Multi-Pass Deep Q-Networks for Reinforcement Learning with Parameterised Action Spaces

Craig James Bester

783135

Supervised by:

Pravesh Ranchod

Steven James

George Konidaris

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

June, 2019

Johannesburg

Abstract

Parameterised actions in reinforcement learning are composed of discrete actions with continuous action-parameters. This provides a framework capable of solving complex domains that require learning high-level action policies with flexible control. Recently, deep Q-networks have been extended to learn over such action spaces with the P-DQN algorithm. However, the method treats all action-parameters as a single joint input to the Q-network, invalidating its theoretical foundations. We demonstrate the disadvantages of this approach and propose two solutions: using split Q-networks, and a novel multi-pass technique. We also propose a weighted-indexed action-parameter loss function to address issues related to the imbalance of sampling and exploration between different parameterised actions. We empirically demonstrate that both our multi-pass algorithm and weighted-indexed loss significantly outperform P-DQN and other previous algorithms in terms of data efficiency and converged policy performance on the Platform, Robot Soccer Goal, and Half Field Offense domains.

Declaration

I, Craig James Bester, hereby declare this dissertation to be my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

[illegible]
Signature

2019/09/06
Date

Acknowledgements

This research has been supported in part by the National Research Foundation of South Africa (Grant Number: 113737).

Contents

Preface

Abstract . . . i
Declaration . . . ii
Acknowledgements . . . iii
Table of Contents . . . iv
List of Figures . . . vi
List of Tables . . . vii

1 Introduction 1

2 Background and Related Work 3

2.1 Reinforcement Learning . . . 3
2.1.1 Value Function Approximation . . . 5
2.1.2 Continuous Action Spaces . . . 7
2.1.3 Parameterised Action Spaces . . . 10
2.2 Deep Reinforcement Learning . . . 12
2.2.1 Deep Q-Networks . . . 13
2.2.2 Deep Deterministic Policy Gradients . . . 15
2.2.3 PA-DDPG . . . 17
2.2.4 Exploration Strategies . . . 18
2.2.5 Action-Parameter Bounding . . . 19
2.3 Domains . . . 20
2.3.1 Platform . . . 20
2.3.2 Robot Soccer Goal . . . 22
2.3.3 Half Field Offense . . . 23
2.4 Related Work . . . 24
2.5 Summary . . . 25

3 Parameterised Deep Q-Networks 26

3.1 P-DQN . . . 26
3.1.1 Multi-Step Returns . . . 29
3.2 Comparison Study . . . 31
3.2.1 Experimental Methodology . . . 31
3.2.2 Results . . . 33
3.3 Summary . . . 36

4 Addressing Action Sampling Imbalance 37

4.1 Indexed Action-Parameter Loss . . . 38
4.1.1 Gradient-Zeroing for Action-Parameter Update Independence . . . 39
4.2 Weighted Action-Parameter Loss . . . 40
4.3 Experiments . . . 41
4.3.1 (E1) Indexed Loss . . . 41
4.3.2 (E2) Weighted Loss . . . 44
4.3.3 (E3) Weighted-Indexed Loss . . . 46

4.4 Summary 49

5 Independent Action-Parameter Methods 50

5.1 Q-Value Action-Parameter Sensitivity 50
5.2 Problems with Q-Value Sensitivity to Unassociated Action-Parameters 52
5.2.1 False Action-Parameter Gradients 52
5.2.2 Discrete Action Policy Perturbation 52
5.3 Split Q-Networks 54
5.3.1 Split Q-Networks with Shared Layers 55
5.4 Multi-Pass Q-Networks 56
5.5 Experiments 58
5.5.1 (E1) Separate Q-Networks 58
5.5.2 (E2) Split Q-Networks with Shared Layers 61
5.5.3 (E3) Multi-Pass Q-Networks 63
5.5.4 (E4) Computational Overhead 67
5.5.5 (E5) Weighted-Indexed Loss Ablation Study 69
5.6 Summary 71

6 Conclusion and Future Work 72

6.1 Future Work 73

Appendix A Hyperparameters 75

Appendix B Alternating Actor-Critic Updates 79

Appendix C Action-Parameter Scaling 82

Appendix D Initial Action-Parameter Policies for Platform and Robot Soccer Goal 86

D.1 Passthrough Layer 87
D.2 Comparison to Uninitialised Policies 90

Appendix E Effects of β Parameter for Mixed n-Step Targets 93

Appendix F Network Size Sensitivity 95

References 97

List of Figures

2.1 Reinforcement learning agent-environment interaction cycle. 3
2.2 Example of an artificial neural network with a single fully connected hidden layer 12
2.3 Abstraction of a deep Q-network 13
2.4 Illustration of the actor-critic network architecture for DDPG 15
2.5 Illustration of the PA-DDPG network architecture. 17
2.6 Different types of noise used for exploration 19
2.7 Platform domain 21
2.8 Successful episode of the Platform domain 21
2.9 Robot Soccer Goal domain 22
2.10 Half Field Offense domain 23

3.1 Illustration of the network architectures for P-DQN and PA-DDPG 27
3.2 Results of Q-PAMDP, P-DQN, and PA-DDPG on Platform, Robot Soccer Goal, and HFO 35

4.1 Plot of action selection ratios for P-DQN on Platform, Robot Soccer Goal and HFO . . . 37
4.2 Results of indexed action-parameter loss 43
4.3 Results of weighted action-parameter loss 45
4.4 Results of combined weighted-indexed action-parameter loss 48

5.1 Sensitivity of Q-values to their own action-parameters versus those of other actions . . . 51
5.2 Example of sensitivity to unrelated action-parameters affecting discrete action selection . 53
5.3 Separate Q-networks architecture for P-DQN 54
5.4 Split Q-network architecture with shared feature extraction layers for P-DQN 55
5.5 Multi-pass Q-network architecture 57
5.6 Results of separate Q-networks 60
5.7 Results of split Q-networks with shared layers on Half Field Offense 61
5.8 Results of split Q-networks with shared layers on Platform and Robot Soccer Goal 62
5.9 Results of multi-pass Q-networks 64
5.10 Comparison between multi-pass and separate Q-network results 66
5.11 Computational overhead comparison 67
5.12 Results of multi-pass Q-network with weighted-indexed loss ablation study 70

B.1 Graphs of results for simulataneous versus alternating actor-critic updates for P-DQN . . 81

C.1 Results of scaled versus unscaled action-parameters for P-DQN 84
C.2 Results of scaled versus unscaled action-parameters for Q-PAMDP 85

D.1 Passthrough layer architecture 87
D.2 Results of using a static versus a dynamic passhtrough layer for P-DQN 89
D.3 Results of initialised versus uninitialised action-parameters for P-DQN 91
D.4 Results of initialised versus uninitialised action-parameters for Q-PAMDP 92

E.1 Learning curves and trend lines of P-DQN and PA-DDPG over different values of β . . . 94

F.1 Results of P-DQN and PA-DDPG using different network sizes on HFO 96

List of Tables

3.1 Results of Q-PAMDP, P-DQN, and PA-DDPG on Platform, Robot Soccer Goal, and HFO 33

4.1 Results of indexed action-parameter loss function 42
4.2 Results of weighted action-parameter loss comparison 44
4.3 Results of weighted-indexed action-parameter loss 47

5.1 Results of separate Q-networks 59
5.2 Results of split Q-networks with shared layers 61
5.3 Results of multi-pass Q-networks 63
5.4 Comparison between multi-pass and separate Q-network results 65
5.5 Timing comparison between split and multi-pass Q-networks 68
5.6 Results of multi-pass Q-network with weighted-indexed loss ablation study 69

A.1 Hyperparameter search ranges 75
A.2 Hyperparameters for Platform 76
A.3 Hyperparameters for Robot Soccer Goal 77
A.4 Hyperparameters for Half Field Offense 78

B.1 Results of simultaneous versus alternating actor-critic updates for P-DQN 80

C.1 Results of using scaled versus unscaled action-parameters for P-DQN and Q-PAMDP . . 83

D.1 Results of using a static versus a dynamic passhtrough layer for P-DQN 88
D.2 Results of uninitialised versus intialised action-parameters for P-DQN and Q-PAMDP . . 90

E.1 Results of P-DQN and PA-DDPG using different values of β on HFO 94

F.1 Results of P-DQN and PA-DDPG on HFO using different network sizes 96

Chapter 1

Introduction

Reinforcement learning (RL) is a machine learning paradigm for control problems which aims to mimic how humans learn tasks by trial-and-error. It involves learning to solve tasks by selecting actions that maximise some reward structure. Previous work in reinforcement learning has focused on problems with either discrete or continuous actions, which often do not adequately reflect real-world tasks.

Consider the task of learning to score a goal in soccer. This problem requires knowing how to kick the ball to a certain position, inside the goals. A human would practise kicking the ball, experimenting with kicking at different angles and with different forces until some combination of angle and force results in the ball going into the goals. One would then try to perform subsequent kicks with a similar angle and force that previously resulted in a goal. Consider another soccer task: passing the ball to a teammate. While the basic action of kicking the ball to a certain position is the same as scoring a goal, we know that the position kicked to must intersect the trajectory of the teammate if they are moving, and the force applied, in general, should be less than when trying to score a goal. So while both tasks have the same underlying action—kicking the ball—how it is kicked changes depending on the task we want to achieve. If we had only discrete actions, each kick with a different angle and force would be considered a separate action and learning would be slow, since there is no generalisation or knowledge transfer between these similar actions. With continuous actions, a learning agent would try optimise the angle and force components of the kick in all possible cases, but would not easily be able to distinguish a goal-kick from a pass, and so the learned parameters would be unlikely to achieve either task. The kick action can instead be split into two *parameterised actions*: pass and goal-kick, each having their own angle and force parameters.

Parameterised actions thus combine discrete and continuous actions: they comprise a set of discrete actions where each is associated with one or more continuous *action-parameters* providing fine-grained control. This creates a more robust representation capable of solving complex problems, without necessarily having to search an entire continuous action space. In particular, such a parameterisation lends itself well to robotics, where actuators require a mix of different high-level actions with flexible control in order to perform different tasks. Learning with such parameterised actions has been shown to result in better control policies than using discrete or continuous actions alone [Masson *et al.* 2016].

The recent P-DQN deep reinforcement learning algorithm [Xiong *et al.* 2018] has demonstrated state-of-the-art performance with parameterised actions on tasks such as playing soccer and controlling agents in the complex multiplayer-online-battle-arena game, King of Glory. The focus of this research is to identify and improve aspects of P-DQN specific to how it handles action-parameters.

We briefly review the foundational concepts and techniques of modern reinforcement learning in Chapter 2. We discuss several RL algorithms for discrete, continuous, and parameterised action spaces in Section 2.1, and the extensions of those algorithms to use artificial neural networks for deep function approximation in Section 2.2. Chapter 3 details the P-DQN algorithm and our initial modifications to it for improved stability, followed by a comparison study against prior methods. In Chapter 4, we explore how the imbalance between actions explored during training detrimentally affects P-DQN and propose two modifications to its action-parameter loss function to account for this imbalance. Chapter 5 discusses issues resulting from the over-parameterisation of Q-values in P-DQN, which use all action-parameters as input, and propose a solution in the form of a novel multi-pass method. Finally, Chapter 6 discusses our findings and summarises our contributions.

Chapter 2

Background and Related Work

In this chapter we discuss modern reinforcement learning techniques for Markov decision processes and approaches for several different action spaces: discrete, continuous, and parameterised. For discrete actions, we focus on value-function based learning methods. In Section 2.1.1, we discuss the use of function approximation to deal with continuous state spaces, which are the norm for most modern domains. Policy search methods for continuous action spaces are briefly discussed in Section 2.1.2. Section 2.1.3 details the formulation of parameterised action spaces and the Q-PAMDP algorithm. We then discuss how these techniques have been extended with deep function approximation in Section 2.2, specifically by the DQN, DDPG, and PA-DDPG methods. Lastly, we discuss research related to parameterised action spaces in Section 2.4.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a form of trial-and-error learning in which an agent learns to perform a task by maximising numeric rewards based on the actions it executes in an environment. The agent repeatedly executes an action, transitions to a new state of the environment, and receives a reward, until a terminal state is reached. A *state* refers to the information given by the environment at a certain timestep and we refer to the sequence of steps from an initial state to a terminal state as an *episode* or *trajectory*.

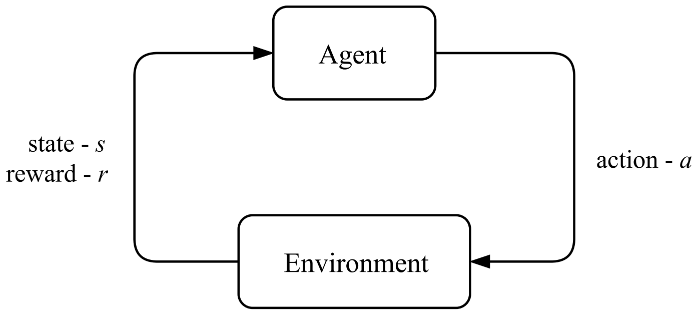


Figure 2.1: Reinforcement learning agent-environment interaction cycle.

Reinforcement learning algorithms typically consider domains where the state information sent to the agent contains all relevant information required to determine the next transition. This is the Markov property, which ensures the transition to a future state is dependent only on the action taken in the present state, and not on any preceding states. Such domains are formulated as *Markov decision processes* (MDPs)

[Sutton and Barto 1998]. An MDP is a tuple, $M = (S, A, P, R)$ where S is the set of possible states, A is the set of actions available to the agent, $P(s'|s, a)$ is the probability of transitioning to state s' after taking action a in state s, and $R(s, a, s')$ is the reward r obtained by taking action a in state s and transitioning to state s'.

The actions of an agent are determined by its *action policy* $\pi(a|s)$, which gives the probability of choosing action a in state s. Deterministic policies are denoted $\pi(s)$, the action chosen in state s. The aim of reinforcement learning algorithms is to find an action policy that maximises the cumulative discounted reward of the agent,

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}),$$

where $0 \leq \gamma \leq 1$ is a discount factor chosen by the algorithm that determines the importance of future rewards. State values, denoted by $V^\pi(s)$, represent the expected cumulative discounted reward under action policy π beginning from state s. This reward can be maximised by finding a solution to the *Bellman optimality equation* [Sutton and Barto 1998]:

$$V^*(s) = \max_a \sum_{s'} P(s'|s, a)\big(R(s, a, s') + \gamma V^*(s')\big), \tag{2.1}$$

where $V^* = V^{\pi^*}$ is an optimal value function, and π^* is an optimal action policy.

Value iteration is an algorithm that iteratively learns such an optimal value function. Starting from an arbitrary initialisation of V, the algorithm sweeps across all state values and updates them according to the Bellman optimality equation (2.1) at each iteration. A deterministic, optimal action policy is then given by greedily selecting the action that maximises V^*:

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s, a)\big(R(s, a, s') + \gamma V^*(s')\big).$$

Using this method, the greedy policy is guaranteed to converge in a finite number of steps even if the value function has yet to converge [Bertsekas 1987].

Clearly, iterating over all possible states is slow, usually infeasible and assumes explicit knowledge of the state transition and reward functions. State-action values, named *Q-values* and denoted by $Q^\pi(s, a)$, are similar to state values except they map state-action pairs. They represent the expected cumulative discounted reward for executing action a in state s and following the policy π thereafter. The Bellman optimality equation then becomes:

$$\begin{aligned} Q^*(s, a) &= \mathop{\mathbb{E}}_{r,s'}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right] && (2.2) \\ &= \sum_{s'} P(s'|s, a)\big(R(s, a, s') + \gamma \max_{a'} Q^*(s', a')\big). && (2.3) \end{aligned}$$

Using Q-values, one can learn a value function using samples gathered by the agent directly acting in the environment, without knowing the transition or reward functions of the domain. An optimal policy then, for a state s, greedily selects the action that maximises $Q^*(s, a)$.

Q-learning [Watkins and Dayan 1992] is a model-free *temporal-difference* (TD) algorithm that iteratively learns the optimal Q-value function for an MDP. The update rule is derived from the Bellman

Equation (2.3): for each new sample $(s_t, a_t, r_{t+1}, s_{t+1})$ from the environment at time step t,

$$Q^{\pi}_{t+1}(s_t, a_t) = Q^{\pi}_t(s_t, a_t) + \alpha_t\big(r_{t+1} + \gamma \max_{a'} Q^{\pi}_t(s_{t+1}, a') - Q^{\pi}_t(s_t, a_t)\big), \tag{2.4}$$

where $0 < \alpha_t \leq 1$ is the *learning rate*, or *step-size*. This update is independent of the next action chosen by the agent, hence Q-learning is *off-policy*. The purpose of the learning rate to prevent overshooting the optimal value by bounding the magnitude of updates. This is particularly important to prevent divergence during value-function approximation updates, which are covered in the next section.

2.1.1 Value Function Approximation

Value functions for finite discrete state spaces can usually be represented and learned using tabular methods. Continuous state spaces, on the other hand, are composed of real-valued state variables, $S \subseteq \mathbb{R}^n$, making the number of states infinite.

Linear value-function approximation [Parr *et al.* 2008] is an approach for cases where the value function cannot be represented exactly. A linear combination of features is used to represent the value-function:

$$\hat{V}(s) = \sum_{i=1}^{m} w_i \phi_i(s) = \mathbf{w}^{\top}\Phi(s),$$

where $\Phi = \{\phi_1, \phi_2, ..., \phi_m\}^{\top}$ is a set of features, $\phi_i(s)$ is the value of feature i in state s, and $\mathbf{w} \in \mathbb{R}^m$ is the parameter or weight vector. The approximation for $\hat{Q}^{\pi}$ is similarly defined by making the features functions of state and action. Practically, this results in the features being indexed by action. Parametric function approximation generalises to unexplored states since similar states produce similar values from the function, so fewer training samples are required. Hence function approximation avoids the time, memory, and data requirements of large value tables [Sutton and Barto 1998].

The representational power of the features determines the quality of the approximation, and hence the performance of the derived policy. Most value functions are too complex to be represented as combinations of the state variables alone, so functions of the variables are used to form some basis. The Fourier basis proposed by Konidaris *et al.* [2011] is one option, where terms of the Fourier series over different combinations of the state variables are used up to some order.

The next problem is finding a parameter vector **w** such that $\hat{V} \approx V^*$. In general, a perfect set of weights cannot be found to exactly approximate the function over all states since the basis has limited resolution. Gradient descent methods such as TD(λ) are typically employed for iterative learning of **w** by minimising the TD error. Assuming a fixed number of features and that the approximate value function $\hat{V}$ is a smooth differentiable function of **w** for all $s \in S$, the update rule for TD(λ) at time step t with sample $(s_t, a_t, r_{t+1}, s_{t+1})$ is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \delta_t \mathbf{e}_t, \tag{2.5}$$

where δ is the TD error:

$$\delta_t = r_{t+1} + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t),$$

and $\mathbf{e} \in \mathbb{R}^k$ is the vector of eligibility traces for each component of $\mathbf{w}_t$. Eligibility traces are a technique of increasing the learning rate by temporal credit assignment, updating the previous states encountered along the trajectory of samples to the current state. The update rule for **e** is

$$\mathbf{e}_{t+1} = \lambda \mathbf{e}_t + \nabla_{\mathbf{w}_t} \hat{V}(s_t),$$

where $\mathbf{e}_0 = \mathbf{0}$ and $0 \leq \lambda \leq 1$ is the exponential eligibility trace decay. When $\lambda = 0$, the update rule for TD(0) reduces to a more familiar form of gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \delta \nabla_{\mathbf{w}_t} \hat{V}(s_t).$$

A useful property of linear function approximation is that the derivative of $\hat{V}(s_t)$ with respect to the weight vector **w** is just the feature vector at that state:

$$\nabla_{\mathbf{w}_t} \hat{V}(s) = \Phi(s).$$

This property greatly simplifies update computations and is one reason for the former popularity of linear function approximation [Sutton and Barto 1998], along with convergence guarantees.

Sarsa(λ)

Sarsa(λ) is an iterative learning algorithm that applies TD(λ) to Q-values. Samples used in updates now include the action chosen for the next state, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. The parameter vector update step is the same as in Equation (2.5) but the calculations for δ and the eligibility traces change to

$$\delta_t = r_{t+1} + \gamma \hat{Q}_t^{\pi}(s_{t+1}, a_{t+1}) - \hat{Q}_t^{\pi}(s_t, a_t)$$

and

$$\mathbf{e}_{t+1} = \gamma \lambda \mathbf{e}_t + \nabla_{\mathbf{w}_t} \hat{Q}^{\pi}(s_t, a_t).$$

The main distinction between this and Watkin's Q-learning, which can similarly be extended with eligibility traces and TD(λ) updates for value function approximation, is that Sarsa(λ) is *on-policy*: the update rule depends on the next action chosen by the agent. This affects the learned policy because it takes into account any randomness of the agent's action selection, such as with ϵ-greedy policies [Sutton and Barto 1998].

Multi-Step Returns

TD(λ) methods use λ to more quickly backpropagate credit assignment to transitions experienced earlier in a trajectory. It can be seen as incorporating the total discounted future return from a state at time t,

$$r_t^{(\infty)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots . \tag{2.6}$$

However, one can also explicitly store the trajectory of transitions during learning and use the return directly in updates. In *n-step Q-learning* [Peng and Williams 1996] for instance, a finite horizon *n-step return* target is constructed of exponentially decayed rewards:

$$r_t^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n-1} \tag{2.7}$$

$$= \sum_{i=t}^{t+n-1} \gamma^{i-t} r_i \tag{2.8}$$

Algorithm 1 Sarsa(λ) with function approximation

```
Input: $\Phi$, $\alpha$, $\gamma$, $\lambda$, initial $\mathbf{w}$
Output: $\mathbf{w}$
  $\mathbf{e} = \mathbf{0}$
  $s =$ initial state from environment
  $a =$ action selected by policy $\pi(s)$
  repeat
    Execute action $a$
    Observe reward $r$, state $s'$
    $a' = \pi(s')$
    $\mathbf{e} = \gamma\lambda\mathbf{e} + \Phi(s, a)$
    $\delta = r + \gamma\mathbf{w}^\top\Phi(s', a') - \mathbf{w}^\top\Phi(s, a)$
    $\mathbf{w} = \mathbf{w} + \alpha\delta\mathbf{e}$
    $s = s'$
    $a = a'$
  until $s$ is a terminal state
```

which is then used as a target for the Q-value function during temporal difference updates

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n \max_{a'} Q(s_{t+n}, a') - Q(s_t, a_t) \right). \tag{2.9}$$

This is particularly impactful for long-running tasks or domains with sparse rewards because it can more quickly backpropagate the reward than with one-step lookahead updates. However, this does introduce bias and makes the algorithm on-policy.

2.1.2 Continuous Action Spaces

Value-function based reinforcement learning algorithms usually consider discrete actions, where selecting an action is the process of scanning through each possible action to determine which one maximises the current value or state-action value function. This is no longer possible with continuous action spaces, where actions lie in some real-valued and possibly multidimensional range, $a \in \mathbb{R}^m$.

Policy Search

Policy search [Deisenroth *et al.* 2013] uses a different approach to value-function methods. Instead of learning a value function that determines state quality and using it to derive a policy, an action policy is learned directly without discretising the action value range. Like value function approximations, however, the policy can be represented as as a linear combination of features: $\theta^\top\Phi(s)$. We use θ to represent the parameter vector for the action policy to distinguish it from the value-function parameter vector $\mathbf{w}$, although it serves a similar purpose. We denote a policy π using such a representation under weight vector θ by π_θ. Continuous action policies are often made stochastic by, for one example, effectively representing the policy as a normal distribution around such a linear combination of features with some standard deviation σ, which may be fixed or learned:

$$\pi_\theta(a|s) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big(- \frac{(a - \theta^\top \Phi(s)^2)}{2\sigma^2}\Big). \tag{2.10}$$

A trajectory, denoted τ, is a collection of samples (s, a, r, s') gathered over an episode. The accumulated reward, or *return*, for a trajectory is given by

$$R(\tau) = \sum_{t=0}^{\mathcal{T}} r_t(s_t, a_t).$$

Policy search methods attempt to update the policy's parameter vector based on these sampled trajectories such that trajectories with higher accumulated rewards become more likely. This increases the expected return given by

$$J(\theta) = \mathbb{E}\Big[R(\tau)|\theta\Big] = \int_\tau P_\pi(\tau)R(\tau)\,\mathrm{d}\tau, \tag{2.11}$$

where $P_\pi(\tau)$ is the distribution over the trajectories based on the current action policy:

$$P_\pi(\tau) = P(s_0)\prod_{t=0}^{T-1} P(s_{t+1}|s_t, \pi_\theta(s_t)).$$

Policy Gradient

In order to find an optimal θ^*, we can use iterative gradient ascent to maximise the expected return J_θ. This approach is known as *policy gradient* [Deisenroth *et al.* 2013; Sutton *et al.* 1999]. The update rule for θ at time t is

$$\theta_{t+1} = \theta_t + \alpha\nabla_\theta J(\theta), \tag{2.12}$$

where $\nabla_\theta J(\theta)$ is the policy gradient defined in this case by

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta P_\theta(\tau)R(\tau)\,\mathrm{d}\tau. \tag{2.13}$$

The next problem is how to calculate $\nabla_\theta J(\theta)$. The *Episodic Natural Actor Critic* (eNAC) algorithm [Peters *et al.* 2005] estimates it using the natural gradient, which optimises parameterised probability distributions faster than using the traditional gradient. The natural gradient uses a Fisher information matrix to account for successive policies possibly causing large differences to the trajectory distribution:

$$\begin{aligned} F_\theta &= \mathbb{E}_{P(\tau)}\Big[\nabla_\theta \log P_\theta(\tau)\nabla_\theta \log P_\theta(\tau)^\top\Big] \\ &= \mathbb{E}_{P(\tau)}\Bigg[\Big(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\Big)\Big(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\Big)^\top\Bigg]. \end{aligned}$$

The eNAC policy gradient is then

$$\nabla_\theta^{eNAC} J(\theta) = F_\theta^{-1}\nabla_\theta J_\theta = F_\theta^{-1}\mathbb{E}_{P(\tau)}\Bigg[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\Bigg].$$

The eNAC algorithm estimates the expectations of these terms by averaging the summations over multiple episodes using the current action policy π_θ. This requirement of episodic rollouts to perform each update, as opposed to TD methods which update at each time step of an episode, makes algorithms such as eNAC data inefficient and ill-suited to domains with very long episodes or unending tasks. One can instead incorporate value functions in a similar fashion to Q-learning for incremental updates, as we see in the next section.

Deterministic Policy Gradients

A more convenient definition of the policy gradient uses a state-action value function to estimate the expected return, rather than rollouts [Sutton *et al.* 1999]:

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_S \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)\,\mathrm{d}s\,\mathrm{d}a && (2.14)\\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\Big], && (2.15)\end{aligned}$$

where $\rho^\pi(s)$ is the discounted state visitation distribution for π. This is a different form of the policy gradient shown in Equation (2.13), where terms are defined over trajectory distributions due to the lack of an explicitly learned state-value function encoding the expected return.

Policy gradient methods based on the above are typically on-policy with stochastic action policies, however Silver *et al.* [2014] introduce a more efficient variant for deterministic policies which integrates only over the state space, rather than the state and action spaces. The *deterministic policy gradient* (DPG) is defined as follows:

$$\begin{aligned}\nabla_\theta J_d(\theta) &= \int_S \rho^\beta \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)\,\mathrm{d}s\,\mathrm{d}a && (2.16)\\ &= \mathbb{E}_{s\sim\rho^\beta}\Big[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)}\Big], && (2.17)\end{aligned}$$

where $\beta = \beta(a|s)$ is a stochastic policy corresponding to π_θ with added noise used for exploration during training. Silver *et al.* [2014] show that by re-parametrising a stochastic policy $\pi_\theta(a|s)$ by a deterministic policy $\pi_\theta(s)$ with a variation variable σ, Equation (2.16) is in fact a special case of the stochastic policy gradient in equation (2.14) when $\sigma \to 0$.

Actor-critic style algorithms can be employed to learn with the above. Such algorithms are comprised of two components: an actor which adjusts the parameters of policy $\pi_\theta(s)$, and a critic which estimates a differentiable action-value function $Q^\omega(s,a) \approx Q^\pi(s,a)$. For example, the *off-policy deterministic actor-critic* (OPDAC) algorithm [Silver *et al.* 2014] uses Q-learning updates for the critic while the deterministic policy gradient (2.17) is used to update the actor:

$$\begin{aligned}\delta_t &= r_t + \gamma Q^\omega(s_{t+1}, \pi_\theta(s_{t+1}) - Q^\omega(s_t, a_t)), && (2.18)\\ \omega_{t+1} &= \omega_t + \alpha_\omega \delta_t \nabla_\omega Q^\omega(s_t, a_t), && (2.19)\\ \theta_{t+1} &= \theta_t + \alpha_\theta \nabla_\theta \pi_\theta(s_t) \nabla_a Q^\omega(s_t, a_t)|_{a=\pi_\theta(s)}, && (2.20)\end{aligned}$$

where α_θ and α_ω are the learning rates for the actor and critic respectively. The use of incremental updates makes this more data efficient than rollout-based methods such as eNAC.

2.1.3 Parameterised Action Spaces

Parameterised action spaces [Masson *et al.* 2016] consist of a set of discrete actions, $\mathcal{A}_d = [K] = \{k_1, k_2, ..., k_K\}$, where each k has a corresponding continuous action-parameter $x_k \in \mathcal{X}_k \subseteq \mathbb{R}^{m_k}$, where m_k is the dimensionality. Adopting the notation of Xiong *et al.* [2018], the parameterised action space is thus:

$$\mathcal{A} = \bigcup_{k \in [K]} \{(k, x_k) | x_k \in \mathcal{X}_k\}. \tag{2.21}$$

A Markov decision processes with such an action space is referred to as a parameterised action Markov decision process (PAMDP). For a PAMDP $M = (S, \mathcal{A}, P, R)$, the action space $\mathcal{A}$ is composed of actions with their parameter spaces, and the transition probability function $P(s'|s, k, x_k)$ and reward function $R(s, k, x_k, s')$ now depend on the given action and its parameter.

