s) w(s,a)]$ where $w(s,a)$ is a weighting function (see paper for details).

5.3.1 Baselines

For our baselines, we focus on three algorithms that can be applied to any continuous-action off-policy algorithm with minor modifications.

- DDPG-fD and SAC-fD (Vecerík et al., 2017) (originally DDPGfD based on DDPG) introduce three main ideas as described above: transitions from demonstrations are added to the replay buffer, demonstration data is sampled more often from the buffer, and a mix of 1-step and n-step return losses is used.
- DDPG-BC and SAC-BC (Nair et al., 2018a) (has no name and originally based on DDPG) also introduce three main ideas as described above: transitions from demonstrations are added to a separate additional replay buffer, an auxiliary behaviour cloning loss is applied to samples from this buffer, and some episodes are reset to a state sampled uniformly from a demonstration. The authors additionally present other ideas regarding the multi-goal setting which we will not cover in this work.
- DDPG-SAIL and SAC-SAIL (Ferret, Pietquin, and Geist, 2020) (originally SAIL based on DQN). This algorithm was presented in Section 4.2 in Chapter 4. It is a self-imitation RL algorithm, but we modify it by introducing demonstrations in the replay buffer like the other baselines.
- DDPG-Demo and SAC-Demo. Simple baseline introduced in Chapter 4. Simply the base DDPG and SAC algorithms, but just like with SAIL, we introduce demonstrations in the replay buffer.

5.4 STIR2: Self and Teacher Imitation by Reward Relabeling

In this section we introduce SAC-STIR^2 and DDPG-STIR^2 (acronym for Self and Teacher Imitation by Reward Relabeling), two new algorithms that integrate four main improvements into the base SAC and DDPG algorithms. All these improvements could also be added to any other continuous-action off-policy RL algorithm. Let's define $\mathcal{L}(\pi_\theta)$ and $\mathcal{L}(Q_\phi)$ as the usual SAC and DDPG losses for policy and Q-function respectively. Other than the reward-relabeling hyper-parameter L discussed in Chapter 4, the full algorithm introduces four additional hyper-parameters: n, λ_n, λ_{BC}, and λ_{SAIL}. We now explain each improvement in more details.

Reward Relabeling.

The reward relabeling method from our algorithm SAC-R^2 presented in Chapter 4.

N-step loss. Following SAC-fD (Vecerík et al., 2017), we use the n-step loss $\mathcal{L}_n$ to train the critic. This loss is a modified version of the standard Q-function loss $\mathcal{L}(Q_\phi) = \mathcal{L}_1(Q_\phi)$, with n-step returns replacing the immediate reward. Using a larger look-ahead n can be particularly useful for tasks with sparse rewards, since it increases the chances of encountering a reward. For instance, for the simpler DDPG case we have:

$$\mathcal{L}_n(Q_\phi) = \mathbb{E}_{\mathcal{D}}\left[\left(\sum_{k=0}^{n-1}\gamma^k r_{t+k} + \gamma^n \hat{Q}^\pi(s_{t+n}, a_{t+n}) - Q_\phi(s_t, a_t)\right)^2\right]$$

The total critic loss becomes:

$$\mathcal{L}_{\text{Critic}}(Q_\phi) = \mathcal{L}_1(Q_\phi) + \lambda_n \mathcal{L}_n(Q_\phi) \tag{5.2}$$

Behaviour-Cloning loss.

Following SAC-BC (Nair et al., 2018a), we use a behaviour-cloning loss to train the actor. This loss prevents the policy from deviating too much from the demonstrations, and accounts for sub-optimimality of the demonstrations by filtering out updates where the critic under-performs.

$$\mathcal{L}_{\text{BC}}(\pi_\theta) = \sum_{i \in \text{Demo}} ||\pi_\theta(s_i) - a_i||_2^2 \mathbb{1}_{Q_\phi(s_i, a_i) > Q_\phi(s_i, \pi_\theta(s_i))}$$

The total actor loss becomes:

$$\mathcal{L}_{\text{Actor}}(\pi_\theta) = \mathcal{L}(\pi_\theta) + \lambda_{\text{BC}} \mathcal{L}_{\text{BC}}(\pi_\theta) \tag{5.3}$$

Modified reward derived from Advantage Learning.

Following SAIL (Ferret, Pietquin, and Geist, 2020), we use a modified reward that promotes self-imitation. The idea is to increase the current action-value for actions whose returns are unexpectedly good. The new reward can also be smaller than the original reward, compensating also for negative experiences. Let $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ be the return-to-go at each time-step. The modified reward is:

$$\tilde{r}(s_t, a_t) = r(s_t, a_t) + \lambda_{\text{SAIL}} \left[\max\left(G_t, Q_{\phi'}(s_t, a_t)\right) - \hat{V}(s_t) \right] \tag{5.4}$$

For DDPG, since the policy is deterministic, we can directly use $\hat{V}(s_t) = Q_{\phi'}(s_t, \pi_{\theta'}(s_t))$ as the value-function estimator. For SAC, we use $\hat{V}(s_t) = V_{\psi'}(s_t)$ where V_ψ is a separate network. Since the first version of the SAC algorithm had a value-function network, we choose the same loss function to train V_ψ:

$$\mathcal{L}(V_\psi) = \mathbb{E}_{s \sim \mathcal{D}} \left[\left(V_\psi(s) - \mathbb{E}_{a \sim \pi_\theta(\cdot|s)} \left[Q_\phi(s, a) - \alpha \log \pi_\theta(a|s) \right] \right)^2 \right]$$

The new reward $\tilde{r}$ replaces the standard reward in the Bellman equation for the critic update. For instance, for DDPG the critic loss becomes:

$$\mathcal{L}(Q_\phi) = \mathop{\mathbb{E}}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}} \left[\left(\tilde{r}_t + \gamma \hat{Q}^\pi(s_{t+1}, a_{t+1}) - Q_\phi(s_t, a_t) \right)^2 \right]$$

Algorithm 2: STIR^2

Require: L amount of steps to relabel.
Set b according to equation 4.2.
Initialize buffer with demonstrations, set reward $r = b$ for the last L non-final transitions.
Initialize empty episode.
while *not converged* **do**
  sample a batch $\mathcal{B}$ from the buffer
  if $b > 0$ **then**
    $b \leftarrow$ decay_rule(b) according to equation 4.3
  else
    set $r = 0$ for all non-final transitions in $\mathcal{B}$
  end
  do off-policy RL update (one step of gradient descent following equations 5.2, 5.3 and 5.4)
  if *len(episode) == 0* **then**
    collect one episode
    if *episode is successful* **then**
      set $r = b$ for the last L non-final transitions
  pop a transition (s, a, s', r) from the episode and add it to the buffer
end

5.5 Experiments

5.5.1 Setup

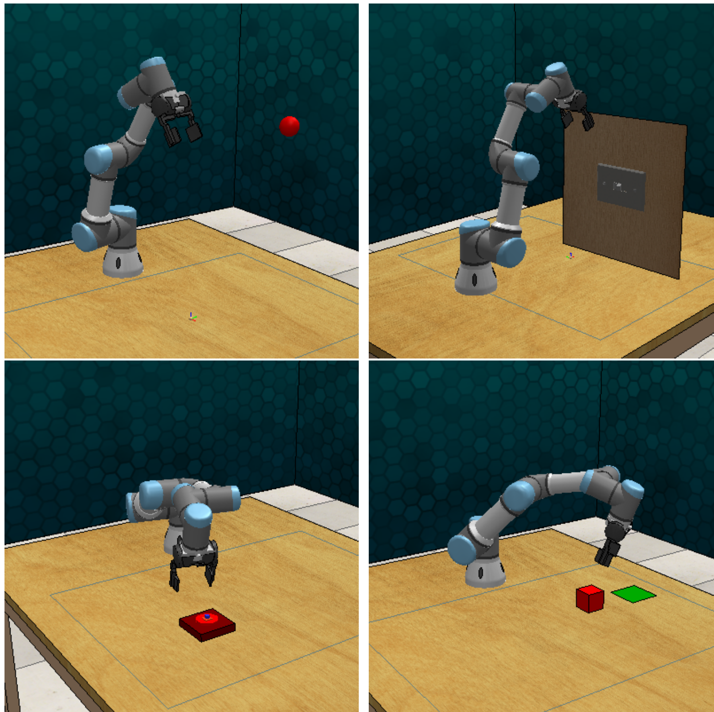


FIGURE 5.1: Snapshots from each RLBench task: reach target, flip switch, push button, slide block.

We test our method on two different simulation environments: RLBench (James et al., 2020) and Meta-World (Yu et al., 2019). For both environments, an episode ends once the robot has completed the task, or after expiration (from 50 to 100 time-steps depending on the task). The reward is fully sparse, and is equal to +100 if the robot solves the task and 0 otherwise.

RLBench. We evaluate our method on four simulated tasks for a 6-degrees-of-freedom robot manipulator: reaching a ball, pushing a button, flipping a switch, and sliding a block to a target square. The target object (ball, button, switch, block and square) appears randomly at the beginning of each episode within the reach of the robot: the ball appears anywhere in the 3D space, the button, block and square are bound to the tabletop, and the switch is on a wall which appears randomly in the scene facing the robot. The initial position of the target object changes after each episode, but it does not move during an episode. The initial state of the robot is always the same, close to an upright position. The state has $19 + x$ dimensions, 19 from the robot proprioceptive state (joint angles, joint speeds, gripper pose) and x from the task-related information (3D coordinates of each target object). For the more contact-based sliding task, the robot receives 3D end-effector position commands (fixed orientation). For the other tasks, the robot receives 6-dimensional joint speed commands.

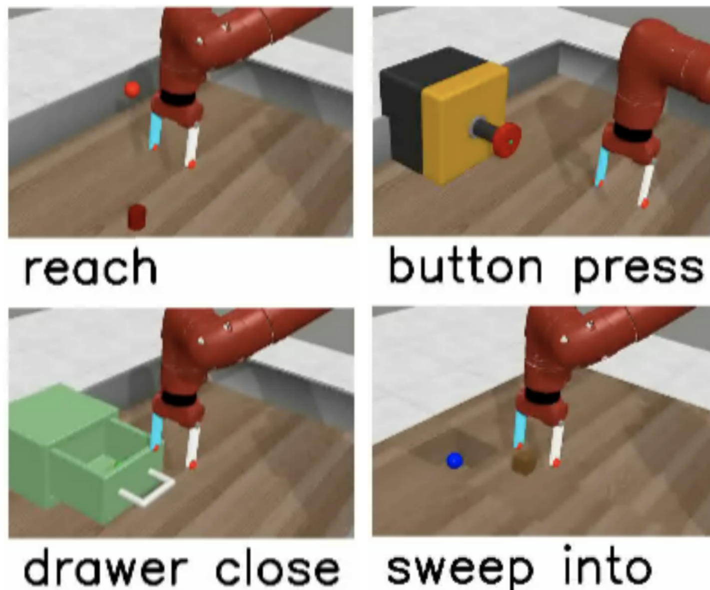


FIGURE 5.2: Snapshots from each MetaWorld task.

Meta-World. We evaluate our method on four simulated tasks for a 7-degrees-of-freedom robot manipulator: reaching a ball, pressing a button, closing a drawer, and sliding a block to a target location. The target object (ball, button, drawer, block) appears randomly at the beginning of each episode within the reach of the robot: the ball appears anywhere in the 3D space, while the drawer, (horizontal) button, and block are bound to the tabletop. We refer to the Meta-World paper for the action and state spaces which we do not change.

For all tasks, success is defined by reaching the end-goal state. In particular, whenever a condition or set of conditions is met (e.g. a proximity condition between the end-effector and the target for the reaching task).

Demonstrations. For RLBench, the expert demonstrations are provided by the environment and rely on OMPL (Sucan, Moll, and Kavraki, 2012) for motion planning. For Meta-World, we first train an RL agent for each task from dense rewards, and retain a batch of its successful episodes at test-time as demonstrations.

Experiments. The basic SAC algorithm without demonstrations is able to solve these tasks on some runs. Our goal is to solve the tasks consistently and reduce the amount of training steps required to do so. We want to answer three questions: How does our method perform? How robust is it to the hyper-parameters? How does it perform under rougher conditions (weaker base algorithm, few demonstrations available, no demonstrations available)?

In order to answer to these questions, we compare our methods SAC-R^2 and SAC-STIR^2 to a simple baseline SAC-Demo, which we define as SAC with demonstrations in the buffer, as well as SAC-fD (Vecerík et al., 2017), SAC-BC (Nair et al., 2018a) and SAC-SAIL (Ferret, Pietquin, and Geist, 2020). We apply the following modifications to all 6 algorithms:

- For all methods except SAC-BC: Single buffer initially filled with 200 demonstrations, and kept thereafter at a ratio of 10% demonstration data. We decide not to set a limit on the number of unique demonstrations available, meaning that around 500-1000 more demonstrations are added to the buffer (depending on the task and number of iterations). For SAC-BC: Two separate buffers, one exclusively filled with demonstrations (with a comparable amount).
- We additionally put 1000 random interactions alongside the demonstrations in the buffer, and pre-train during 3000 iterations before collecting any data. The idea is to have some negative examples during the pre-training phase so that the agent can disriminate between good and bad transitions.
- For all methods except SAC-fD: The data is sampled from the buffer according to prioritized experience replay (PER) (Schaul et al., 2016). For SAC-fD: A modified version of PER which boosts the sampling of demonstration data is used.
- The replay ratio on the collected data is set to 32. Since the batch size is set to 64, the agent takes two environment steps per training step.
- L2 regularization losses on the weights of the critic and the actor.

Nothe that some of these parameters (amount of pre-training steps, amount of initial demonstrations, ratio of demonstrations in the buffer) were fine-tuned on the reaching task with SAC-Demo, while others (replay ratio, batch size) were not thoroughly tested and were chosen with the limitations of the hardware in mind. On top of these changes, SAC-BC uses an auxiliary behaviour-cloning loss and resets some episodes to demonstration states, SAC-fD uses an auxiliary n-step loss and boosts demonstrations in PER, SAC-R^2 uses the reward bonus presented in Chapter 4, and SAC-SAIL uses a modified reward (see (5.4)). For SAC-R^2, we use the scheduled decay presented in equation 4.3. If the success rate doesn't improve over a period of 5000 iterations, the safety decay is applied to completely erase the bonus. We always choose the largest bonus allowed by the condition 4.2. Unless stated otherwise, we set $L = 10$, $\lambda_{\text{SAIL}} = 0.9$, $\lambda_{\text{BC}} = 2$, $\lambda_n = 1$ and $n = 5$.

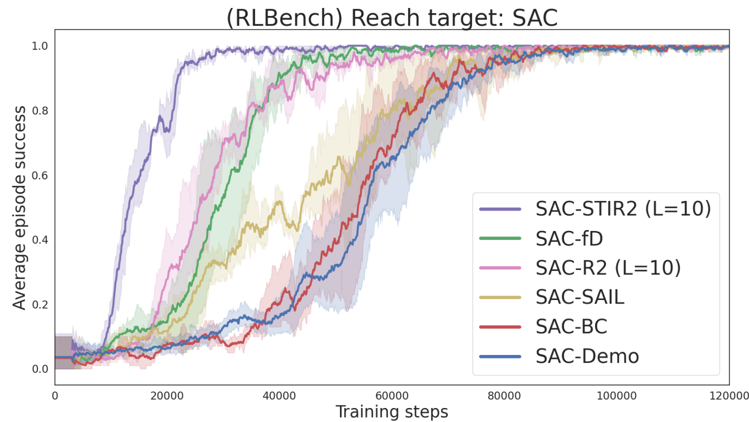
All the results are smoothed with a rolling window of 100 episodes, and the standard error is computed on three random seeds.

5.5.2 Results

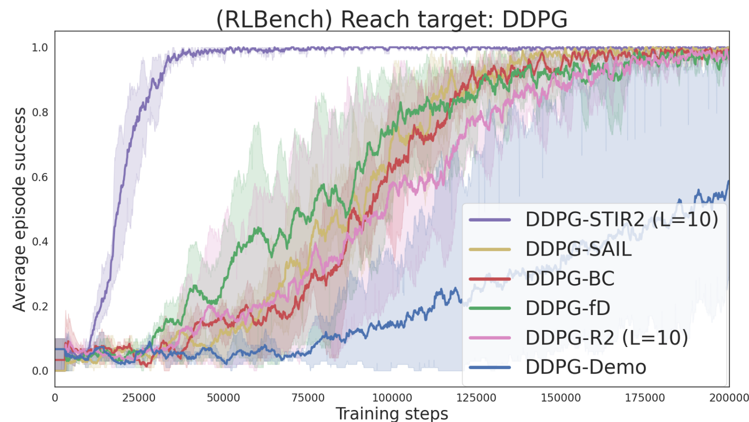
The results are reported in Table 5.1 and show that SAC-R^2 has comparable results to the other baselines. Overall, the best methods are SAC-fD in the RL-Bench tasks, and SAC-BC in the Meta-World tasks. However, our full algorithm SAC-STIR^2 does outperform all baselines in all tasks.