Q-PAMDP

Q-PAMDP(κ), proposed by Masson *et al.* [2016], is a model-free reinforcement learning method for parameterised action spaces. Instead of directly learning the policy $\pi(k, x_k|s)$ to select both a discrete action and its parameter at once, the policy is split into a policy for discrete action selection, denoted $\pi^d(k|s)$, and another policy for action-parameter selection, $\pi^k(x_k|s)$, which is indexed by the action k. A parametric value-function approximation with weight vector **w** is used for the discrete action policy, so the action policy using **w** is denoted $\pi^d_{\mathbf{w}}(k|s)$. Similarly, a parametric function approximation with weight vector θ is used for the action-parameter policy, denoted $\pi^k_\theta(x_k|s)$. Then for a PAMDP, $M = (S, \mathcal{A}, P, R)$, the discrete MDP using a fixed action-parameter policy with a certain weight vector θ is given by $M_\theta = (S, \mathcal{A}_d, P_\theta, R_\theta)$, where $\mathcal{A}_d$ is the discrete action set. The transition and reward functions $P_\theta(s'|s, k)$ and $R_\theta(s, k, s')$ act as their parameterised action counterparts $P(s'|s, k, x_k)$ and $R(s, k, x_k, s')$, but with x_k determined by the fixed action-parameter policy π^k_θ.

Q-PAMDP (Algorithm 2) comprises a Q-learning algorithm for learning **w**, and an algorithm such as policy search for learning the parameter policy weight vector θ by optimising the expected discounted return:

$$J(\theta, \mathbf{w}) = \mathbb{E}_{s_0}\Big[V^\pi(s_0)\Big], \tag{2.22}$$

where s_0 is a state sampled according to the initial state distribution of the MDP, and V is a value function. With **w** fixed, Equation (2.22) is denoted $J_{\mathbf{w}}(\theta)$. The algorithms are alternated each iteration, with the action policy fixed while the parameter is updated for κ steps before fixing the parameter policy and updating the action policy to convergence. In Algorithm 2, P-UPDATE denotes the algorithm chosen for updating the parameter policy weight vector θ with respect to $J_{\mathbf{w}}(\theta)$, and Q-LEARN denotes a Q-learning algorithm.

The algorithm is proven to converge to a locally optimal solution for $\kappa = 1$, provided the choice of P-UPDATE algorithm converges to a local or global optimum. For $\kappa = \infty$, where θ is updated to convergence each iteration, P-UPDATE must to converge to a global optimum for the algorithm to converge locally.

Algorithm 2 Q-PAMDP(κ)

```
Input: $\theta_0$, $\mathbf{w}_0$
  $\mathbf{w} = $ Q-LEARN$^{(\infty)}(M_\theta, \mathbf{w}_0)$
  repeat
    $\theta = $ P-UPDATE$^{(\kappa)}(J_{\mathbf{w}}(\theta), \theta)$
    $\mathbf{w} = $ Q-LEARN$^{(\infty)}(M_\theta, \mathbf{w})$
  until $\theta$ converges
```

Using Sarsa(λ) for Q-LEARN and eNAC for P-UPDATE, Masson *et al.* [2016] empirically show that Q-PAMDP(1) and Q-PAMDP(∞) outperform both a discrete Sarsa(λ) agent using a fixed action-parameter policy, and a direct policy search agent using eNAC over the entire space on the Platform domain (Section 2.3.1) and Robot Soccer Goal domain (Section 2.3.2). This result indicates that a full search over the action-parameter space is not necessary to learn a good policy. Interestingly, eNAC on its own was shown to converge early to a very suboptimal policy on Platform.

2.2 Deep Reinforcement Learning

Deep learning is an alternative to basis functions and linear function approximation in reinforcement learning. It involves using artificial neural networks characterised by feedforward processing layers with non-linear transformations, or *activation functions*, between successive layers. This allows for high-level features to be learned from state variables directly, avoiding extensive feature engineering. The simplest layer representation is a set of dense, fully connected artificial neurons as shown in Figure 2.2. The capacity of a network is controlled by the number of hidden layers and artificial neurons used per layer, or depth and width respectively.

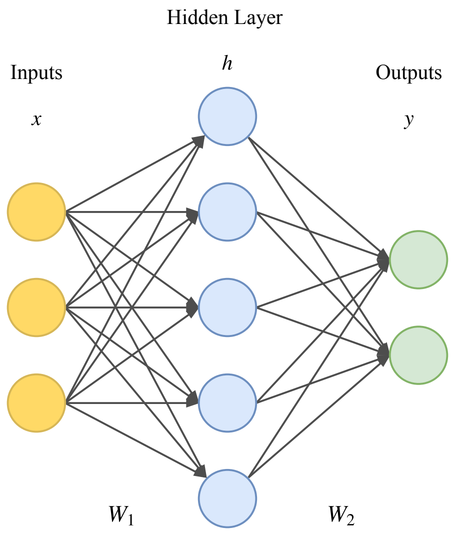


Figure 2.2: Example of a feedforward artificial neural network with a single fully connected hidden layer. The parameters of the network are given by the layer weights, $\theta = \{W_1, W_2\}$.

The network parameters are typically trained using some form of stochastic gradient descent but more complex and efficient optimisers such as RMSProp [Hinton *et al.* 2012] or Adam [Kingma and Ba 2014] have seen wide use. Other forms of artificial neural networks exist that account for different types of input data. Convolutional neural networks (CNNs) [Krizhevsky *et al.* 2012], for instance, exploit spacial information over input data typically from pixels in images, and recurrent neural networks (RNNs) [Medsker and Jain 1999] account for temporal correlations in sequences of data. For further information on the extensive topic of deep learning we refer the reader to Goodfellow *et al.* [2016].

Deep reinforcement learning has achieved super-human performance on tasks with high-dimensional spaces such as playing Atari games from pixels [Mnih *et al.* 2015], the game of Go [Silver *et al.* 2016], as well as robotic control [Levine *et al.* 2016; Lillicrap *et al.* 2015]. We now discuss several particular deep RL algorithms used for discrete, continuous, and parameterised action spaces.

2.2.1 Deep Q-Networks

Mnih *et al.* [2013 2015] extend Q-learning for discrete actions by using an artificial neural network to approximate the state-action value function $Q(s, a; \theta_Q) \approx Q^*(s, a)$, yielding a deep Q-network (DQN) with parameters θ_Q, illustrated in Figure 2.3.

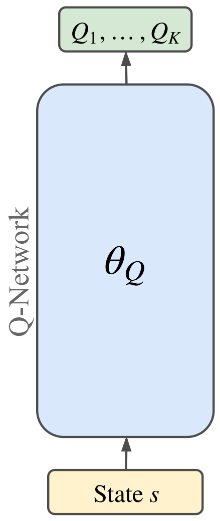


Figure 2.3: Abstraction of a deep Q-network. Given a state s as input, the network outputs a Q-value for each discrete action $a_i, i = 1, \ldots, K$.

Minibatch updates are performed during training by sampling a number of transitions uniformly randomly from *experience replay memory*—a finite buffer storing the last N transitions experienced by the agent. This is done to enhance data efficiency and helps avoid oscillation and divergence during learning by averaging updates over a number of samples. The network is trained using stochastic gradient methods to minimise the following least-squares loss function based on the Bellman Equation (2.2) at every iteration:

$$L(\theta_Q) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim D}\left[\frac{1}{2}\big(y - Q(s, a; \theta_Q)\big)^2\right], \tag{2.23}$$

where

$$y = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_Q^-) \tag{2.24}$$

is the update target and D is the replay memory. A clone of the Q-network with parameters θ_Q^- is used to calculate the update targets. This *target network* is synchronised with the latest parameters $\theta_Q^- = \theta_Q$ every C iterations; this reduces oscillations and the chance of divergence during learning. The derivative of the loss function with respect to the network weights gives the following gradient:

$$\nabla_{\theta_Q} L(\theta_Q) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\Big(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_Q^-) - Q(s, a; \theta_Q)\Big)\nabla_{\theta_Q} Q(s, a; \theta_Q)\right]. \tag{2.25}$$

The full DQN method is detailed in Algorithm 3. While standard Q-learning converges to an optimal policy, this approach lacks any theoretical convergence guarantees as a result of using non-linear function approximation.

Algorithm 3 Deep Q-Learning with Experience Replay [Mnih *et al.* 2015]

```
Input: Learning rate $\alpha$, target network update frequency $C$, exploration parameter $\epsilon$, minibatch size $B$, probability distribution $\xi$, replay memory capacity $N$, discount factor $\gamma$, maximum episodes $E_{max}$.
Initialise replay memory $D$ to capacity $N$
Initialise network weights $\theta_Q$
Initialise target network weights $\theta_Q^- = \theta_Q$
Initialise counters $t = 0, e = 0$
repeat
  $s_t =$ initial state from environment
  repeat
    Select action $a_t = \begin{cases} \text{argmax}_{a \in \mathcal{A}} Q(s_t, a; \theta_Q) & \text{with probability } 1 - \epsilon \\ \text{random sample from distribution } \xi & \text{with probability } \epsilon \end{cases}$
    Execute action $a_t$, observe reward $r_t$ and next state $s_{t+1}$
    Store transition $\{s_t, a_t, r_t, s_{t+1}\}$ in $D$
    Sample a minibatch of $B$ transitions $\{s_b, a_b, r_b, s_{b+1}\}_{b \in [B]}$ randomly from $D$
    Set $y_b = \begin{cases} r_b & \text{if } s_{b+1} \text{ is terminal} \\ r_b + \max_{a' \in \mathcal{A}} \gamma Q(s_{b+1}, a'; \theta_Q^-) & \text{otherwise} \end{cases}$
    With $\{s_b, a_b, y_b\}_{b \in [B]}$:
      Compute $\nabla_{\theta_Q} L_Q$ according to Equation (2.25) and update $\theta_Q$
    Every $C$ steps, update target network: $\theta_Q^- = \theta_Q$
    $t = t + 1$
  until $s_t$ is terminal
  $e = e + 1$
until $e \geq E_{max}$
```

2.2.2 Deep Deterministic Policy Gradients

Lillicrap *et al.* [2015] present an actor-critic method with deep function approximation for high dimensional, continuous action spaces. Their Deep Deterministic Policy Gradient (DDPG) algorithm extends DPG (Section 2.1.2) by using artificial neural networks to represent the actor and critic policies.

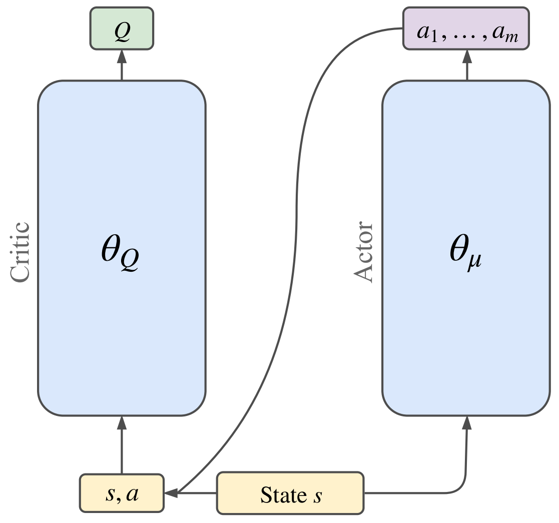


Figure 2.4: Illustration of the actor-critic network architecture for DDPG. Given a state, the actor network produces an m-dimensional continuous action vector and the critic predicts a scalar Q-value for that particular state and action.

The actor determines the continuous action policy $\pi(s;\theta_\mu)$ with parameters θ_μ while, like DQN, the critic minimises least squares Bellman error to learn an estimate of the state-action value function:

$$L(\theta_Q) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim D}\left[\frac{1}{2}\Big(r + \gamma Q(s', \pi(s';\theta_\mu);\theta_Q^-) - Q(s,a;\theta_Q)\Big)^2\right]. \tag{2.26}$$

The gradient for the critic loss is the same as in DQN, Equation (2.23). Updates to the actor network aim to find a policy that maximises the expected return, which is equivalent to minimising the negative Q-value function in the following loss:

$$L_\mu(\theta_\mu) = \mathop{\mathbb{E}}_{s}\Big[-Q\big(s, \pi(s;\theta_\mu);\theta_Q\big)\Big]. \tag{2.27}$$

Gradients with respect to the actions are backpropagated through the critic network and used to update the actor network with the policy gradient, based on (2.17):

$$\nabla_{\theta_\mu} L(\theta_\mu) = \mathbb{E}_s\Big[\nabla_a Q(s, \pi(s;\theta_\mu);\theta_\mu)\nabla_{\theta_\mu}\pi(s;\theta_\mu)\Big]. \tag{2.28}$$

Similar to DQN, target networks for the actor and critic are used for stability during training. Instead of synchronising the weights every few steps, however, Polyak averaging is used to *soft update* the target networks every iteration:

$$\theta^- = \tau\theta + (1-\tau)\theta^-, \tag{2.29}$$

where the averaging factor $\tau \in (0,1]$ determines the mixing proportion between the weights of the original and target networks, with $\tau = 1$ corresponding to copying the weights of the original network.

Algorithm 4 DDPG [Lillicrap *et al.* 2015]

```
**Input:** Learning rates $\{\alpha_Q, \alpha_\mu\}$, target networks averaging factor $\tau$, exploration noise $\mathcal{N}$, minibatch size $B$, replay memory capacity $N$, discount factor $\gamma$, maximum episodes $E_{max}$.
Initialise replay memory $D$ to capacity $N$
Initialise network weights $\theta_Q$, $\theta_\mu$
Initialise target network weights $\theta_Q^- = \theta_Q$, $\theta_\mu^- = \theta_\mu$
Initialise counters $t = 0, e = 0$
repeat
  $s_t =$ initial state from environment
  repeat
    Select action $a_t = \pi(s_t; \theta_\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$, observe reward $r_t$ and next state $s_{t+1}$
    Store transition $\{s_t, a_t, r_t, s_{t+1}\}$ in $D$
    Sample a minibatch of $B$ transitions $\{s_b, a_b, r_b, s_{b+1}\}_{b\in[B]}$ randomly from $D$
    Set $y_b = \begin{cases} r_b & \text{if } s_{b+1} \text{ is terminal} \\ r_b + \gamma Q(s_{b+1}, \pi(s_{b+1}; \theta_\mu^-); \theta_Q^-) & \text{otherwise} \end{cases}$
    With $\{s_b, a_b, y_b\}_{b\in[B]}$:
      Update the critic parameters $\theta_Q$ by minimising $L_Q$ according to Equation (2.26)
      Update the actor parameters $\theta_\mu$ using the policy gradient Equation (2.28)
    Update target networks $\theta_Q^- = \tau\theta_Q + (1-\tau_Q)\theta_Q^-$, $\theta_\mu^- = \tau\theta_\mu + (1-\tau)\theta_\mu^-$
    $t = t + 1$
  until $s_t$ is terminal
  $e = e + 1$
until $e \geq E_{max}$
```

2.2.3 PA-DDPG

Hausknecht and Stone [2016a] present the first method of applying deep reinforcement learning to parameterised action spaces. They do so by treating both the discrete actions and continuous action-parameters as a single continuous action vector. The discrete action policy is represented by continuous values $f_1, f_2, \dots, f_K$ where $f_i \in [-1, 1]$. An ϵ-greedy or softmax policy is then used to select a discrete action. This essentially relaxes a parameterised action space (2.21) into a continuous one:

$$\mathcal{A} = \{(f_{1:K}, x_{1:K}) | f \in \mathbb{R}, x_k \in \mathcal{X}_k \forall k \in [K]\}. \tag{2.30}$$

The standard DDPG algorithm can then be applied, with the actor network outputting the joint policy vector $(f_1, \dots, f_K, x_1, \dots, x_K)$ and the critic network providing Q-values for the entire policy vector. Following Wei *et al.* [2018], we henceforth refer to this method as *parameterised action DDPG* (PA-DDPG).

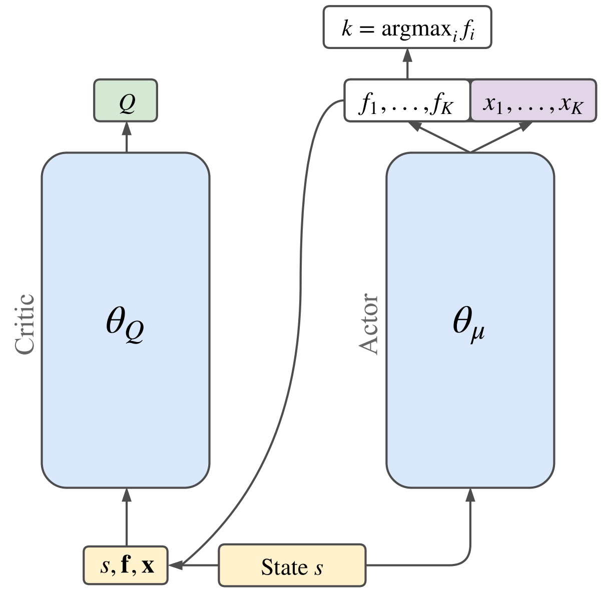


Figure 2.5: Illustration of the PA-DDPG network architecture.

No information is given to the network regarding which action is executed during training, nor which action-parameter is associated with which action. Optimisation is therefore performed over the entire joint policy vector $(\mathbf{f}, \mathbf{x})$. Despite this, PA-DDPG still demonstrates good performance when learning to score goals in Half Field Offense (Section 2.3.3).

Mixed n-Step Return Targets

Hausknecht and Stone [2016b] further extend PA-DDPG to use n-step returns (Section 2.1.1) to improve learning. However, since n-step returns makes updates on-policy, samples are required to be decorrelated. While experience replay memory—which is commonly employed by off-policy algorithms with neural network function approximation such as DQN—does decorrelate updates to some extent, it is unsuitable for on-policy algorithms because it includes data generated by older policies.

Hausknecht and Stone [2016b] therefore propose mixed n-step return targets: a technique that compromises between off-policy and on-policy updates by introducing a mixing ratio $\beta \in [0, 1]$. An update target is then constructed out of both one-step and n-step returns:

$$y_t = \beta \Bigg[\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i \Bigg] + (1 - \beta) \Big[r_t + \gamma \max_{a'} Q(s_t, a') \Big]. \tag{2.31}$$

The first term represents the on-policy discounted return computed over the agent's trajectory up to n steps from the current state (to account for long or unending tasks), while the second term is the standard off-policy one-step lookahead target. The choice of n and β can be seen as deciding the trade-off between bias and variance, similar to λ in TD(λ) [Watkins 1989]. In general, values of β closer to zero are found to be better, which we discuss further in Section 3.1.1 and Appendix E.

2.2.4 Exploration Strategies

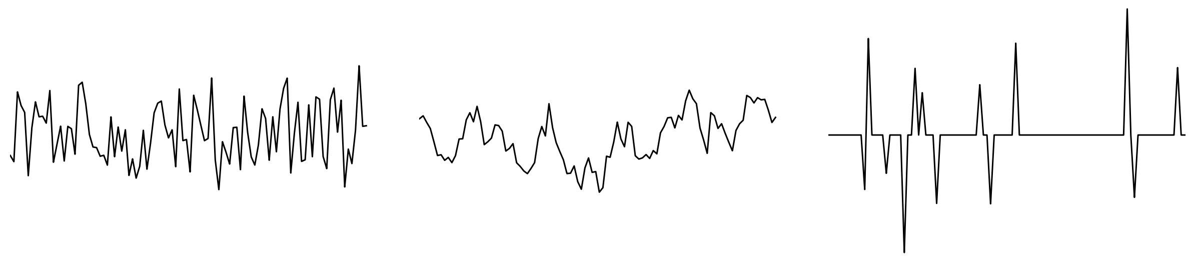
Exploration is essential in reinforcement learning to learn accurate value function estimates; ensuring different state and action combinations are adequately visited during training avoids agents getting stuck exploiting suboptimal policies. Different exploration strategies are employed depending on the type of action space used. We detail several common approaches in this section, particularly those used for parameterised action spaces.

Discrete actions: An ϵ-greedy strategy is typically employed for exploration, where the agent uniformly randomly chooses a discrete action with probability ϵ, and acts greedily with probability $1 - \epsilon$. Another options is to use softmax exploration, where actions are selected based on a Boltzmann distribution over the Q-values:

$$\pi(a|s) = \frac{\exp(Q(s, a)/\tau)}{\sum_{a'} \exp(Q(a')/\tau)},$$

where $\tau \in \mathbb{R}^+$ in this case represents a *temperature* parameter dictating the spread of probabilities over the actions: a large temperature tends towards a more uniform probability distribution, while in the limit $\tau \to 0$, softmax exploration tends to a greedy action selection. The main difference between this and ϵ-greedy is that softmax exploration picks actions with higher Q-values more often, while ϵ-greedy is as likely to pick the worst action as the best during exploration.

Continuous actions: Policies with Gaussian noise $\mathcal{N}(\mu, \sigma^2)$ are commonly used for exploration when using continuous actions [Deisenroth *et al.* 2013; Silver *et al.* 2014]. However, Lillicrap *et al.* [2015] found that using an Ornstein-Uhlenbeck (OU) process [Uhlenbeck and Ornstein 1930] to generate noise for exploration resulted in better performance on physical control tasks with momentum. The OU process models the velocity of a Brownian particle with friction, which results in temporally correlated values centred around 0. The process is characterised by three variables: $\theta > 0$, $\sigma > 0$, and a drift term μ (typically 0).



(a) Gaussian (b) Ornstein-Uhlenbeck (c) ϵ-uniform

Figure 2.6: Different types of noise used for exploration. The scale of the Gaussian and Ornstein-Uhlenbeck noise samples is exaggerated for illustrative purposes, since they produce small deviations from the mean while the ϵ-uniform approach selects values over the entire range.

Parameterised actions: The discrete action and continuous action-parameter policies can be treated separately for exploration. This was the approach by Masson *et al.* [2016], where a softmax discrete action policy is chosen with additive Gaussian noise for the action-parameters. Hausknecht and Stone [2016ab] and Xiong *et al.* [2018], on the other hand, use an ϵ-greedy discrete action policy with uniform random exploration for the continuous action-parameters. So with probability ϵ, choose a random discrete action with its action-parameter sampled uniformly randomly within its range, and act greedily otherwise. For brevity, we refer to this strategy as ϵ-*uniform* exploration. A visual comparison between Gaussian noise, OU noise, and ϵ-uniform exploration is shown above in Figure 2.6.

2.2.5 Action-Parameter Bounding

Action-parameters, and continuous actions in general, typically have well-defined bounds on the range of values available to them. These usually reflect physical limitations, for example maximum actuator torques or joint rotation angles for robots in the real world. We discuss the three strategies used to enforce these bounds for parameterised actions in deep reinforcement learning methods.

Squashing Functions

The most common way of enforcing these bounds is by applying an invertible squashing function, usually the hyperbolic tangent ($\tanh$), to the action vector chosen by the agent. While this ensures the agent can never exceed the boundaries of its actions, this approach can be problematic if the squashing function becomes saturated. When this occurs, the gradient of the squashing function tends to 0, making updates have very small impact on the policy. This strategy is used by Wei *et al.* [2018] to bound action-parameters.

Inverting Gradients

Another approach is to alter the update gradients such that they push the policy to within valid ranges. The *inverting gradients* method [Hausknecht and Stone 2016a] downscales gradients as they approach the action boundaries, and inverts their direction if the boundaries are exceeded:

$$\nabla_x = \nabla_x \cdot \begin{cases} (x_{\text{max}} - x)/(x_{\text{max}} - x_{\text{min}}) & \text{if } \nabla_x \text{ suggests increasing } x \\ (x - x_{\text{min}})/(x_{\text{max}} - x_{\text{min}}) & \text{otherwise.} \end{cases} \tag{2.32}$$

This avoids the problem of saturation and thus allows for faster policy changes. For example, if the critic indicates that the action-parameter should decrease when its at the upper bound, inverting gradients would respond immediately while a saturated squashing function would require many updates to actually change the policy. While the inverting gradients approach does not explicitly limit the range of values available, Hausknecht and Stone [2016a] found that agents did not exceed the maximum or minimum values in their experiments.

Squared-Loss Penalty

A simpler alternative used by Xiong *et al.* [2018] is to add a term to the loss function that discourages values exceeding the given bounds. For instance, a squared-loss penalty on out-of-bounds values:

$$L_x + \begin{cases} (x_{\text{max}} - x)^2 & \text{if } x > x_{\text{max}} \\ (x - x_{\text{min}})^2 & \text{if } x < x_{\text{min}} \\ 0 & \text{otherwise,} \end{cases} \tag{2.33}$$

where L_x is the original loss. In practice this penalty term is calculated per dimension for each action-parameter, as with inverting gradients. This has the benefit of avoiding the problems associated with squashing functions without complicated gradient tinkering. At the time of writing there has been no study on the effectiveness of this approach compared to inverting gradients or squashing functions.

2.3 Domains

Unlike the set of Atari games for discrete actions [Mnih *et al.* 2015], and MuJoCo for continuous actions [Todorov *et al.* 2012], there is not yet any widely accepted set of benchmark domains for parameterised action spaces. However, the Platform and Robot Soccer Goal domains introduced by Masson *et al.* [2016], along with the more complex Half Field Offense used by Hausknecht and Stone [2016a], form a decent set of tasks with varying levels of difficulty and number of state features between them.

2.3.1 Platform

The Platform domain, introduced by Masson *et al.* [2016], consists of a 2-dimensional game world with stationary platforms, moving enemies, and the agent. Two basic actions are available to the agent: run and jump, which continue for a fixed period or until the agent lands again respectively. The jump is parameterised into two separate actions: a hop to get over enemies, and a leap to traverse the gaps between platforms. Initially, the leap parameter is too small to clear the gaps between the platforms. The reward function is based on the normalised distance the agent moves towards the goal. Each platform has a single enemy which patrols back and forth at a constant speed. The enemies cannot jump and move only when the agent is within the horizontal range of the platform; this is a simplification that s the feature set to consider only a single enemy's horizontal position and velocity, $(x_e, \dot{x_e})$, at a time. An episode ends when the agent touches an enemy, falls into a gap between two platforms, or reaches the goal.

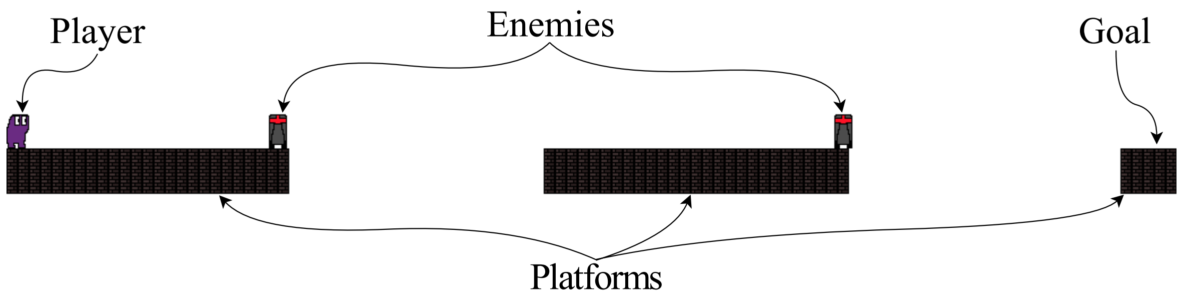


Figure 2.7: Initial configuration of the Platform domain. The agent starts on the edge of the left platform and has to traverse over enemies and gaps to reach the goal on the far right.

The basic feature set is composed of the agent's horizontal position and velocity and that of the enemy on the current platform: $(x, \dot{x}, x_e, \dot{x_e})$. The enhanced feature set, $(x, \dot{x}, x_e, \dot{x_e}, x_p, w_p, w_{p'}, g, h)$, includes the horizontal position of the current platform, x_p, the widths of the current and next platforms, w_p and $w_{p'}$, and the gap and height between the current and next platforms, g and h respectively.

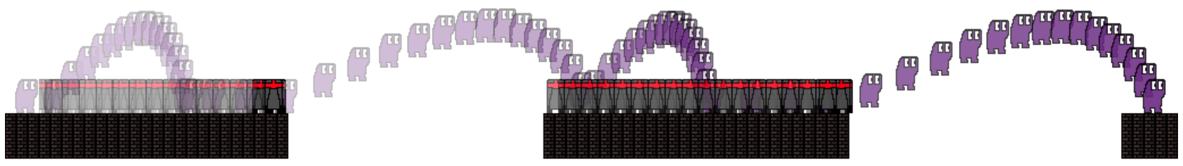


Figure 2.8: Successful episode of the Platform domain.

Based on the source code generously released by Masson *et al.* [2016], we reimplemented the Platform domain using the OpenAI Gym interface [Brockman *et al.* 2016]. Source code for our version is publicly available online.[1]

[1]`https://github.com/cycraig/gym-platform`

2.3.2 Robot Soccer Goal

The Robot Soccer Goal domain is adapted by Masson *et al.* [2016] from 2D RoboCup [Kitano *et al.* 1997], a robot soccer challenge domain. It focuses on a particular task in soccer: a striker scoring by shooting the ball past a defensive goal keeper and into the goals. Two basic actions are available to the agent: kick-to(x, y), which kicks to ball towards position (x, y) on the field; and shoot-goal(h), which shoots the ball towards a position h along the goal line. Gaussian noise is added to each action. The shoot-goal action is split into two parameterised actions: shoot-goal-left and shoot-goal-right. This is because the agent has to shoot around the keeper to score a goal, either to the left or right of it but never directly at it. The agent automatically moves towards the ball whenever it is not in possession of it. The keeper either moves towards the ball or, if the agent shoots at the goal, moves to intercept it.

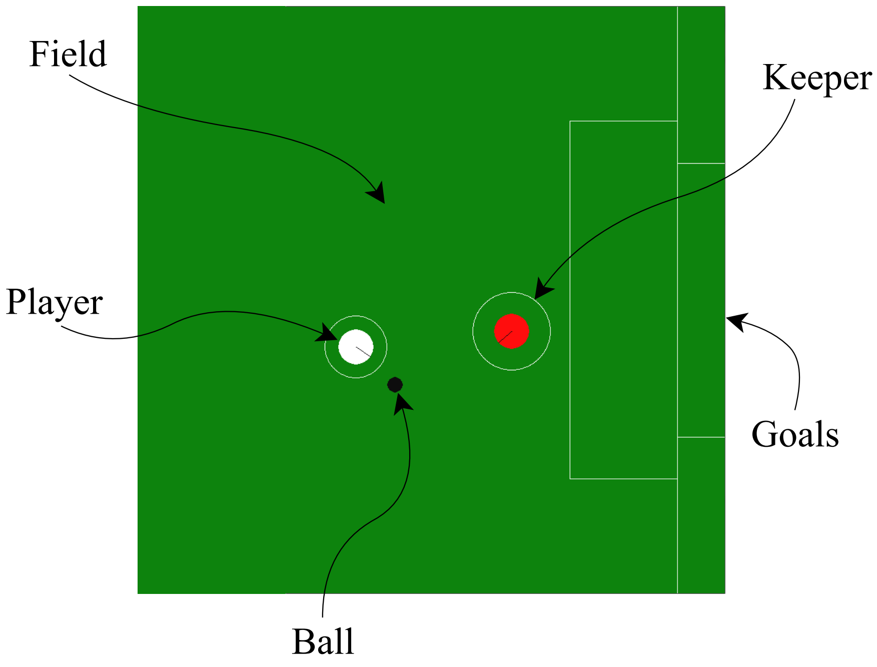


Figure 2.9: Robot Soccer Goal domain.