	SAC-Demo	SAC-BC	SAC-SAIL	SAC-fD	SAC-R2	SAC-STIR2
RlBench						
Reach	1.0 / 95k	1.0 / 95k	1.0 / 90k	1.0 / 70k	1.0 / 70k	**1.0 / 50k**
Push	.60 / 170k	.70 / 170k	.95 / 140k	.95 / 140k	.95 / 140k	**.95 / 110k**
Flip	.40 / 160k	.80 / 160k	.90 / 160k	**.95 / 120k**	.90 / 120k	**.95 / 120k**
Slide	.78 / 300k	.78 / 300k	.80 / 300k	.80 / 300k	.78 / 300k	**.82 / 300k**
MetaWorld						
Sweep	0.0 / 50k	.40 / 50k	.30 / 50k	.30 / 50k	.30 / 50k	**.42 / 50k**
Close	1.0 / 15k	1.0 / 10k	1.0 / 10k	1.0 / 10k	1.0 / 15k	**1.0 / 5k**
Press	.80 / 25k	**1.0 / 5k**	.85 / 25k	.90 / 15k	.85 / 15k	**1.0 / 5k**
Reach	.90 / 25k	1.0 / 20k	.90 / 25k	1.0 / 15k	.90 / 25k	**1.0 / 10k**

TABLE 5.1: Summary of the training results (Figures 5.3, 5.4, 5.5). The values correspond to the rounded final training success rate and the amount of iterations needed to converge. The column-wise best results are marked in bold.



(A)



(B)

FIGURE 5.3: Learning curves for the RLBench reaching task.

RLBench Results

The results (Figures 5.3a, 5.4a, 5.4b, 5.4c) show that SAC-STIR^2 is significantly better than the other methods on the *reach target* and *push button* tasks: It

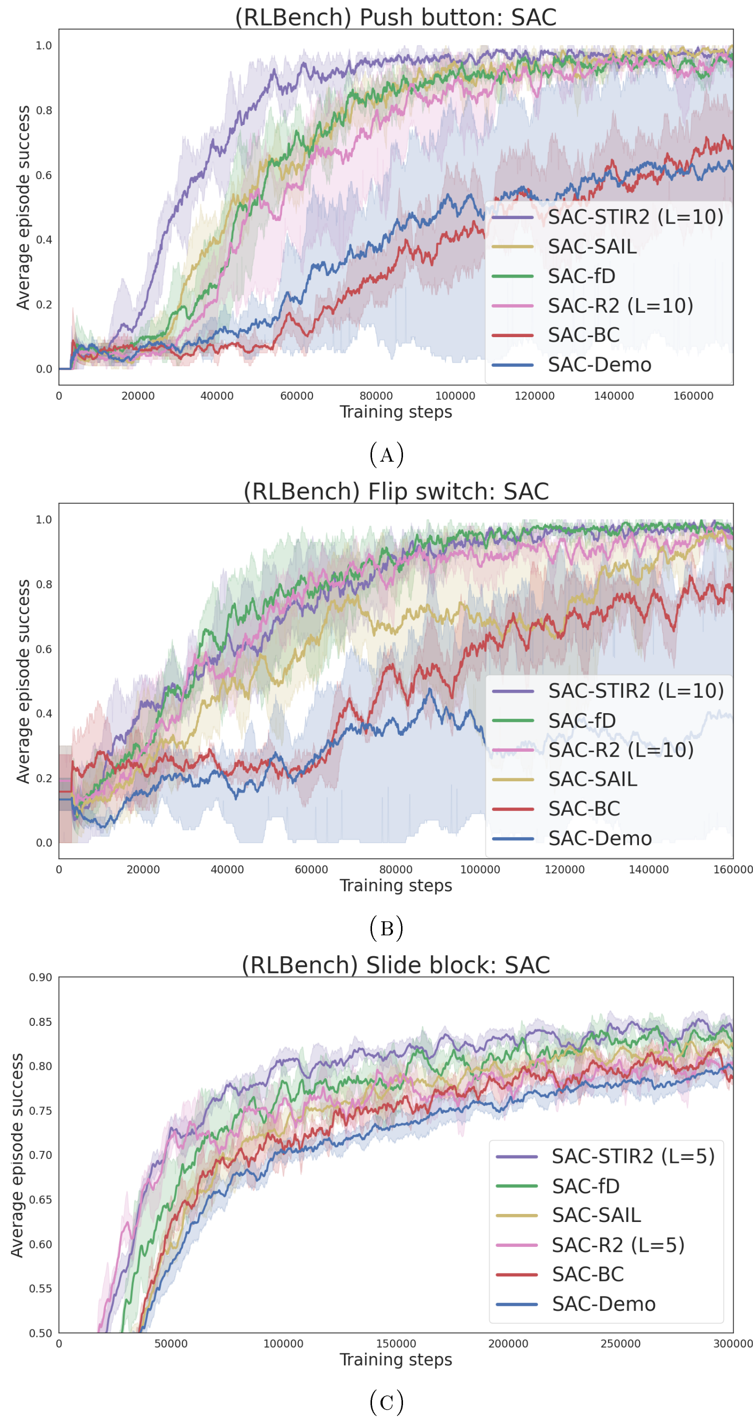


FIGURE 5.4: Learning curves for three RLBench tasks.

is $\sim$ 1.5 times faster to reach a relative success rate of 90%. It also has the best performance on the *flip switch* task, but is tied with both SAC-R^2 and SAC-fD. On the more difficult *slide block* task, SAC-STIR^2 is able to reach a higher success rate than the other methods ($\sim$ 84%, compared to $\sim$ 82% for second-best SAC-fD and $\sim$ 78% for SAC-Demo).

From SAC to DDPG: Figure 5.3b shows that DDPG-R^2, DDPG-BC, DDPG-fD and DDPG-SAIL have similar performances in the *reach target* task, while

DDPG-Demo struggles and DDPG-STIR^2 outpaces everybody by a factor of 4 to reach a relative success rate of 90%.

Note that the hyper-parameter L is always chosen equal to 10, except for the RLBench sliding task (5 instead). The action space used for that task is based on end-effector position commands, since none of the methods made any progress with the lower-level joint-speed commands. Therefore, the episodes tend to be much shorter, as the task can be completed in just a few steps. $L = 5$ is more consistent with the other experiments in terms of average ratio of relabeled transitions per episode, even if the results were only marginally better than $L = 10$.

Meta-World Results

The results (Figure 5.5) show that SAC-STIR^2 is also the best method across the four Meta-World tasks. It is faster to converge on the easier tasks, and attains a higher final performance (almost one point higher) on the more difficult *sweep into goal* task. However, the overall results are only marginally better than those of SAC-BC, which is the strongest among all baselines. Surprisingly, SAC-BC was one of the weakest baselines on the RLBench tasks. We believe this is tied to the origin of the demonstrations. Indeed, SAC-BC's behaviour-cloning loss pushes the agent to closely imitate the expert's actions, which is easier if the expert is also an RL agent, rather than a motion-planning expert who might exhibit different tendencies.

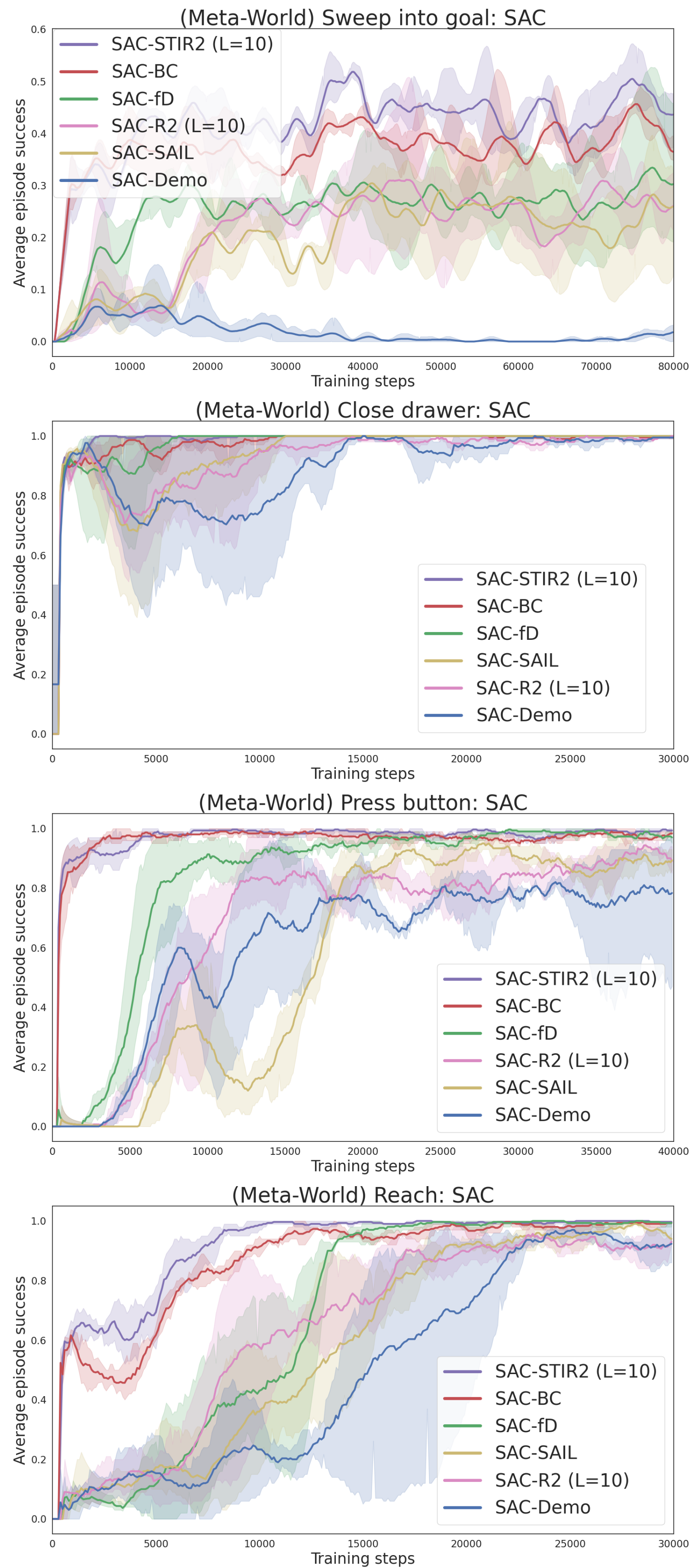


FIGURE 5.5: Learning curves for four Meta-World tasks.

5.5.3 Ablation study

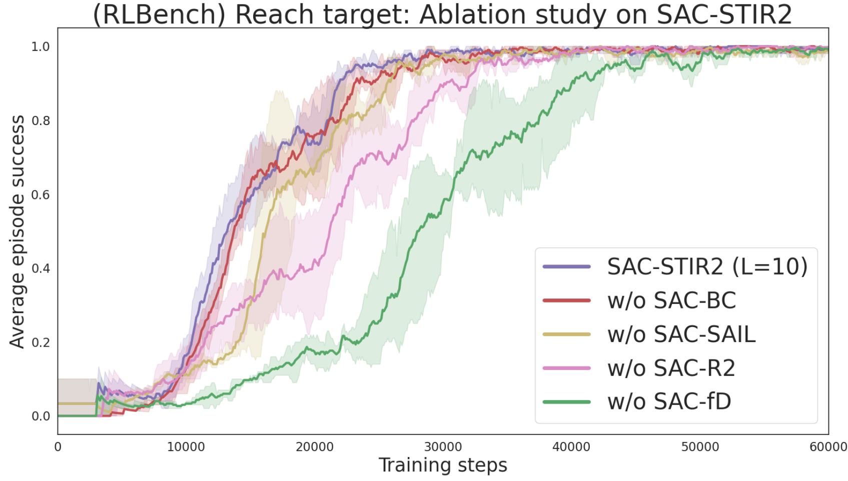
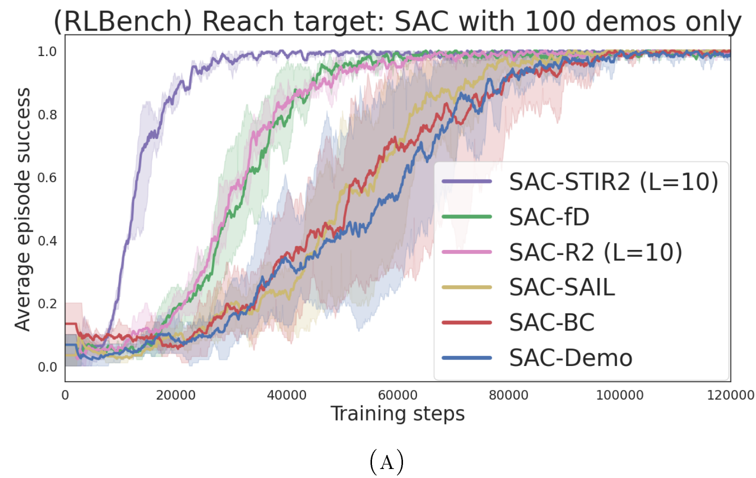


FIGURE 5.6: Ablation curves on SAC-STIR^2.

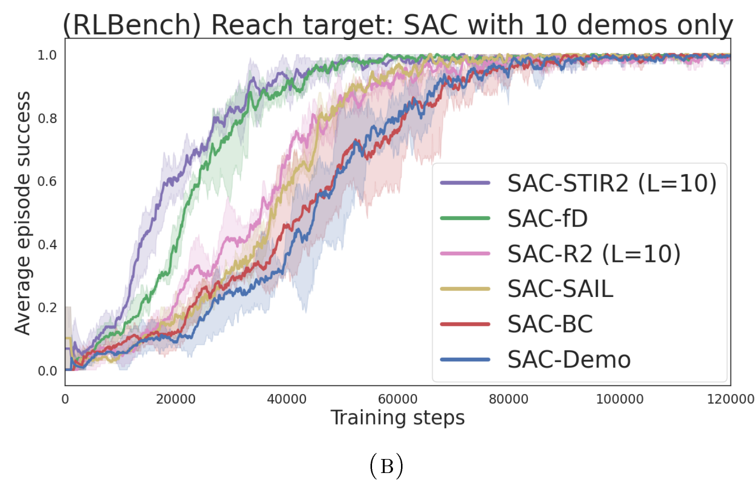
Ablation study on SAC-STIR^2. Figure 5.6 shows that the contributions of all four methods (SAC-R^2, SAC-BC, SAC-fD and SAC-SAIL) are important to SAC-STIR^2, since removing either of them causes the performance to drop. For this task, SAC-fD is clearly the most important element, followed by SAC-R^2. SAC-BC performs particularly bad on this task (Figure 5.3a), so it is unsurprising that it has the lesser impact.

Low-data regime. The previous results were obtained with a large amount of demonstrations, which is unrealistic for real-world applications. What happens if we limit the amount of available demonstrations to just 100? Figure 5.7a shows that the results are very similar (but slightly worse for all methods) to the high-data regime (Figure 5.3a). What about 10 demonstrations? The results shown in Figure 5.7b are also similar, but our method SAC-R^2 actually has the highest drop in performance, while SAC-fD appears to be responsible for most of SAC-STIR^2's success under these circonstances. One possible explanation is that SAC-R^2 is the only method that explicitely modifies the transitions that come from the demonstrations added to the buffer (by giving them a bonus), so it might be more sensible to the degree in which they cover the entire state-action space.

Learning without demonstrations. Finally, can SAC-R^2 also improve the performance when no demonstrations are available, by just relabeling successful episodes? Figure 5.8 shows similar results to those with demonstrations in the buffer: SAC-STIR^2 performs the best, while the weakest SAC baseline is only able to solve the task on some runs. Note that SAC-BC is not included in the figure because its behaviour-cloning loss relies on demonstration data.



(A)



(B)

FIGURE 5.7: Learning curves exploring low-data regimes on the RLBench reaching task.

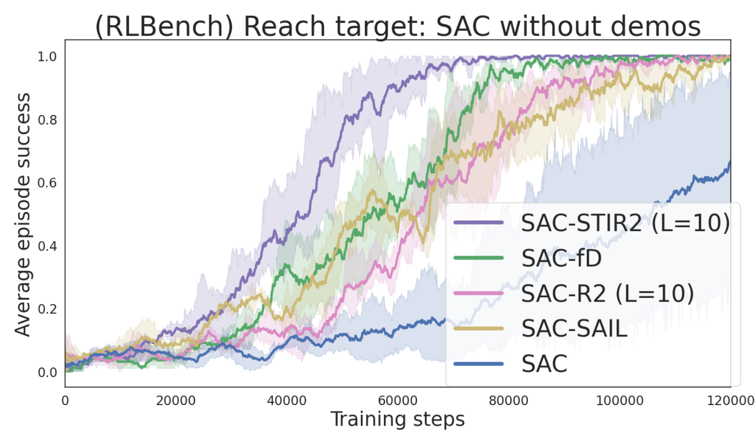


FIGURE 5.8: Learning curves without demonstrations on the RLBench reaching task.

5.6 Additional Results

As described in Section 5.3, the baselines we introduced actually contain other elements that we decided not to integrate into our final algorithm SAC-STIR^2. Why is so? In this section we carry a study to isolate the impact of each element on the basic SAC-Demo baseline.