The agent starts in possession of the ball, at a random position along the left bound of the field. The keeper is positioned between the ball and the goal. The agent has a position (x, y), velocity $(\dot{x}, \dot{y})$ and orientation (θ), as does the keeper. The ball also has a position and velocity, so each state has 14 variables: $(x, y, \dot{x}, \dot{y}, \theta, x_k, y_k, \dot{x_k}, \dot{y_k}, \theta_k, x_b, y_b, \dot{x_b}, \dot{y_b})$, where the subscripts k and b distinguish the variables of the keeper and ball respectively. An episode ends when the keeper intercepts the ball, the agent scores a goal, or the ball leaves the field. The agent receives a reward of 50 scoring a goal, $-d(b, g)$ when the ball is intercepted or leaves the field, where $d(b, g)$ is the distance of the ball to the goal, and 0 otherwise. Based on the version by Masson *et al.* [2016], source code for our Open AI Gym implementation of Robot Soccer Goal is available online.[2]

[2]`https://github.com/cycraig/gym-goal`

2.3.3 Half Field Offense

The Half Field Offense (HFO) domain is a more general abstraction of 2D RoboCup. Hausknecht and Stone [2016a] use HFO to simulate the task of a single agent learning to score a goal in soccer without a goalie.

At the beginning of each episode, the agent and ball are randomly positioned on the offensive (right) half of the field. The agent must learn to approach and kick the ball into the goals using three parameterised actions: dash, turn, and kick. The state space is composed of 58 continuously-valued variables, describing the position, velocity, orientation, and stamina of the agent, the ball's location and velocity, the relative distance to landmark features such as the goals and so on.[3]

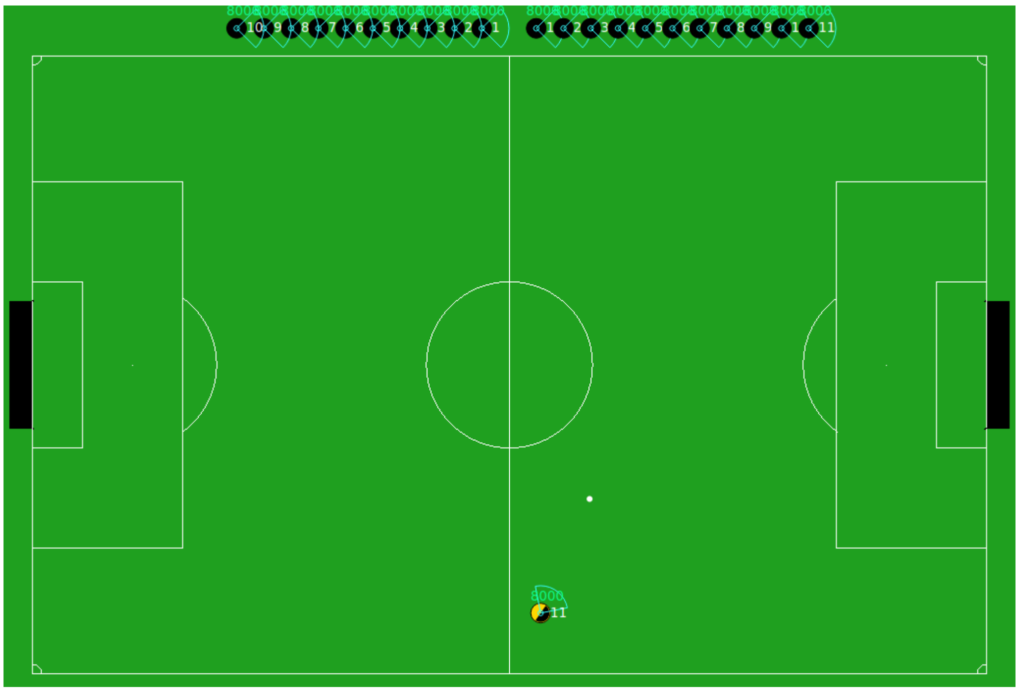


Figure 2.10: Half Field Offense domain. The agent is tasked with learning to kick the ball into the rightmost goals without a goalie present.

Hausknecht and Stone [2016a] use a dense, hand-crafted reward function:

$$r_t = d_{t-1}(a,b) - d_t(a,b) + \mathbb{I}_{t+1}^{\text{kick}} + 3\big(d_{t-1}(a,b) - d_t(b,g)\big) + 5\mathbb{I}_t^{\text{goal}}, \tag{2.34}$$

where $d(a,b)$ and $d(b,g)$ are the distances between the ball and agent or centre of goals respectively, and $\mathbb{I}^{\text{kick}}$ is a binary reward given the first time the agent is within kicking distance of the ball, and $\mathbb{I}^{\text{goal}}$ is a terminal reward for scoring a goal.

[3]A full description of the environment can be found at `https://github.com/mhauskn/HFO/blob/master/doc/manual.pdf`.

While the task appears simpler than Robot Soccer Goal at first, the relatively large number of state variables and finer-grain control required, with the agent first having to learn to approach the ball using the turn and dash actions rather than moving towards it automatically, result in HFO being more difficult.

2.4 Related Work

Several recent deep reinforcement learning approaches for parameterised actions follow the strategy by Hausknecht and Stone [2016a] of collapsing the parameterised action space into a continuous one.

Hussein *et al.* [2018], for instance, present a deep imitation learning approach for parameterised actions using long-short-term-memory (LSTM) networks with a joint action and action-parameter policy. They demonstrate their method learns to score goals on HFO with less task-specific knowledge than PA-DDPG when using a less complex reward function.

Agarwal [2018] introduces skills for multi-goal environments with parameterised action spaces to achieve multiple related goals. They similarly treat the parameterised actions as continuous and demonstrate success on robotic manipulation tasks by combining PA-DDPG with hindsight experience replay and their skill library.

One can alternatively view parameterised actions as a two-level hierarchy: Klimek *et al.* [2017] use this approach to learn a reach-and-grip task using a single network to represent a distribution over macro (discrete) actions and their lower-level action-parameters. Wei *et al.* [2018] also take a hierarchical approach but instead condition the action-parameter policy on the discrete action chosen to avoid predicting all action-parameters at once. The authors use this strategy to develop a parameterised action version of Trust Region Policy Optimisation (TRPO) [Schulman *et al.* 2015], yielding PATRPO. While their preliminary results show the method achieves good performance on the Platform domain, it fails to learn to score goals on HFO.

At the time of writing, we note that only one of the deep reinforcement learning approaches discussed above, PATRPO, explicitly treats parameterised actions as separate. We further note that none of these works compare against Q-PAMDP and, while some use HFO, most define their own domains to use as benchmarks.

2.5 Summary

We briefly covered modern reinforcement learning and deep RL techniques for Markov decision processes with discrete action spaces (Sarsa(λ), DQN) and continuous action spaces (eNAC, DDPG). Masson *et al.* [2016] formalised the notion of parameterised action spaces, a combination of discrete and continuous actions, and introduced the Q-PAMDP algorithm. Hausknecht and Stone [2016a] proposed PA-DDPG to learn in a parameterised action space by collapsing the space into a continuous one, a method employed by most subsequent deep RL approaches for parameterised actions. Three benchmark domains for parameterised actions were presented: Platform, Robot Soccer Goal, and Half Field Offense. In the next chapter, we detail the P-DQN algorithm for parameterised actions, along with our modifications to it, and compare it to previous methods.

Chapter 3

Parameterised Deep Q-Networks

The initial deep reinforcement learning approaches for parameterised action spaces treat both the discrete actions and their action-parameters as a joint continuous action vector [Hausknecht and Stone 2016ab; Hussein *et al.* 2018], allowing for standard continuous action algorithms such as DDPG to be applied directly. As discussed in Section 2.2.3, this can be seen as relaxing the parameterised action space

$$\mathcal{A} = \bigcup_{k \in [K]} \{(k, x_k) | x_k \in \mathcal{X}_k\}. \tag{3.1}$$

into a continuous action space:

$$\mathcal{A} = \{(f_{1:K}, x_{1:K}) | f \in \mathbb{R}, x_k \in \mathcal{X}_k \forall k \in [K]\}. \tag{3.2}$$

However, not only does this fail to exploit the disjoint nature of different parameterised actions, but optimising over the joint action and action-parameter space can lead to premature convergence to suboptimal policies, as occurred in experiments by Masson *et al.* [2016]; Wei *et al.* [2018].

To address this, Xiong *et al.* [2018] introduce a method that works in the parameterised action space directly, without relaxing it into a continuous action space, by combining DQN and DDPG. Their *Parameterised Deep Q-Network* (P-DQN) algorithm uses a standard Q-network to approximate Q-values used for discrete action selection, in addition to providing critic gradients for an actor network that determines the continuous action-parameter values for all actions. While Xiong *et al.* [2018] show that P-DQN outperforms PA-DDPG on Half Field Offense, they do so using asynchronous parallel workers which are not present in PA-DDPG or other previous algorithms for parameterised actions. In this chapter, we discuss the original P-DQN algorithm and our modifications to it before concluding with a fair comparison study against prior methods.

3.1 P-DQN

By framing the problem as a PAMDP directly, rather than alternating between a discrete and continuous action MDP as with Q-PAMDP [Masson *et al.* 2016] or a single continuous action MDP as with PA-DDPG [Hausknecht and Stone 2016a], P-DQN necessitates a change to the Bellman Equation (2.2) to incorporate continuous action-parameters:

$$Q(s, k, x_k) = \mathop{\mathbb{E}}_{r,s'} \left[r + \gamma \max_{k' \in [K]} \sup_{x_{k'} \in \mathcal{X}_{k'}} Q(s', k', x_{k'}) \middle| s, k, x_k \right]. \tag{3.3}$$

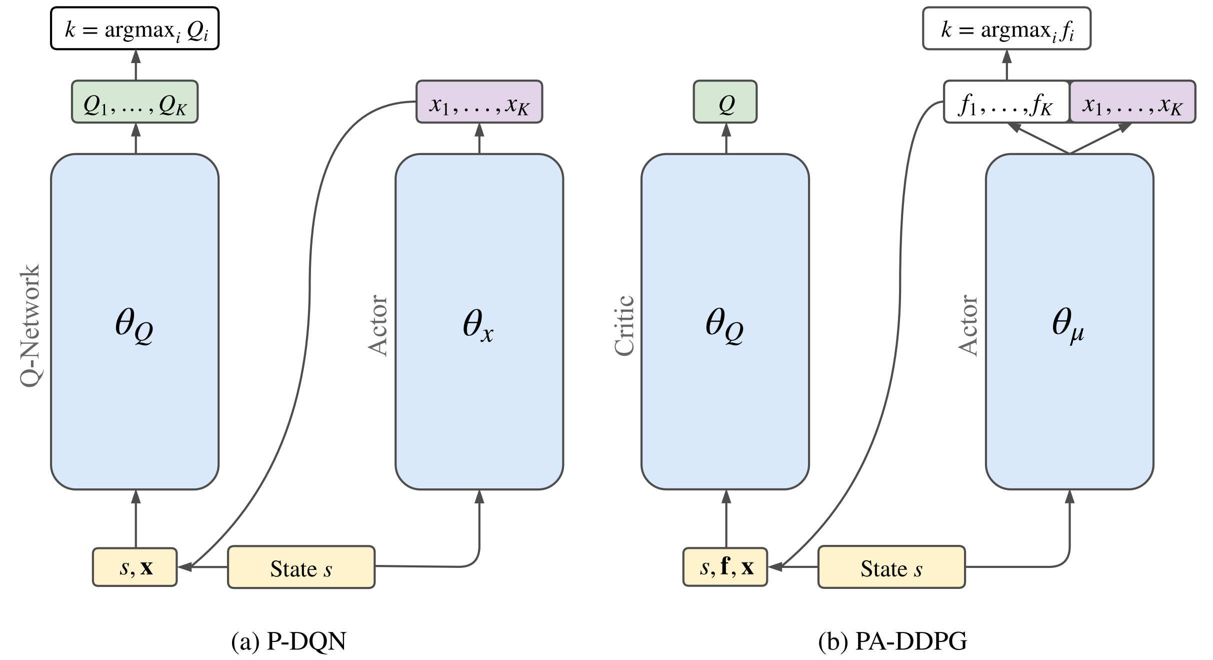


Figure 3.1: Illustration of the P-DQN [Xiong *et al.* 2018] and PA-DDPG [Hausknecht and Stone 2016a] artificial neural network architectures for parameterised action spaces. The primary difference is that PA-DDPG predicts a scalar Q-value and uses a continuous parameterisation of discrete actions $f_i, i = 1, \dots, K$, while P-DQN selects discrete actions by maximising the Q-values explicitly. Take special note that the entire action-parameter vector $\mathbf{x} = (x_1, \dots, x_K)$ is fed into the Q-network of P-DQN, as we show this becomes an issue in later chapters.

To avoid the computationally intractable calculation of the supremum over $\mathcal{X}_k$, Xiong *et al.* [2018] state that when the Q function is fixed, one can view $\text{argsup}_{x_k \in \mathcal{X}_k} Q(s, k, x_k)$ as a function $x_k^Q : S \to \mathcal{X}_k$ for any state $s \in S$ and $k \in [K]$. This allows the Bellman Equation for parameterised actions (3.3) to be rewritten as:

$$Q(s, k, x_k) = \mathop{\mathbb{E}}_{r,s'} \left[r + \gamma \max_{k' \in [K]} Q(s', k', x_{k'}^Q(s')) \middle| s, k, x_k \right]. \tag{3.4}$$

P-DQN uses a deep neural network with parameters θ_Q to represent $Q(s, k, x_k; \theta_Q)$, and a second deterministic actor network with parameters θ_x to represent the action-parameter policy $x_k(s; \theta_x) : S \to \mathcal{X}_k$, an approximation of $x_k^Q(s)$. With this formulation it is easy to apply the standard DQN approach of minimising the mean-squared Bellman error to update the Q-network using minibatches sampled from replay memory D, replacing a with (k, x_k):

$$L_Q(\theta_Q) = \mathop{\mathbb{E}}_{(s,k,x_k,r,s') \sim D} \left[\frac{1}{2} \big(y - Q(s, k, x_k; \theta_Q) \big)^2 \right], \tag{3.5}$$

where $y = r + \gamma \max_{k' \in [K]} Q(s', k', x_{k'}(s'; \theta_x); \theta_Q)$ is the update target derived from Equation (3.4). Then, the loss for the actor network in P-DQN is given by the negative sum of Q-values:

$$L_x(\theta_x) = \mathop{\mathbb{E}}_{s \sim D} \left[-\sum_{k=1}^{K} Q\big(s, k, x_k(s; \theta_x); \theta_Q\big) \right], \tag{3.6}$$

and although this choice was not motivated by Xiong *et al.* [2018], it resembles the deterministic policy gradient loss used by PA-DDPG where a scalar critic value is used over all action-parameters [Hausknecht and Stone 2016a]:

$$L_\mu(\theta_\mu) = \mathop{\mathbb{E}}_{s\sim D}\Big[-Q\big(s,\pi(s;\theta_\mu);\theta_Q\big)\Big]. \tag{3.7}$$

This interlinks the actor and critic networks since the estimated Q-values are backpropagated through the critic to the actor, producing gradients indicating how the action-parameters should be updated to increase the Q-values. A comparison between the networks used by P-DQN and PA-DDPG is shown in Figure 3.1.

The complete P-DQN algorithm with experience replay is shown in Algorithm 5. Although not originally employed by Xiong *et al.* [2018], we added target networks with soft updates (Polyak averaging) [Lillicrap *et al.* 2015] to improve stability, as in PA-DDPG and DQN [Mnih *et al.* 2015]. This changes the update target calculation in Equation (3.5) to:

$$y = r + \gamma \max_{k'\in[K]} Q(s',k',x_{k'}(s';\theta_x^-);\theta_Q^-), \tag{3.8}$$

where θ_Q^-, θ_x^- are the parameters of the target networks.

Algorithm 5 Parameterised Deep Q-Network (P-DQN) with Experience Replay and Target Networks

```
**Input:** Learning rates $\{\alpha_Q, \alpha_x\}$, target network averaging factors $\{\tau_Q, \tau_x\}$, exploration parameter $\epsilon$, minibatch size $B$, probability distribution $\xi$, replay memory capacity $N$, discount factor $\gamma$, maximum episodes $E_{max}$.
Initialise replay memory $D$ to capacity $N$
Initialise network weights $\theta_Q$, $\theta_x$
Initialise target network weights $\theta_Q^- = \theta_Q$, $\theta_x^- = \theta_x$
Initialise counters $t = 0, e = 0$
repeat
  $s_t$ = initial state from environment
  repeat
    Compute all action parameters $x_k(s_t;\theta_x)$
    Select action $a_t = \begin{cases}(k_t, x_{k_t}) : k_t = \text{argmax}_{k\in[K]} Q(s_t,k,x_{k_t};\theta_Q) & \text{with probability } 1-\epsilon \\ \text{random sample from distribution } \xi & \text{with probability } \epsilon\end{cases}$
    Execute action $a_t$, observe reward $r_t$ and next state $s_{t+1}$
    Store transition $\{s_t, a_t, r_t, s_{t+1}\}$ in $D$
    Sample $B$ transitions $\{s_b, a_b, r_b, s_{b+1}\}_{b\in[B]}$ randomly from $D$
    Set $y_b = \begin{cases} r_b & \text{if } s_{b+1} \text{ is terminal} \\ r_b + \max_{k\in[K]} \gamma Q(s_{b+1},k,x_k(s_{b+1};\theta_x^-);\theta_Q^-) & \text{otherwise}\end{cases}$
    With minibatch $\{s_b, a_b, y_b\}_{b\in[B]}$:
      Compute $\nabla_{\theta_Q} L_Q$ and update $\theta_Q$          # using Adam or any other optimiser
      Compute $\nabla_{\theta_x} L_x$ and update $\theta_x$
    Update target networks $\theta_Q^- = \tau_Q\theta_Q + (1-\tau_Q)\theta_Q^-$, $\theta_x^- = \tau_x\theta_x + (1-\tau_x)\theta_x^-$
    $t = t + 1$
  until $s_t$ is terminal
  $e = e + 1$
until $e \geq E_{max}$
```

3.1.1 Multi-Step Returns

Xiong *et al.* [2018] also introduce a version of P-DQN using n-*step* returns (Section 2.1.1). We discuss two approaches to incorporating n-step return targets below, both of which have been used in the parameterised action-space context.

Parallel Workers

The approach used by *asynchronous P-DQN* [Xiong *et al.* 2018] is to gather samples from multiple instances of the environment using parallel agents to decorrelate samples from the current policy. In asynchronous algorithms with n-step returns, each worker independently collects samples and performs updates to a global shared network, with which workers synchronise at the start of each local episode. This technique has been used to great effect by algorithms such as *asynchronous advantage actor-critic* (A3C) [Mnih *et al.* 2016], which surpassed state-of-the-art algorithms of the time on Atari games and significantly reduced the training time required. However, we note that implementing parallel sample collection requires that the environment can be duplicated, which is impossible when learning online in most real-world scenarios, and complex environments may require powerful hardware to simulate in parallel. Moreover, the performance of such parallel algorithms is strongly correlated with the number workers used [Mnih *et al.* 2016]. Thus with limited hardware or real-world environments, implementing n-step returns by means of parallel workers is infeasible.

Mixed Targets

An alternative to parallel workers is using mixed n-step return targets. As discussed in Section 2.2.3, by using an update target constructed out of both one-step and n-step return targets,

$$y_t = \beta \left[\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i \right] + (1-\beta) \left[r_t + \gamma \max_{a'} Q(s_t, a') \right], \tag{3.9}$$

one can incorporate data-efficient n-step returns using a single agent while retaining a replay memory buffer. This approach was proposed by Hausknecht and Stone [2016b] and applied to PA-DDPG on HFO. However, the value of the mixing ratio β has a massive impact on performance and appears to depend on the domain as well as the algorithm used; see Appendix E for our analysis on the effect of different β values on HFO.

Based on the PA-DDPG algorithm by Hausknecht and Stone [2016b], we present our own version of P-DQN with mixed n-step return targets in Algorithm 6. This includes a sample-to-update ratio parameter u, which makes the algorithm perform one update every $1/u$ samples instead of for every transition. This is done to reduce training time and to be consistent with PA-DDPG.

Algorithm 6 P-DQN with Mixed n-Step Return Targets

```
**Input:** Learning rates $\{\alpha_Q, \alpha_x\}$, target network averaging factors $\{\tau_Q, \tau_x\}$, exploration parameter $\epsilon$, minibatch size $B$, probability distribution $\xi$, replay memory capacity $N$, maximum length of multistep returns $n$, discount factor $\gamma$, maximum episodes $E_{max}$, sample-to-update ratio $u$.
Initialise replay memory $D$ to capacity $N$
Initialise network weights $\theta_Q$, $\theta_x$
Initialise target network weights $\theta_Q^- = \theta_Q$, $\theta_x^- = \theta_x$
Initialise counters $t = 0, e = 0$
repeat
  $t_{\text{start}} = t$
  $s_t$ = initial state from environment
  repeat
    Compute all action parameters $x_k(s_t; \theta_x)$
    Select action $a_t = \begin{cases} (k_t, x_{k_t}) : k_t = \text{argmax}_{k\in[K]}\, Q(s_t, k, x_{k_t}; \theta_Q) & \text{with probability } 1-\epsilon \\ \text{random sample from distribution } \xi & \text{with probability } \epsilon \end{cases}$
    Execute action $a_t$, observe reward $r_t$ and next state $s_{t+1}$
    $t = t + 1$
  until $s_t$ is terminal or $t - t_{\text{start}} \geq n$
  $y_{\text{acc}} = 0$
  for $i = t - 1 \dots t_{\text{start}}$ do
    $y_{\text{acc}} = r_i + \gamma y_{\text{acc}}$
    $\hat{y}_i = y_{\text{acc}}$
    Store $\{s_i, a_i, r_i, \hat{y}_i, s_{i+1}\}$ in $D$
  end for
  for $j = 1 \dots u(t - t_{\text{start}})$ do
    Sample $B$ transitions $\{s_b, a_b, r_b, \hat{y}_b, s_{b+1}\}_{b\in[B]}$ randomly from $D$
    Set $y_b = \begin{cases} \beta\hat{y}_b + (1-\beta)r_b & \text{if } s_{b+1} \text{ is terminal} \\ \beta\hat{y}_b + (1-\beta)\Big(r_b + \max_{k\in[K]} \gamma Q(s_{b+1}, k, x_k(s_{b+1}; \theta_x^-); \theta_Q^-)\Big) & \text{otherwise} \end{cases}$
    With minibatch $\{s_b, a_b, y_b\}_{b\in[B]}$:
      Compute $\nabla_{\theta_Q} L_Q$ and update $\theta_Q$          # using Adam or any other optimiser
      Compute $\nabla_{\theta_x} L_x$ and update $\theta_x$
    Update target networks $\theta_Q^- = \tau_Q\theta_Q + (1-\tau_Q)\theta_Q^-$, $\theta_x^- = \tau_x\theta_x + (1-\tau_x)\theta_x^-$
  end for
  $e = e + 1$
until $e \geq E_{max}$
```

3.2 Comparison Study

We compare our modified P-DQN against Q-PAMDP and PA-DDPG on the Platform, Robot Soccer Goal, and Half Field Offense domains. Previously, Xiong *et al.* [2018] compared only against the results reported by Hausknecht and Stone [2016a] for PA-DDPG on HFO without reproducing them. This comparison is not quite fair, however, since Xiong *et al.* [2018] use asynchronous P-DQN with 24 parallel workers and dueling networks [Wang *et al.* 2016] while PA-DDPG uses a single worker with mixing n-step returns. The dueling networks technique is an enhancement for DQN that decomposes Q-values as a sum of the state value and advantage functions; it is not part of the core P-DQN algorithm. Although Wei *et al.* [2018] present preliminary results for PA-TRPO on Platform and HFO, we exclude it from our comparison as it failed to learn to score goals on HFO.

3.2.1 Experimental Methodology

For each of the chosen benchmark domains we perform a hyperparameter search for P-DQN and PA-DDPG to determine the best performing network size, learning rate, target network Polyak averaging factor, and minibatch update size. The values searched and final hyperparameters are given in Appendix A; these are used in all experiments unless otherwise specified. Adam [Kingma and Ba 2014] is used to optimise the neural network parameters for P-DQN and PA-DDPG. Layer weights are initialised following the *Kaiming normal* strategy of He *et al.* [2015] with rectified linear unit (ReLU) [Glorot *et al.* 2011] activation functions applied after every layer except the last. We employ the inverting gradients approach to account for bounded action-parameters, the ranges of which are scaled to $[-1, 1]$ in all domains as we found this improves the performance in general; see Appendix C for more information. Experiments with the final hyperparameters decided by grid search are performed over 30 uniquely seeded random runs per algorithm in each domain. At the end of training, agents are evaluated without random exploration over an additional 1000 episodes. We use somewhat more episodes than necessary to train all agents to ensure sufficient time for convergence, since we are as interested in the training dynamics of each algorithm as the final performance during evaluation. Performance is measured in terms of area under the training curve of episodic scores, average mean evaluation score, and median agent evaluation score. We include the median because several outliers occur where agents fail to complete the given task, pulling down the mean score significantly. We use the mixing n-step returns variants of PA-DDPG and P-DQN (Algorithm 6) on HFO, and regular one-step lookahead targets (Algorithm 5) on Platform and Robot Soccer Goal. The following hardware is used for our experiments: Intel Core i7-7700 CPU, 16GB DRAM, NVidia GTX 1060 GPU. Our experiments are implemented in Python using the PyTorch framework [Paszke *et al.* 2017] and OpenAI Gym interface [Brockman *et al.* 2016]. Our source code is publicly available online at `https://github.com/cycraig/MP-DQN` to encourage reproducibility.

Platform

Agents are trained on the Platform domain over 80 000 episodes. We use the same hyperparameters for Q-PAMDP as Masson *et al.* [2016] except we reduce the learning rate for eNAC from 1 to 0.1 to account for the scaled action-parameters. The additive Gaussian noise variance is also reduced from 0.1 to 0.0001 for the same reason.

For P-DQN, a shallow network consisting of a single hidden layer of 128 neurons was found to perform best with a minibatch size of 128, and learning rates of 10^{-3} and 10^{-4} for the critic and actor networks respectively. The Polyak averaging factors for the target networks were chosen to be 0.1 and 0.001, although changing them had a minor impact on performance compared to the learning rate during the search. A larger network of two hidden layers with 256 and 128 neurons respectively was found to be better for PA-DDPG, with coincidentally the same learning rates of 10^{-3} and 10^{-4} as used for P-DQN, a minibatch size of 32, and Polyak averaging factors of 0.01 and 0.01. The same replay memory size of $10\,000$ samples was used for both algorithms. We use an ϵ-greedy discrete action policy with additive Ornstein-Uhlenbeck noise for action-parameter exploration, as we found this gives slightly better performance than using Gaussian noise. A passthrough layer is used to initialise the action-parameter policy of the actor networks for P-DQN and PA-DDPG to the same as Q-PAMDP, see Appendix D.

Robot Soccer Goal

Training consists of $100\,000$ episodes on Robot Soccer Goal. We again use the same hyperparameters for Q-PAMDP as Masson *et al.* [2016] except we reduce the eNAC learning rate from 0.1 to 0.06 and the noise variance from 0.01 to 0.0001.

P-DQN seems to perform better with shallower networks as a single hidden layer of 128 neurons was also the best choice for this domain, with learning rates of 10^{-3} and 10^{-5}, Polyak averaging factors of 0.1 and 0.001, and a minibatch size of 128. Similar to Platform, PA-DDPG uses a larger network of two hidden layers of 128 and 64 neurons, learning rates of 10^{-4} and 10^{-5}, Polyak averaging factors of 0.01 and 0.01, and a minibatch size of 64. Both algorithms use a replay memory size of $20\,000$ and the same action-parameter policy initialisation as Q-PAMDP with additive Ornstein-Uhlenbeck noise.

Half Field Offense

We use $30\,000$ episodes for training on HFO. This is more than the $20\,000$ episodes (or roughly 3 million transitions) used by Hausknecht and Stone [2016a] and Xiong *et al.* [2018] because, as mentioned before, we want to give enough opportunity for the algorithms to converge in order to fairly evaluate the final policy performance. We use the same network structure as previous works with hidden layers of 256-128-64 neurons for P-DQN and 1024-512-256-128 neurons for PA-DDPG. The ReLU activation function with negative slope 10^{-2} is used on HFO because of these deeper networks. We also use the same hyperparameters as Hausknecht and Stone [2016b] apart from the β value, which we set to 0.25, and the network learning rates which are adjusted due to using scaled action-parameters: $10^{-3}, 10^{-5}$ for P-DQN and $10^{-3}, 10^{-3}$ for PA-DDPG. In absence of an initial action-parameter policy, we use the ϵ-greedy with uniform random action-parameter exploration strategy. In general we keep as many factors consistent between the two algorithms as possible for a fair comparison.

Q-PAMDP on the other hand, having never been applied to HFO before to the best of our knowledge, requires some feature engineering and manual tuning. Since the full state observation space consists of 58 features, we selectively choose 10 of the most relevant ones to avoid intractable Fourier basis calculations over the entire space. These features include: player orientation, stamina, proximity to ball, ball angle, ball-kickable, goal centre position, and goal centre proximity. Orientations are encoded as two values: the $\sin$ and $\cos$ of the angle. Even with this significantly reduced feature space, we found at most a 2nd-order Fourier basis could be used. The same alpha scaling strategy as Masson *et al.* [2016] is used for Sarsa(λ) with an eNAC learning rate of 0.2. The Q-PAMDP agent initially learns with Sarsa(λ) for a period of 1000 episodes before alternating between $k = 50$ eNAC updates of 25 rollouts each, and 1000 episodes of discrete action re-exploration with Sarsa(λ).

3.2.2 Results

The results of training P-DQN, PA-DDPG, and Q-PAMDP on the benchmark domains are shown in Table 3.1 and Figure 3.2.