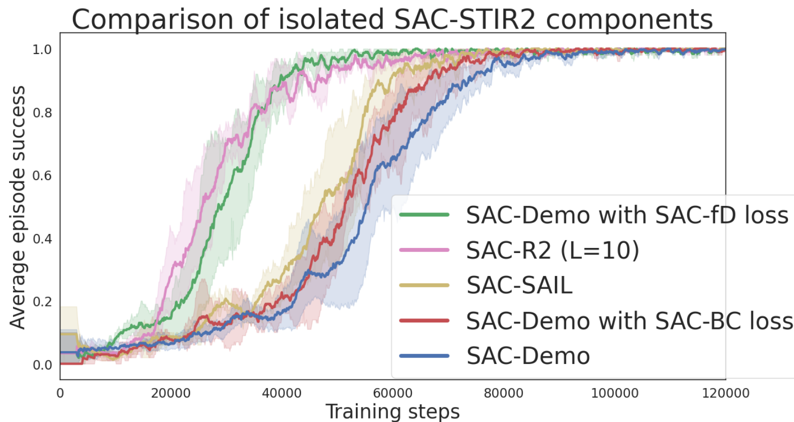


FIGURE 5.9: Curves of the isolated SAC-STIR^2 components on SAC-Demo.

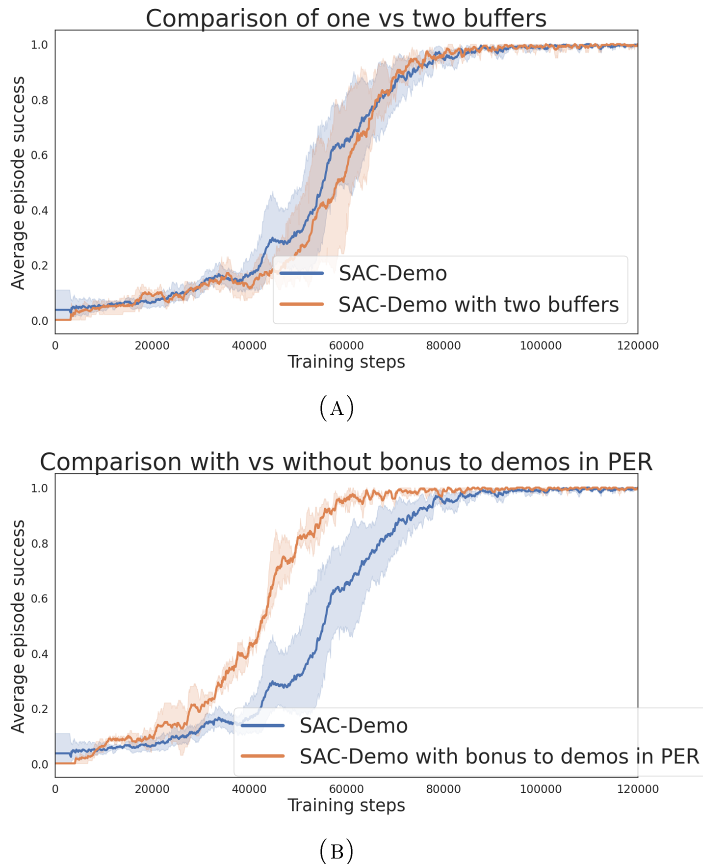
As introduced in Section 5.4, SAC-STIR^2 has four main components: additional n-step critic loss 5.2, additional behaviour-cloning actor loss 5.3, modified self-imitation reward 5.4, and our reward-relabeling method presented in Chapter 4. Figure 5.9 shows the impact of each element on isolation on top of the baseline SAC-Demo. As expected, all four elements provide a boost in performance.

One vs two buffers.

The main difference between these two approaches is the ratio of demonstrations in the sampled batches. With two buffers, this ratio can be fixed, for instance SQIL (Reddy, Dragan, and Levine, 2020) uses 50% and SAC-BC roughly 10%. With one buffer, the ratio varies from batch to batch and is on average equal to the ratio of demonstration data in the buffer if the sampling is uniform. Since we sample according to PER, the ratio is actually slightly higher, but the results (Figure 5.10a) show that the two configurations yield almost identical results. One practical advantage of using two buffers over one is that we can easily have two different replay ratios for the demonstration data and online data, but in our experiments we introduced an equivalent amount of demonstrations in both settings to keep things fair.

Prioritized replay.

Following SAC-fD, we modify the PER strategy with two additional terms: a term representing the actor loss, and a constant bonus applied to all transitions coming from demonstrations. We recall that in PER, the probability of sampling a particular transition is proportional to its priority p_i, which is commonly computed from the transition's temporal difference (TD) error δ_i,



(A)

(B)

FIGURE 5.10: Additional curves (buffer configuration + PER) on SAC-Demo.

for instance $p_i = \delta_i^2 + \epsilon$ where ϵ is a bonus given to all transitions. SAC-fD adds a square term representing the actor loss and a second bonus ϵ_D given only to the demonstrations. We use the same hyper-parameters as in SAC-fD.

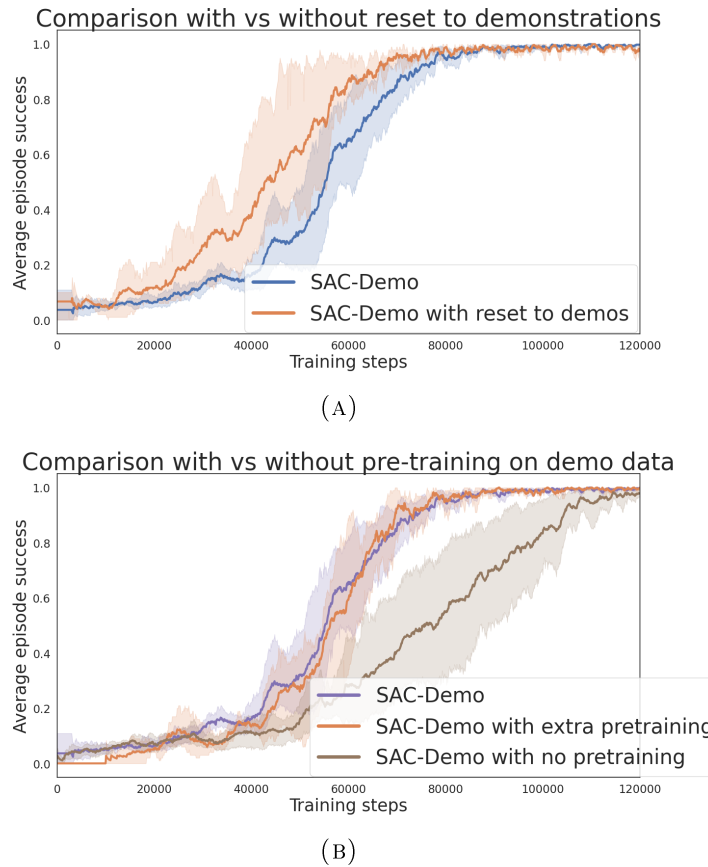
From our limited experiments, this new strategy didn't have a great impact in terms of ratio of demonstration data in the sampled batches. With both PER and the modified PER the ratio is close to 11% (we recall that 10% of the transitions in the buffer come from demonstrations). However, the results (Figure 5.10b) do show a major initial boost during training.

Reset to demonstrations.

Following SAC-BC, we reset some episodes (10%) to a demonstration: the position of the target ball is the same as in the demonstration, and the initial state of the robot is randomly sampled from the demonstration's states. This should act as a form of curriculum learning and lead to more successful episodes early on. The results (Figure 5.11a) do show an initial boost in the learning process, but the improvement isn't too significant.

Pre-training on demonstrations.

Pre-training on demonstrations is one of the most common approaches to



(A)

(B)

FIGURE 5.11: Additional curves (pretraining + resetting to demos) on SAC-Demo.

leverage demonstrations in the literature, and does seem like a good idea according to the results (Figure 5.11b). However, too much pre-training (10000 iterations on 600 demonstrations rather than 3000 iterations on 200 demonstrations) also decreases performance. One possible explanation is that the agents winds up forgetting what it initially learnt from the demonstrations when it first encounters subpar trajectories from its collected experience. This drop in performance from excessive pre-training appears even clearlier in Figure 5.12 with STIR^2 as the base algorithm.

Conclusion

The results of the study show that the n-step loss from SAC-fD, our approach SAC-R^2, the behaviour-cloning loss from SAC-BC, and the modified reward from SAC-SAIL are the four improvements with the greatest increase in performance over the baseline SAC-Demo. These four components are the 4 main ingredients of our STIR^2 algorithm presented in Section 5.4.

To further validate these results, we evaluate STIR^2 in Figure 5.12, and show that none of the rejected components particularly move the needle anymore. However, these results come from a single task, and we don't know how they would translate to other tasks, in particular more complex tasks where we try to

increase the success rate rather than the sample efficiency. Although we cannot draw any categorical conclusions from these studies, they do allow us to justify our choices for the components integrated into STIR^2, where the goal was to balance simplicity and performance.

5.7 Discussion

We propose Self and Teacher Imitation by Reward Relabeling (STIR^2), a new algorithm which combines our Reward Relabeling (R^2) method from Chapter 4 with the n-step loss from SAC-fD, the behaviour-cloning loss from SAC-BC, and the modified reward from SAIL. We show that these methods stack together, as the best results were obtained with SAC-STIR^2 on all tasks. Similar results were obtained with DDPG as the base algorithm, and in theory it could be implemented on top of any continuous-action off-policy RL algorithm.

The questions raised at the end of Chapter 4 also apply to the full algorithm STIR^2, so addressing those issues remains a possible path for future work. Other baselines based on more recent works could also be compared against and potentially integrated into the algorithm, in particular a pure imitation-learning algorithm such as Implicit BC (Florence et al., 2022). Finally, it would be interesting to test our method with a Q-learning-type base algorithm on a task with discrete actions.

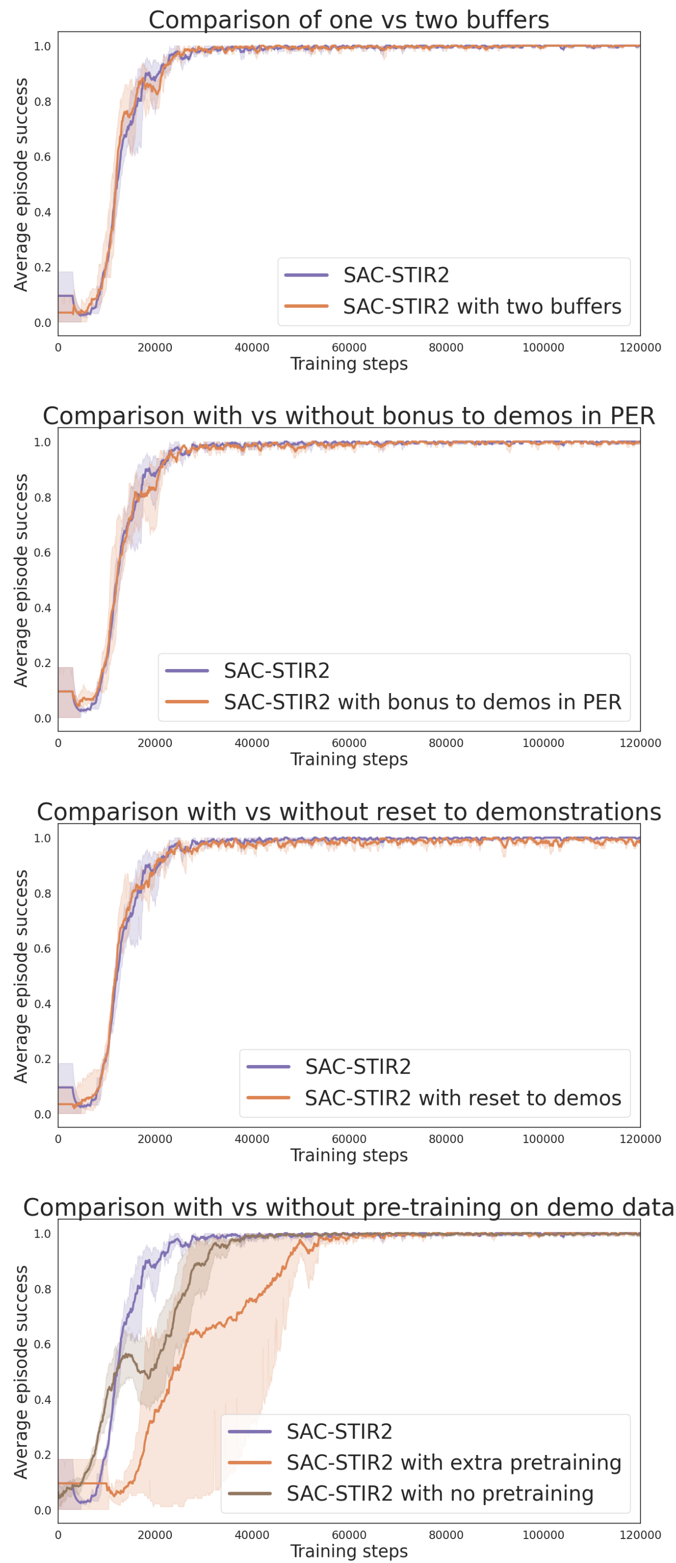


FIGURE 5.12: Adding extra components to SAC-STIR^2 doesn't result in further improvements.

Chapter 6

Pre-trained Vision Encoder for Data-Efficient Reinforcement Learning

6.1 Introduction

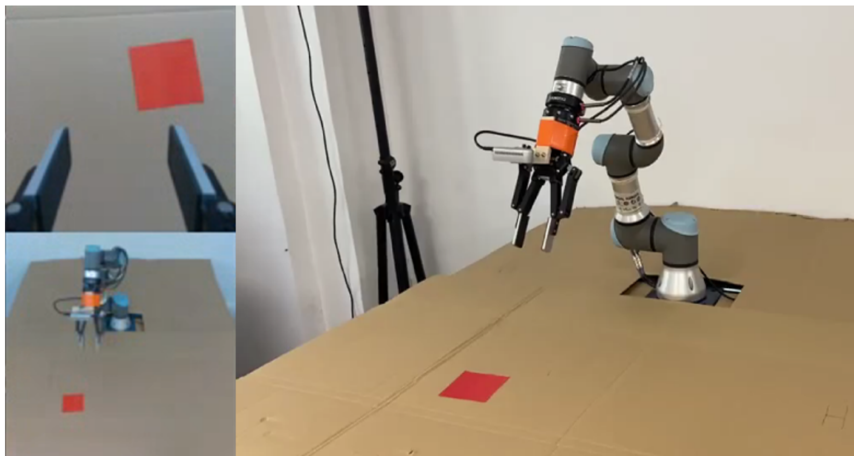


FIGURE 6.1: Real-world setting with a Universal Robots' UR3 model. The inputs to the agent are the views from two cameras as shown in the left.

In the previous chapters, all the experiments were done entirely from a scalar state, where the main limitation is that the agent needs to be granted access to information that is not easily available in the real world, such as the exact position of every object in the scene. In this chapter we move onto the more realistic setting of vision-based reinforcement learning. As we will see, demonstration data can also be used to increase sample efficiency in this domain.

To tackle this problem, we design a two-stage (optionally three) training pipeline (see Figure 6.2): first learn a visual representation of the scene by pre-training an encoder from multiple supervised computer-vision objectives, then train a reinforcement-learning agent which can focus solely on solving the task. Despite all the data being collected in simulation, the experiments include one

sim-to-real example to show that these techniques can translate to real-world controlled environments.

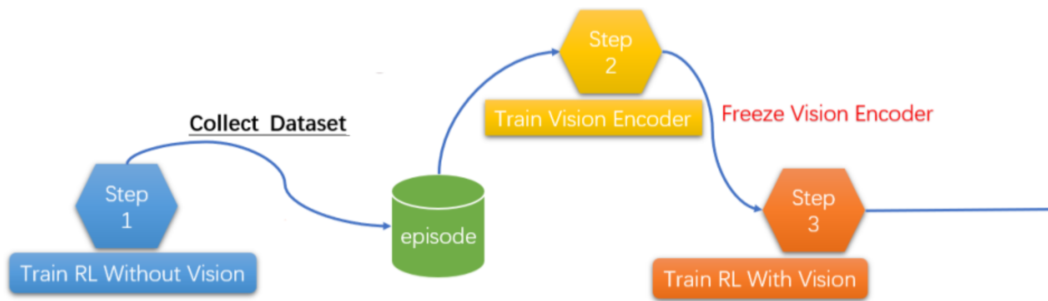


FIGURE 6.2: Proposed pipeline.

The results show that not only is our method more sample-efficient than any of the end-to-end baselines, but it also achieves a higher final success rate in all tasks, including one task where the baselines are unable to make any progress.

6.2 Background - Challenges in Robotics

Solving robotic-manipulation tasks with RL is no easy endeavor. However, multiple tasks have been attempted and solved in the past:

- Simple tasks: non-prehensile (reaching, pushing, button-pressing, pouring) and prehensile (grasping, picking and placing, door opening).
- Sequential tasks: block stacking (Nair et al., 2018b), dish placing (Sermanet et al., 2018), table clean-up (Xu et al., 2018)...
- Contact-rich tasks: peg insertion (Lee et al., 2019), sweeping with a tool (Xie et al., 2019; Schmeckpeper et al., 2019)...
- Complex-dynamics tasks: pouring water (Schenck and Fox, 2017), playing table tennis (Muelling et al., 2012), throwing objects (Ghadirzadeh et al., 2017; TossingBot: Zeng et al., 2019)...