Platform

	Mean	Median	Area Under Curve
Q-PAMDP	0.789 ± 0.188	0.795	50 397
P-DQN	$\mathbf{0.964 \pm 0.068}$	**0.997**	**64 705**
PA-DDPG	0.284 ± 0.061	0.308	23 247

Robot Soccer Goal

	Mean	Median	Area Under Curve
Q-PAMDP	0.452 ± 0.093	0.456	39 340
P-DQN	$\mathbf{0.668 \pm 0.062}$	**0.656**	**61 445**
PA-DDPG	0.006 ± 0.020	0.000	1791

Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
Q-PAMDP	0 ± 0	0	n/a	15 924
P-DQN (ours)	$\mathbf{0.883 \pm 0.085}$	0.917	111 ± 11	182 573
PA-DDPG (ours)	0.875 ± 0.182	**0.945**	$\mathbf{95 \pm 7}$	**193 687**
Asynchronous P-DQN[1]	0.989 ± 0.006	0.991	81 ± 3	-
PA-DDPG[2]	0.923 ± 0.073	0.960	112 ± 5	-

Table 3.1: Results of the comparison study between Q-PAMDP, P-DQN, and DDPG on Platform (top), Robot Soccer Goal (middle) and HFO (bottom). We include previously published results from Hausknecht and Stone [2016a] and Xiong *et al.* [2018] on HFO, although they are not directly comparable with ours as we use a longer training period and have a much larger sample size of agents—30 versus 7 and 9 respectively—and asynchronous P-DQN, as mentioned before, uses 24 parallel workers to implement n-step returns rather than the mixing strategy of PA-DDPG. Our results are similar to those of Hausknecht and Stone [2016a] for PA-DDPG, except we observe more outliers where agents fail to score reliably, shown in Figure 3.2b.

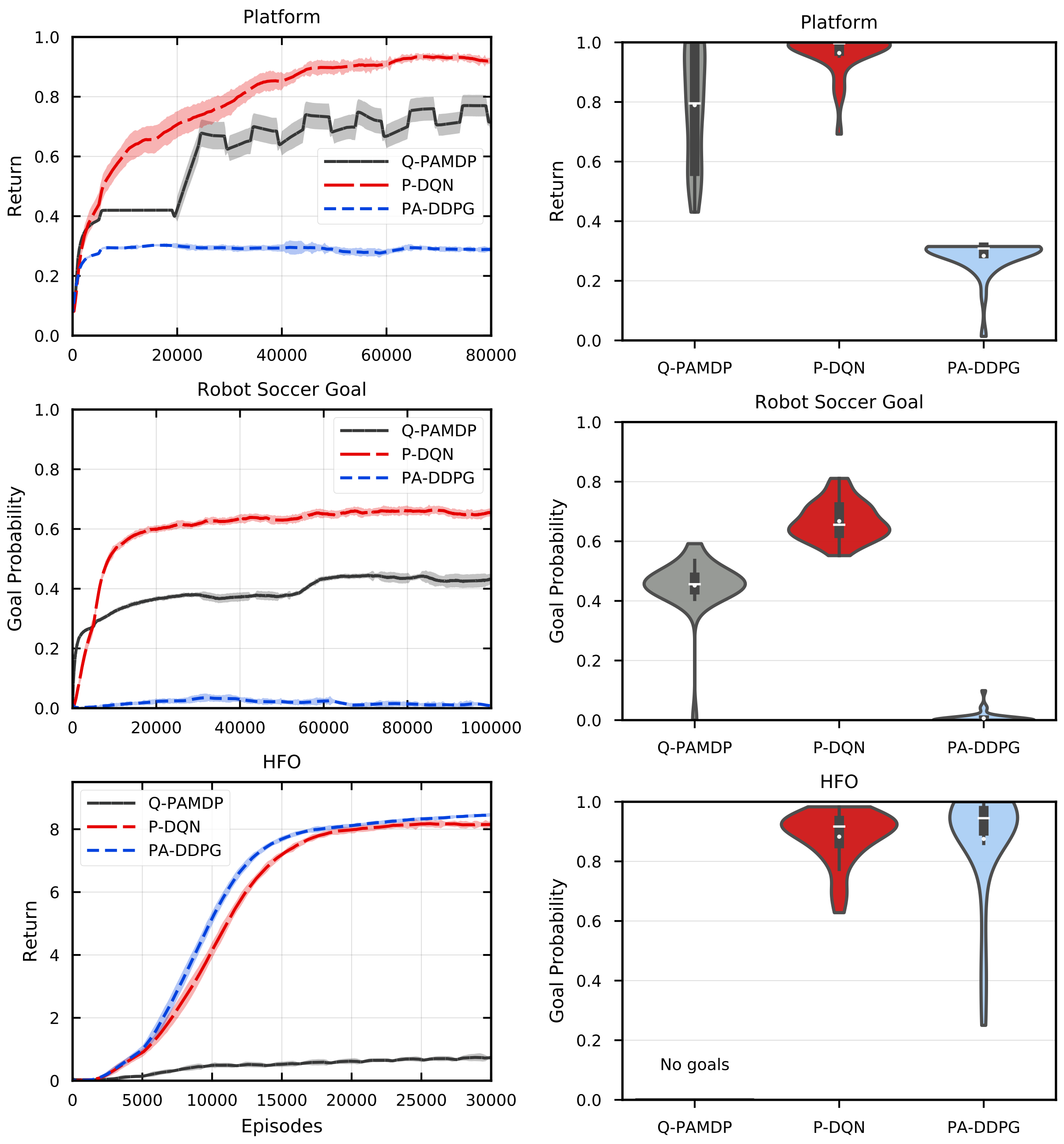
[1]Reported average over 9 runs with 24 parallel workers for n-step returns [Xiong *et al.* 2018].

[2]Reported average over 7 runs with mixing n-step returns [Hausknecht and Stone 2016a].

While Q-PAMDP shows learns slightly faster than P-DQN in the first few thousand episodes, particularly on Robot Soccer Goal, P-DQN quickly outperforms Q-PAMDP and reaches a significantly higher average return by the end of training. On Platform, for example, the mean P-DQN evaluation score is 0.964 versus 0.789 for Q-PAMDP, indicating almost all P-DQN agents manage to reach the final platform consistently, bar one or two outliers. PA-DDPG, on the other hand, fails to learn to traverse past the first platform with a mean score of 0.284; this is similar to the results reported by Wei *et al.* [2018]. Upon closer inspection, PA-DDPG consistently learns to leap over the first enemy into the gap because this gives a greater immediate reward than hopping over the enemy.[3] Similarly on Robot Soccer Goal, PA-DDPG completely fails to optimise discrete action selection and changes the action-parameter policy such that it is almost impossible to score goals. This behaviour highlights the problem with updating the action and action-parameter policies at the same time leading to premature convergence to suboptimal policies. This problem is also observed when using direct policy search with eNAC in experiments by Masson *et al.* [2016].

In Half Field Offense, the most complex of the benchmark domains, PA-DDPG agents learn faster on average and score goals more consistently than P-DQN agents. This confirms that the use of asynchronous parallel workers was a major factor influencing the performance of P-DQN on HFO in the experiments by Xiong *et al.* [2018], and under similar conditions with a single worker using mixing n-step returns it is somewhat inferior to PA-DDPG. However, we note that PA-DDPG produces outliers with poorer goal-scoring capabilities than the worst performing agents of P-DQN. This leads P-DQN to have a higher mean evaluation score but a lower median evaluation score. Q-PAMDP learns to approach the ball but otherwise fails to score any goals, which is not unexpected given that agents do not have access to the full feature space and the action-parameter policy is restricted to a linear combination of features rather than a non-linear neural network function approximator. Further feature engineering would likely improve the performance of Q-PAMDP but the fact that P-DQN and PA-DDPG learn without complex hand-crafted features is a major practical advantage.

[3]There is no negative reward for the agent dying in Platform, only a positive reward based on the distance traversed.



(a) Learning curves (b) Evaluation scores

Figure 3.2: Results of Q-PAMDP, P-DQN, and PA-DDPG on Platform (top), Robot Soccer Goal (middle) and HFO (bottom) over 30 seeded-random runs per algorithm per domain. The learning curves during training (a) show the running average scores—episodic return for Platform and HFO, and goal scoring probability for Robot Soccer Goal—smoothed over 5000 episodes including random exploration of the agents; shaded areas represent the standard deviation of the running averages over the different agents. The violin plots in (b) show the distribution of mean scores over 1000 evaluation episodes post-training. Note that the average goal probability is reported for evaluation on HFO instead of the episodic return, as done by Hausknecht and Stone [2016a]; Xiong *et al.* [2018]. The inner box plot of each violin shows the interquartile range (IQR) with whiskers representing data within 1.5 times the IQR above and below the first and third quartiles; anything outside of that range is considered an outlier, as is convention [Tukey 1977]. Mean average scores are indicated by white dots, and white lines show the median agent's average score.

3.3 Summary

We detailed the core P-DQN algorithm and presented our own version using mixing n-step returns, an alternative to the multi-worker asynchronous P-DQN. Our comparison study shows that P-DQN outperforms Q-PAMDP in general, the previous state-of-the-art algorithm on Platform and Robot Soccer Goal, while PA-DDPG prematurely converges to poor action policies on those domains. Furthermore, our results show that P-DQN with mixing n-step returns performs slightly worse than PA-DDPG on HFO, indicating that the results of the comparison by Xiong *et al.* [2018] were largely due to the use of asynchronous parallel workers and that there is room for improvement for P-DQN on HFO.

Chapter 4

Addressing Action Sampling Imbalance

One of the main differences between P-DQN and PA-DDPG is the learning of a Q-value per action instead of a single critic value over a combined action and action-parameter vector. This difference introduces potential problems stemming from the natural imbalance in the distribution between actions taken by agents during learning. Figure 4.1 shows that such action sampling imbalance occurs on all three benchmark domains.

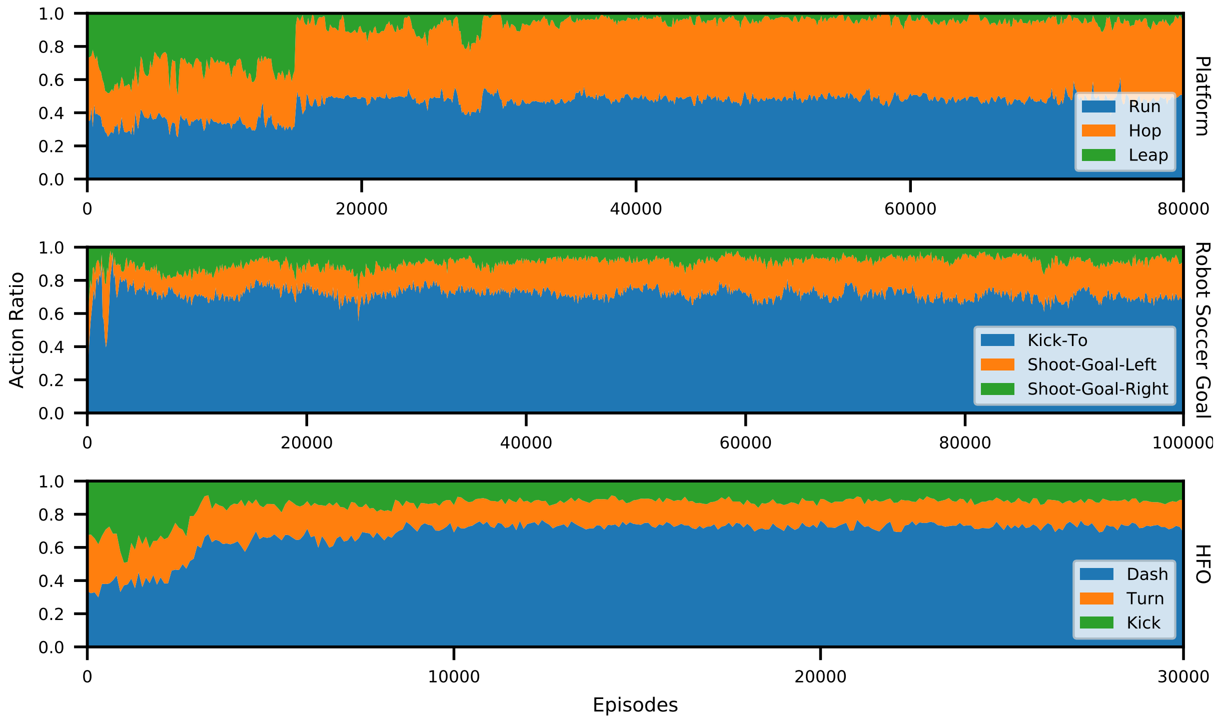


Figure 4.1: Plot of action selection ratios for P-DQN over the course of training on Platform (top), Robot Soccer Goal (middle) and HFO (bottom). The action counts are averaged over 100 episode intervals recorded from a single agent per domain, including any random exploration. There is clearly a significant imbalance between actions on all three domains: for example on HFO, 70% of the agent's actions are dashes because most of an episode consists of moving towards the ball, while on Platform the agent eventually chooses the leap action less than 3% of the time.[2]

[0]This is a consequence of the hop action having a higher apogee than the leap action, thus allowing the agent to jump somewhat further. This is likely unintentional but we retain this behaviour in our version of Platform for consistency.

The main issue with this imbalance for P-DQN is that each Q-value is only updated using transitions corresponding to their associated action, leading to some Q-values possibly being less accurate than others. This is primarily a problem when novel states are first explored by the agent, since if not all Q-values have sufficiently generalised to similar states they could provide incorrect gradients to their associated action-parameters in the summation loss function. Only the Q-value of the action taken in the novel state, say Q_k for action k, contains valuable information for x_k, but all $x_j, j \neq k$ are still updated even though the agent has never executed those actions in that state. As exploration decreases during training, the actions chosen by the agent favour exploitation which further entrenches the imbalance of actions chosen. While all Q-values would theoretically provide accurate critic estimates for the action-parameters if all actions were explored an infinite number of times in all states, this is not the case in practice due to limited time and resources, and the imbalance would still be an issue early on in training during exploration. To address this, we propose two alternatives to the summation action-parameter loss used by P-DQN: the indexed loss, and the weighted loss.

4.1 Indexed Action-Parameter Loss

Recall the original summation action-parameter loss function for P-DQN:

$$L_x(\theta_x) = \mathop{\mathbb{E}}_{s}\left[-\sum_{k=1}^{K} Q\big(s, k, x_k(s;\theta_x);\theta_Q\big)\right]. \tag{4.1}$$

When using minibatch updates, Equation (4.1) can be written as:

$$L_x(\theta_x) = \frac{1}{|B|}\sum_{s_b \in B}\left[-\sum_{k=1}^{K} Q\big(s, k, x_k(s_b;\theta_x);\theta_Q\big)\right], \tag{4.2}$$

where B is a minibatch of transitions sampled from replay memory D. Because action sampling imbalance causes potential for inaccurate gradients from Q-values for actions other than the one taken in an update sample, we propose changing the loss function to use only the Q-value corresponding to the action chosen per sample in the minibatch. This gives us the *indexed loss*:

$$L_x(\theta_x) = \frac{1}{|B|}\sum_{s_b, k_b \in B}\Big[-Q\big(s, k_b, x_{k_b}(s_b;\theta_x);\theta_Q\big)\Big]. \tag{4.3}$$

The loss now depends on the action k_b from each sample in the minibatch as well as the state, removing the summation over all actions. This is similar to the Q-value loss, Equation (3.5), in that only the value corresponding to a single action is used per sample in the minibatch. Under the assumption that each action is explored infinitely many times in every state, using the indexed loss should converge to the same policy as the summation loss. Note that there are no guarantees of convergence to globally optimal solutions when non-linear function approximation is used in DQN or DDPG, and the same is true for P-DQN. One potential disadvantage of the indexed loss is that if each Q-value is relatively accurate in the states sampled from a minibatch, then the summation loss could update the policy faster since it affects all action-parameters. However, this is unlikely in most situations due to the aforementioned action sampling imbalance.

4.1.1 Gradient-Zeroing for Action-Parameter Update Independence

We now take a small detour to highlight a different problem with P-DQN and how it affects our proposed indexed loss function. Recall that the P-DQN architecture inputs the joint action-parameter vector over all actions to the Q-network, as illustrated in Figure 3.1a. While this may seem like an inconsequential implementation detail at first, it in fact changes the formulation of the Bellman Equation for parameterised actions (3.4) such that each Q-value is a function of the joint action-parameter vector $\mathbf{x} = (x_1, \dots, x_K)$, rather than only the action-parameter x_k corresponding to the associated action:

$$Q(s, k, \mathbf{x}) = \mathop{\mathbb{E}}_{r,s'} \left[r + \gamma \max_{k' \in [K]} Q(s', k', \mathbf{x}^Q(s')) \middle| s, k, \mathbf{x} \right]. \tag{4.4}$$

This in turn affects both the updates to the Q-values and the action-parameters. For now, we consider only the effect on the action-parameter loss, specifically that each Q-value produces gradients for all action-parameters. Consider for demonstration purposes the summation loss taken over a single sample with state s:

$$L_x(\theta_x) = -\sum_{k=1}^{K} Q\big(s, k, \mathbf{x}(s; \theta_x); \theta_Q\big). \tag{4.5}$$

The policy gradient is then given by:

$$\nabla_{\theta_x} \mathbf{x}(s; \theta_x) = -\sum_{k=1}^{K} \nabla_{\mathbf{x}} Q\big(s, k, \mathbf{x}(s; \theta_x); \theta_Q\big) \nabla_{\theta_x} \mathbf{x}(s; \theta_x). \tag{4.6}$$

Expanding the gradients with respect to the action-parameters,

$$\nabla_{\mathbf{x}} Q = \left(\tfrac{\partial Q_1}{\partial x_1} + \tfrac{\partial Q_2}{\partial x_1} + \dots + \tfrac{\partial Q_K}{\partial x_1}, \quad \cdots \quad, \tfrac{\partial Q_1}{\partial x_K} + \tfrac{\partial Q_2}{\partial x_K} + \dots + \tfrac{\partial Q_K}{\partial x_K} \right), \tag{4.7}$$

where $Q_k := Q(s, k, \mathbf{x}(s; \theta_x); \theta_Q)$. Theoretically, if each Q-value were a function of just x_k as the original P-DQN formulation intended, then $\partial Q_k / \partial x_j = 0 \; \forall j, k \in [K], j \neq k$ and $\nabla_{\mathbf{x}} Q$ simplifies to

$$\nabla_{\mathbf{x}} Q = \left(\tfrac{\partial Q_1}{\partial x_1}, \quad \tfrac{\partial Q_2}{\partial x_2}, \quad \cdots \quad, \tfrac{\partial Q_K}{\partial x_K} \right). \tag{4.8}$$

However this is not the case in P-DQN, so the gradients with respect to other action-parameters are not zero in general. This is a problem because each Q-value is only updated when its corresponding action is sampled, as per Equation (3.5), and thus has no information on what effect other action-parameters $x_j, j \neq k$ have on transitions or how they should be updated to maximise the expected return. They therefore produce what we term *false gradients*. This effect may be mitigated by the summation loss function, since the gradients from each Q-value are summed and then averaged over a minibatch, hopefully causing the signal from the true gradient to overpower the conflating false gradients. When using the proposed indexed loss, however, it is possible that the false gradients become more prominent since the loss is a function of a single Q-value per sample, but still averaged over a minibatch.

One approach to overcome this is by explicitly setting the gradients of Q-values for action-parameters other than their own to zero during updates: $\frac{\partial Q_k}{\partial x_j} \leftarrow 0 \; \forall k, j \in [K], j \neq k$. This is not trivial when using the summation loss function because the gradients are summed. Using the indexed loss, on the other hand, one can simply use the same index as the sampled action to exclude other gradients. We refer to this technique as *gradient-zeroing*. Note that this only solves the issue introduced by having joint inputs for action-parameter updates, and only for the indexed loss function thus far. The Q-value issues we alluded to before are discussed and addressed in Chapter 5, with solutions applicable to any loss function. For the purposes of this chapter, however, gradient-zeroing suffices.

4.2 Weighted Action-Parameter Loss

The second action-parameter loss function we propose addresses a different aspect of the action sampling imbalance. The transitions stored in replay memory will obviously mimic the distribution of actions taken by the agent in the environment. The imbalance between actions thus also extends to minibatches that are sampled uniformly randomly from replay memory. This means that not all actions are represented equally during an update.

Consider running P-DQN on Half Field Offense. There are three parameterised actions available: dash, turn, and kick. As seen in Figure 4.1, the dash action is chosen around 70% of the time, turn 18%, and kick 12%. Thus with a minibatch size of 32, on average one would expect roughly 22 of the samples to be transitions where the dash action was taken, 6 for the turn action, and 4 for the kick action. During updates, the dash action-parameter gradient is therefore likely to be a better estimate since it is averaged over more samples, whereas the gradients for the turn and kick action-parameters might be misled by a few states that suggest an incorrect or suboptimal direction of increase.

Explicitly balancing the number of transitions sampled from replay memory to be equal for each action is unwise as it can prioritise outdated transitions if a certain action is seldom taken, such as the leap action on the Platform domain. Prioritised experience replay [Schaul *et al.* 2015] could be employed to bias sampling towards transitions where the critic has high error, but that would not address the imbalance of samples possibly leading to poor gradient estimates. So, instead of changing the replay memory sampling strategy, we propose a simple change to the loss function to incorporate knowledge of the action distribution imbalance. Using the summation loss from Equation (4.1), we weight each Q-value according to the ratio of transitions from the current minibatch where that action is executed. Our proposed *proportionally weighted loss*, or simply *weighted loss*, is defined as follows:

$$L_x(\theta_x) = \frac{1}{|B|} \sum_{s_b \in B} \left[-\sum_{k=1}^{K} c_k Q\big(s, k, x_k(s_b; \theta_x); \theta_Q\big) \right], \tag{4.9}$$

where $c_k = \sum_{k_b \in B} [k_b = k]/|B|$ is the proportion of transitions where action k is taken in the current minibatch B. Note that $\sum_{k=1}^{K} c_k = 1$, meaning the magnitude of the loss is only ever decreased. This has the effect of asymmetrically downscaling the loss generated by each Q-value, creating a more conservative estimate of the gradient for each action-parameter based on how confident we are in the loss. This is not the same as reducing the learning rate since the c_k weighting terms change per minibatch and in general differ per action. We can also extend the indexed loss function proposed in Section 4.1 with weighting terms, forming the *weighted-indexed loss*:

$$L_x(\theta_x) = \frac{1}{|B|} \sum_{s_b, k_b \in B} \Big[- c_k Q\big(s, k_b, x_{k_b}(s_b; \theta_x); \theta_Q\big) \Big], \tag{4.10}$$

where c_k is the same as in Equation (4.9).

The potential disadvantages of how c_k is defined are that it makes action-parameter gradients strictly more conservative, even when the Q-values become accurate enough to provide useful estimates over fewer transitions, and the weightings would decrease if the number of actions increased, since the ratio of samples corresponding to each action would be smaller. Ideally, one would calculate the confidence interval of each Q-value, for example by bootstrapping [White and White 2010], and scale the action-parameter gradients accordingly. However more complex techniques would introduce additional computational complexity, whereas our weighted loss function has negligible overhead and will be shown to have a positive impact on performance despite its simplicity.

4.3 Experiments

We devise three sets of experiments to empirically evaluate the performance of the proposed action-parameter loss functions for P-DQN:

(E1) Indexed loss

(E2) Weighted loss

(E3) Weighted-indexed loss

In all cases, we use the same experimental configuration for P-DQN detailed in Section 3.2.1 on the three benchmark domains.

4.3.1 (E1) Indexed Loss

We first examine the impact of the indexed action-parameter loss, with and without gradient-zeroing, on the performance of P-DQN versus the summation loss as a baseline. The results, shown in Table 4.1 and Figure 4.2, indicate a clear improvement in terms of training and evaluation performance when using the indexed loss over the baseline P-DQN agents on Platform and HFO. Using the gradient-zeroing technique shows an even greater improvement due to the removal of false gradients, confirming not only their existence in practice but that they are detrimental. On Robot Soccer Goal, however, agents with the indexed loss function have reduced performance, with an 11% lower probability of scoring goals on average compared to baseline agents. Even with gradient-zeroing, the area under the curve is lower than that of P-DQN, although the average evaluation performance is higher than the baseline by the end of training.

So why is the indexed loss beneficial on some domains while reducing performance on others? We suspect this can be attributed to the difference in the visitation of states by agents between the domains. On Platform, the agent continues to see novel states as training progresses, traversing past enemies to new platforms until it learns to reach the goal. Similarly on HFO, the agent does not experience states or rewards where the ball moves until it learns to approach and kick it. On Robot Soccer Goal, however, the agent has access to almost all states at the beginning of training because each of its actions are related to kicking the ball in a different manner. The agent's initial position at the beginning of each episode is randomised, so most states are visited during initial exploration. Thus P-DQN possibly learns to generalise its Q-values more quickly than on Platform and HFO due to the relative lack of novel states. This is supported by the fact that the learning curve starts to plateau relatively early in Robot Soccer Goal, around

20 000 episodes. This explains the decrease in performance since, as mentioned in Section 4.1, the indexed loss is less efficient than the summation loss when the Q-values of all actions produce accurate gradients in most states. There is also a larger difference between the indexed loss with gradient-zeroing and without on Robot Soccer Goal, indicating the magnitude of false gradients is greater on that domain than the other two: this is confirmed by our sensitivity analysis in Chapter 5.

The fact that the indexed loss without gradient-zeroing significantly reduces performance of P-DQN on Robot Soccer Goal, and causes worse outliers on Platform and HFO, serves to highlight the problem of false gradients caused by each Q-value depending on all action-parameters. While gradient-zeroing can be employed to eliminate false gradients, this technique is specific to the indexed loss function because it relies on the action taken in the sampled transition.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.964 ± 0.068	**0.997**	64 705
Indexed with Gradient-Zeroing	$\mathbf{0.990 \pm 0.016}$	**0.996**	**73 567**
Indexed	0.945 ± 0.150	**0.995**	66 998

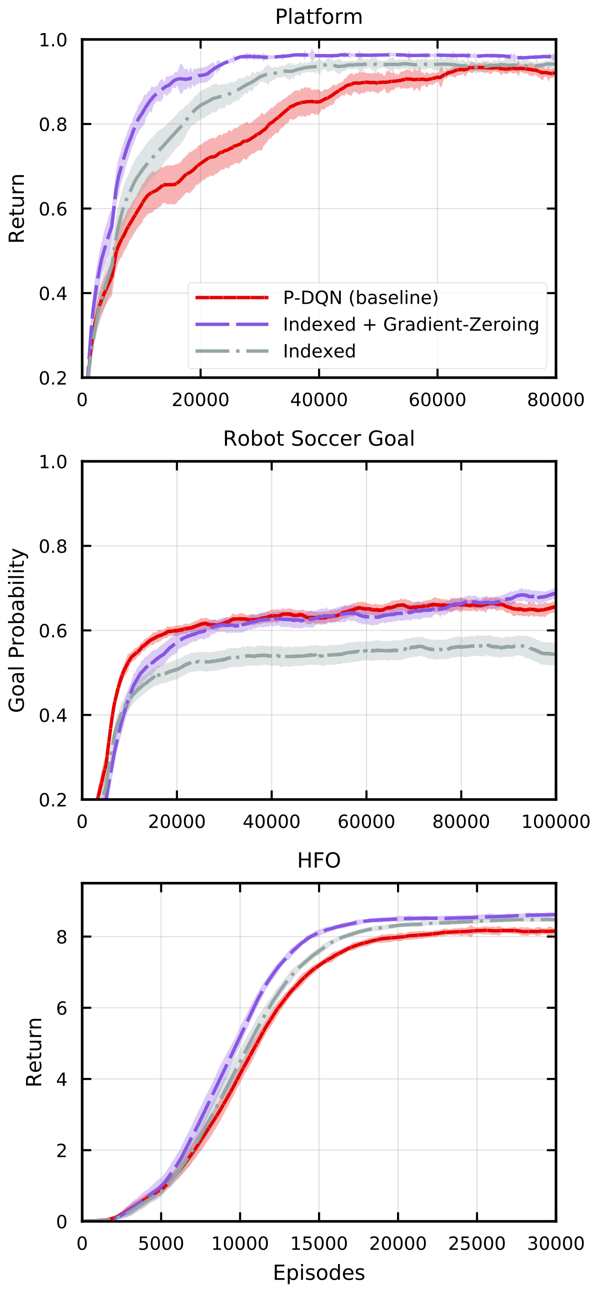
Robot Soccer Goal

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	**61 445**
Indexed with Gradient-Zeroing	$\mathbf{0.700 \pm 0.073}$	**0.720**	60 119
Indexed	0.558 ± 0.147	0.578	52 140

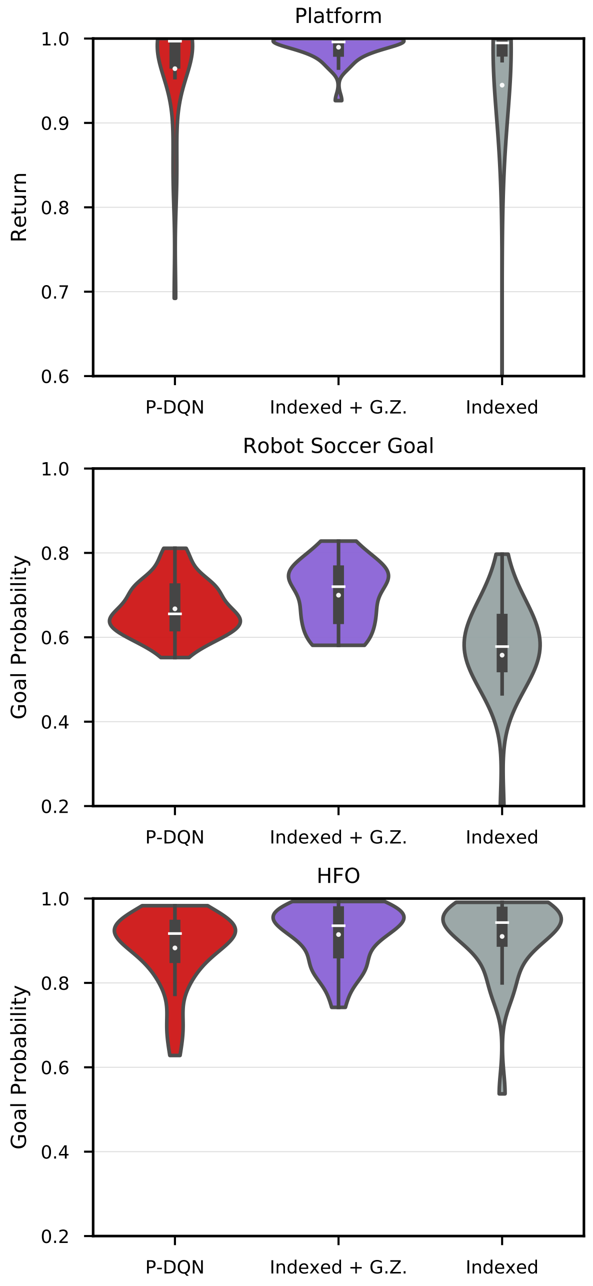
Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	0.883 ± 0.085	0.917	111 ± 11	182 573
Indexed with Gradient-Zeroing	$\mathbf{0.915 \pm 0.066}$	0.936	$\mathbf{106 \pm 10}$	**199 747**
Indexed	0.910 ± 0.092	**0.943**	$\mathbf{106 \pm 11}$	190 742

Table 4.1: Results of using the proposed indexed action-parameter loss function for P-DQN with and without gradient-zeroing. We observe the indexed loss alone achieves a mean score lower than the baseline on Platform and Robot Soccer Goal, but a higher mean on HFO.



(a) Learning curves



(b) Evaluation scores

Figure 4.2: Graphs of the results of training P-DQN agents with the proposed indexed action-parameter loss function with and without gradient-zeroing, compared to baseline P-DQN using the original summation loss.

4.3.2 (E2) Weighted Loss

Our second set of experiments evaluates the proposed weighted action-parameter loss function. We compare three different choices of the c_k weighting terms to determine whether any change in performance is due to selectively conservative gradients or smaller losses in general. The three choices of c_k we evaluate are:

1. **Proportional**: $c_k = \sum_{k_b \in B}[k_b = k]/|B|$. This is the same version we propose in Section 4.2.
2. **Average**: $c_k = 1/K$. This is equivalent to simply lowering the learning rate, since the loss is downscaled by a constant factor.
3. **Random**: $c_k = U([0,1])/\sum_{k=1}^{K} c_k$. In this case the weightings are uniformly randomly sampled from the range $[0,1]$ in each update, then normalised such that $\sum_{k=1}^{K} c_k = 1$ to be consistent with the other two approaches.

We compare these variants of the weighted loss against P-DQN with the summation action-parameter loss as a baseline. We present the results of this comparison in Table 4.2 and Figure 4.3.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.964 ± 0.068	**0.997**	64 705
Proportional	**0.972 ± 0.057**	**0.997**	65 587
Average	**0.971 ± 0.054**	**0.996**	65 260
Random	0.943 ± 0.137	0.993	**66 901**

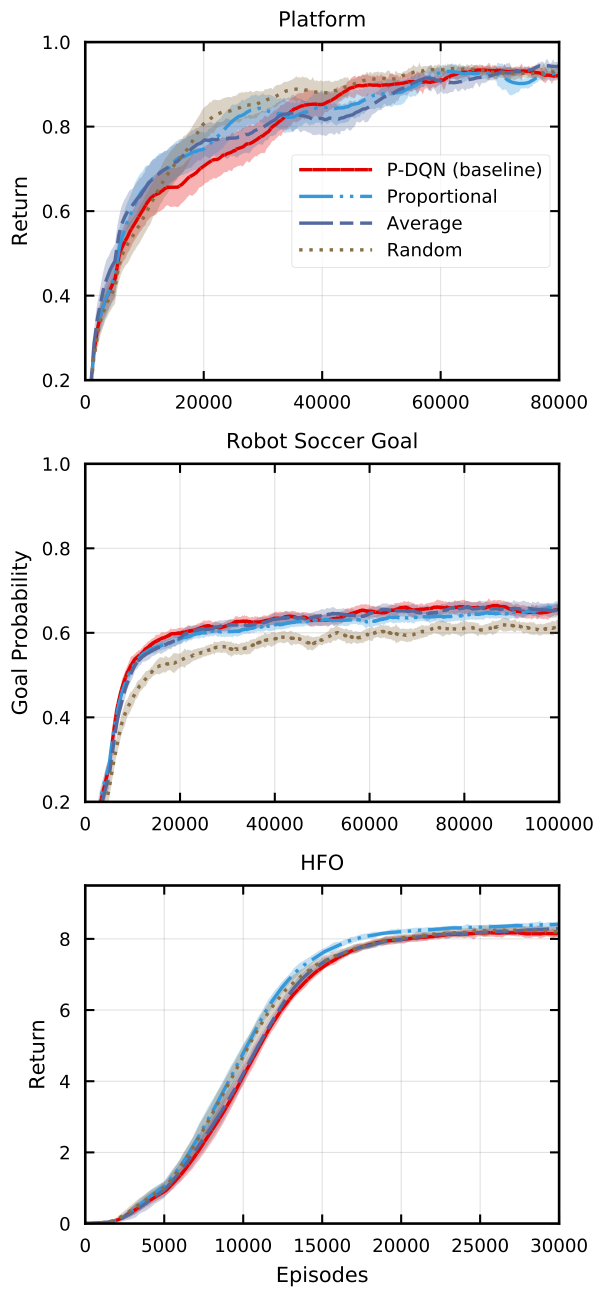
Robot Soccer Goal

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	**61 445**
Proportional	**0.683 ± 0.069**	0.666	60 312
Average	**0.682 ± 0.083**	**0.680**	61 006
Random	0.624 ± 0.063	0.625	56 135

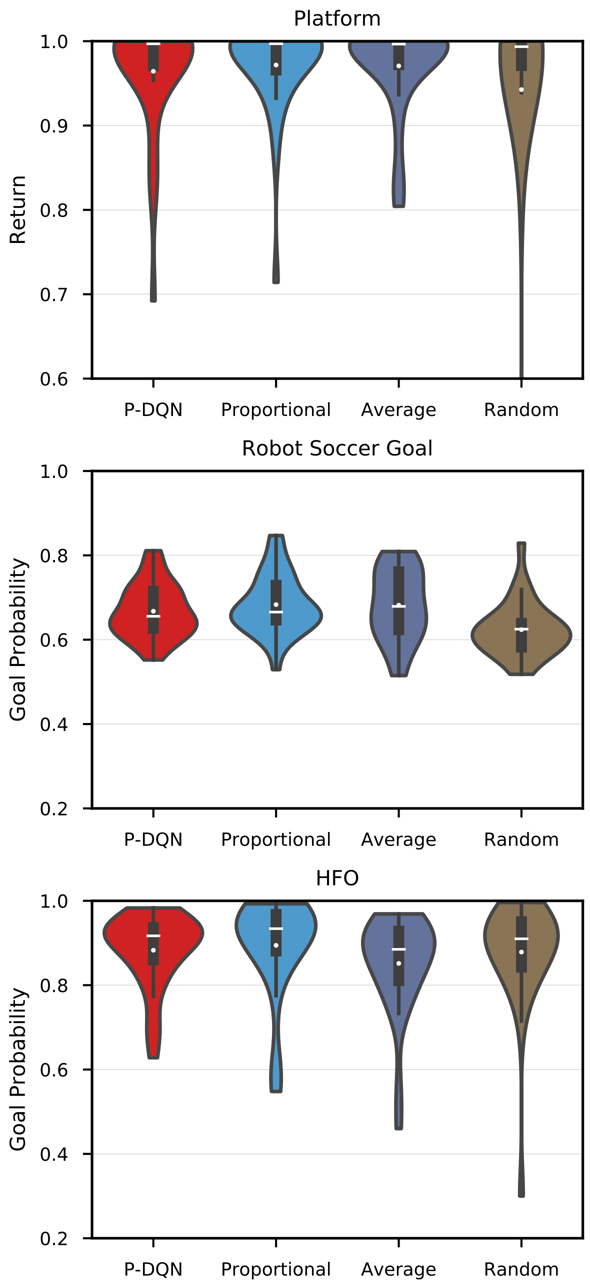
Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	0.883 ± 0.085	0.917	**111 ± 11**	182 573
Proportional	**0.894 ± 0.116**	**0.934**	**110 ± 9**	**191 873**
Average	0.852 ± 0.113	0.885	116 ± 12	184 620
Random	0.879 ± 0.125	0.910	114 ± 15	187 618

Table 4.2: Results of training P-DQN agents with each of the weighted action-parameter loss function variants compared to baseline P-DQN. The highest mean scores are achieved by the loss with proportional c_k weightings.



(a) Learning curves



(b) Evaluation scores

Figure 4.3: Graphs of the results of using each of the weighted action-parameter loss function variants for P-DQN compared to the summation loss as a baseline. There is little difference between the learning curves of the different approaches in (a), although a slight increase in average evaluation scores for the proportional weighted loss function is observed in (b).

We observe no significant changes in training performance when using any of the weighted loss functions: the proportional and average weightings have roughly the same area under the curve as baseline P-DQN, while agents with random weightings learn slightly faster on Platform and slower on Robot Soccer Goal. There is a difference in evaluation scores, however, where we see proportional weightings produce higher mean and median scores than the baseline on all three domains. Average weightings are competitive with proportional weightings on Platform and Robot Soccer Goal but have lower-than-baseline scores on HFO. Random weightings, on the other hand, produce significantly worse evaluation scores than the other variants and baseline in all cases. The similarity between scores of agents using the average and proportional weightings on Platform and Robot Soccer Goal may indicate that a lower action-parameter learning rate would show similar improvements. The hyperparameter search only included learning rates in decreasing powers of 10—10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}—so it is possible that the ideal learning rate is between these. However this is not the case on HFO, where we see proportional weightings outperform the baseline while average weightings reduce performance. This seems to indicate that while having more conservative gradient estimates may be a factor, asymmetrically and proportionally downscaling action-parameter gradients to account for the imbalance of actions in minibatch samples is the main reason for the observed increase in average evaluation scores.

We suspect that an unintended effect of having asymmetric weightings for the Q-values is that the balance of false gradients in the summation loss function, discussed in Section 4.1.1, is changed. This could lead to the true action-parameter gradient becoming less or more prominent depending on how often the associated action is sampled. This might then explain why using random weightings increases the learning speed on Platform, but decreases it on Robot Soccer Goal where we suspect the effects of action sampling imbalance not to be as significant, as mentioned in Section 4.3.1. Despite this, the loss with proportional c_k weightings, henceforth simply referred to as the weighted loss, still shows an increase in evaluation performance overall.

4.3.3 (E3) Weighted-Indexed Loss

Finally, we evaluate whether the combination of the proposed weighted and indexed action-parameter losses—the weighted-indexed loss, Equation (4.10)—further increases the performance of P-DQN. We compare against results from previous experiments for the indexed loss with gradient-zeroing and the proportionally weighted loss, and P-DQN with the summation loss again used as a baseline. We only test the weighted-indexed loss when also using gradient-zeroing, as this was shown in Section 4.3.1 to be strictly better than the indexed loss without gradient-zeroing and this removes any impact of false gradients. We present the results of this experiment in Table 4.3 and Figure 4.4.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.964 ± 0.068	**0.997**	64 705
W	0.972 ± 0.057	**0.997**	65 587
I+GZ	**0.990 ± 0.016**	**0.996**	**73 567**
W+I+GZ	**0.989 ± 0.019**	**0.995**	72 144

Robot Soccer Goal

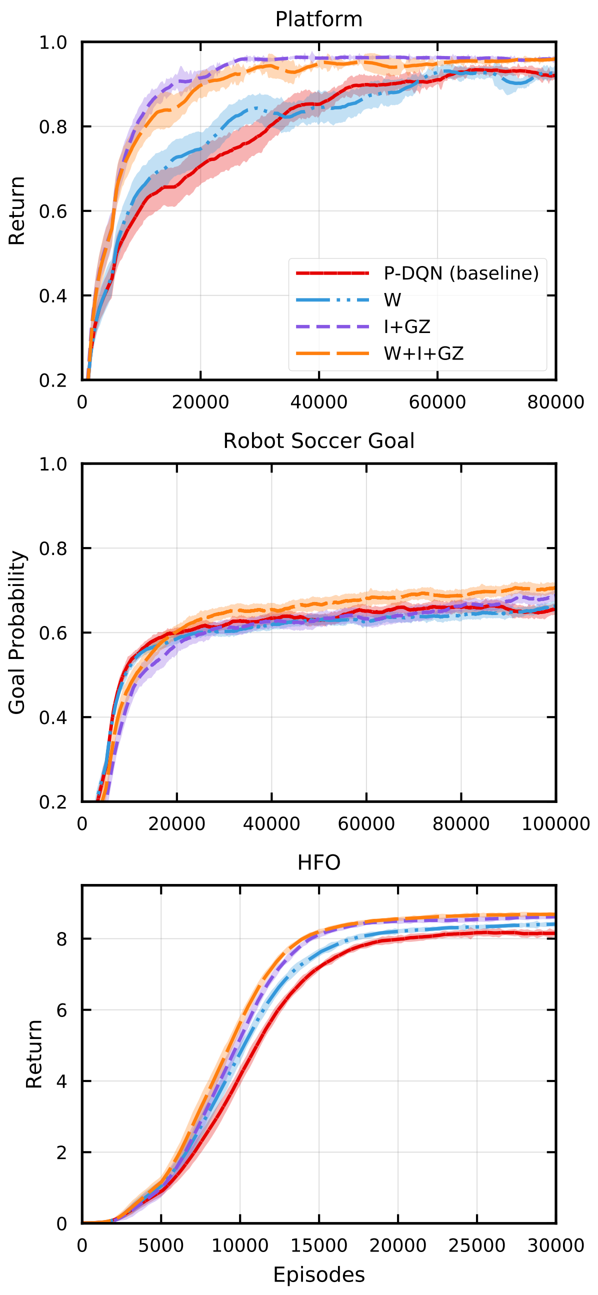
	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	61 445
W	0.683 ± 0.069	0.666	60 312
I+GZ	0.700 ± 0.073	0.720	60 119
W+I+GZ	**0.734 ± 0.073**	**0.756**	**63 429**

Half Field Offense

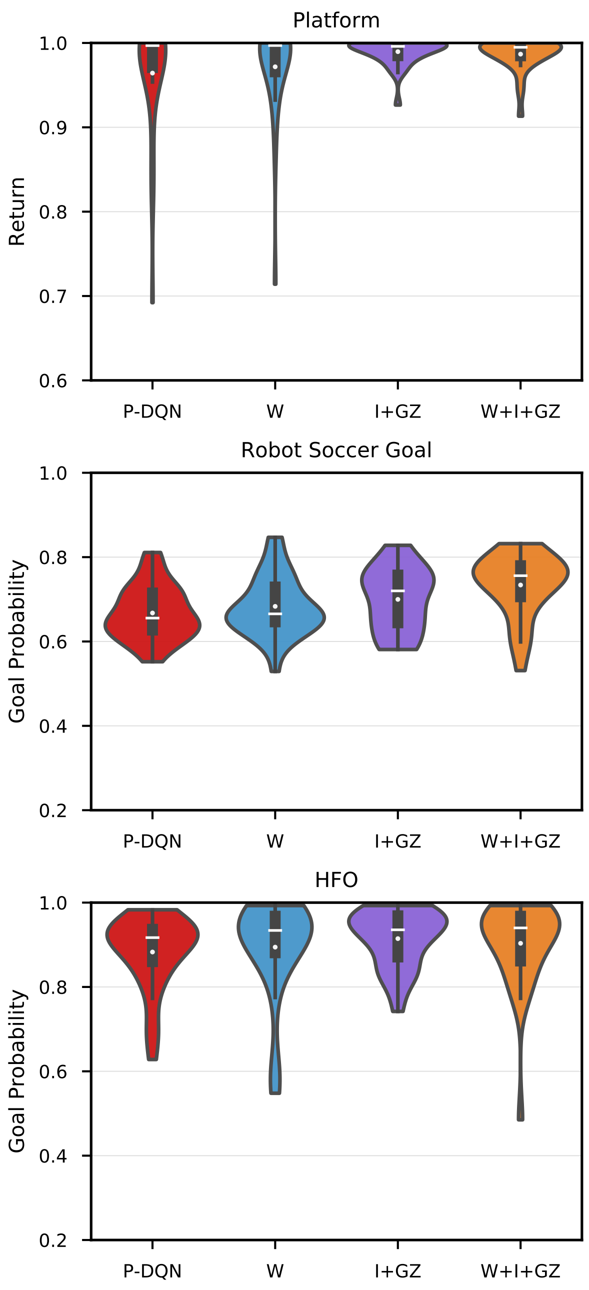
	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	0.883 ± 0.085	0.917	111 ± 11	182 573
W	0.894 ± 0.116	0.934	110 ± 9	191 873
I+GZ	**0.915 ± 0.066**	0.936	**106 ± 11**	199 747
W+I+GZ	0.904 ± 0.100	**0.940**	**107 ± 9**	**204 382**

Table 4.3: Results of using the combined weighted-indexed action-parameter loss with gradient-zeroing (W+I+GZ) compared to using either the weighted loss (W), or indexed loss with gradient-zeroing (I+GZ) alone. The difference between the weighted-indexed and indexed losses with gradient-zeroing on Platform is negligible because almost all agents learn to reach the goal consistently in the first place, leaving little room for improvement bar the reduction of outliers. Robot Soccer Goal shows the clearest increase in performance when using the weighted-indexed loss both in terms of evaluation scores and area under the curve. It also achieves the best performance on HFO across all measures except the mean score, which is due to an outlier shown in Figure 4.4b reducing the average.

We see that the combined weighted-indexed loss with gradient-zeroing achieves the highest median evaluation scores across all domains. The mean score on HFO is lower than that of just the indexed loss with gradient-zeroing but we see in Figure 4.4b that this is due to a single outlier pulling down the average. Surprisingly, the addition of weighting terms to the indexed loss with gradient-zeroing shows an increase in area under the curve on Robot Soccer Goal, contrary to the results of the previous experiment with the weighted loss alone. The gradient-zeroing technique removes any problems related to the balance of true and false action-parameter gradients that may have been an issue for the weighted loss in the previous experiment. The results of this experiment demonstrate that both the indexed and weighted loss functions are complementary and beneficial in addressing different problems associated with action sampling imbalance, since the combined weighted-indexed loss shows greater performance than using either loss alone.



(a) Learning curves



(b) Evaluation scores

Figure 4.4: Graphs of the results of using the combined weighted-indexed action-parameter loss with gradient-zeroing (W+I+GZ) compared to using either the weighted loss (W), or indexed loss with gradient-zeroing (I+GZ) alone. The learning curves in (a) show a negligible difference between the weighted-indexed and indexed losses on Platform and HFO during training, but the learning curve for Robot Soccer Goal is notably higher. The evaluation scores in (b) indicate a more significant increase as a result of the combined weighted-indexed loss than during training.

4.4 Summary

We discussed two of the problems associated with the summation action-parameter loss function for P-DQN as a result of the imbalance between actions experienced by agents during learning. To address this, we proposed the indexed and weighted loss functions as alternatives. Our experiments empirically show that the use of these loss functions improves the performance of P-DQN in general, and that the combined weighted-indexed loss function is better than using either the weighted or indexed loss alone. Furthermore, we showed that the dependence of the Q-network in P-DQN on the joint action-parameter vector over all actions can cause what we term false gradients during updates. Our gradient-zeroing technique for the indexed loss eliminates this problem, improving performance on all domains.

Chapter 5

Independent Action-Parameter Methods

Recall that the joint vector $\mathbf{x} = (x_1, \ldots, x_K)$ over all action-parameters is input to the Q-network for P-DQN (Figure 3.1a), causing the Q-values to become dependent on the parameters of all actions, not just their own. In this chapter, we explore the detrimental effects this has not only on the action-parameter gradients, as was seen in Chapter 4, but the discrete action policy too. We then propose two solutions for P-DQN to ensure that Q-values depend only on their associated action-parameters.

5.1 Q-Value Action-Parameter Sensitivity

To investigate whether the Q-network in P-DQN learns to compensate and discount the impact of unassociated action-parameters, we measure the sensitivity of each Q-value to its own action-parameter versus others. To do so, we employ a technique similar to Grad-CAM [Selvaraju *et al.* 2017], whereby we record the magnitude of the gradients each Q-value produces with respect to the action-parameters during backward passes of the Q-network. We define the sensitivity of a Q-value in our case as the L2-norm over its gradient vector. This can be calculated separately over the associated action-parameter,

$$\text{sensitivity}\left(Q_k, x_k\right) := \left\| \left(\tfrac{\partial Q_k}{\partial x_k} \right) \right\|_2 , \tag{5.1}$$

and unassociated action-parameters,

$$\text{sensitivity}\left(Q_k, x_{1,\ldots,k-1,k+1,\ldots,K}\right) := \left\| \left(\tfrac{\partial Q_k}{\partial x_1}, \quad \cdots \quad , \tfrac{\partial Q_k}{\partial x_{k-1}}, \quad \tfrac{\partial Q_k}{\partial x_{k+1}}, \quad \cdots \quad , \tfrac{\partial Q_k}{\partial x_K} \right) \right\|_2 . \tag{5.2}$$

These two values can then be used to quantify how sensitive a Q-value is to changes in its own action-parameter relative to those of other actions. We record these sensitivities while training a P-DQN agent for each of the benchmark domains; plots of these values are shown in Figure 5.1. We observe a strong correlation between the sensitivities of the associated and other action-parameters on Platform and HFO. While there is a downward trend of sensitivity over time on Platform, there is instead an upward trend on HFO. The correlation is not as prominent on Robot Soccer Goal but there is still no indication of the sensitivity to other action-parameters decreasing during training. We can therefore infer from these results that the learned Q-values do not, in general, become independent of other action-parameters. In fact, we observe that the Q-values of some actions—hop on Platform, kick-to on Robot Soccer Goal, and dash on HFO—actually become more sensitive to the parameters of other actions.

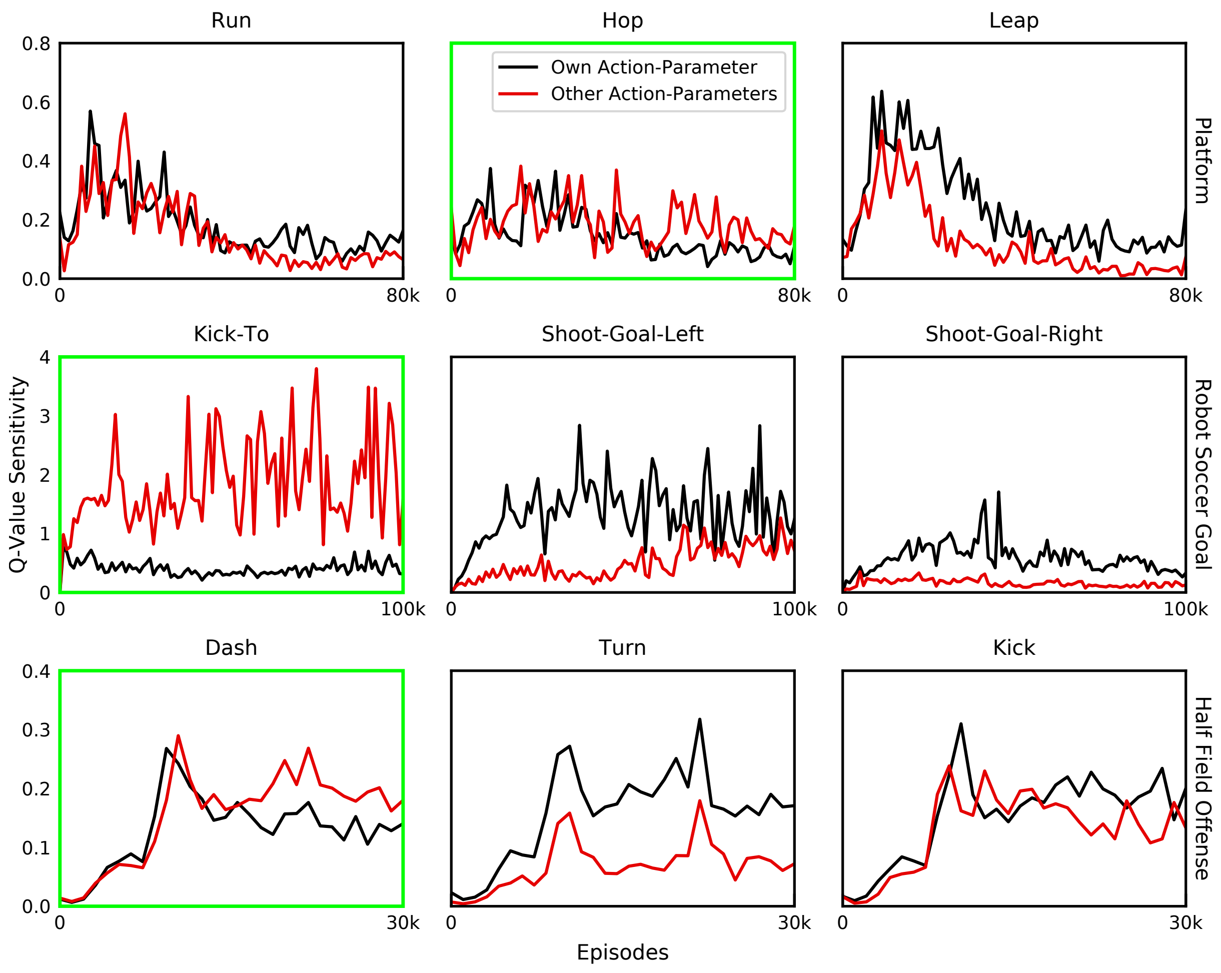


Figure 5.1: Sensitivity of each Q-value to its own action-parameter (black) versus those of other actions (red), sampled every 1000 episodes from a single agent over the course of training on Platform (top), Robot Soccer Goal (middle), and HFO (bottom). Each sensitivity calculation is averaged over 10 episodes. Despite the unassociated action-parameters having no bearing on the resulting transition when any action other than their own is executed, Q-values under P-DQN do not become independent of these conflating values in general. Instead, we see several cases (highlighted in green) where the Q-value of an action becomes more sensitive to the action-parameters of other actions. We can see from the range of values that P-DQN on Robot Soccer Goal is far more sensitive to its action-parameters than on Platform and HFO. This may be a factor in why the indexed loss function without gradient-zeroing performs worse only on Robot Soccer Goal in Section 4.3.1.

5.2 Problems with Q-Value Sensitivity to Unassociated Action-Parameters

We now know that the P-DQN Q-network does not learn to discount the impact of unassociated action-parameters for each Q-value. We identify and discuss two problems with Q-values being dependent on the combined action-parameter vector $\mathbf{x}$ instead of x_k.

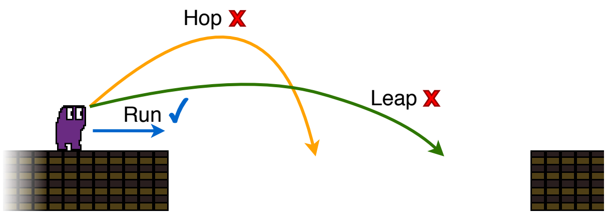
5.2.1 False Action-Parameter Gradients

We saw in Section 4.1.1 how each Q-value, Q_k, produces false gradients for unassociated action-parameters $x_j, j \in [K]\ j \neq k$. We also saw how explicitly removing the impact of these false gradient by setting them to zero, $\partial Q_k / \partial x_j \leftarrow 0\ \forall k, j \in [K], j \neq k$, improves performance of P-DQN. The sensitivity analysis in Figure 5.1 gives additional insight into this as it shows that some Q-values, particularly the kick-to action on Robot Soccer Goal, contribute more to other action-parameters than the Q-values of those actions themselves.

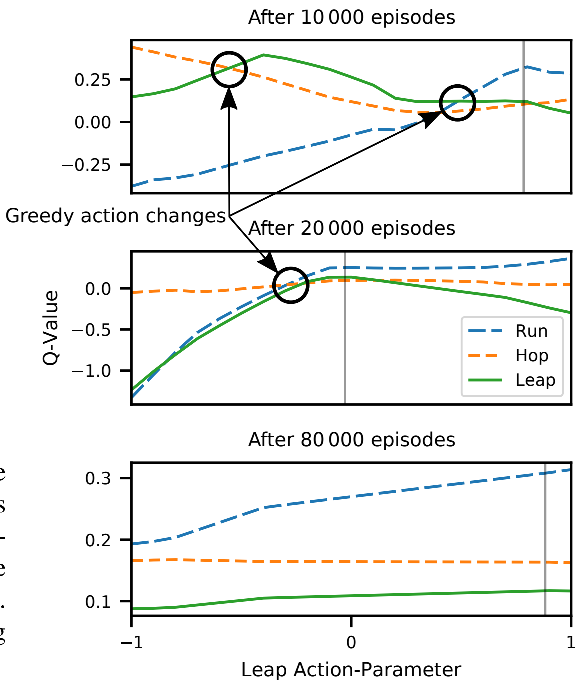
Recall that false gradients are produced by Q-values that have no information on how those action-parameters affect transitions from any given state. One may argue, however, that feeding $x_j, j \neq k$ as input to Q_k gives it information on the current action-parameter policy in future states and how it may affect the expected future return which the Q-value encodes. However, $Q(s, k, \mathbf{x})$ only gives an estimate of the expected return from the current state s, and there is not necessarily any correlation between the x_j chosen in state s and the x_j chosen in some future state s' in general. This argument is further refuted by the fact that, again, Q_k is only updated with samples where action (k, x_k) is executed and not (j, x_j). This is the key difference between PA-DDPG and P-DQN: while PA-DDPG also uses a joint action-parameter vector as input to its critic, the Q-value it calculates is over all actions representing both action and action-parameter policies. The critic in PA-DDPG is also updated using samples from all actions, unlike the action-specific Q-values in P-DQN.

5.2.2 Discrete Action Policy Perturbation

The second problem with Q-value sensitivity to all action-parameters is with the discrete action policy. Specifically, updating the continuous action-parameter policy of any action can affect the discrete action policy by perturbing the Q-values of all actions, and not just of the action associated with that parameter. This can lead to the relative ordering of Q-values changing, which in turn changes any greedy action policy to be possibly suboptimal. We demonstrate a situation where this occurs on the Platform domain in Figure 5.2.



(a) Agent in the Platform domain near the edge of the second platform. Only the run action is optimal in this situation as the maximum hop or leap distance is insufficient to traverse the gap without moving closer to the edge first, and hopping slightly forward is slower than running. Leaping in this state will always result in the agent falling into the gap and dying.



(b) Predicted Q-values for the state in (a) shown varying with the leap action-parameter at different points during training while the run and hop action-parameters are kept fixed. The vertical lines indicate the leap value actually chosen by the policy. Crossover points where the maximum Q-value changes are circled.

Figure 5.2: Example of sensitivity to unrelated action-parameters affecting discrete action selection on the Platform domain. In a particular state (a), the optimal action is to run forward to be able to traverse a gap. Choosing to leap here is suboptimal and would result in the agent dying irrespective of the associated action-parameter. The Q-value of the leap action changes with its action-parameter, however (b) shows that varying the leap action-parameter changes the Q-values predicted by P-DQN for *all* actions. Near the start of training, changing the leap value can alter the discrete policy such that a suboptimal action is chosen. At the end of training, after 80 000 episodes, the agent correctly learns to choose the optimal action regardless of the unrelated leap value; the other Q-values still vary but to a lesser extent than during training.

The observation that Q-values are learned such that they retain relative ordering when one of the action-parameters is perturbed may suggest that the false gradients $\partial Q_k / \partial x_j, j \neq k$ are not based on an expected improvement in the state-action value function with respect to action-parameter policy, but rather to correct or maintain the discrete action policy: changing the action-parameters similar to the weights of the Q-network to better fit the Q-value function.

5.3 Split Q-Networks

The obvious solution to joint action-parameter inputs in P-DQN is to split the Q-network, maintaining a separate, identical Q-network for each action. Then, one can input only the relevant action-parameter x_k to the network corresponding to action k. The resulting architecture is illustrated below in Figure 5.3.