In terms of RL algorithm choice, there is a trade-off between sample efficiency and model complexity. For robotic tasks, the optimal policy often has fewer parameters than the optimal value function or optimal dynamics model. However, methods like policy gradient that look directly for such a policy are the least sample-efficient ones, while model-based methods are the most sample efficient.

In terms of design choices, robotic tasks are often best modelled as POMDPs with finite horizon episodic tasks. There are lots of other choices to make: discretize the state/action spaces or not, choice of time-step, initial state distribution, state representation, action space, action repeat, episode termination, torque limits...

Let's look in more detail at the main challenges that make robotics such a tricky field for RL. The main sources for this section are the surveys (Andrew Bagnell, 2014) and (Kroemer, Niekum, and Konidaris, 2019).

Challenges

- *Curse of dimensionality.* The state and action spaces are continuous for robot manipulation tasks. We have seen several algorithms that can handle continuous spaces, like policy gradient or actor-critics. We could also opt for a (smart) discretization of the space, or aggregations (form clusters of states/actions and consider their transitions). We refer to (Sigaud and Stulp, 2019) for a comprehensive overview of the main families of methods handling continuous action spaces.

  One approach especially suited for robotics is to choose a particular structure for our policy that best adapts to our problem: motor primitives (e.g. Muelling et al., 2012), locally linear controllers (e.g. Levine et al., 2016), hierarchical structure (learned high-level actions, hard-coded low-level actions)...

- *Real-world problems.* Many potential issues related to the robot's hardware: the dynamics of a robot can change over time due to external factors, there is measurement noise from the sensors, actions may be delayed and its effects can appear several time steps later (latency, which violates the MDP assumption of synchronous execution), the upper bound on the rate of temporal discretization is determined by the robot's sampling frequency... Also, the main problem is that real-world samples are expensive in terms of time and labor.

  However, recent work towards large-scale robotics has proven to be viable. For instance, in QT-Opt (Kalashnikov et al., 2018) they exploit a dataset of about 800 robot hours collected over the course of four months.

- *Exploration.* Exploration is particularly challenging for robotic manipulation. Random actions can be a suitable exploration strategy for other tasks such as navigation, but random torques on the joints will spend most of the time doing useless exploration and not interacting with the environment. Also, simply adding random noise to the actions may produce jerky motions that could damage the robot.

  For instance, (Ibarz et al., 2021) propose different options to smooth out the actions and avoid the jerkiness: low-pass filters, temporal-coherent noise, reward shaping...

- *Safety.* Other than safe exploration, many other safety measures need to be implemented. Typically, many model-based or heuristic-based safety checks are used, such as self-collision detection, power/torque limit, collision detection from force-torque sensors... More sophisticated methods can also be used.

For instance, (Ibarz et al., 2021) use a geometric model of the robot and the world in order to reject actions that violate kinematic or geometric constraints.

To address these problems, we dispose of two main tools:

- *Prior knowledge* from other tasks (meta-learning, multi-task learning...), from demonstrations (imitation learning), from task structure (hierarchical RL)... These can alleviate the challenge of *goal specification* (specifying a suitable reward function), which is particularly difficult in real-world settings.
- *Simulation*. Can we transfer a policy/model learned in simulation to the real robot? Oftentimes, a model of the dynamics will transfer more easily, especially if we reduce the simulation biases by introducing some stochasticity to the model, or by explicitly training for robustness (referred to as under-modeling). Generally, the actuator dynamics and the lack of latency modeling are the main causes of the model error.

6.3 Background - Simulation to Reality

Transfer learning is a well-studied field in computer vision, and it has been used in robotics to address the *sim-to-real* problem: from simulation to the real world. The most common techniques usually involve GANs or other generative networks, where the goal is to update the distribution of simulated data to match that of the real world. These techniques are known as *domain adaptation*.

Rather than working towards a specific target domain, which usually requires collecting data from that domain (in this case the real world), more recent methods aim for a more general transfer. The idea is to train a model that is able to generalize across an entire distribution of simulated environments, which hopefully covers the desired test environment (again, the real world). This techniques are known as *domain randomization*. Let's look at a few examples.

For instance, (Pashevich et al., 2019) propose to augment synthetic depth images through a set of random transformations, while (Tobin et al., 2018) focuses on generating random synthetic objects for grasp planning. In (James, Davison, and Johns, 2017), domain randomisation is used to augment robot trajectories to train a CNN mapping observed images to velocities, and achieve to do so even under dynamic lighting conditions.

In (Horváth et al., 2022), domain randomization is used for object detection by generating synthetic objects. For the trickier detection classes where randomized synthetic data is not enough, they show that as little as one single real-world image can be added to considerably improve the results.

In (James et al., 2019) they introduce Randomized-to-Canonical Adaptation Networks (RCAN). Rather than feeding an agent a wide array of diverse data so that it can generalize to any situation, they first train an image-conditional GAN to transform those randomized images into "plain" (canonical) images. When

the agent is deployed, the real-world images are also transformed into their canonical versions, which means the agent sees no difference from its training episodes.

Like the previous methods, in this work we are interested in the visual aspect of sim-to-real, dealing with how to process real images at test time after training with simulated images. However, domain randomization is not limited to images, it can be used to randomize the dynamics of the environment as well as other factors. Let's look at some examples.

In (Kumar et al., 2021), they introduce Rapid Motor Adaptation (RMA), where a legged robot learns how to walk on difficult terrains. The idea is to train a supervised adaptation module, whose goal is to predict the extrinsics of the environment (a latent representation of the environmental parameters). This information is fed into the policy as an additional input, and the policy is trained accros a wide variety of simulated environment, which should cover the conditions encountered in the real world.

In (Pinto et al., 2017), they introduce Robust Adversarial RL (RARL). Both modeling errors and differences in training and test scenarios can be viewed as extra disturbances in the system. The idea is to train an agent in the presence of these disturbances. Rather than sampling all possible disturbances, they jointly train a second agent (the adversary, rewarded only for the failure of the main agent) that applies disturbance forces that incorporate domain knowledge (e.g. suddenly change a physical parameter like frictional coefficient or mass).

Finally, the distribution of data where we want to bridge the sim-to-real gap isn't necessarily the agent's state space. Recent works use domain adaptation/randomization for other more original use-cases. Let's look at one such example.

In (Liu et al., 2018), they introduce Context Translation, which achieves a wide range of real-world robotic tasks from videos of a human demonstrator (sweeping, pushing objects...). The goal is to learn a model that can convert a demonstration from one context (a third person viewpoint and a human demonstrator) to another context (a first person viewpoint and a robot), and then learn a policy that optimally tracks the translated demonstration in the target context. To do so, a translation model looks at a single observation from a target context j, and predicts what future observations in that context will look like by translating a demonstration D_i from a source context i.

6.4 Related work

Representation Learning is particularly challenging in RL, because it is intrinsically tied with the agent's exploration. Exploring the environment provides the dataset to learn the representations, but effective non-random exploration is difficult without good representations. We refer to the survey (Lesort et al.,

2018) for a thorough review of the algorithms and objectives that have been used in recent years. Let's look at some examples.

SAC+AE (Yarats et al., 2021b) augments SAC with a regularized deterministic autoencoder, which is updated with gradients from the reconstruction and soft Q-learning objectives.

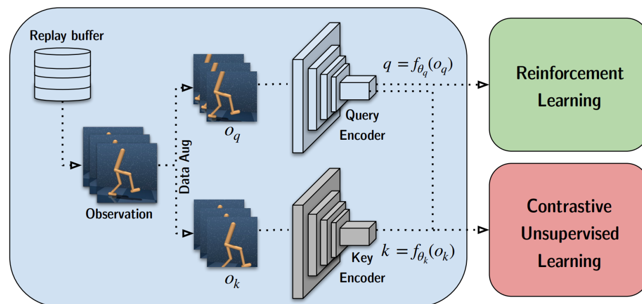


FIGURE 6.3: CURL: Contrastive Unsupervised Representations for Reinforcement Learning (source: Srinivas, Laskin, and Abbeel, 2020)

Contrastive Unsupervised Representations for RL (CURL) (Srinivas, Laskin, and Abbeel, 2020) (see Figure 6.3) extracts high-level features from raw pixels using contrastive learning, and simultaneously performs off-policy control on top of the extracted features. Trajectories sampled from the replay buffer are data-augmented twice to form key-query pairs, and then encoded with the key and query encoders. While the full key-query pairs are used to compute the contrastive-learning objective, only the queries are passed on to the RL algorithm.

Proto-RL (Yarats et al., 2021a) trains two agents simultaneously. An exploration agent is trained from an intrinsic reward to learn task-agnostic representations: a set of vectors that will act as a basis for representing observations. The projected observations are fed to the main RL agent, which receives the task reward.

Policy Adaptation during Deployment (PAD) (Hansen et al., 2020) operates in two steps. During the first training, the RL and self-supervised objectives are jointly optimized. During the second training (referred to as adaptation), only the self-supervised objective is optimized in the test environment. They test two different self-supervised objectives: learning an inverse-dynamics model (usually better for RL tasks that require more motor control) and learning to predict the rotation angle applied as augmentation to the original image (usually better for navigation tasks).

Active Pre-Training (APT) (Liu and Abbeel, 2021) designs an intrinsic reward that maximizes entropy in an abstract representation space, and can be used to pre-train a policy that can be later fine-tuned for downstream tasks. This reward decreases to 0 as most of the state space is visited, and it is derived from a nonparametric particle-based entropy estimator $H_k(z) \propto \sum_{i=1}^{n} \log ||z_i - z_i^{(k)}||$, where $z^{(k)}$ is the k-th closest neighbor, and the latent

representation z is obtained from data augmentation and contrastive learning within each batch of transitions sampled from the replay buffer.

6.4.1 Baselines

For our baselines, we focus on two of the aforementioned algorithms:

- SAC+AE (Yarats et al., 2021b).
- SAC-CURL (Srinivas, Laskin, and Abbeel, 2020).
- SAC+AE-CURL: A combination of the two methods above.
- SAC-Demo. Simple baseline introduced in Chapter 4. The base SAC algorithm with demonstrations in the replay buffer. Note that all baselines use SAC-Demo rather than SAC as the base algorithm, but we omit the Demo in the nomenclature for ease of reference.

6.5 Training the Encoder

As shown in Figure 6.2, our method consists in two main steps (steps 2 and 3 in the figure). First, we train an encoder on a collection of supervised computer-vision objectives. Then, we freeze the weights of the encoder and use it as the backbone of all RL networks to learn a task. Let $s = (s_{\text{image}}, s_{\text{proprioception}})$ the environment state. The goal of the first step presented in this section is to learn a more compact representation $\tilde{s}_{\text{image}}$, and the goal of the second step is to train an RL agent from the new states $\tilde{s} = (\tilde{s}_{\text{image}}, s_{\text{proprioception}})$.

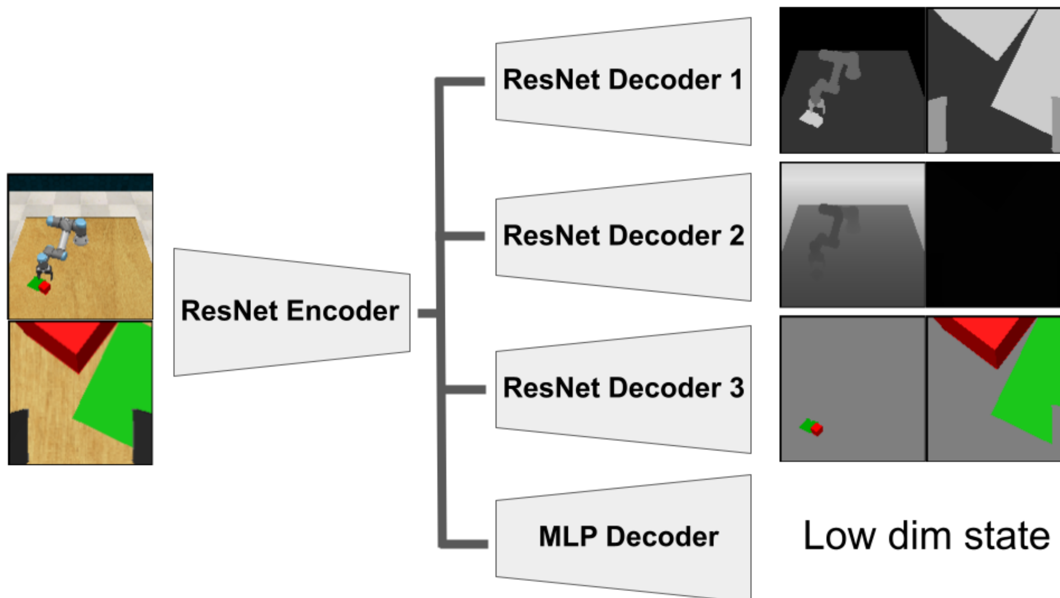


FIGURE 6.4: Our four computer-vision objectives to learn an encoded representation of the scene.

Before training the encoder, one preliminary step is required (step 1 in Figure 6.2): obtaining a dataset to train on. We choose to gather the data by collecting

episodes with a pre-trained RL agent, but any source of data that sufficiently covers the entire state space would work, such as expert demonstrations. This agent was trained in a much simpler setting that the one we are interested in, as it was trained with dense rewards and without image inputs (instead, it has access to the full state of the environment, namely the coordinates of the objects in the scene).

Similar to works such as *Mid-Level Visual Representations* (Chen et al., 2020), we consider multiple computer-vision objectives in order to learn good encoded features. However, rather than having one encoder per objective, we train a single encoder on all the objectives at once. These are the objectives we train on simultaneously: Image Segmentation, Depth Prediction, Auto-encoding, and State Regression. A simple ResNet decoder is used for the first three objectives, and a simple MLP for regressing the state of the environment.

In order to be able to solve manipulation tasks, it is most important to accurately process the scene and the objects within it. We apply the following modifications to the standard computer-vision objectives in order to prioritize the objects over the other visual information:

- Image Segmentation: We simply give a bigger weight to the object class in the cross-entropy loss (10 for the object class, 1 for the rest). We consider 5 different tasks for segmentation: Floor and Walls, Table, Robot, Gripper, Objects. Let $c = \mathrm{Im}_i^{\mathrm{segm}}(n,m)$ the target class of pixel (n,m) in image i, and w_c the corresponding weight:

$$\mathcal{L}_{\text{segmentation}} = \sum_i^N \sum_{n,m} -w_c \log \sigma \left(\hat{\mathrm{Im}}_i^{\mathrm{segm}}(n,m,c) \right) \tag{6.1}$$

- Depth Prediction: Similarly, we use the segmented ground-truth images $\mathrm{Im}^{\mathrm{segm}}$ to multiply by a factor of 10 the loss value of all pixels belonging to the object class. Without this modification, the network might choose to ignore the objects since they represent a tiny portion of the full image. Let $\mu = 0.1$, I the identity matrix, and I_i^{obj} a binary mask matrix equal to 1 for every pixel in the object class:

$$\mathcal{L}_{\text{depth}} = \frac{1}{N} \sum_i^N \left\| \left(\mathrm{Im}_i^{\mathrm{depth}} - \hat{\mathrm{Im}}_i^{\mathrm{depth}} \right) \odot \left((1-\mu) I_i^{\mathrm{obj}} + \mu I \right) \right\|_1 \tag{6.2}$$

- Auto-encoding. Learning to reconstruct the full image would be redundant with the information extracted by the other two decoders. Instead, we want this decoder to further help the network focus on the objects. To do so, we use the same object-segmentation mask to only reconstruct the objects in the scene, ignoring all the rest. Let $\mathrm{Im}_i = s_i^{\mathrm{image}}$:

$$\mathcal{L}_{\text{auto-encoding}} = \frac{1}{N} \sum_i^N \left\| \left(\mathrm{Im}_i - \hat{\mathrm{Im}}_i \right) \odot I_i^{\mathrm{obj}} \right\|_2^2 \tag{6.3}$$

- State regression. This objective already focuses on the objects, as the goal is to reconstruct the scene objects' coordinates as well as a 20-dimensional vector containing the robot proprioception (joint positions, joint speeds, gripper position and orientation, gripper state).

$$\mathcal{L}_{\text{regression}} = \frac{1}{N} \sum_{i}^{N} \left\| s_i^{\text{proprioception}} - \hat{s}_i^{\text{proprioception}} \right\|_2^2 \tag{6.4}$$

The final loss function is a simple weighted sum of all 4 losses to adjust for the different magnitudes of each loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{regression}} + \lambda^{\text{rgb}} \mathcal{L}_{\text{auto-encoding}} + \lambda^{\text{depth}} \mathcal{L}_{\text{depth}} + \lambda^{\text{segm}} \mathcal{L}_{\text{segmentation}} \tag{6.5}$$

6.5.1 Architecture

All networks, one per task, were trained with identical hyperparameters and using a ResNet-50 encoder (He et al., 2016). As shown in Figure 6.4, our setup consists of two cameras, one mounted on the wrist of the robot and one in front of it. The encoder takes a single image as input, and all three vision decoders take the corresponding 32-dimensional encoded representation as input. However, the MLP decoder for state regression takes in the concatenation of both encoded vectors (one for each camera). This fully concatenated 64-dimensional vector corresponds to $\tilde{s}_{\text{image}}$, and is part of the RL state that will be used in the following stage of the pipeline.