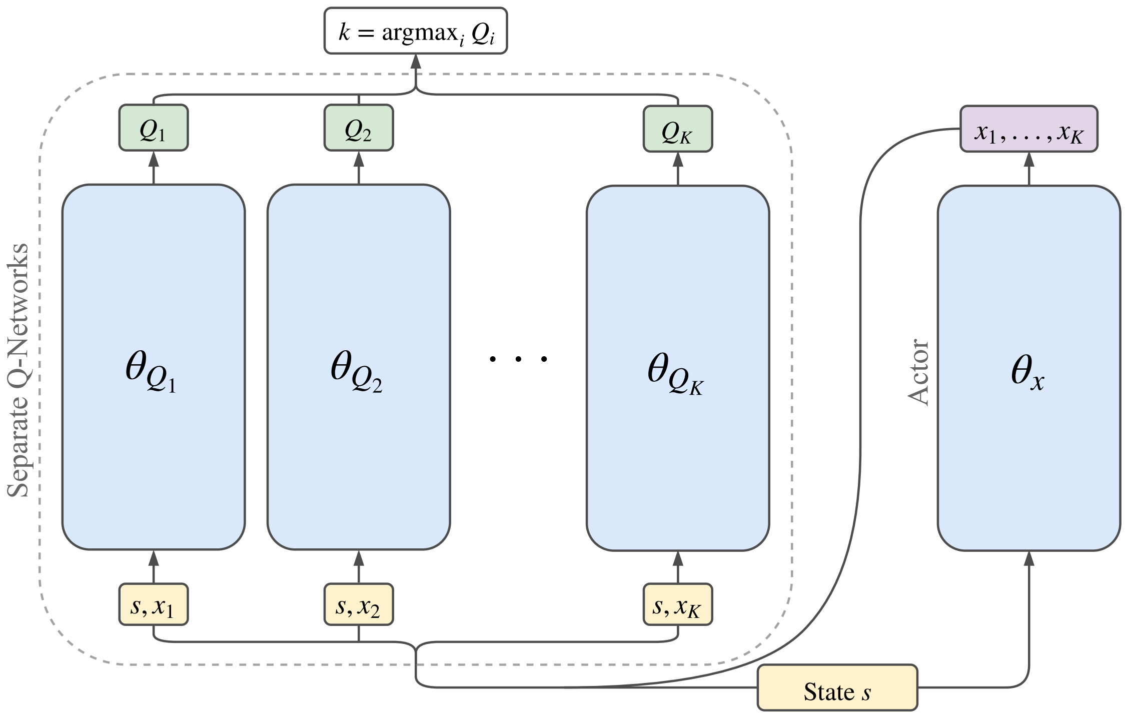


Figure 5.3: Separate Q-networks used for P-DQN. Each action has its own corresponding Q-network, which takes only the state s and associated action-parameter x_k as input.

With this change, each Q_k is reliant only on its action-parameter x_k and no others, since the inputs are explicitly separated between the different networks. This addresses both of the problems detailed in Section 5.2 as Q-values are not influenced by unassociated action-parameters nor produce any gradients for them. There are, however, other consequences that come from duplicating and separating the Q-network:

1. There is no shared feature extraction learned between the separate Q-networks.
2. Additional computational complexity is introduced.

Having no shared layers between the networks could be advantageous as the network capacity is increased: each Q-network then learns only the feature extraction and Q-value approximation specific to a single action, rather than having to generalise over all actions. On the other hand, having to learn a separate feature representations could lead to some Q-values taking longer to learn than with a joint representation, if their associated actions are seldom explored for instance. This can occur as a result of action sampling imbalance, or the fact that certain actions have no effect until the agent is in specific states explored later in training—the kick action in HFO is one such example, as it requires the agent first learns to approach the ball before the kick has any effect on the environment.

With regards to computational complexity: for a standard DQN, only the weights of the last hidden layer are duplicated based on the number of actions, but in our approach we duplicate the entire Q-network for each action. This worsens with the number of hidden layers and neurons used, so the storage and memory requirements of the Q-network portion of P-DQN increase from $\mathcal{O}(W)$ to $\mathcal{O}(KW)$, where W is the total number of weights. Duplicating the trainable network parameters also increases the computational requirements of predicting Q-values and optimising the networks, thus the cost of inference and updates similarly scales with the number of actions to a greater extent than with a single Q-network.

5.3.1 Split Q-Networks with Shared Layers

We can partially address these concerns by introducing shared layers to the split Q-network, allowing a partial feature representation to be shared and reducing the number of duplicated parameters. The action-parameters are then input after the shared layers to the separate Q-value heads. The proposed architecture is shown below in Figure 5.4.

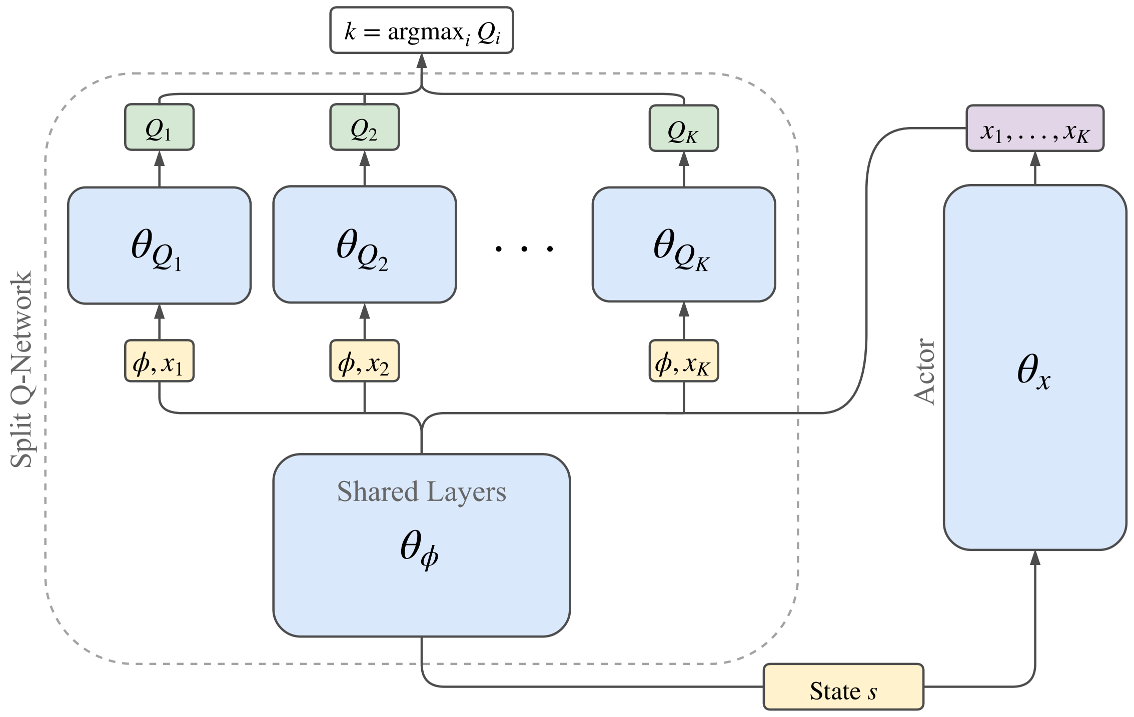


Figure 5.4: Split Q-network architecture with shared feature extraction layers for P-DQN. The shared layers compute a feature representation ϕ from the state variables s before being fed into separate Q-value heads along with the corresponding action-parameter x_i for each action.

This introduces a new design choice: how many layers should be shared and how many should be split?

There is a trade-off between the capacity of the network dedicated to shared feature representation and Q-value function approximation. The further along the network the action-parameters are input, the less representational capacity there is available to model the Q-values in relation to the action-parameters. This forms an upper limit on the number of shared layers: at most $\ell - 1$ hidden layers can be shared, where ℓ is the total number of hidden layers, because inputting the action-parameters at the output layer of each head assumes a strictly linear relationship between the Q-values and action-parameters. This is an unreasonable assumption that does not apply in general.

5.4 Multi-Pass Q-Networks

Even with shared layers, the approach of having split or separate Q-networks significantly increases the computational complexity of the algorithm. So we consider an alternative approach that does not involve architectural changes to the network structure of P-DQN. Instead of splitting the network in order to separate the joint action-parameter vector, we can perform multiple forward passes over a single Q-network. We refer to this as the *multi-pass*, or *multi-pass Q-network*, technique.

Separating the action-parameters in a single forward pass of a single Q-network with fully connected layers is impossible. So instead, we perform a forward pass once per action k with the state s and action-parameter vector $\mathbf{x}\mathbf{e}_k$ as input, where $\mathbf{e}_k$ is the standard basis vector for dimension k. Then $\mathbf{x}\mathbf{e}_k = (0, \dots, 0, x_k, 0, \dots, 0)$ is the joint action-parameter vector where each $x_j, j \neq k$ is set to zero. This firstly causes all false gradients to be zero, ${}^{\partial Q_k}/_{\partial x_j} = 0$, and secondly, it completely negates the impact of the network weights for unassociated action-parameters x_j from the input layer, making Q_k only depend on x_k, that is,

$$Q\left(s, k, \mathbf{x}\mathbf{e}_k\right) = Q\left(s, k, x_k\right). \tag{5.3}$$

Thus both of the problems in Section 5.2 are addressed without introducing any additional neural network parameters.

This requires a total of K forward passes to predict all Q-values instead of one. However, we can make use of the parallel minibatch processing capabilities of artificial neural networks, provided by libraries such as PyTorch and Tensorflow, to perform this in a single parallel pass, or *multi-pass*. A multi-pass with K actions is processed in the same manner as a minibatch of size K. Because a forward pass of the Q-network cannot avoid generating all Q-values, some unnecessary computation is introduced: each pass still generates all K Q-values even though only one is used. In fact, a total of K^2 Q-values are generated from this method:

$$\begin{pmatrix} Q\left(s, \cdot, \mathbf{x}\mathbf{e}_1; \theta_Q\right) \\ \vdots \\ Q\left(s, \cdot, \mathbf{x}\mathbf{e}_K; \theta_Q\right) \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} & \cdots & Q_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{K1} & Q_{K2} & \cdots & Q_{KK} \end{pmatrix}, \tag{5.4}$$

where Q_{ij} is the Q-value for action j generated on the i^{th} pass where x_i is non-zero. Thus only the diagonal elements Q_{ii} are valid and used in the final output $Q_i \leftarrow Q_{ii}$. This process is illustrated in Figure 5.5. Note that a single forward pass of the Q-network produces K Q-values regardless. The only unnecessary computation is at the output layer for Q-values where $i \neq j$, so the overhead does not scale quadratically but rather linearly with the number of actions.

For a concrete example, consider using the multi-pass technique with P-DQN on HFO. There are three actions available, $K = 3$. During environment interaction, the agent receives a single state at a time and uses a multi-pass of size three to predict the Q-values for the discrete action policy:

$$\begin{pmatrix} Q\left(s, \cdot, \mathbf{xe}_1; \theta_Q\right) \\ Q\left(s, \cdot, \mathbf{xe}_2; \theta_Q\right) \\ Q\left(s, \cdot, \mathbf{xe}_3; \theta_Q\right) \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} & Q_{13} \\ Q_{21} & Q_{22} & Q_{23} \\ Q_{31} & Q_{32} & Q_{33} \end{pmatrix},$$

from which we extract the diagonal elements $Q_1 \leftarrow Q_{11}, Q_2 \leftarrow Q_{22}, Q_3 \leftarrow Q_{33}$. During updates on HFO, a minibatch of size 32 is used. This is similarly duplicated 3 times to form a multi-pass minibatch of size 96 during forward passes. However since a multi-pass only extracts a single Q-value from each of the duplicates, the backward pass to calculate gradients is still over 32 samples, not 96. In practice, the computation graph used for automatic differentiation would include extra operations but the gradient accumulation is performed only for those Q-values used in the loss functions, and never $Q_{ij}, j \neq i$.

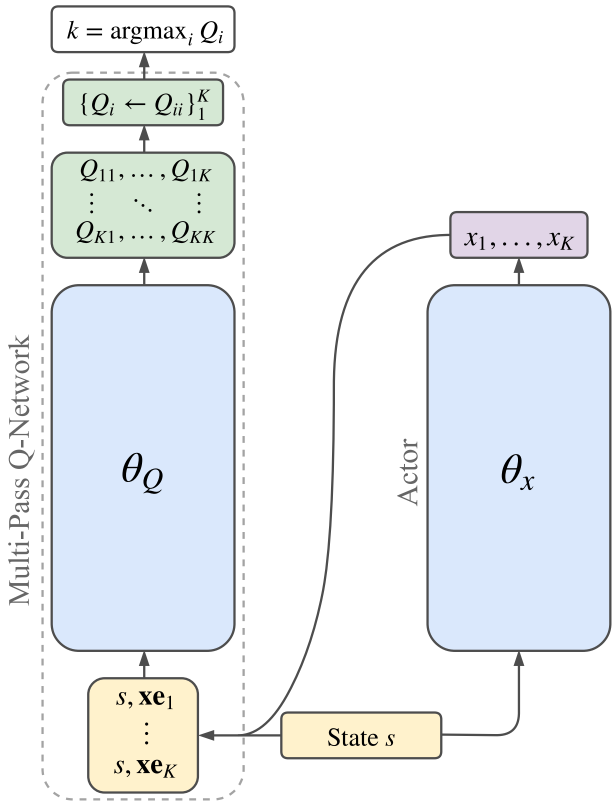


Figure 5.5: Multi-pass Q-network architecture for P-DQN. The input to the network is duplicated K times, once for each action, to form a minibatch. The row for action k uses $\mathbf{xe}_k$ as the action-parameter input, where only x_k is non-zero, instead of the full joint action-parameter vector $\mathbf{x}$.

Compared to split Q-networks, our multi-pass technique introduces a relatively minor amount of overhead during forward passes. The computational complexity of this overhead scales linearly with the number of actions $O(K)$ for inference on a single state, and also scales with the minibatch size B during updates, $O(KB)$. This is because multi-passes are efficiently processed in parallel as regular minibatches. So unlike split Q-networks, even when a larger Q-network with more hidden layers and neurons is used, if the number of actions does not change then the overhead of multi-passes would be the same as with a smaller Q-network. This is shown empirically in Section 5.5.4.

5.5 Experiments

We design four sets of experiments to evaluate different aspects of the two proposed methods, and a fifth to re-evaluate the weighted and indexed action-parameter loss functions from Chapter 4 in combination with the multi-pass technique:

(E1) Separate Q-Networks

(E2) Split Q-Networks with Shared Layers

(E3) Multi-Pass Q-Networks

(E4) Computational Overhead

(E5) Weighted-Indexed Loss Ablation Study

In each experiment, we use the same configuration for P-DQN from Section 3.2.1 on the three benchmark domains.

5.5.1 (E1) Separate Q-Networks

We first evaluate P-DQN with the use of separate Q-networks without shared layers versus a single Q-network as the baseline. To determine whether any changes are due to the independence of Q-values from unassociated action-parameters or simply a result of having separate networks, we also evaluate using duplicate Q-networks. For this, we use an identical Q-network for each action but input the joint action-parameter vector $\mathbf{x}$, instead of only the corresponding x_k as done for separate Q-networks. Our results are shown in Table 5.1 and Figure 5.6.

We observe significant improvements in the learning efficiency and evaluation scores of P-DQN when using separate Q-networks on Platform and Robot Soccer Goal. Duplicate Q-networks, on the other hand, produce similar results to baseline P-DQN with a single, standard Q-network: the learning curve is slightly higher on Platform but almost identical on Robot Soccer Goal. This shows that an increased network capacity and lack of shared features does not improve performance on the toy domains, but that the separation of action-parameter inputs does, as expected. The results on the more complex HFO domain, however, show that agents using separate Q-networks learn faster initially but eventually taper off and achieve significantly lower evaluation scores than the baseline. Agents using duplicate Q-networks show even worse performance, with a lower average score and a reduction in learning speed compared to the baseline.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	**0.964** ± **0.068**	0.997	64 705
Separate Q-Networks	0.941 ± 0.164	**0.999**	**70 358**
Duplicate Q-Networks	**0.951** ± **0.109**	0.994	66 315

Robot Soccer Goal

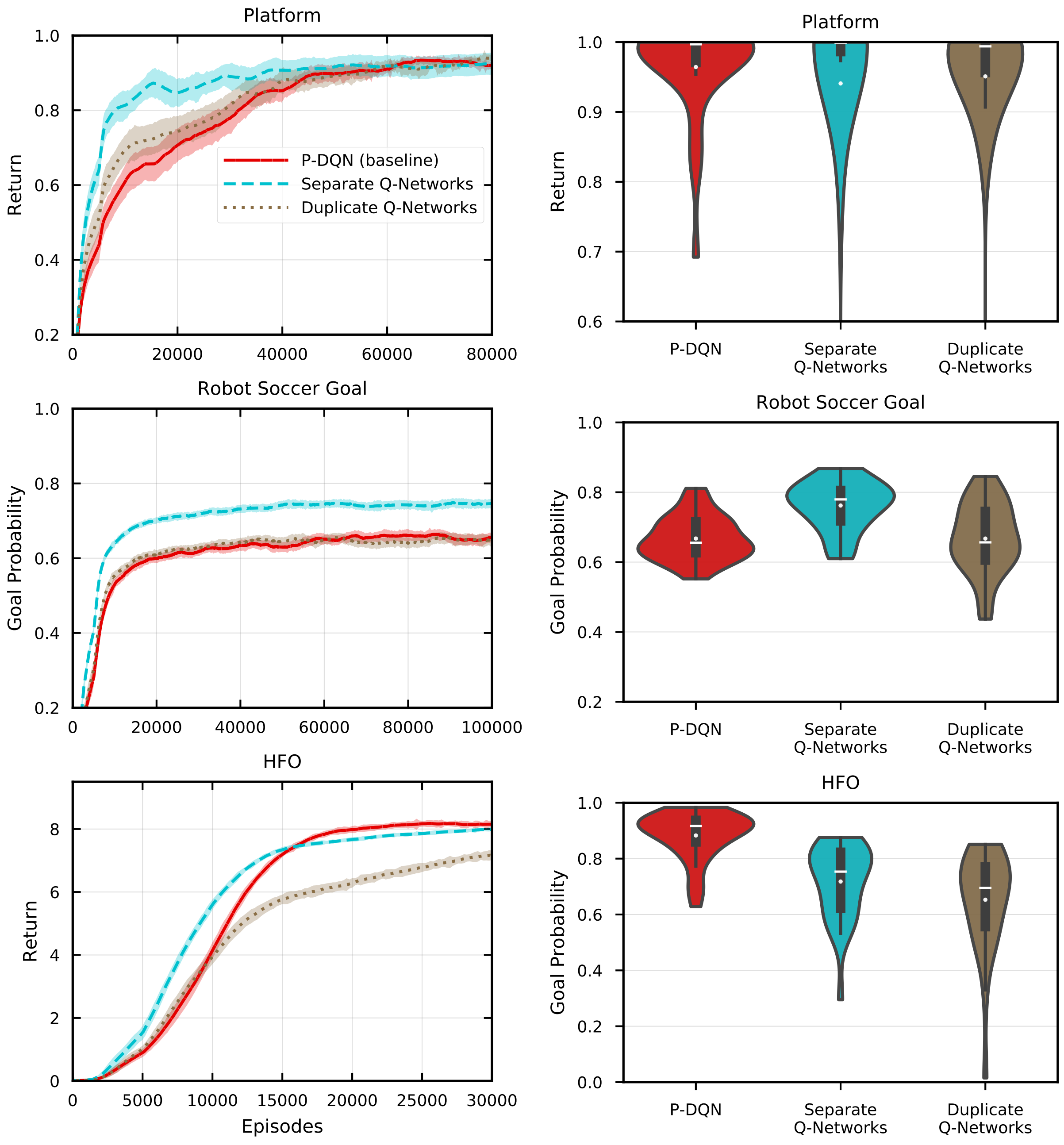
	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	61 445
Separate Q-Networks	**0.762** ± **0.070**	**0.780**	**71 121**
Duplicate Q-Networks	0.667 ± 0.100	0.657	61 791

Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	**0.883** ± **0.085**	**0.917**	111 ± 11	182 573
Separate Q-Networks	0.718 ± 0.131	0.754	**99** ± **7**	**189 916**
Duplicate Q-Networks	0.653 ± 0.174	0.695	119 ± 15	154 756

Table 5.1: Results of using the proposed architecture with separate Q-networks versus duplicate Q-networks for P-DQN. Separate Q-networks achieve a higher median score and area under the curve on Platform but a lower-than-baseline mean score; this is a result of a few outliers shown in Figure 5.6b. On Robot Soccer Goal the scores are significantly higher across the board for separate Q-networks. In both these domains, duplicate Q-networks perform very similarly to the baseline. HFO, however, shows both separate and duplicate Q-networks achieve evaluation scores significantly lower than the baseline.

This raises the question of why the observed behaviour of separate Q-networks on HFO is different than on Platform and Robot Soccer Goal. Our hyperparameter search found that P-DQN on the two toy domains performs best with a relatively small network of one hidden layer, while three hidden layers are best for HFO. This indicates that less complex feature representations and Q-value function approximations are required on Platform and Robot Soccer Goal, so the lack of shared layers may not be detrimental. HFO has far more state variables—51 versus 9 and 14—and thus more complex features, requiring a wider and deeper network for function approximation. Thus we suspect the observed decrease in performance is primarily due to the lack of a shared feature representation between the separate Q-networks on HFO. This is supported by the fact that duplicate Q-networks perform poorly on HFO to a much worse extent than separate Q-networks in both training and evaluation. So while separating the action-parameter inputs is definitely beneficial, it is not enough to overcome the lack of a shared feature representation. To further investigate this claim, we assess the use of split Q-networks with shared layers for P-DQN on HFO in Section 5.5.2.



(a) Learning curves

(b) Evaluation scores

Figure 5.6: Graphs of the results of using separate Q-networks compared to duplicate Q-networks for P-DQN, with a single Q-network as a baseline. The learning curves of agents with separate Q-networks on Platform and Robot Soccer Goal show a significant improvement over the baseline during training, but a reduction in performance on HFO; the evaluation scores in (b) similarly reflect this. Note that more outliers are observed on Platform for separate and duplicate Q-networks.

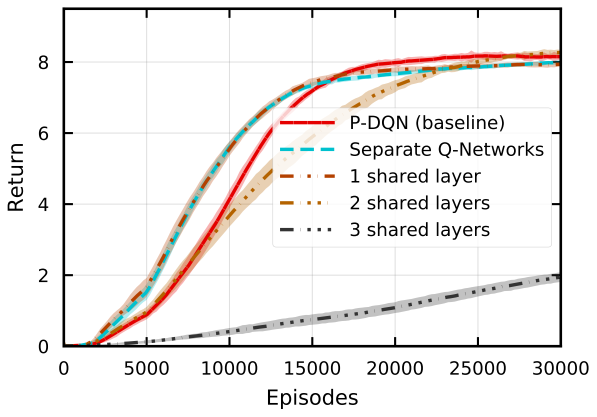
5.5.2 (E2) Split Q-Networks with Shared Layers

P-DQN on HFO uses a Q-network with three hidden layers of 256, 128, 64 neurons respectively. This means at most two hidden layers should be shared when splitting the Q-network; sharing three hidden layers would mean inputting the action-parameters at the output layers in order to still keep them separate, which as we mentioned before would assume a linear relationship between the Q-values and action-parameters. In this experiment, however, we examine the performance of P-DQN using a split Q-network in the case of zero (separate Q-networks), one, two, and three shared hidden layers. The results are shown below in Table 5.2 and Figure 5.7.

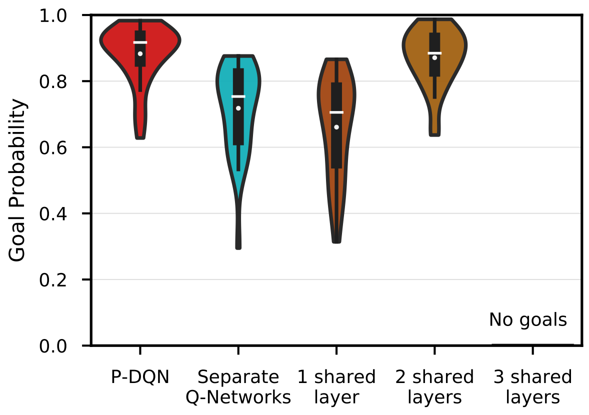
Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	**0.883** ± **0.085**	**0.917**	111 ± 11	182 573
Separate Q-Networks	0.718 ± 0.131	0.754	**99** ± **7**	189 916
Split (1 Shared Layer)	0.661 ± 0.148	0.706	107 ± 10	**191 242**
Split (2 Shared Layers)	0.871 ± 0.086	0.885	109 ± 10	170 643
Split (3 Shared Layers)	0 ± 0	0	n/a	29 289

Table 5.2: Results of using separate and split Q-networks with varying numbers of shared feature extraction layers compared to baseline P-DQN with a single Q-network. The baseline P-DQN agents still achieve the highest mean and median evaluation scores. Agents using split Q-networks with three shared layers completely fail to score any goals, as expected.



(a) Learning curves



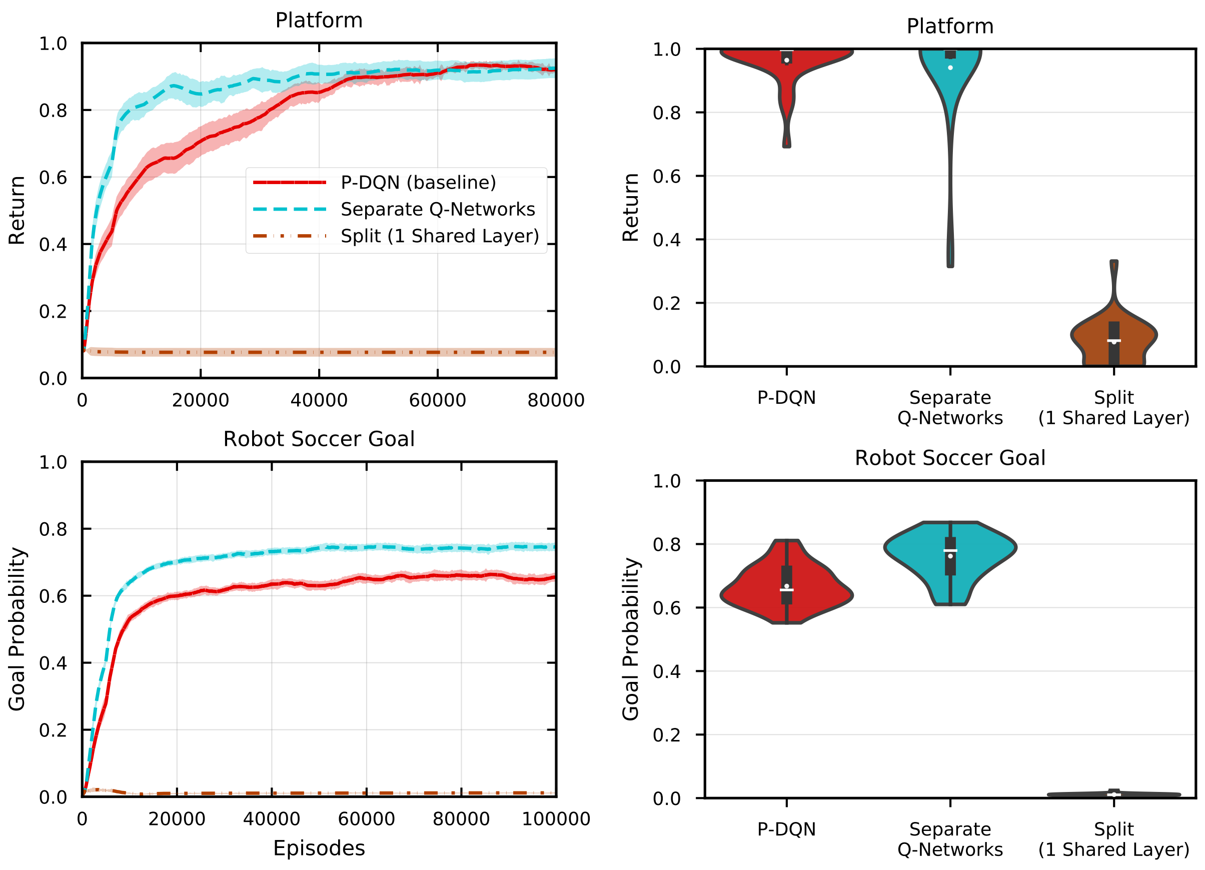
(b) Evaluation scores

Figure 5.7: Graphs of the results of using split Q-networks with shared layers on Half Field Offense.

Unexpectedly, the learning curves in Figure 5.7a show that increasing the number of shared layers beyond one decreases the learning speed, yet using two shared layers results in the highest evaluation scores second only to the baseline. This shows that a shared feature representation is important but the trade-off between shared layers and the representational capacity with respect to the action-parameters harms performance the later the action-parameters are input to the network.

Linear Action-Parameter Relationship Assumption

To reinforce our assertion that the poor performance of split Q-networks with three shared layers on HFO is due to the assumption of a linear relationship between Q-values and action-parameters forced by the network architecture, we perform similar experiments on Platform and Robot Soccer Goal. Since the Q-networks used for P-DQN on those domains have only one hidden layer, sharing just that layer means the action-parameters are introduced at the output layer of the Q-network.



(a) Learning curves (b) Evaluation scores

Figure 5.8: Graphs of the results of using split Q-networks with shared layers on Platform and Robot Soccer Goal. The agents with split Q-networks do not show any improvement during training and fail to solve either task.

It is clear from Figure 5.8 that P-DQN agents using split Q-networks with all hidden layers shared completely fail to learn on these domains. To avoid this situation, one could expand the network by adding extra hidden layers to the separate sub-networks for each action. However, that would be an unsatisfactory solution as it introduces another architectural design choice and unnecessarily increases the number of parameters, which we want to avoid.