Other than ResNet, we also tried different architectures. The one that yielded the second-best results was the Vision Transformer (ViT) (Dosovitskiy et al., 2020), used as a backbone for both encoder and decoder. However, Figure 6.5 shows that the qualitative results in the segmentation and depth tasks (we discarded ViT before adding the auto-encoding task) are noticeably worse than those of the ResNet counterpart (Figures 6.8 and 6.7). The depth predictions in particular appear to be very low-resolution, which might be due to the patch-decomposition nature of the ViT architecture.

Note that we borrow pre-trained versions (on ImageNet) of both the ResNet and ViT encoders, from their official Github repositories. In the case of the chosen ResNet architecture, only the first and last CNN layers are trained from scratch, because we need them to match our desired input and output shapes respectively. The ResNet decoders are more shallow than the encoder, and are fully trained from scratch. Note that all three decoders share most of the architecture, and only differ at the very end, each having a separate single-layer CNN head. We adopt this architecture because we believe that the same low-level features are needed to solve all three computer-vision tasks.

6.5.2 Experiments

Figures 6.8, 6.7, 6.9 and 6.6 show some examples of ground-truth images used to train the encoder, as well as the corresponding predictions after training.

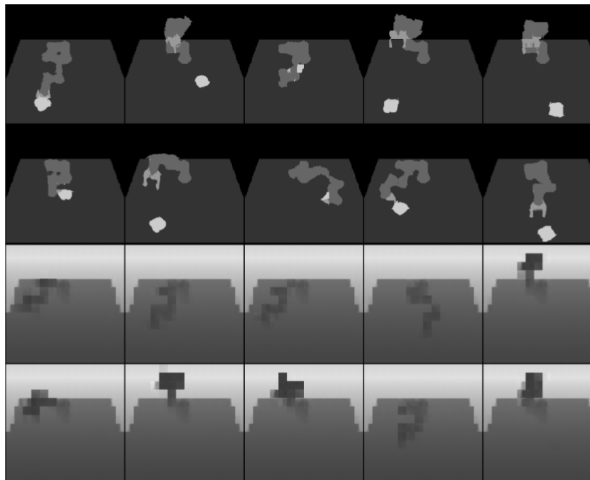


FIGURE 6.5: Qualitative results with a ViT encoder for the pushing task. The upper two rows show prediction results for the image segmentation task, the lower two rows for the depth prediction task.

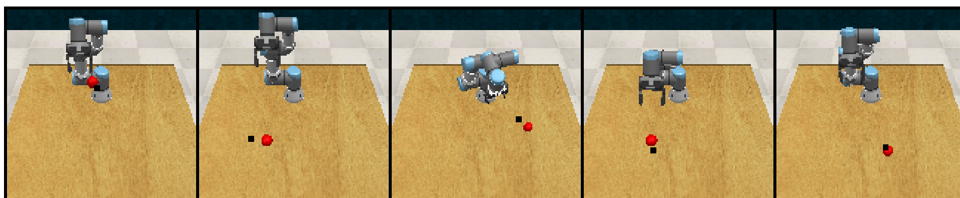


FIGURE 6.6: State regression: Qualitative results of the pre-trained encoder for the reaching task on some test images. The black square shows the 2D projection of the 3D output for the ball's coordinates.

The segmentation results are key for the agent to accurately locate the objects in the scene. Despite some errors in the boundaries between classes, the predictions closely match the ground truth, even when the different objects overlap (objects, gripper, and robot overlap often).

The depth results further help in locating the robot, but the objects are trickier to spot because the differences in depth are very minor. In particular, the front camera (upper two rows in Figure 6.7) fails to detect the button on the table, but it does appear in the wrist camera.

The auto-encoding results show that the encoder is able to accurately differentiate between the two objects in the scene (red block and green target). The un-masked predictions are shown in the bottom row of Figure 6.9.

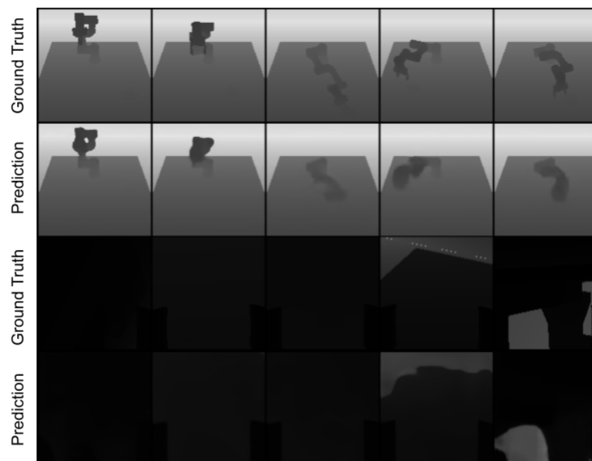


FIGURE 6.7: Depth prediction: Qualitative results of the pre-trained encoder for the pushing task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs.

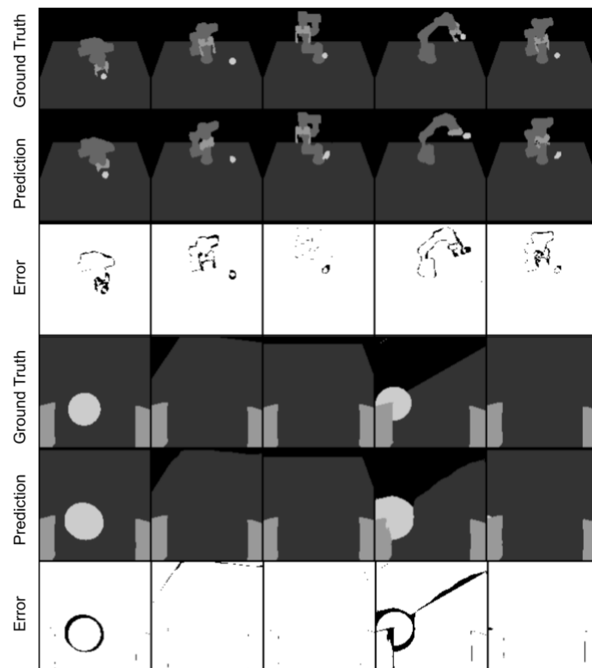


FIGURE 6.8: Image segmentation: Qualitative results of the pre-trained encoder for the reaching task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs.

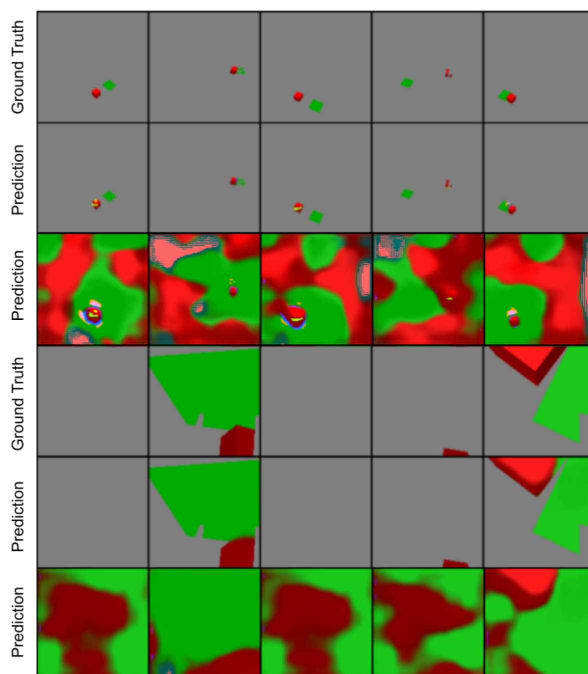


FIGURE 6.9: Auto-encoding: Qualitative results of the pre-trained encoder for the sliding task on some test images. The first row for each view shows the ground-truth, the second row shows the network outputs.

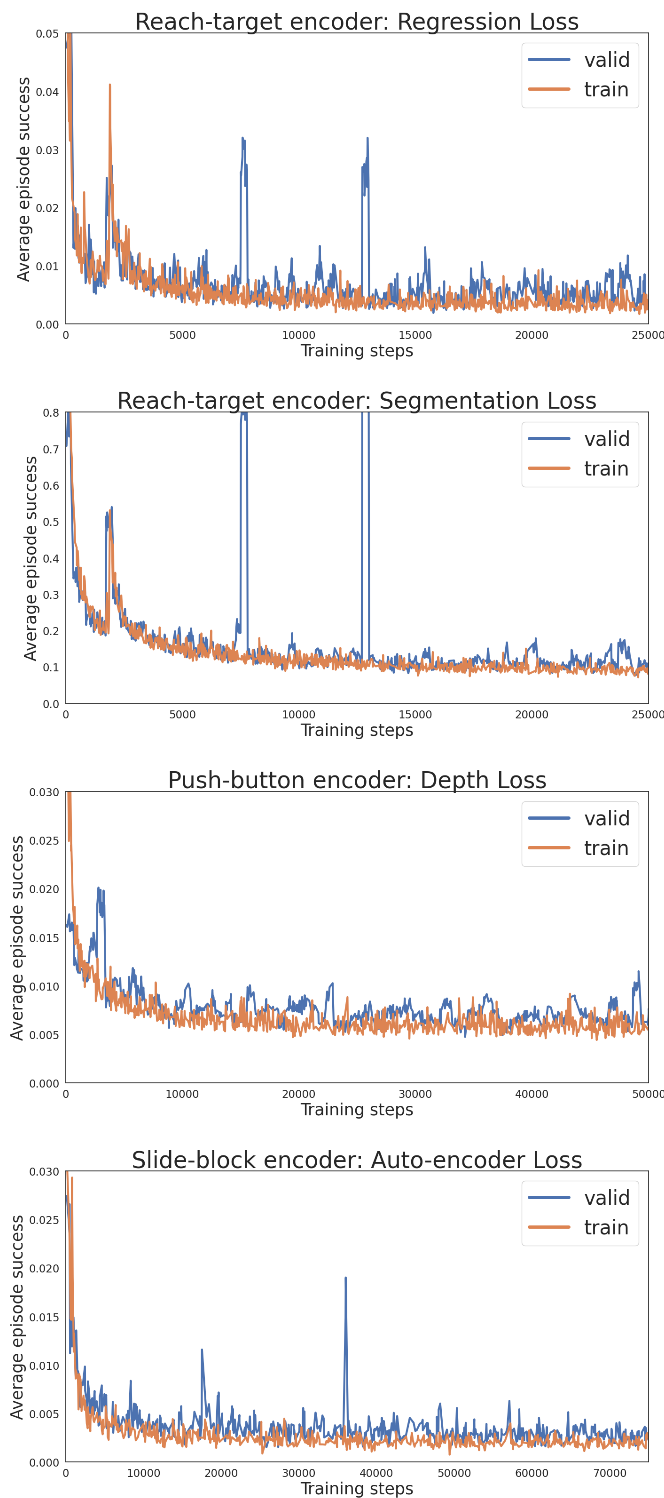


FIGURE 6.10: Learning curves for the encoder's supervised objectives.

6.6 Training the Agent

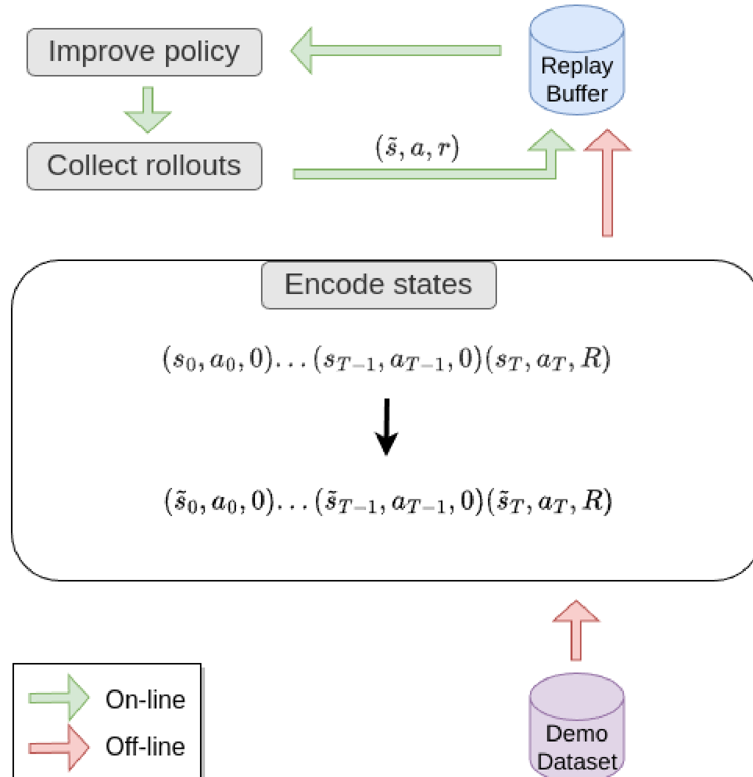


FIGURE 6.11: State-encoding procedure.

Once the encoder is trained, we follow by learning an agent with RL. Since we use SAC (most specifically SAC-Demo, the modified version introduced in Chapter 4) as the base algorithm for all our experiments, we name our method SAC-PVE, as in Pre-trained Vision Encoder.

Similar to works such as SEER (Chen et al., 2021a), the encoder is frozen during the totality of the training process. Other than hopefully being more sample efficient, another advantage of using a pre-trained encoder is that the replay buffer has a smaller memory footprint, since the encoded representations are much smaller than the full images.

Some minor modifications to the standard SAC are required to store the intermediate encoded states $\tilde{s}$ in the buffer. One idea would be to use a callback function whenever an episode is complete, to transform the transitions (s_t, a_t, r_t, s_{t+1}) into $(\tilde{s}_t, a_t, r_t, \tilde{s}_{t+1})$. However, we already compute the intermediate encoded states during exploration when we compute the actions $a = \pi(\text{Enc}(s)) = \pi(\tilde{s})$, so we simply need to keep track of them.

What about demonstration data (or any other type of off-policy data that might be used)? This time the callback function is actually required to compute the encoded states in the demonstrations' transitions. As shown in Figure 6.11, this step can entirely be done off-line. However, if GPU memory is not the bottleneck, in practice a separate copy of the frozen encoder can be loaded to encode the demonstations as needed.

6.6.1 Experiments

Setup

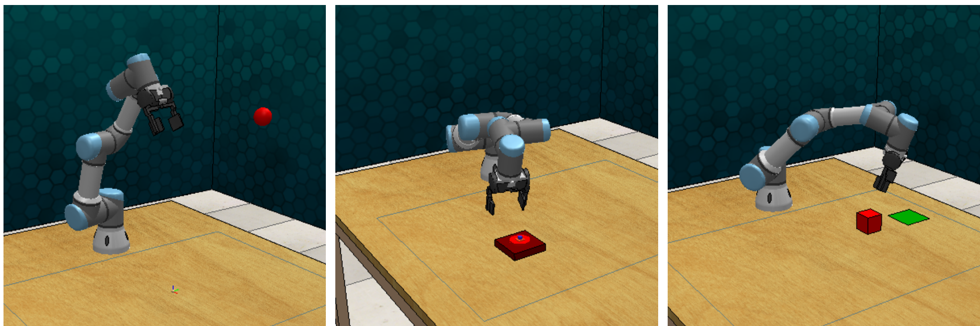


FIGURE 6.12: Snapshots from each task: reach target, push button, slide block.

We evaluate our method on three simulated RLBench James et al., 2020 tasks for a 6 degrees-of-freedom robot manipulator: reaching a ball, pushing a button, and sliding a block to a target square. Other than the state space, the experimental setup is the same as in Chapter 5, so we refer to it for a detailed description of the scene parameters. All the results are smoothed with a rolling window of 100 episodes, and the standard error is computed on three random seeds.

Observation space and Policy architecture. The encoder receives 64x64x6 images (two cameras) and outputs a 64-dimensional vector. This vector is concatenated to a 20-dimensional vector containing the robot proprioception (joint positions, joint speeds, gripper position and orientation, gripper state). This 84-dimensional vector is then fed into an MLP that outputs the action.

Results

Figure 6.13 shows that learning with a pre-trained encoder is not only much more sample-efficient, but the final performance is also better.

For the reaching task, the agent with a pre-trained encoder reaches a performance close to 100% after just 20.000 steps, while the end-to-end agents have a high variance across runs. The best baseline SAC+AE-CURL is also able to reach a perfect success rate on one of the seeds, but fails to make any progress on another. The converge speed of the end-to-end baselines is also considerably slower. For reference, an agent learning from the full state of the environment (i.e. object coordinates) rather than image inputs, is also able to reach a training success rate close to 100% at a similar convergence speed.

For the pushing task, the agent with a pre-trained encoder reaches a performance close to 100% after just 25.000 steps, while the end-to-end agents seem to perform better than in the reaching task, particularly SAC+AE-CURL and SAC-Demo. The most probable explanation is that the reaching task presents the additional challenge of having to reach a target in the 3D space, which

	Reach	**Push**	**Slide**
no vision	99.3 (12.2)	97.7 (19.8)	85.2 (4.66)
SAC-PVE	**99.2** (12.1)	**98.2** (19.4)	**72.9** (6.76)
SAC+AE-CURL	97.1* (12.4)	97.2 (21.3)	5.90 (4.56)
SAC-Demo	94.4 (14.0)	94.2 (22.2)	16.8 (6.66)

Table 6.1: Comparison on evaluation episodes. The first value is the average success rate, and the second value (between parentheses) is the average episode length (of succesful episodes only), calculated across 3000 evaluation episodes (3 seeds, 1000 per seed, except for SAC+AE-CURL where we exclude the seed that made zero progress in the reach task). The column-wise best vision results are marked in bold.

wasn't really a problem in previous chapters when learning from object coordinates. For reference, in the pushing task such an agent is also able to reach close to 100% accuracy, but takes significantly longer to do so (around 100.000 iterations). One possible explanation is that knowing the position of the button is only helpful to reach it, but doesn't really help with the actual pushing down (the gripper would sometimes get stuck in the sides of the button and the agent is unable to recover).

In the case of the more complicated sliding task, all end-to-end baselines are unable to make any significant progress, while the agent with the pre-trained encoder solves the task consistently, reaching a performance close to 70% after 200.000 steps. This task is also particularly hard from image inputs because the wrist camera is not as useful as in the other two tasks, as the target is almost never in frame, and the block is only in frame when already making contact with it. For reference, an agent learning from the full state of the environment is able to reach $\sim 80\%$ accuracy after 200.000 iterations.

Being more sample-efficient than end-to-end methods is an expected result, what's more interesting is that the final success rate is also higher. Table 6.1 compiles these results on evaluation episodes. We include a baseline *no vision*, the best method from Chapter 5 that takes in the true state of the environment, including the target object's coordinates. The results show that on every task, SAC-PVE has the highest success rate and shortest trajectories.

Qualitative results for all three tasks are shown in the following video. The quality of the learnt policy is noticeably better when learning with a pre-trained encoder.

drive.google.com/file/d/14c1MPRvfumvos2o8tDsIXMF1MLDpiPSh/view

6.7 Towards a Shared Encoder across tasks

The main obvious disadvantage of our method is that it requires an additional non-negligeable step: pre-training the encoder. However, do we really need to train a separate encoder for every task we want to solve? The encoder doesn't

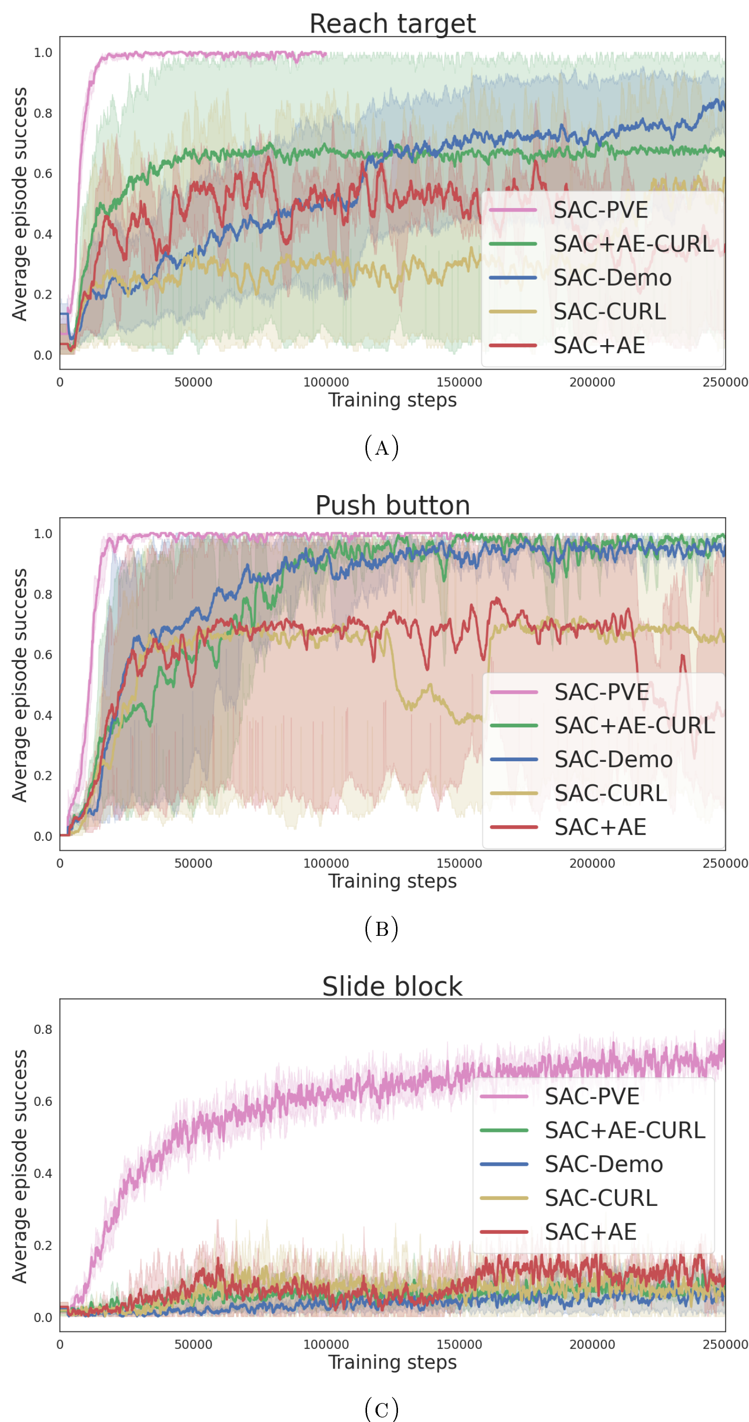


FIGURE 6.13: Learning curves of SAC-PVE on three RLBench tasks.

care about the actual task and how to solve it, it only cares about the visual representation of the scene. If we had a shared encoder across tasks we would only need 1 additional training rather than N additonal trainings, where N is the number of tasks we want to solve.

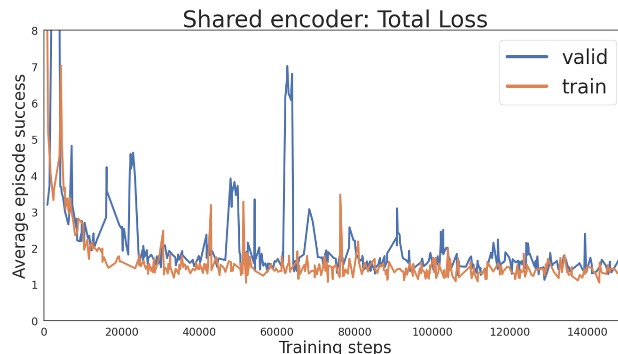


FIGURE 6.14: Learning curves of the shared encoder.

	Reach	Push	Slide
SAC-PVE (shared encoder)	**99.9** (11.3)	**99.7** (17.7)	**87.7** (4.79)
SAC-PVE	99.2 (12.1)	98.2 (19.4)	72.9 (6.76)

TABLE 6.2: SAC-PVE, with and without a shared encoder, compared on evaluation episodes. Same conditions as in Table 6.1.

Here are the two main modifications we bring to our encoder-training procedure in order to train such a shared encoder.

- We combine the per-task datasets into a single shared dataset. During training, each batch of data contains exactly the same amount of elements from each task.
- While the vision decoders (depth, segmentation, autoencoder) can remain the same, the state decoder presents a problem, because its output is different depending on the task. For instance, for our particular setup, the sliding task contains two objects in the scene, while the other two tasks just one. We want a decoder that can predict the coordinates of every object in the scene, regardless of how many there are. To do so, we replace the simple MLP decoder with a Permutation-Invariant Set Autoencoder (PISA), as introduced in (Kortvelesy, Morad, and Prorok, 2023). A PISA decoder is a set decoder able to decode fixed-size embeddings into variable-size output sets.

Figure 6.15 shows that not only do we not pay a price for training a single encoder, but the results are actually better with the shared encoder than with the task-specific ones, especially for the sliding task where the increase in performance is most noticeable. But Table 6.2 shows that the performance improves for all three tasks across evaluation episodes. For reaching and pushing, where SAC-PVE was already close to a perfect success rate, the shared-encoder version is slightly better and the quality of the planner is noticeably better (shorter trajectories). This improvement is probably due to the larger training dataset for the shared encoder, leading to improved generalization.

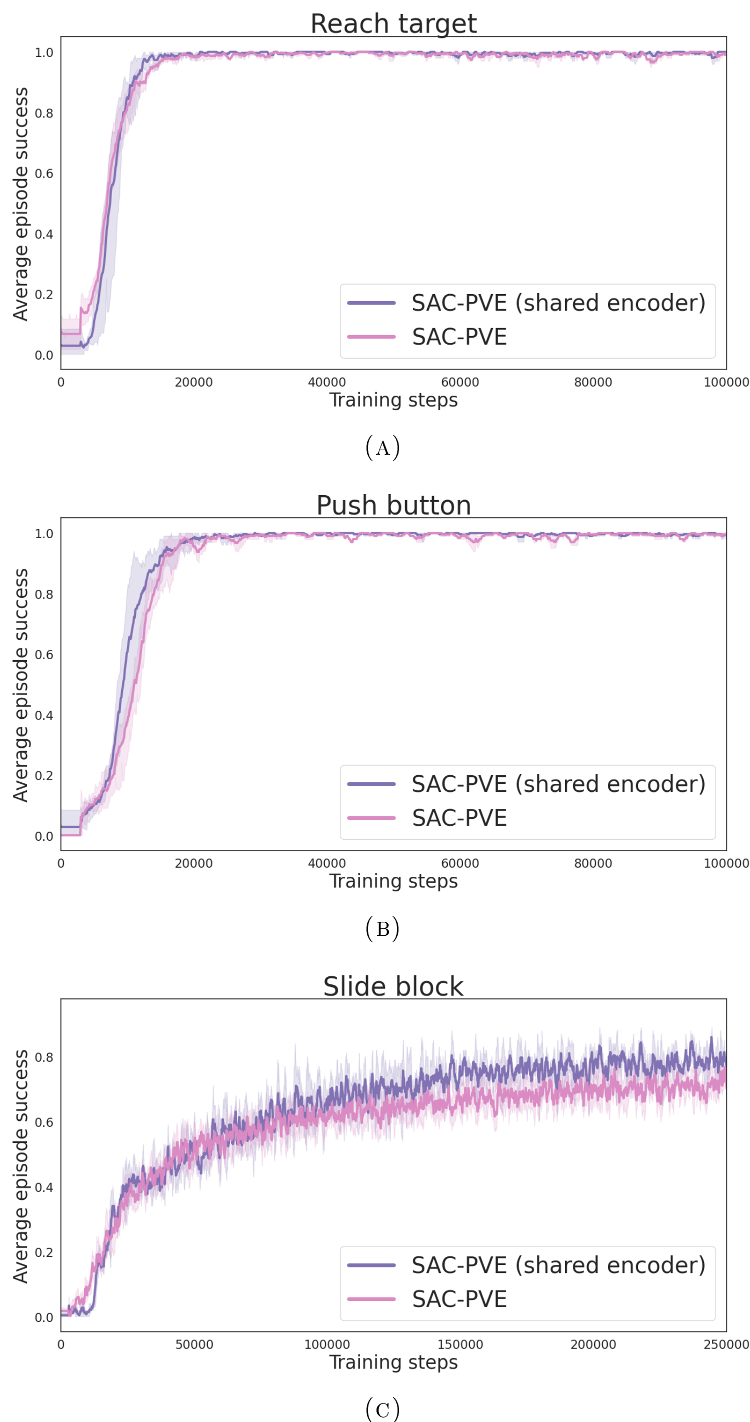


FIGURE 6.15: Learning curves of SAC-PVE (shared encoder) on three RLBench tasks.

6.8 Towards a Sim-to-Real pipeline

Other than learning how to solve the task, sim-to-real transfer presents an additional challenge to the RL agent: learning a robust image representation that can generalize to the target test environment. By having a two-step process, we can alleviate the burden of the RL algorithm and rely instead on the more

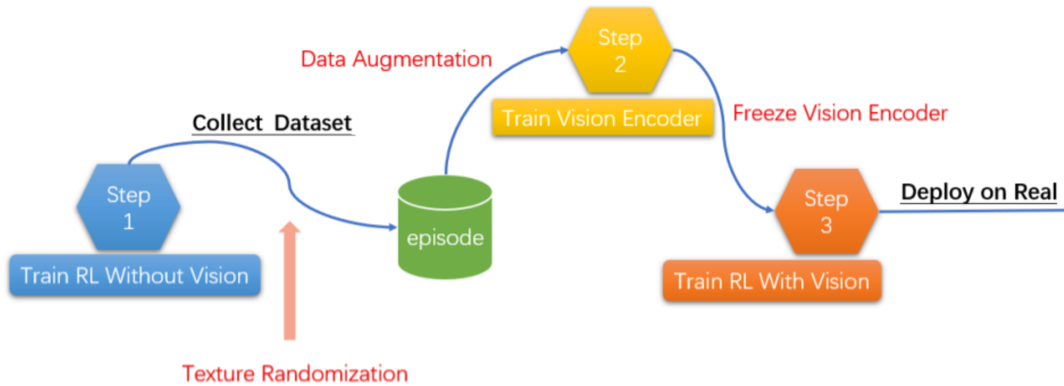


FIGURE 6.16: Full sim-to-real pipeline. Two additional steps with respect to the standard pipeline: texture randomization and data augmentation.

stable supervised-learning training. We introduce a simple sim-to-real pipeline to show the advantages and possibilities of such a framework. As shown in Figure 6.16, the sim-to-real pipeline is very similar to the one discussed so far. Two additional steps are required.

- First, we apply texture randomisation to generate the dataset that is used to train the encoder, as shown in Figure 6.17. In our case, rather than adopting a full domain-randomisation approach, we use some domain knowledge and collect textures from the real target environment (Figure 6.1) under different lighting and camera conditions. If the transfer domain were completely unknown, a much more varied dataset would be required for the final agent to be able to adapt to it (we have discussed some domain-randomization examples in Section 6.3). Note that this same texture randomisation is also applied during Step 3 when training the final vision agent.

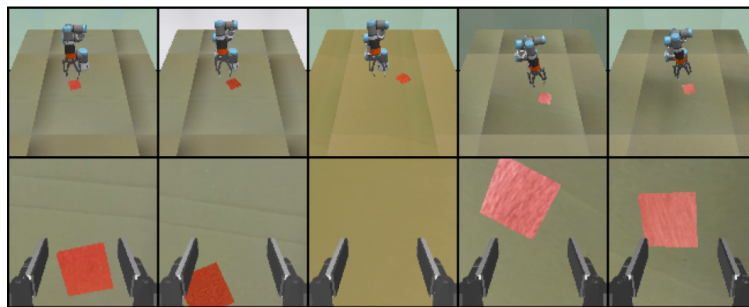


FIGURE 6.17: Texture randomization: the texture of the walls, table and target are randomized to bring variety to the dataset.

- Second, Figure 6.18 shows the standard computer-vision data-augmentation transformations we apply to the data during the training process of the encoder, including classic augmentations (hue, contrast, blur, saturation...) as well as small image translations to account for any differences in position between the simulated and real-world cameras.

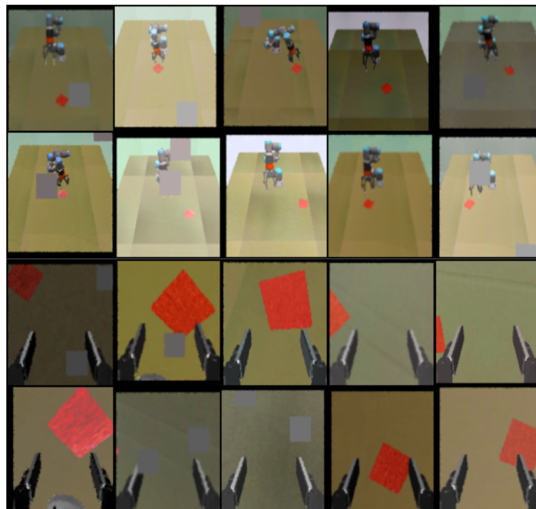


FIGURE 6.18: Data augmentation: during training of the encoder, visual transformations are applied to further diversify the data.

As shown in Figure 6.1, the task we tackle in the real world is a simpler version of the reaching task, with a 2D target on the table. We only provide qualitative results for this experiment, available at the end of the video below. A zero-shot transfer results in a policy that is able to consistently solve the task in the real-world setup, and is even able to adapt mid-episode if the target is displaced.

drive.google.com/file/d/14c1MPRvfumvos2o8tDsIXMF1MLDpiPSh/view

6.9 Discussion

In this chapter we propose a two-stage pipeline in order to obtain faster and most sample-efficient RL trainings, while also obtaining a higher final success rate than the end-to-end baselines. Our simple pipeline accelerates RL algorithms with the help of a pre-trained vision encoder, and vastly outperforms end-to-end alternatives across three different manipulation tasks.