5.5.3 (E3) Multi-Pass Q-Networks

We now evaluate our second proposed solution, the multi-pass technique. Similar to our first experiment, we investigate whether any performance changes from using multi-pass Q-networks are due to the separation of action-parameter inputs to the Q-value functions or the change in method. In this case, we compare against duplicate passes, where samples are duplicated as with multi-passes but the joint action-parameter vector $\mathbf{x}$ is used instead of $\mathbf{x}\mathbf{e}_k$, leaving all duplicates identical. What we expect is that using duplicate passes produce similar results to the baseline, since the loss functions should average out to be the same as without duplicates. The results of our experiments compared to baseline P-DQN are shown in Table 5.3 and Figure 5.9.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.964 ± 0.068	0.997	64 705
Multi-Pass	**0.987 ± 0.039**	**0.999**	**72 390**
Duplicate Passes	0.956 ± 0.052	0.979	67 716

Robot Soccer Goal

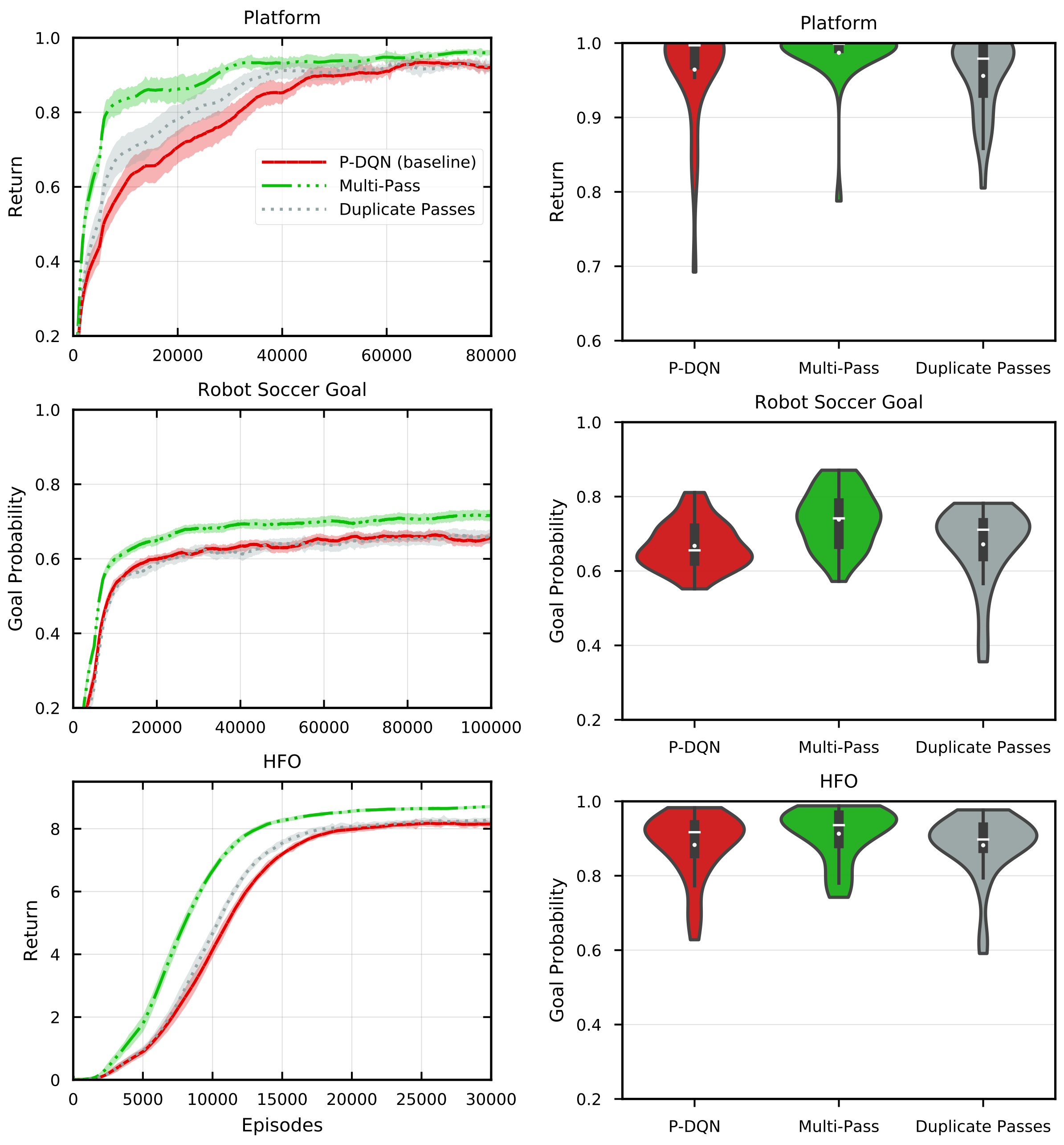
	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	61 445
Multi-Pass	**0.738 ± 0.076**	**0.742**	**67 116**
Duplicate Passes	0.672 ± 0.103	0.711	60 828

Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	0.883 ± 0.085	0.917	111 ± 11	182 573
Multi-Pass	**0.913 ± 0.070**	**0.936**	**99 ± 12**	**213 125**
Duplicate Passes	0.882 ± 0.084	0.898	113 ± 9	188 591

Table 5.3: Graphs of the results of training P-DQN using Q-networks with multi-passes compared to duplicate passes, and baseline P-DQN with single passes. We see that agents using the multi-pass technique achieve the best scores across the board on Platform, Robot Soccer Goal, and HFO. Duplicate passes achieve mean scores similar to the baseline.

The multi-pass technique clearly demonstrates significantly improved performance on all three benchmark domains, including HFO where a much higher learning curve is observed, unlike with split Q-networks. Contrary to our expectation, the use of duplicate passes appears to affect evaluation scores in Figure 5.9b. This could simply be a result of random deviation, however, and we note that the learning curves in Figure 5.9a are largely unaffected and the mean scores from Table 5.3 for duplicate passes and baseline P-DQN are almost identical on all three domains. This indicates that the difference in performance when using the multi-pass technique is due to the independence of Q-values from unrelated action-parameters and not the duplication of passes.



(a) Learning curves (b) Evaluation scores

Figure 5.9: Graphs of the results of training P-DQN using Q-networks with multi-passes compared to duplicate passes, and baseline P-DQN with single passes. The learning curves of multi-pass P-DQN agents are consistently higher than the baseline on all three domains. This is reflected by the evaluation scores, where the multi-pass technique on average achieves the highest scores with fewer outliers and a lower variance between agents, particularly on Platform.

Comparison to Separate Q-Networks

We now use the results from experiments in Sections 5.5.1 and 5.5.3 to directly compare the performance of multi-passes and separate Q-networks. We exclude the results of split Q-networks with shared layers on HFO, as both the training performance and evaluation scores were still inferior to baseline P-DQN.

Platform

	Mean	Median	Area Under Curve
P-DQN (baseline)	0.964 ± 0.068	0.997	64 705
Multi-Pass Q-Network	**0.987 ± 0.039**	**0.999**	**72 390**
Separate Q-Networks	0.941 ± 0.164	**0.999**	70 358

Robot Soccer Goal

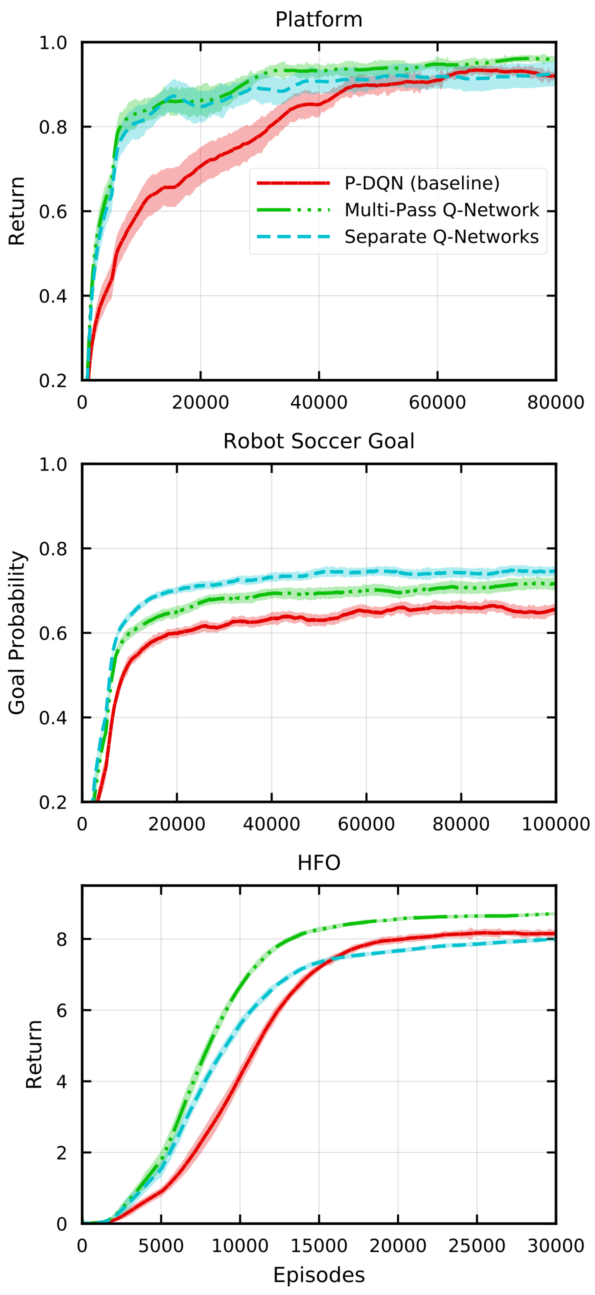
	Mean	Median	Area Under Curve
P-DQN (baseline)	0.668 ± 0.062	0.656	61 445
Multi-Pass Q-Network	0.738 ± 0.076	0.742	67 116
Separate Q-Networks	**0.762 ± 0.070**	**0.780**	**71 121**

Half Field Offense

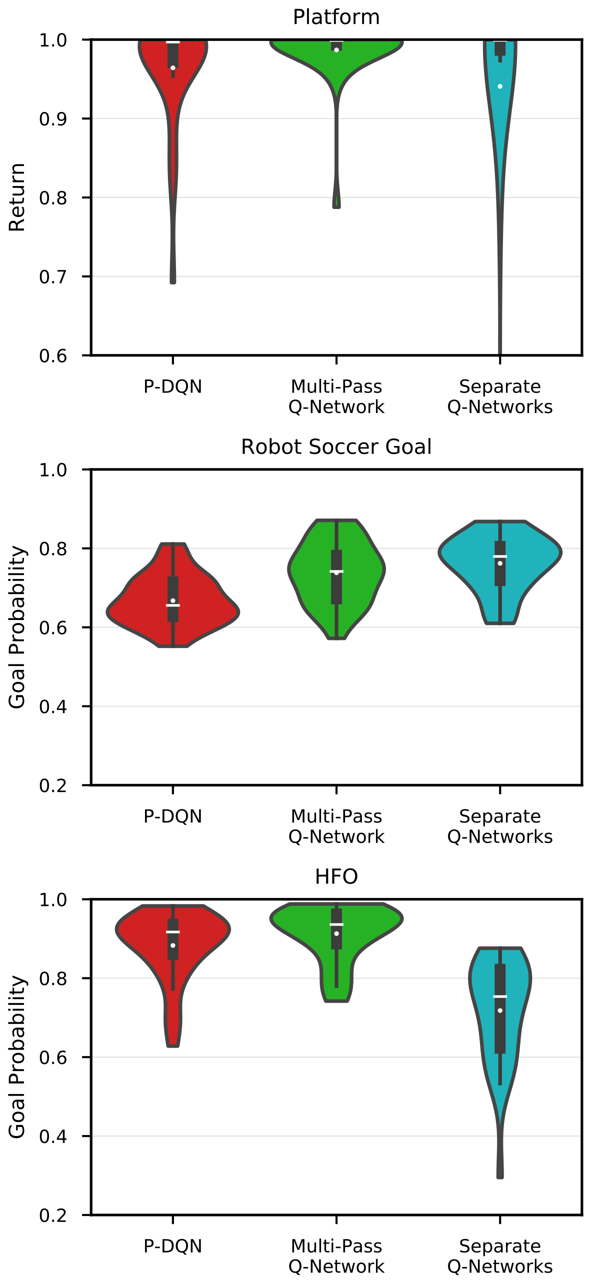
	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN (baseline)	0.883 ± 0.085	0.917	111 ± 11	182 573
Multi-Pass Q-Network	**0.913 ± 0.070**	**0.936**	**99 ± 12**	**213 125**
Separate Q-Networks	0.718 ± 0.131	0.754	**99 ± 7**	189 916

Table 5.4: Comparison between the results of multi-pass and separate Q-networks for P-DQN. While the multi-pass technique achieve higher scores on Platform and HFO, separate Q-networks beat it on Robot Soccer Goal. Note the smaller standard deviation of the multi-pass Q-network on Platform and HFO.

We see in Table 5.4 and Figure 5.10 that the multi-pass technique beats separate Q-networks on Platform and, more importantly, HFO. There is a minor difference in performance between the two methods on Robot Soccer Goal but both still beat the baseline. This difference can possibly be attributed to a greater network capacity being dedicated to approximating each Q-value when using separate Q-networks. On both Platform and HFO, we also observe fewer outliers and a smaller variance between agents using the multi-pass technique. We can reasonably conclude then that the multi-pass technique is more stable and consistent, allowing Q-values to be independent of unrelated action-parameters without suffering the disadvantages of duplicating network parameters, which demonstrably reduces performance with larger networks.



(a) Learning curves



(b) Evaluation scores

Figure 5.10: Graphs comparing the learning curves and evaluation scores of P-DQN with multi-pass Q-networks versus separate Q-networks from previous experiments. The learning curves are similar between the two methods on Platform. On Robot Soccer Goal, the learning curve of agents using separate Q-networks is initially similar to that of multi-pass P-DQN, but converges to a somewhat higher average score. The converse is true on HFO, where both methods show faster-than-baseline learning initially, except P-DQN with separate Q-networks starts plateauing at a lower score than the baseline, whereas multi-pass P-DQN continues to improve.

5.5.4 (E4) Computational Overhead

Both the proposed multi-pass and split Q-network approaches introduce computational overhead to the P-DQN algorithm to different extents. To quantify this additional computation, we examine the total time taken for each technique to process 100 000 independent samples from Platform, Robot Soccer Goal, and HFO. Specifically, we record the cumulative wall time taken to infer an action and perform an update once per sample. Action inference and update timings are performed separately. This methodology is preferred over reporting the training times from our previous experiments as episodes can vary in length based on agent performance—for example on the Platform domain where dying to the first enemy ends an episode with fewer transitions than making it to the goal. This methodology eliminates any influence of the environment on timings other than the number of actions and state variables. The timing results along with the number of trainable neural network parameters are listed in Table 5.5. The cumulative overhead, calculated as the difference in wall time of the proposed methods versus baseline P-DQN, is illustrated in Figure 5.11.

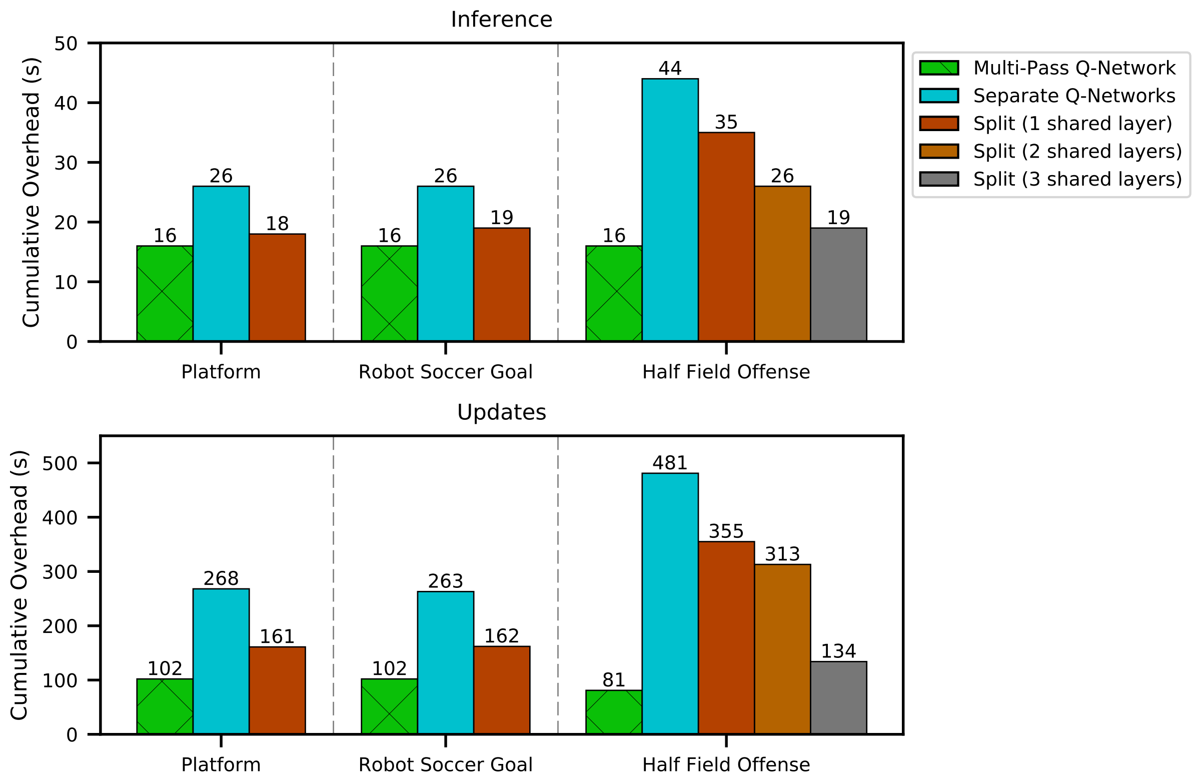


Figure 5.11: Comparison of the cumulative overhead in seconds for inference (top) and updates (bottom) using 100 000 samples for the multi-pass technique compared to separate and split Q-networks. The overhead is calculated as the difference in time taken to process all samples versus baseline P-DQN from the measurements in Table 5.5; lower is better. Using a multi-pass Q-network clearly introduces less computational overhead than using separate Q-networks, which have an overhead roughly 3 times higher for inference on HFO and 6 times higher for updates. Even the smallest split Q-network on HFO with 3 shared layers has higher overhead. The update overhead for a multi-pass Q-network on HFO is slightly lower than on Platform and Robot Soccer Goal due to a smaller minibatch size—32 instead of 128.

Platform

	Inference Time (s)	Update Time (s)	No. Trainable Parameters
Single Q-Network (baseline)	32	387	3718
Multi-Pass Q-Network	48	489	3718
Separate Q-Networks	58	655	6278
Split (1 Shared Layer)	50	548	3337[1]

Robot Soccer Goal

	Inference Time (s)	Update Time (s)	No. Trainable Parameters
Single Q-Network (baseline)	32	393	6023
Multi-Pass Q-Network	48	495	6023
Separate Q-Networks	59	656	10 631
Split (1 Shared Layer)	51	555	5515[1]

Half Field Offense

	Inference Time (s)	Update Time (s)	No. Trainable Parameters
Single Q-Network (baseline)	52	614	114 824
Multi-Pass Q-Network	68	695	114 824
Separate Q-Networks	96	1095	227 848
Split (1 Shared Layer)	87	969	196 488
Split (2 Shared Layers)	78	927	130 376
Split (3 Shared Layers)	71	748	113 549[1]

Table 5.5: Total time taken in seconds for P-DQN with single, multi-pass, separate, and split Q-networks to process 100 000 action inference calculations and updates on each domain (lower is better). The number of trainable neural network parameters is also listed, giving an indication of the storage and memory requirements of each method (lower is better).

The increase in time taken during inference and updates using separate or split Q-networks is significantly greater than when using the multi-pass technique, which is expected since the former duplicates the number of network parameters per action. This difference is more pronounced on HFO where a larger network is used, although the use of split networks with shared layers does alleviate this due to fewer network parameters being duplicated. Multi-passes, on the other hand, add a seemingly constant cumulative overhead of 16 seconds during inference over 100 000 samples on each domain, even with a larger network on HFO; this is because all domains have the same number of parameterised actions. Due to a smaller minibatch size of 32 being used on HFO rather than 128, the increase in update time from using multi-pass Q-network (81s) is actually lower than on Platform and Robot Soccer Goal (102s), despite a much larger network.

These results confirm that the computational overhead of the multi-pass technique is not influenced by the network size, and instead scales with the number of discrete actions and minibatch size used for updates.

[1]The number of trainable parameters for a split Q-network with all hidden layers shared is less than when using a single Q-network because the action-parameters are introduced at the output layer rather than to a hidden layer.

5.5.5 (E5) Weighted-Indexed Loss Ablation Study

Our final experiment re-evaluates the weighted-indexed action-parameter loss function proposed in Chapter 4. We test multi-pass P-DQN with each of the weighted, indexed, and weighted-indexed losses in turn. The multi-pass technique is selected because previous experiments showed it to be the more consistent in improving performance across all benchmark domains than split Q-networks. We also compare against the weighted-indexed loss with gradient-zeroing. This is done to examine the difference in performance with and without the influence of unrelated action-parameters on the discrete action policy, since gradient-zeroing only resolves the issue of false gradients. The results of this experiment are shown in Table 5.6 and Figure 5.12.

Platform

	Mean	Median	Area Under Curve	Δ Mean	Δ Area
P-DQN	0.964 ± 0.068	0.997	64 705	-	-
W+I+GZ	**0.989 ± 0.019**	0.995	72 144	**2.6%**	11.5%
MP+I	0.961 ± 0.134	**0.999**	72 690	−0.3%	12.3%
MP+W	0.971 ± 0.060	**0.999**	**73 071**	0.7%	**12.9%**
MP+W+I	**0.990 ± 0.024**	**0.999**	72 751	**2.7%**	12.4%

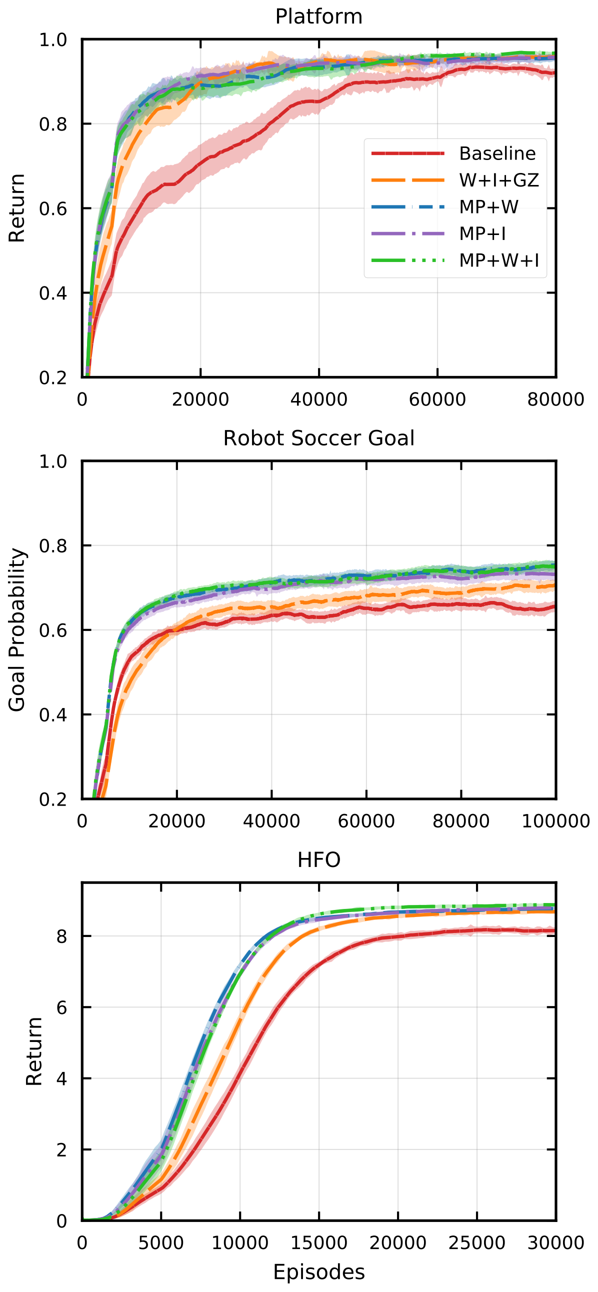
Robot Soccer Goal

	Mean	Median	Area Under Curve	Δ Mean	Δ Area
P-DQN	0.668 ± 0.062	0.656	61 445	-	-
W+I+GZ	0.734 ± 0.073	0.756	63 429	9.9%	3.2%
MP+I	0.762 ± 0.060	**0.771**	**69 644**	14.1%	**13.3%**
MP+W	0.756 ± 0.073	0.739	68 530	13.2%	11.5%
MP+W+I	**0.769 ± 0.066**	**0.770**	**69 665**	**15.1%**	**13.4%**

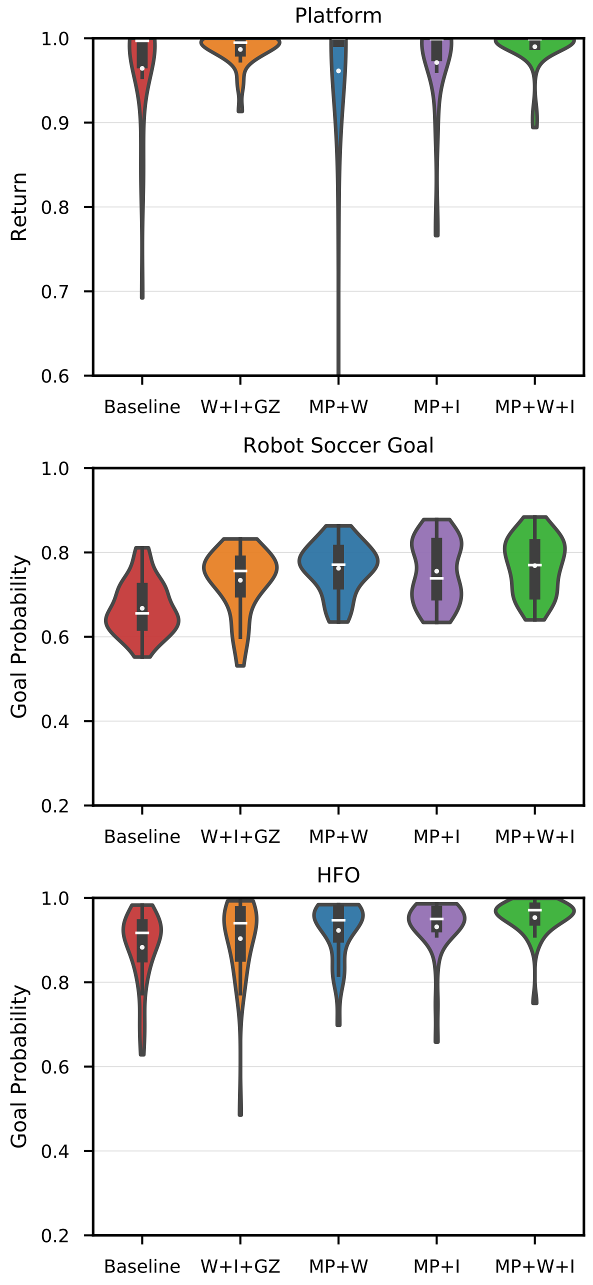
Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve	Δ Mean	Δ Area
P-DQN	0.883 ± 0.085	0.917	111 ± 11	182 573	-	-
W+I+GZ	0.904 ± 0.100	0.940	107 ± 9	204 382	2.4%	11.9%
MP+I	0.923 ± 0.069	0.947	99 ± 9	218 653	4.5%	19.7%
MP+W	0.931 ± 0.068	0.950	**97 ± 7**	217 150	5.4%	18.9%
MP+W+I	**0.953 ± 0.045**	**0.971**	**97 ± 6**	**219 192**	**7.9%**	**20.0%**

Table 5.6: Results of using multi-passes (MP) for P-DQN with the indexed (I), weighted (W), and weighted-indexed (W+I) action-parameter loss functions. We also report the difference in mean scores and area under the curve as a percentage relative to baseline P-DQN. We see multi-passes with the weighted-indexed loss (MP+W+I) achieves the highest mean and median scores on all three domains. There is a notable difference in performance, particularly area under the curve, between the weighted-indexed loss with gradient-zeroing (W+I+GZ) and with multi-passes (MP+W+I); this is most significant on Robot Soccer Goal and Half Field Offense. Performance between the two is almost identical on Platform likely because there is not much room for improvement compared to baseline P-DQN anyway.



(a) Learning curves



(b) Evaluation scores

Figure 5.12: Graphs of the learning curves and evaluation scores of P-DQN with multi-passes (MP) and the weighted (W), indexed (I), and weighted-indexed (W+I) action-parameter loss functions. There is no observable difference between the learning curves when using the various loss functions. The curve of the weighted-indexed loss with gradient-zeroing (W+I+GZ), however, is lower than those agents using multi-passes. The evaluation scores in (b) indicate a trend of improvement with each loss function. The individual losses (MP+I and MP+W) appear to worsen the effects of outliers on Platform, however the combination does not suffer from this problem.

The change in loss functions has a negligible effect on learning efficiency. They do, however, improve the average evaluation scores on all three benchmark domains. We can conclude then that the combined weighted-indexed action-parameter loss function still complements multi-passes in increasing performance of P-DQN. This is expected since the loss function addresses action sampling imbalance, which is separate to the problems presented by joint action-parameter inputs and the multi-pass solution. The second observation of note is the difference between the weighted-indexed loss with gradient-zeroing versus multi-passes, the latter of which is better in terms of both learning speed and evaluation scores. This shows that the influence of unassociated action-parameters on the discrete action policy has a tangible impact on performance, and that addressing false gradients alone by means of gradient-zeroing is insufficient.

Although we used the synchronous, single worker version of P-DQN throughout our experiments, none of the techniques we propose are restricted to this setting and can be readily applied to asynchronous version of P-DQN with parallel workers.

5.6 Summary

We showed that the sensitivity of Q-values to unrelated action-parameters as a result of the joint action-parameter input to the Q-network in P-DQN is problematic. Two solutions to separate the action-parameters inputs were proposed: split Q-networks, and a novel multi-pass technique. Our experiments show that split Q-networks improve performance on Platform and Robot Soccer Goal, but reduce performance on HFO as a result of changes to the network architecture. Furthermore, the computational complexity of split Q-networks scales poorly with larger networks. Our multi-pass technique, on the other hand, was shown to significantly improve performance on all three benchmark domains with minimal overhead. Finally, we further improved the performance of P-DQN by using the weighted-indexed action-parameter loss function from Chapter 4 in conjunction with multi-passes.

Chapter 6

Conclusion and Future Work

Parameterised action spaces are gaining momentum in the reinforcement learning community for their ability to naturally represent complex high-level actions with fine-grained control, allowing for more difficult tasks to be tackled such as learning to play soccer.

We presented an improved version of the recent P-DQN deep reinforcement learning algorithm for parameterised action spaces [Xiong *et al.* 2018], the state-of-the-art approach at the time of writing. Our modified version includes mixed n-step return updates and employs target networks for stability. Our comparison study shows that P-DQN outperforms previous methods Q-PAMDP [Masson *et al.* 2016] and PA-DDPG [Hausknecht and Stone 2016ab] on Platform and Robot Soccer Goal, but fails to beat PA-DDPG on HFO outright without asynchronous parallel workers and dueling networks. We also observe that PA-DDPG prematurely converges to poor action policies on the simpler Platform and Robot Soccer Goal domains, which we blame on the algorithm's simultaneous optimisation of both the discrete action and action-parameter policies.