The better results are actually obtained with a single shared encoder across tasks, rather than having to train one encoder per task. It would be interesting to see if this holds true for a larger amount of tasks and objects, but the current results show that the scalability of the model is promising. The next step would be to have a fully multi-task model by training a single multi-task RL agent on top of our single task-agnostic encoder.

We also show that, by adding texture randomization and data augmentation, the pipeline is suited for a zero-shot sim-to-real transfer in a reaching task. Many improvements can be brought to our simple sim-to-real setup. For instance, other than texture randomization, it would be interesting to see if we could generalize to previously-unseen objects in the real world after training on a

variety of synthetic objects. We also wish we had the time to test more tasks in the real robot.

Chapter 7

Conclusion

7.1 Summary of Contributions

Overall, this thesis focused on two main research directions:

- **Using demonstrations to improve the performance and efficiency of off-policy reinforcement-learning algorithms.** We proposed R^2, a reward-bonus method at the intersection of imitation learning, self-imitation learning, and hindsight experience replay. Other than improving the overall performance and efficiency, it benefits from being easy to implement and simple to understand, with a single hyper-parameter that intuitively corresponds to the amount of steps that the user wants to relabel, requiring very little knowledge of the task at hand. After comparing our method to concurrent works, we carried a thorough experimental study to determine the key methods in the literature that make the better use of the demonstrations in the buffer, and combined them together into our second algorithm $STIR^2$. While some methods struggled in a particular task or simulation environment, $STIR^2$ always had the better performance, showing that the selected methods don't conflict with each other but rather elevate the performance in all the tested scenarios.
- **Using demonstrations to accelerate vision-based reinforcement learning.** Rather than an end-to-end controller, directly mapping observations to controls, we introduced a perception module in a two-stage pipeline. We used the demonstrations to train an encoder from multiple computer-vision objectives simultaneously: Image Segmentation, Depth Prediction, Auto-encoding, and State Regression. The pre-trained encoder was then frozen and used by the reinforcement-learning agent to bypass the raw images. Despite all the demonstrations being collected in simulation, we showed that by adding a handful of real-world texture images, the pipeline is suited for a zero-shot sim-to-real transfer in a reaching task. We carried multiple experiments showing the benefits over the end-to-end alternatives.

7.2 Perspectives and Future Work

Exploiting offline data is very much a promising research axis in the field of RL, with recent advances such as offline RL (Levine et al., 2020) or decision transformers (Chen et al., 2021b). Here are some perspectives to improve on the methods that we presented in this thesis, as we continue to scale up towards more complex tasks.

- The biggest missing piece of our reward-relabelling method R^2 is the lack of a proper theoretical proof. We provided some intuition and believe that our choices for the equations governing the evolution of the hyper-parameters are sound, but we failed to provide an actual theoretical guarantee that could really benefit the method. We believe the most interesting research direction on this work is to tackle more complex tasks where credit assignment isn't obvious. The popular and well-studied attention mechanism immediately comes to mind as a possible way to determine which steps need to be relabelled, rather than relying on the rigid heuristic of our current method.
- The experiments of our two-stage vision pipeline weren't as thorough as we wish since we didn't test for generalization to randomized objects and scenes. Other than carrying those tests, as well as more real-world tests, the encoder can be easily improved by adopting more recent computer-vision objectives and architectures. We believe the most interesting research direction to be multi-task learning. We already showed that a multi-task vision encoder is possible and even desirable in terms of performance, probably due to the fact that robot-manipulation tasks present a significant degree of visual overlap. As we move to more complicated tasks, we believe that the overlap in behaviour could lead to similar results for the second part of the pipeline by adopting a multi-task agent.
- Overall, we believe that in order to make the best use of the demonstration data (or any other type of additional data), it is better to revert back from a fully end-to-end robot-control pipeline, as having separate modules allows to attack specific problems in a more precise manner, providing each module with the parts of the data that are most important. For instance, as we saw with our two-stage vision papeline, when the RL agent can focus solely on solving the task rather than also learning a state representation simultaneously, it is able to reach a much higher performance. Other than off-policy RL, there are other frameworks that lay themselves better to such a modular approach, and are better tailored to exploit large quantities of offline data, such as model-based RL, offline RL, or hierarchical RL. Our most promising experiments came in the latter, where we began to design a two-level hierarchy with a pure IL agent at the higher level and a RL agent at the lower level. Sadly, we couldn't finish that work due to lack of time, but we believe it to be a promising direction for future work.

Bibliography

Abbeel, Pieter and Andrew Y Ng (2004). "Apprenticeship learning via inverse reinforcement learning". In: *Proceedings of the twenty-first international conference on Machine learning*, p. 1.

Abdolmaleki, Abbas, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller (2018). "Maximum a posteriori policy optimisation". In: *arXiv preprint arXiv:1806.06920*.

Andrew Bagnell, J. (2014). "Reinforcement Learning in Robotics: A Survey". In: *Springer Tracts in Advanced Robotics*. Vol. 97, pp. 9–67. DOI: 10.1007/978-3-319-03194-1_2. URL: https://www.ias.informatik.tu-darmstadt.de/uploads/Publications/Kober{_}IJRR{_}2013.pdf.

Andrychowicz, Marcin, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba (2017). "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2017/file/453fadbd8a1a3af50a9df4df899537b5-Paper.pdf.

Barth-Maron, Gabriel, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap (2018). "Distributed distributional deterministic policy gradients". In: *arXiv preprint arXiv:1804.08617*.

Bellemare, Marc, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos (2016). "Unifying count-based exploration and intrinsic motivation". In: *Advances in neural information processing systems*, pp. 1471–1479.

Bellemare, Marc G, Will Dabney, and Rémi Munos (2017). "A distributional perspective on reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 449–458.

Bertsekas, Dimitri (2019). *Reinforcement Learning and Optimal Control*. URL: http://web.mit.edu/dimitrib/www/RLbook.html.

Bharadhwaj, Homanga, Aviral Kumar, Nicholas Rhinehart, Sergey Levine, Florian Shkurti, and Animesh Garg (2020). "Conservative safety critics for exploration". In: *arXiv preprint arXiv:2010.14497*.

Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba (2016). *End to End Learning for Self-Driving Cars*. Tech. rep. arXiv: 1604.07316v1.

Burda, Yuri, Harrison Edwards, Amos Storkey, and Oleg Klimov (2018). "Exploration by random network distillation". In: *arXiv preprint arXiv:1810.12894*.

Chebotar, Yevgen, Karol Hausman, Marvin Zhang, Gaurav Sukhatme, Stefan Schaal, and Sergey Levine (2017). "Combining model-based and model-free updates for trajectory-centric reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 703–711.

Chen, Bryan, Alexander Sax, Gene Lewis, Iro Armeni, Silvio Savarese, Amir Zamir, Jitendra Malik, and Lerrel Pinto (2020). "Robust policies via mid-level visual representations: An experimental study in manipulation and navigation". In: *arXiv preprint arXiv:2011.06698*.

Chen, Lili, Kimin Lee, Aravind Srinivas, and Pieter Abbeel (2021a). "Improving computational efficiency in visual reinforcement learning via stored embeddings". In: *Advances in Neural Information Processing Systems* 34.

Chen, Lili, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch (2021b). "Decision transformer: Reinforcement learning via sequence modeling". In: *Advances in neural information processing systems* 34, pp. 15084–15097.

Chen, Xi, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel (2016). "Infogan: Interpretable representation learning by information maximizing generative adversarial nets". In: *Advances in neural information processing systems*, pp. 2172–2180.

Chua, Kurtland, Roberto Calandra, Rowan McAllister, and Sergey Levine (2018). "Deep reinforcement learning in a handful of trials using probabilistic dynamics models". In: *Advances in Neural Information Processing Systems*, pp. 4754–4765.

Cobbe, Karl W, Jacob Hilton, Oleg Klimov, and John Schulman (2021). "Phasic policy gradient". In: *International Conference on Machine Learning*. PMLR, pp. 2020–2027.

Dayan, Peter (1993). "Improving generalization for temporal difference learning: The successor representation". In: *Neural computation* 5.4, pp. 613–624.

Deisenroth, Marc and Carl E Rasmussen (2011). "PILCO: A model-based and data-efficient approach to policy search". In: *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472.

Della Vecchia, Riccardo, Alena Shilova, Philippe Preux, and Riad Akrour (2022). "Entropy Regularized Reinforcement Learning with Cascading Networks". In: *arXiv preprint arXiv:2210.08503*.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. (2020). "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929*.

Eberhard, Onno, Jakob Hollenstein, Cristina Pinneri, and Georg Martius (2022). "Pink noise is all you need: Colored noise exploration in deep reinforcement learning". In: *The Eleventh International Conference on Learning Representations*.

Eysenbach, Ben, Xinyang Geng, Sergey Levine, and Russ R Salakhutdinov (2020). "Rewriting history with inverse rl: Hindsight inference for policy

improvement". In: *Advances in neural information processing systems* 33, pp. 14783–14795.

Eysenbach, Ben, Sergey Levine, and Russ R Salakhutdinov (2021). "Replacing rewards with examples: Example-based policy search via recursive classification". In: *Advances in Neural Information Processing Systems* 34.

Fedus, William, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney (2020). "Revisiting fundamentals of experience replay". In: *International Conference on Machine Learning*. PMLR, pp. 3061–3071.

Feinberg, Vladimir, Alvin Wan, Ion Stoica, Michael I Jordan, Joseph E Gonzalez, and Sergey Levine (2018). "Model-based value estimation for efficient model-free reinforcement learning". In: *arXiv preprint arXiv:1803.00101*.

Ferret, Johan, Olivier Pietquin, and Matthieu Geist (2020). "Self-Imitation Advantage Learning". In: *CoRR* abs/2012.11989. URL: https://arxiv.org/abs/2012.11989.

Finn, Chelsea, Paul Christiano, Pieter Abbeel, and Sergey Levine (2016). *A Connection Between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models*. Tech. rep. arXiv: 1611.03852v3.

Finn, Chelsea, Sergey Levine, and Pieter Abbeel (Mar. 2016). "Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization". In: arXiv: 1603.00448. URL: http://arxiv.org/abs/1603.00448.

Flet-Berliac, Yannis, Johan Ferret, Olivier Pietquin, Philippe Preux, and Matthieu Geist (2021). "Adversarially guided actor-critic". In: *arXiv preprint arXiv:2102.04376*.

Flet-Berliac, Yannis and Philippe Preux (2019). "Merl: Multi-head reinforcement learning". In: *arXiv preprint arXiv:1909.11939*.

Florence, Pete, Corey Lynch, Andy Zeng, Oscar A Ramirez, Ayzaan Wahid, Laura Downs, Adrian Wong, Johnny Lee, Igor Mordatch, and Jonathan Tompson (2022). "Implicit behavioral cloning". In: *Conference on Robot Learning*. PMLR, pp. 158–168.

Fortunato, Meire, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. (2017). "Noisy networks for exploration". In: *arXiv preprint arXiv:1706.10295*.

Fujimoto, Scott, Herke Van Hoof, and David Meger (2018). "Addressing function approximation error in actor-critic methods". In: *arXiv preprint arXiv:1802.09477*.

Gao, Yang, Huazhe(Harry) Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell (2018). *Reinforcement Learning from Imperfect Demonstrations*. URL: https://openreview.net/forum?id=BJJ9bz-0-.

Ghadirzadeh, Ali, Atsuto Maki, Danica Kragic, and Marten Bjorkman (Dec. 2017). "Deep predictive policy training using reinforcement learning". In: *IEEE International Conference on Intelligent Robots and Systems*. Vol. 2017-Septe. Institute of Electrical and Electronics Engineers Inc., pp. 2351–2358. ISBN: 9781538626825. DOI: 10.1109/IROS.2017.8206046. arXiv: 1703.00727. URL: http://arxiv.org/abs/1703.00727.

Ghosh, Dibya, Abhishek Gupta, Ashwin Reddy, Justin Fu, Coline Devin, Benjamin Eysenbach, and Sergey Levine (2019). "Learning to reach goals via iterated supervised learning". In: *arXiv preprint arXiv:1912.06088*.

Gu, Shixiang, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine (2016a). "Q-prop: Sample-efficient policy gradient with an off-policy critic". In: *arXiv preprint arXiv:1611.02247*.

Gu, Shixiang, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine (2016b). "Continuous deep q-learning with model-based acceleration". In: *International Conference on Machine Learning*, pp. 2829–2838.

Guo, Xiaoxiao, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang (2014). "Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning". In: *Advances in neural information processing systems*, pp. 3338–3346.

Haarnoja, Tuomas, Vitchyr Pong, Aurick Zhou, Murtaza Dalal, Pieter Abbeel, and Sergey Levine (2018a). "Composable deep reinforcement learning for robotic manipulation". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 6244–6251.

Haarnoja, Tuomas, Haoran Tang, Pieter Abbeel, and Sergey Levine (Feb. 2017). "Reinforcement Learning with Deep Energy-Based Policies". In: arXiv: 1702.08165. URL: http://arxiv.org/abs/1702.08165.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine (2018b). "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290*.

Hafner, Danijar, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba (2020). "Mastering atari with discrete world models". In: *arXiv preprint arXiv:2010.02193*.

Hafner, Danijar, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap (2023). "Mastering diverse domains through world models". In: *arXiv preprint arXiv:2301.04104*.

Hansen, Nicklas, Rishabh Jangir, Yu Sun, Guillem Alenyà, Pieter Abbeel, Alexei A Efros, Lerrel Pinto, and Xiaolong Wang (2020). "Self-supervised policy adaptation during deployment". In: *arXiv preprint arXiv:2007.04309*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Heess, Nicolas, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa (2015). "Learning continuous control policies by stochastic value gradients". In: *Advances in neural information processing systems* 28.

Hessel, Matteo, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver (2018). "Rainbow: Combining improvements in deep reinforcement learning". In: *Thirty-Second AAAI Conference on Artificial Intelligence*.

Hester, Todd, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys (Apr. 2017). "Deep Q-learning from Demonstrations". In: arXiv: 1704.03732. URL: http://arxiv.org/abs/1704.03732.

Ho, Jonathan and Stefano Ermon (June 2016). "Generative Adversarial Imitation Learning". In: arXiv: 1606.03476. URL: http://arxiv.org/abs/1606.03476.

Horváth, Dániel, Gábor Erdős, Zoltán Istenes, Tomáš Horváth, and Sándor Földi (2022). "Object detection using sim2real domain randomization for

robotic applications". In: *IEEE Transactions on Robotics* 39.2, pp. 1225–1243.

Houthooft, Rein, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel (2016). "Vime: Variational information maximizing exploration". In: *Advances in Neural Information Processing Systems*, pp. 1109–1117.

Ibarz, Julian, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine (2021). "How to train your robot with deep reinforcement learning: lessons we have learned". In: *The International Journal of Robotics Research* 40.4-5, pp. 698–721.

Jaderberg, Max, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu (2016). "Reinforcement learning with unsupervised auxiliary tasks". In: *arXiv preprint arXiv:1611.05397*.

James, Stephen, Andrew J Davison, and Edward Johns (2017). "Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task". In: *Conference on Robot Learning*. PMLR, pp. 334–343.

James, Stephen, Zicong Ma, David Rovick Arrojo, and Andrew J Davison (2020). "Rlbench: The robot learning benchmark & learning environment". In: *IEEE Robotics and Automation Letters* 5.2, pp. 3019–3026.

James, Stephen, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis (2019). "Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12627–12637.

Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine (2019). "When to trust your model: Model-based policy optimization". In: *Advances in Neural Information Processing Systems*, pp. 12498–12509.

Janner, Michael, Igor Mordatch, and Sergey Levine (2020). "Generative temporal difference learning for infinite-horizon prediction". In: *arXiv preprint arXiv:2010.14496*.

Julian, Ryan, Benjamin Swanson, Gaurav S Sukhatme, Sergey Levine, Chelsea Finn, and Karol Hausman (2020). "Efficient Adaptation for End-to-End Vision-Based Robotic Manipulation". In: *arXiv preprint arXiv:2004.10190*.

Kahn, Gregory, Tianhao Zhang, Sergey Levine, and Pieter Abbeel (2017). "Plato: Policy learning using adaptive trajectory optimization". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 3342–3349.

Kalashnikov, Dmitry, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine (June 2018). "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". In: arXiv: 1806.10293. URL: http://arxiv.org/abs/1806.10293.

Kidambi, Rahul, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims (2020). "Morel: Model-based offline reinforcement learning". In: *Advances in neural information processing systems* 33, pp. 21810–21823.

Kortvelesy, Ryan, Steven Morad, and Amanda Prorok (2023). "Permutation-Invariant Set Autoencoders with Fixed-Size Embeddings for Multi-Agent Learning". In: *Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems*. AAMAS '23. International Foundation for Autonomous Agents and Multiagent Systems.

Kroemer, Oliver, Scott Niekum, and George Konidaris (2019). "A review of robot learning for manipulation: Challenges, representations, and algorithms". In: *arXiv preprint arXiv:1907.03146*.

Kumar, Ashish, Zipeng Fu, Deepak Pathak, and Jitendra Malik (2021). "Rma: Rapid motor adaptation for legged robots". In: *arXiv preprint arXiv:2107.04034*.

Kumar, Aviral, Abhishek Gupta, and Sergey Levine (2020). "Discor: Corrective feedback in reinforcement learning via distribution correction". In: *Advances in Neural Information Processing Systems* 33, pp. 18560–18572.

Kumar, Aviral, Xue Bin Peng, and Sergey Levine (2019). "Reward-Conditioned Policies." In: *CoRR* abs/1912.13465. URL: http://arxiv.org/abs/1912.13465.

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine (2020). "Conservative q-learning for offline reinforcement learning". In: *Advances in Neural Information Processing Systems* 33, pp. 1179–1191.

Kurutach, Thanard, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel (2018). "Model-ensemble trust-region policy optimization". In: *arXiv preprint arXiv:1802.10592*.

Lee, Michelle A., Yuke Zhu, Krishnan Srinivasan, Parth Shah, Silvio Savarese, Li Fei-Fei, Animesh Garg, and Jeannette Bohg (Oct. 2019). "Making sense of vision and touch: Self-supervised learning of multimodal representations for contact-rich tasks". In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2019-May, pp. 8943–8950. ISBN: 9781538660263. DOI: 10.1109/ICRA.2019.8793485. arXiv: 1907.13098. URL: http://arxiv.org/abs/1810.10191.

Lesort, Timothée, Natalia Díaz-Rodríguez, Jean-Franois Goudou, and David Filliat (2018). "State representation learning for control: An overview". In: *Neural Networks* 108, pp. 379–392.

Levine, Sergey (Fa2019). *CS285: Deep Reinforcement Learning*. URL: http://rail.eecs.berkeley.edu/deeprlcourse/.

Levine, Sergey and Pieter Abbeel (2014). "Learning neural network policies with guided policy search under unknown dynamics". In: *Advances in Neural Information Processing Systems*, pp. 1071–1079.

Levine, Sergey, Chelsea Finn, Trevor Darrell, and Pieter Abbeel (2016). "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* 17.1, pp. 1334–1373.

Levine, Sergey and Vladlen Koltun (2013). "Guided policy search". In: *International Conference on Machine Learning*, pp. 1–9.

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu (2020). "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems". In: *arXiv preprint arXiv:2005.01643*.

Li, Alexander, Lerrel Pinto, and Pieter Abbeel (2020). "Generalized hindsight for reinforcement learning". In: *Advances in neural information processing systems* 33, pp. 7754–7767.

Li, Yunzhu, Jiaming Song, and Stefano Ermon (2017). "Infogail: Interpretable imitation learning from visual demonstrations". In: *Advances in Neural Information Processing Systems*, pp. 3812–3822.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2016). "Continuous control with deep reinforcement learning." In: *ICLR (Poster)*. URL: http://arxiv.org/abs/1509.02971.

Liu, Guoqing, Li Zhao, Pushi Zhang, Jiang Bian, Tao Qin, Nenghai Yu, and Tie-Yan Liu (2021). "Demonstration actor critic". In: *Neurocomputing* 434, pp. 194–202.

Liu, Hao and Pieter Abbeel (2021). "Behavior from the void: Unsupervised active pre-training". In: *Advances in Neural Information Processing Systems* 34.

Liu, Yuxuan, Abhishek Gupta, Pieter Abbeel, and Sergey Levine (Sept. 2018). "Imitation from Observation: Learning to Imitate Behaviors from Raw Video via Context Translation". In: *Proceedings - IEEE International Conference on Robotics and Automation*. Institute of Electrical and Electronics Engineers Inc., pp. 1118–1125. ISBN: 9781538630815. DOI: 10.1109/ICRA.2018.8462901. arXiv: 1707.03374. URL: http://arxiv.org/abs/1707.03374.

Mnih, Volodymyr, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*, pp. 1928–1937.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Muelling, Katharina, Jens Kober, Oliver Kroemer, and Jan Peters (2012). *Learning to Select and Generalize Striking Movements in Robot Table Tennis*. Tech. rep. URL: www.aaai.org.

Nair, Ashvin, Murtaza Dalal, Abhishek Gupta, and Sergey Levine (2020). "Accelerating online reinforcement learning with offline datasets". In: *arXiv preprint arXiv:2006.09359*.

Nair, Ashvin, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel (2018a). "Overcoming exploration in reinforcement learning with demonstrations". In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE, pp. 6292–6299.

— (2018b). "Overcoming Exploration in Reinforcement Learning with Demonstrations". In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 6292–6299. ISBN: 9781538630815. DOI: 10.1109/ICRA.

2018.8463162. arXiv: 1709.10089. URL: http://ashvin.me/demoddpg-website.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In: *Icml*. Vol. 99, pp. 278–287.

Oh, Junhyuk, Yijie Guo, Satinder Singh, and Honglak Lee (2018). "Self-Imitation Learning". In: *ICML*.

Pascanu, Razvan, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia (2017). "Learning model-based planning from scratch". In: *arXiv preprint arXiv:1707.06170*.

Pashevich, Alexander, Robin Strudel, Igor Kalevatykh, Ivan Laptev, and Cordelia Schmid (2019). "Learning to augment synthetic images for sim2real policy transfer". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 2651–2657.

Pathak, Deepak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell (2017). "Curiosity-driven exploration by self-supervised prediction". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17.

Pinto, Lerrel, James Davidson, Rahul Sukthankar, and Abhinav Gupta (2017). "Robust adversarial reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 2817–2826.

Racanière, Sébastien, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. (2017). "Imagination-augmented agents for deep reinforcement learning". In: *Advances in neural information processing systems*, pp. 5690–5701.

Rajeswaran, Aravind, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine (2017). "Learning complex dexterous manipulation with deep reinforcement learning and demonstrations". In: *arXiv preprint arXiv:1709.10087*.

Ratliff, Nathan D, J Andrew Bagnell, and Martin A Zinkevich (2006). "Maximum margin planning". In: *Proceedings of the 23rd international conference on Machine learning*, pp. 729–736.

Recht, Benjamin (2019). "A tour of reinforcement learning: The view from continuous control". In: *Annual Review of Control, Robotics, and Autonomous Systems* 2, pp. 253–279.

Reddy, Siddharth, Anca D Dragan, and Sergey Levine (2019). "Sqil: Imitation learning via reinforcement learning with sparse rewards". In: *arXiv preprint arXiv:1905.11108*.

Reddy, Siddharth, Anca D. Dragan, and Sergey Levine (2020). "{SQIL}: Imitation Learning via Reinforcement Learning with Sparse Rewards". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=S1xKd24twB.

Ross, Stephane, Geoffrey J. Gordon, and J. Andrew Bagnell (Nov. 2010). "A Reduction of Imitation Learning and Structured Prediction to No-Regret

Online Learning". In: *Journal of Machine Learning Research* 15, pp. 627–635. arXiv: 1011.0686. URL: http://arxiv.org/abs/1011.0686.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver (2015). "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952*.

— (2016). "Prioritized Experience Replay". In: *ICLR (Poster)*. URL: http://arxiv.org/abs/1511.05952.

Schenck, Connor and Dieter Fox (2017). "Visual closed-loop control for pouring liquids". In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2629–2636. ISBN: 9781509046331. DOI: 10.1109/ICRA.2017.7989307. arXiv: 1610.02610.

Schmeckpeper, Karl, Annie Xie, Oleh Rybkin, Stephen Tian, Kostas Daniilidis, Sergey Levine, and Chelsea Finn (2019). "Learning Predictive Models From Observation and Interaction". In: arXiv: 1912.12773. URL: http://arxiv.org/abs/1912.12773.

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. (2020). "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839, pp. 604–609.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015a). "Trust region policy optimization". In: *International conference on machine learning*, pp. 1889–1897.

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel (2015b). "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438*.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Sermanet, Pierre, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain (2018). "Time-Contrastive Networks: Self-Supervised Learning from Video". In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1134–1141. ISBN: 9781538630815. DOI: 10.1109/ICRA.2018.8462891. arXiv: 1704.06888.

Shelhamer, Evan, Parsa Mahmoudieh, Max Argus, and Trevor Darrell (2016). "Loss is its own reward: Self-supervision for reinforcement learning". In: *arXiv preprint arXiv:1612.07307*.

Sigaud, Olivier and Freek Stulp (2019). "Policy search in continuous action domains: an overview". In: *Neural Networks* 113, pp. 28–40.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, p. 484.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller (2014). "Deterministic policy gradient algorithms". In.

Singh, Avi, Albert Yu, Jonathan Yang, Jesse Zhang, Aviral Kumar, and Sergey Levine (2020). "Cog: Connecting new skills to past experience with offline reinforcement learning". In: *arXiv preprint arXiv:2010.14500*.

Srinivas, Aravind, Michael Laskin, and Pieter Abbeel (2020). "Curl: Contrastive unsupervised representations for reinforcement learning". In: *arXiv preprint arXiv:2004.04136*.

Stulp, Freek and Olivier Sigaud (2012). "Path integral policy improvement with covariance matrix adaptation". In: *arXiv preprint arXiv:1206.4621*.

Sucan, Ioan A, Mark Moll, and Lydia E Kavraki (2012). "The open motion planning library". In: *IEEE Robotics & Automation Magazine* 19.4, pp. 72–82.

Sun, Mingfei and Xiaojuan Ma (2019). "Adversarial imitation learning from incomplete demonstrations". In: *arXiv preprint arXiv:1905.12310*.

Sutton, Richard S (1990). "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming". In: *Machine learning proceedings 1990*. Elsevier, pp. 216–224.

Sutton, Richard S and Andrew G Barto (2011). "Reinforcement learning: An introduction". In.

Tang, Haoran, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel (2017). "# exploration: A study of count-based exploration for deep reinforcement learning". In: *Advances in neural information processing systems*, pp. 2753–2762.

Theodorou, Evangelos, Jonas Buchli, and Stefan Schaal (2010). "Learning policy improvements with path integrals". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, pp. 828–835.

Thomas, Philip (2014). "Bias in natural actor-critic algorithms". In: *International conference on machine learning*, pp. 441–448.

Tobin, Josh, Lukas Biewald, Rocky Duan, Marcin Andrychowicz, Ankur Handa, Vikash Kumar, Bob McGrew, Alex Ray, Jonas Schneider, Peter Welinder, et al. (2018). "Domain randomization and generative models for robotic grasping". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 3482–3489.

Toussaint, Marc (2009). "Robot trajectory optimization using approximate inference". In: *Proceedings of the 26th annual international conference on machine learning*, pp. 1049–1056.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI conference on artificial intelligence*.

Vecerik, Mel, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller (2017). "Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards". In: *arXiv preprint arXiv:1707.08817*.

Vecerík, Matej, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin A. Riedmiller (2017). "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards". In: *CoRR* abs/1707.08817. URL: http://arxiv.org/abs/1707.08817.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4, pp. 229–256.

Wu, Yuhuai, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba (2017). "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation". In: *Advances in neural information processing systems*, pp. 5279–5288.

Wulfmeier, Markus, Peter Ondruska, and Ingmar Posner (July 2015). "Maximum Entropy Deep Inverse Reinforcement Learning". In: arXiv: 1507.04888. URL: http://arxiv.org/abs/1507.04888.

Xie, Annie, Frederik Ebert, Sergey Levine, and Chelsea Finn (2019). "Improvisation through Physical Understanding: Using Novel Objects As Tools with Visual Foresight". In: DOI: 10.15607/rss.2019.xv.001. arXiv: 1904.05538. URL: https://sites.google.com/view/gvf-tool.

Xu, Danfei and Misha Denil (2019). "Positive-unlabeled reward learning". In: *arXiv preprint arXiv:1911.00459*.

Xu, Danfei, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese (2018). "Neural Task Programming: Learning to Generalize Across Hierarchical Tasks". In: *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3795–3802. ISBN: 9781538630815. DOI: 10.1109/ICRA.2018.8460689. arXiv: 1710.01813. URL: https://ai.stanford.edu/{~}yukez/papers/icra2018.pdf.

Yarats, Denis, Rob Fergus, Alessandro Lazaric, and Lerrel Pinto (2021a). "Reinforcement learning with prototypical representations". In: *International Conference on Machine Learning*. PMLR, pp. 11920–11931.

Yarats, Denis, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus (2021b). "Improving sample efficiency in model-free reinforcement learning from images". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 12, pp. 10674–10681.

Yu, Tianhe, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine (2019). "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning". In: *arXiv preprint arXiv:1910.10897*.

Yu, Tianhe, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma (2020). "Mopo: Model-based offline policy optimization". In: *Advances in Neural Information Processing Systems* 33, pp. 14129–14142.

Zeng, Andy, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser (Mar. 2019). "TossingBot: Learning to Throw Arbitrary Objects with Residual Physics". In: arXiv: 1903.11239. URL: http://arxiv.org/abs/1903.11239.

Zhou, Allan, Moo Jin Kim, Lirui Wang, Pete Florence, and Chelsea Finn (2023). "Nerf in the palm of your hand: Corrective augmentation for robotics via novel-view synthesis". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 17907–17917.

Zhu, Yuke, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, and Nicolas Heess (Feb. 2018). "Reinforcement and Imitation Learning for Diverse Visuomotor Skills". In: arXiv: 1802.09564. URL: http://arxiv.org/abs/1802.09564.

Ziebart, Brian D, Andrew L Maas, J Andrew Bagnell, and Anind K Dey (2008). "Maximum entropy inverse reinforcement learning." In: *Aaai*. Vol. 8. Chicago, IL, USA, pp. 1433–1438.

RÉSUMÉ

Malgré leur grand succès, les algorithmes d'apprentissage par renforcement doivent encore devenir plus efficaces en termes d'échantillons, en particulier pour la robotique où il est beaucoup plus difficile d'entraîner un agent en dehors d'un environnement de simulation. Alors que la communauté se tourne vers des approches orientées données (apprentissage par renforcement "offline", "decision transformers", etc.), nous nous concentrons dans cette thèse sur l'apprentissage par renforcement "off-policy" et explorons différentes manières d'incorporer des données supplémentaires dans les algorithmes. En particulier, nous nous appuyons sur des démonstrations d'experts, qui peuvent contribuer à l'efficacité ainsi qu'à la performance globale. L'objectif est de concevoir des algorithmes efficaces pour résoudre des tâches de manipulation robotique, comme actionner un interrupteur ou faire glisser un cube sur une table.

Après une étude approfondie de l'apprentissage par renforcement et par imitation, nous présentons tout d'abord notre méthode de ré-étiquetage des récompenses, qui peut être considérée comme une forme de "reward shaping" qui se produit a posteriori, une fois que l'ensemble de l'épisode a été collecté. Cette approche peut s'appliquer à tout algorithme "off-policy" pour bénéficier à la fois des signaux de renforcement et d'imitation. En nous appuyant sur cette méthode, nous présentons ensuite un algorithme plus efficace qui regroupe des travaux antérieurs et concomitants qui traitent également de questions similaires.

Enfin, nous passons au cadre plus réaliste de l'apprentissage par renforcement basé sur la vision. Pour résoudre ce problème, nous concevons un pipeline d'entraînement en deux étapes : d'abord, apprendre une représentation visuelle de la scène en pré-entraînant un encodeur à partir de plusieurs objectifs supervisés de vision, puis entraîner un agent d'apprentissage par renforcement qui peut se concentrer uniquement sur la résolution de la tâche. Bien que toutes les données soient collectées en simulation, les expériences comprennent un exemple de transfert simulation-réalité pour montrer que ces techniques peuvent s'appliquer à des environnements contrôlés du monde réel.

MOTS CLÉS

Robots manipulateurs, Apprentissage par renforcement, Apprentissage par imitation

ABSTRACT

Despite having known great success, reinforcement-learning algorithms still need to become more sample-efficient, particularly for robotics where it is much harder to train an agent outside of simulation. As the community leans towards data-driven approaches (offline reinforcement learning, decision transformers, etc.), in this thesis we focus on off-policy reinforcement learning, and explore different ways of incorporating additional data into the algorithms. In particular, we rely on expert demonstrations, which can help with efficiency as well as overall performance. The goal is to design efficient algorithms to solve a range of robotic-manipulation tasks, such as flipping a switch or sliding a cube on a table.

After a thorough review of the reinforcement-learning and imitation-learning frameworks, we first introduce our reward-relabeling method, which can be seen as a form of reward shaping that happens in hindsight, once the entire episode is collected. This approach can easily extend any off-policy algorithm to benefit from both reinforcement and imitation signals. Building on this method, we then introduce a more efficient algorithm that aggregates previous and concurrent works that also address similar concerns.

Finally, we move onto the more realistic setting of vision-based reinforcement learning. To tackle this problem, we design a two-stage training pipeline: first learn a visual representation of the scene by pre-training an encoder from multiple supervised computer-vision objectives, then train a reinforcement-learning agent which can focus solely on solving the task. Despite all the data being collected in simulation, the experiments include one sim-to-real example to show that these techniques can translate to real-world controlled environments.

KEYWORDS

Robot manipulator, Reinforcement Learning, Imitation Learning