We explored how the imbalance of sampling and exploration between actions can negatively affect learning when using parameterised actions. In particular, we discussed how the action-parameter loss function used by the original P-DQN algorithm can lead to inaccurate updates and proposed our indexed action-parameter loss function as an alternative. We further proposed a weighted loss to address the imbalance of transitions per action in minibatch samples. Both individual losses and the combination of the two, which we call the weighted-indexed loss, were demonstrated to achieve higher average evaluation scores when applied to P-DQN.

The major contribution of our work, however, is the identification that the over-parameterisation of Q-values in P-DQN—that Q-values are a function of *all* action-parameters—changes the Bellman Equation for parameterised actions that Xiong *et al.* [2018] introduced. We showed that this causes two problems: false gradients for action-parameters from unrelated Q-values during updates, and that changing unrelated action-parameters can cause the discrete action policy to become suboptimal.

Two solutions were proposed for this problem: naïvely splitting the deep Q-network used by P-DQN to separate action-parameter inputs, and a novel multi-pass technique that exploits minibatch processing to separate action-parameter inputs to Q-values using parallel passes. The change in network architecture by splitting Q-networks is disadvantageous as it significantly increases the computational complexity of P-DQN and causes issues relating to the lack of a shared feature extraction layer. In particular, split Q-networks were shown to be inferior to original P-DQN on HFO. Our multi-pass technique, which does not alter the structure of the Q-network, does not suffer from these issues and substantially improved both the learning efficiency and converged performance of P-DQN across all domains. The average evaluation scores of our multi-pass P-DQN algorithm were further improved by incorporating the weighted-indexed loss function.

Overall, our work indicates that reinforcement learning algorithms should exploit the separate nature of parameterised actions, and that they should not simply be treated as continuous actions.

6.1 Future Work

There have been many recent advancements in deep reinforcement learning for continuous actions. Methods such as Soft Actor-Critic [Haarnoja *et al.* 2018] and Twin Delayed DDPG [Fujimoto *et al.* 2018] could be applied directly to parameterised actions using the collapsing strategy of Hausknecht and Stone [2016a]. However, as our results indicate, such a strategy is inferior to learning with parameterised actions directly. Thus such algorithms should be altered to account for parameterised actions, similar to how Wei *et al.* [2018] developed PATRPO by extending TRPO [Schulman *et al.* 2015] with hierarchical reinforcement learning techniques. The relation between parameterised and hierarchical action spaces, as noted by Klimek *et al.* [2017] and Wei *et al.* [2018], also warrants further exploration to see how methods for hierarchical actions can be applied to parameterised action spaces or vice-versa.

The imbalance of sampling and exploration between different actions is a problem that seems largely unaddressed hitherto, particularly in relation to parameterised action spaces. Our proposal of a weighted-indexed loss function for P-DQN, while improving performance, does not solve the problem entirely. As we mentioned in Section 4.2, enhanced replay memory sampling methods specific to parameterised actions could be explored, for example by balancing the portion of samples per action. More advanced techniques such as confidence weightings in the loss function using model-based methods could also possibly be used.

The lack of widely accepted and varied benchmark domains with parameterised action spaces have led to limited comparison against well-established prior methods in published works. Many authors instead choose to test their proposed methods only on novel domains with limited comparison to other algorithms. At the time of writing, Half Field Offense is the most commonly used domain in publications with parameterised actions [Hausknecht and Stone 2016ab; Hussein *et al.* 2018; Wei *et al.* 2018; Xiong *et al.* 2018]. However, PA-DDPG failed to learn on Platform and Robot Soccer Goal, despite performing well on HFO. This demonstrates the need for both simple toy domains and more complex tasks. Another issue is that Platform, Robot Soccer Goal, and HFO all use exactly three parameterised actions. New domains are needed that test a wider range of parameterised actions, preferably with the ability to vary the number of actions to facilitate the examination of scaling characteristics in algorithms.

Several parameterised action space domains can naturally be extended with multi-agent tasks. While Hausknecht and Stone [2016ab] train a lone soccer agent in HFO, the environment can be scaled up to include multiple agents on both offence and defence. King of Glory can also incorporate up to five players on each opposing team, although Xiong *et al.* [2018] use solo mode in their experiments. Multi-agent RL with parameterised actions would come with unique considerations, as Q-values and policies for discrete actions and continuous action-parameters could be jointly or separately learned between the agents.

Many real-world application areas stand to benefit from parameterised actions. Beyond robotic control challenges where separate actions or tasks are required, such as human-robot interaction and sports like soccer, we note that operating self-driving cars would be a natural extension. Different manoeuvres such as driving straight, turning, and reversing could be modelled as parameterised actions straightforwardly and combined to achieve different tasks such as lane following, changing lanes and parking. It has also been noted that parameterised action spaces may suit stock trading algorithms [Spooner *et al.* 2018], but they have yet to be extended to such an application. The main problems currently limiting real-world applications are inherent to deep reinforcement learning in general, where neural networks typically take hundreds of thousands to millions of samples to train, which is infeasible when dealing with real equipment.

We note that current reinforcement learning methods for parameterised action spaces, to the best of our knowledge at the time of writing, are all model-free. Data efficiency is a major issue with deep reinforcement learning and parameterised action algorithms could stand to benefit from model-based methods, hopefully reducing the number of samples required for learning from millions, as with HFO, to something manageable and able to be used in real-world applications.

Appendix A

Hyperparameters

We test each combination of hyperparameters in the grid search and select the one with the highest average area under the curve and evaluation score over 5 random runs. Adam [Kingma and Ba 2014] with $\beta_1 = 0.9, \beta_2 = 0.999$ is used to optimise the neural network parameters; this differs from Hausknecht and Stone [2016b] who use Adam with $\beta_1 = 0.95$, and Xiong *et al.* [2018] who use RMSProp [Hinton *et al.* 2012]. For P-DQN and PA-DDPG, the following hyperparameters are tested over 60 000 episodes on Platform and 50 000 episodes on Robot Soccer Goal:

Hyperparameter	Search Range
hidden neurons	$(256, 128), (128, 64), (128)$
α_Q/α_x	$10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$
τ_Q/τ_x	$0.1, 0.01, 0.001$
minibatch size	$128, 64, 32$

Table A.1: Hyperparameter search ranges.

Only combinations where $\alpha_Q \geq \alpha_x$ and $\tau_Q \geq \tau_x$ are tested since in general the actor should be updated slower than the critic. For Half Field Offense, we default to most of the same hyperparameters as Hausknecht and Stone [2016b]: $\tau_Q = \tau_x = 0.001$, sample-to-update ratio $u = 0.1$, and a replay memory size of 500 000. However we do perform a search over the actor and critic learning rates in $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$ to account for our scaling of action-parameters. We also briefly analyse the sensitivity of P-DQN and PA-DDPG to the number of hidden layers used for HFO in Appendix F.

Tables A.2 to A.4 list the final hyperparameters used for P-DQN, PA-DDPG, and Q-PAMDP in experiments on the Platform, Robot Soccer Goal, and Half Field Offense domains respectively. Unless otherwise indicated, the hyperparameters listed for P-DQN are used across all variants of the algorithm. An ϵ-greedy exploration strategy with additive Ornstein-Uhlenbeck noise for the action-parameters is used for P-DQN and PA-DDPG on the Platform and Robot Soccer Goal domains due to the action-parameters being initialised to particular values, while an ϵ-greedy with uniform-random exploration strategy is used on HFO.

Hyperparameter	**P-DQN**	**PA-DDPG**
hidden neurons	128	$256, 128$
α_Q/α_x	$10^{-3}/10^{-4}$	
τ_Q/τ_x	$0.1/0.001$	$0.01/0.01$
γ	0.9	
ϵ_{final}	0.01	
$\mu_{ou}, \theta_{ou}, \sigma_{ou}$	$0, 0.15, 0.0001$	
exploration episodes	1000	
minibatch size	128	32
gradient clip threshold	10	
initial memory threshold	500	
replay memory capacity	10 000	
activation function	ReLu	
weight initialisation	Kaiming normal	
output layer initialisation	$\mathcal{N}(0, 0.0001^2)$	

Hyperparameter	**Q-PAMDP**
κ	180
γ	0.999
initial action learning episodes	10 000
action relearn episodes	1000
eNAC rollouts	50
Fourier basis order	6
λ	0.5
SARSA(λ) α-scaling	Yes
α_{SARSA}	1.0
α_{eNAC}	0.2
normalised eNAC gradient	No
action exploration	Softmax
exploration annealing factor	0.995
action-parameter noise	$\mathcal{N}(0, 0.01^2)$

Table A.2: Hyperparameters for Platform.

Hyperparameter	**P-DQN**	**PA-DDPG**
hidden neurons	128	$128, 64$
α_Q/α_x	$10^{-3}/10^{-5}$	$10^{-4}/10^{-5}$
τ_Q/τ_x	$0.1/0.001$	$0.01/0.01$
γ	0.95	
ϵ_{final}	0.01	
$\mu_{ou}, \theta_{ou}, \sigma_{ou}$	$0, 0.15, 0.0001$	
exploration episodes	1000	
minibatch size	128	64
gradient clip threshold	1	
initial memory threshold	128	
replay memory capacity	$20\,000$	
activation function	ReLu	
weight initialisation	Kaiming normal	
output layer initialisation	$\mathcal{N}(0, 0.00001^2)$	

Hyperparameter	**Q-PAMDP**
κ	100
γ	0.9
initial action learning episodes	4000
action relearn episodes	2000
eNAC rollouts	50
Fourier basis order	7
λ	0.1
SARSA(λ) α-scaling	No
α_{SARSA}	0.01
α_{eNAC}	0.06
normalised eNAC gradient	Yes
action exploration	Softmax
exploration annealing factor	1.0
action-parameter noise	$\mathcal{N}(0, 0.01^2)$

Table A.3: Hyperparameters for Robot Soccer Goal.

Hyperparameter	**P-DQN**	**PA-DDPG**
hidden neurons	$256, 128, 64$	$1024, 512, 256, 128$
α_Q/α_x	$10^{-3}/10^{-5}$	$10^{-3}/10^{-3}$
τ_Q/τ_x	$0.001/0.001$	
β	0.25	
u	0.1	
γ	0.99	
ϵ_{final}	0.1	
exploration episodes	1000	
minibatch size	32	
gradient clip threshold	1	
initial memory threshold	1000	
replay memory capacity	$500\,000$	
activation function	leaky ReLu, negative slope 0.01	
weight initialisation	Kaiming normal for leaky ReLu	
output layer initialisation	$\mathcal{N}(0, 0.01^2)$	

Hyperparameter	**Q-PAMDP**
κ	50
γ	0.99
initial action learning episodes	1000
action relearn episodes	1000
eNAC rollouts	25
Fourier basis order	2
λ	0.5
SARSA(λ) α-scaling	Yes
α_{SARSA}	1.0
α_{eNAC}	0.2
normalised eNAC gradient	No
action exploration	ϵ-greedy
exploration annealing factor	0.996
action-parameter noise	$\mathcal{N}(0, 0.1^2)$

Table A.4: Hyperparameters for Half Field Offense.

Appendix B

Alternating Actor-Critic Updates

Actor-critic algorithms generally calculate the losses for both components then update the actor and critic together [Bhatnagar *et al.* 2009; Silver *et al.* 2014], or less commonly update the critic prior to calculating the actor loss [Lillicrap *et al.* 2015]. We refer to these different approaches as *simultaneous updates* and *alternating updates* respectively. Algorithms that use the critic from the previous timestep to update the actor effectively use simultaneous updates. While the distinction between these two approaches is disregarded in most publications, Schulman *et al.* [2016] state that they update the value function *after* the action policy—falling into the simultaneous update category since their value function update is independent of the policy—to prevent additional bias or overfitting the value function to the point where the policy gradient becomes zero. However the latter concern is an extreme case and unlikely to occur in most complex domains with the use of artificial neural networks for function approximation.

We compare using alternating and simultaneous updates for P-DQN on Platform, Robot Soccer Goal, and HFO. The results presented in Table B.1 and Figure B.1 show a negligible difference in training efficiency between the two techniques overall, but agents using alternating updates on average achieve slightly better evaluation scores across all three domains. While part of the observed difference can be attributed to random variation between runs, the consistent improvement in evaluation scores indicate alternating updates have some empirical benefit. We argue this is because the critic is updated with the most recent information from the current timestep, whereas performing simultaneous updates using the critic from the previous timestep may provide outdated or inaccurate information when calculating the actor loss. For instance, if the current update uses a previously unexplored transition that the critic has yet to generalise to, the actor's gradient—which is calculated using values from the critic—may in the worst case be completely inaccurate without updating the critic first. However the effect of this disadvantage would diminish once the critic has adequately generalised to the available samples. It is also possible that the use of target networks to calculate the critic loss mitigates the effect of any additional bias introduced by alternating updates. Based on these results, we choose alternating updates for our implementations of P-DQN in Algorithms 5 and 6.

Platform

	Mean	Median	Area Under Curve
Alternating Updates	**0.964 ± 0.068**	**0.997**	64 705
Simultaneous Updates	0.955 ± 0.060	0.985	**66 998**

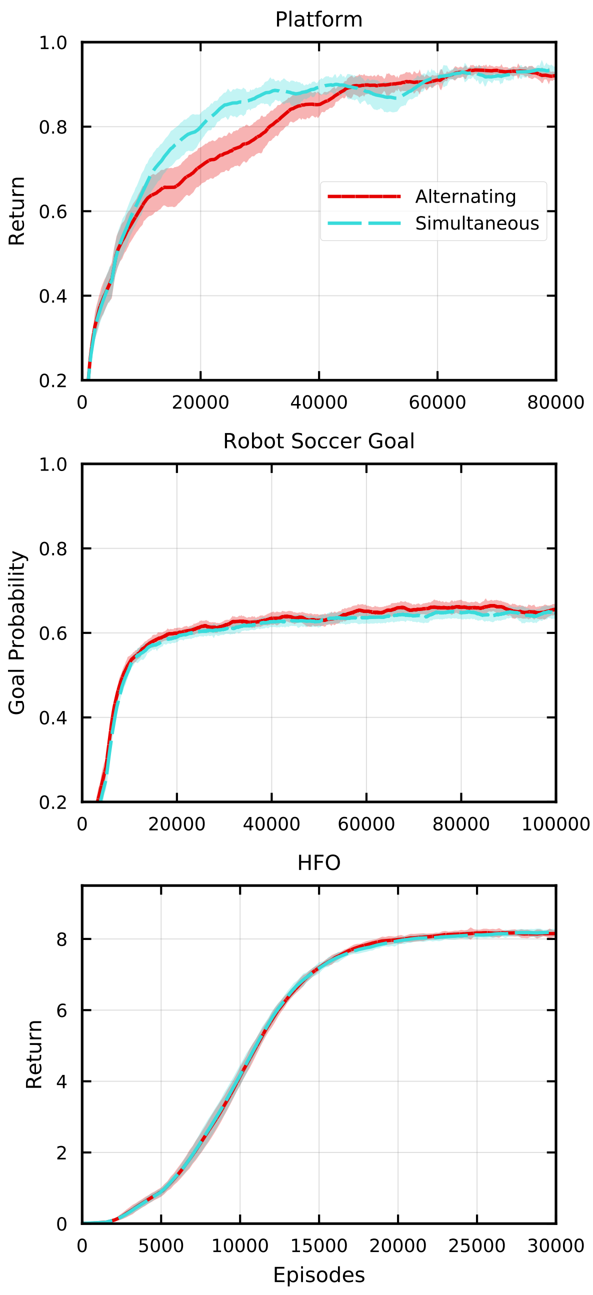
Robot Soccer Goal

	Mean	Median	Area Under Curve
Alternating Updates	**0.668 ± 0.062**	**0.656**	**61 445**
Simultaneous Updates	0.661 ± 0.071	0.646	60 350

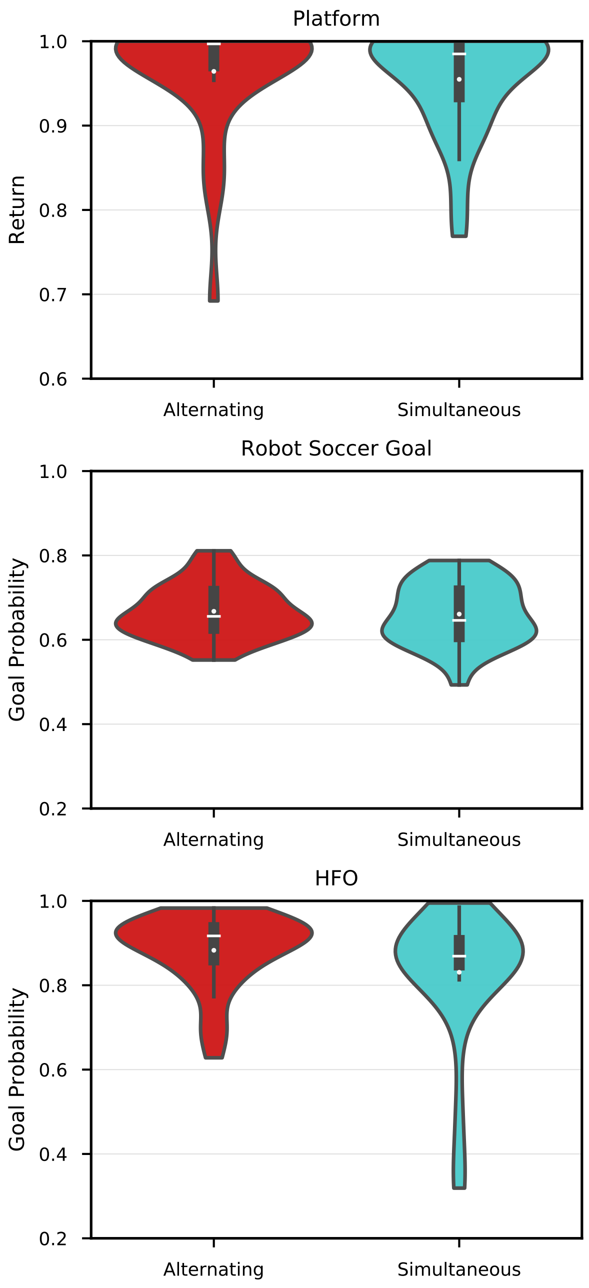
Half Field Offense

	Mean	Median	Avg. Steps to Goal	Area Under Curve
Alternating Updates	**0.883 ± 0.085**	**0.917**	**111 ± 11**	**182 573**
Simultaneous Updates	0.831 ± 0.157	0.869	114 ± 14	**182 372**

Table B.1: Results of using simultaneous versus alternating actor-critic updates for P-DQN. Alternating updates achieve the highest evaluation scores.



(a) Learning curves



(b) Evaluation scores

Figure B.1: Graphs of results using simultaneous versus alternating actor-critic updates for P-DQN over 30 seeded-random runs on each domain. From the learning curves shown in (a), simultaneous updates lead to slightly faster initial learning on Platform but there is almost no difference compared to alternating updates on Robot Soccer Goal and HFO. The average evaluation performance show in (b), on the other hand, indicates agents using alternating updates converge to policies with slightly higher scores in general over those using simultaneous updates, particularly on Platform and HFO.

Appendix C

Action-Parameter Scaling

We choose to scale the action-parameters used by all algorithms in our experiments to $[-1, 1]$ with the aim of avoiding exploding gradients [Goodfellow *et al.* 2016] and enhancing learning speed. This range was chosen over $[0, 1]$ so policies with zero-mean random initialisations start in the middle of the range, rather than at the lower extremum. Our choice matches the $\tanh$ squashing function commonly used to bound continuous outputs in neural networks, which is also zero-centred and has a range of $[-1, 1]$ by default.

We evaluate the effect of action-parameter scaling for P-DQN and Q-PAMDP on Platform, Robot Soccer Goal, and HFO. We manually tuned the eNAC learning rate for Q-PAMDP, setting it at 0.1 for Platform and 0.06 for Robot Soccer Goal compared to 1 and 0.1 used by Masson *et al.* [2016] without scaling. Altering the learning rate for the action-parameter network of P-DQN did not appear to improve performance when using unscaled action-parameters, so we present the results of using the same learning rates detailed in Appendix A for both cases. We did not test Q-PAMDP without scaling on HFO as it failed to score goals despite extensive manual tuning.

The results of our comparisons between scaled and unscaled action-parameters are shown in Table C.1, with graphs for P-DQN in Figure C.1 and Q-PAMDP in Figure C.2. Clearly, downscaling the action-parameters has a massive impact on both the learning and evaluation performance for P-DQN. Without scaling, the algorithm on average learns slower and results in inferior policies across all three domains. The same was found to be true for PA-DDPG, although we did not test to the same extent as P-DQN and thus omit its results. The performance of Q-PAMDP is similarly improved when using scaled action-parameters on Platform, where it learns significantly faster compared to the unscaled version initially used by Masson *et al.* [2016]. The learning speed on Robot Soccer Goal is similar with and without scaling, with a minor increase in evaluation performance.

Overall, our results indicate that action-parameter scaling is beneficial and important for neural network based approaches such as P-DQN to learn efficiently. The faster learning is likely because scaling allows gradients to have a larger impact on the policy with fewer updates. It also complements the gradient clipping strategy in avoiding problems relating to the scale of input variables such as exploding gradients, which can cause divergence during learning.

Platform

Algorithm	Scaled	Mean	Median	Area Under Curve
P-DQN	Yes	**0.964 ± 0.069**	**0.997**	**64 705**
	No	0.385 ± 0.176	0.356	15 270
Q-PAMDP	Yes	**0.789 ± 0.188**	**0.795**	**50 397**
	No	0.608 ± 0.175	0.552	39 520

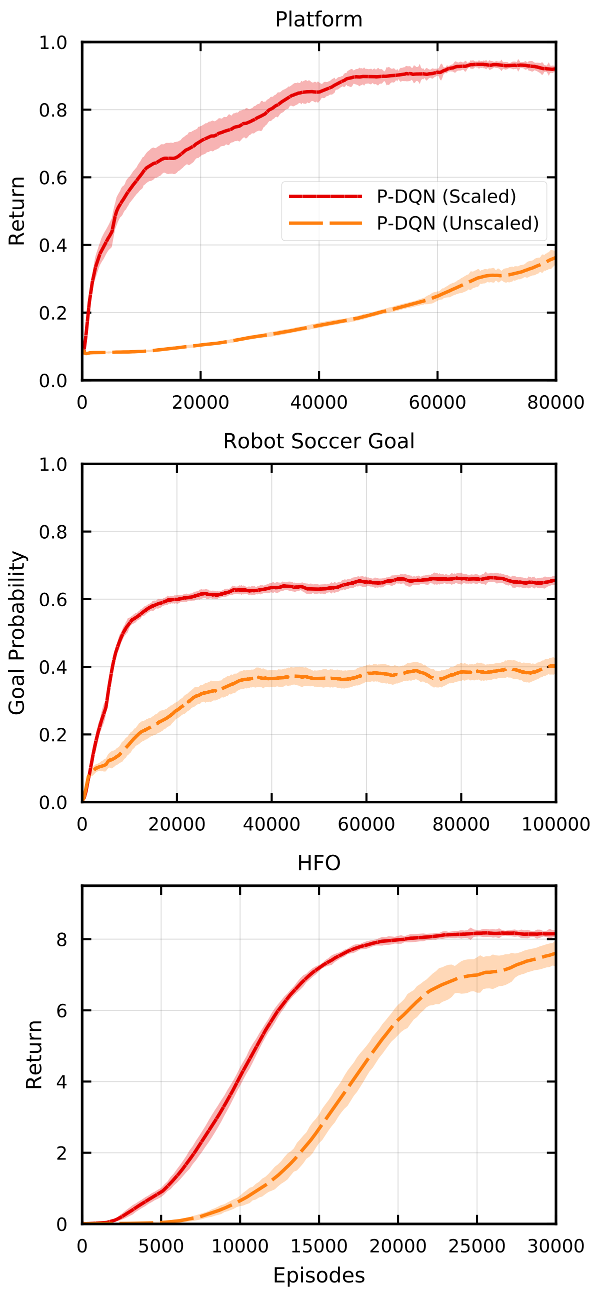
Robot Soccer Goal

Algorithm	Scaled	Mean	Median	Area Under Curve
P-DQN	Yes	**0.668 ± 0.062**	**0.656**	**61 445**
	No	0.399 ± 0.124	0.417	33 599
Q-PAMDP	Yes	**0.452 ± 0.093**	**0.456**	**39 340**
	No	0.428 ± 0.084	0.445	38 724

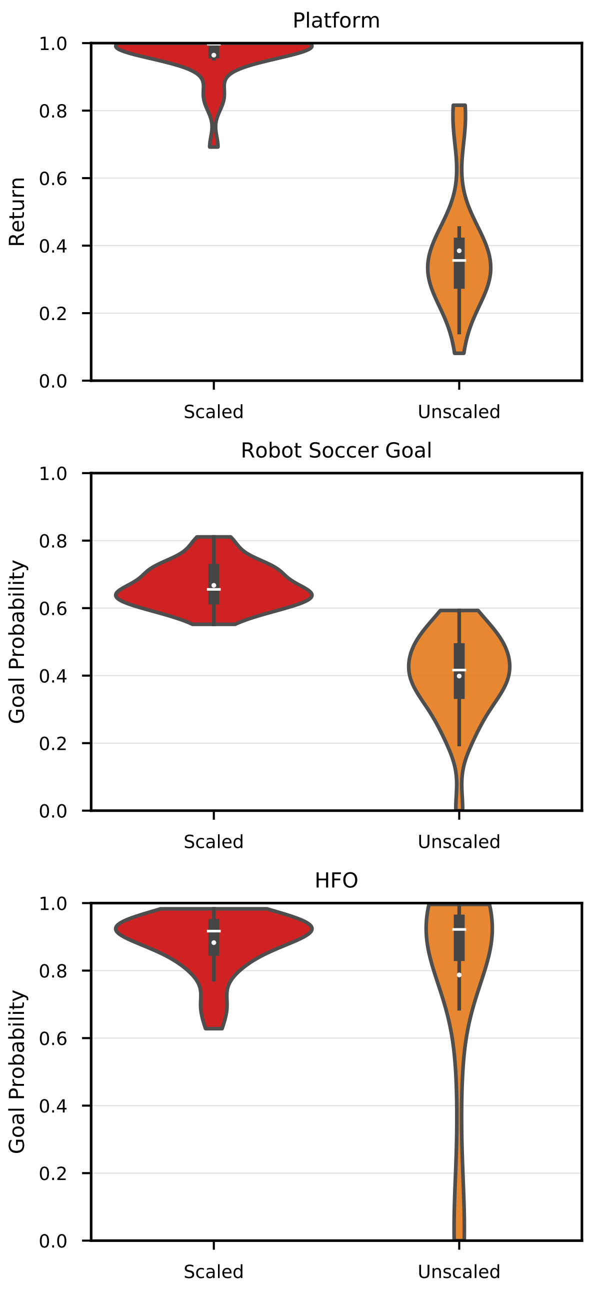
Half Field Offense

Algorithm	Scaled	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN	Yes	**0.883 ± 0.085**	0.917	**111 ± 11**	**182 573**
	No	0.787 ± 0.309	**0.922**	119 ± 23	118 448

Table C.1: Results of using scaled versus unscaled action-parameters for P-DQN and Q-PAMDP on Platform, Robot Soccer Goal, and Half Field Offense. For both algorithms, the agents using action-parameters rescaled to $[-1, 1]$ perform better on all domains.

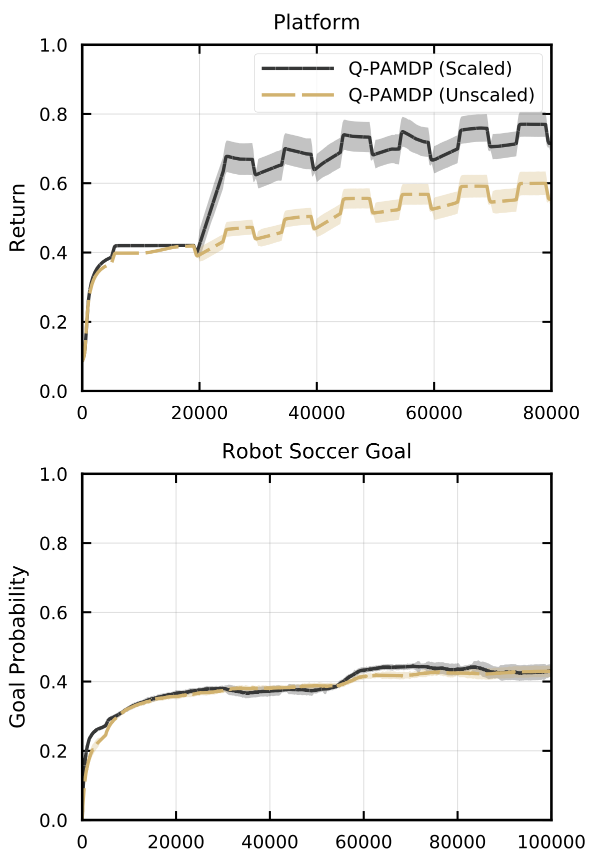


(a) Learning curves

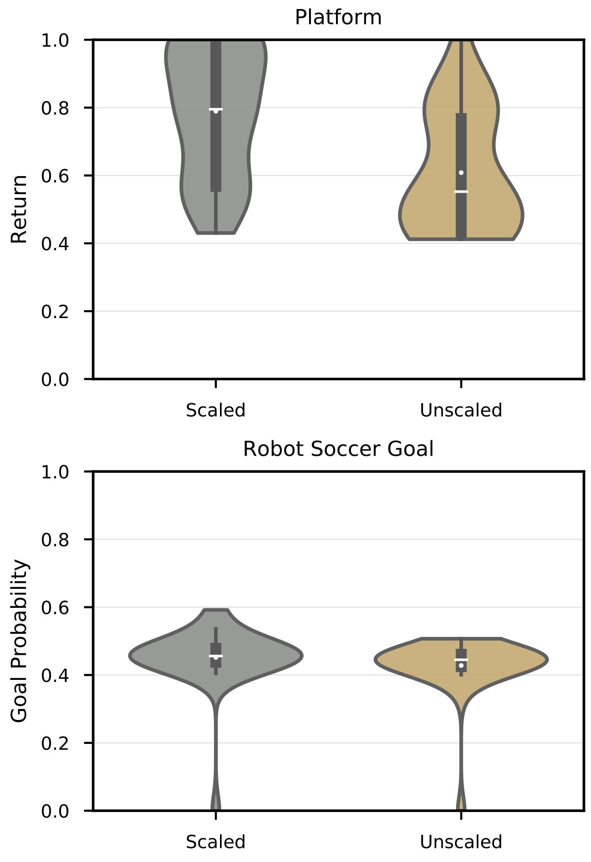


(b) Evaluation scores

Figure C.1: Graphs of the learning curves and evaluation scores of P-DQN with unscaled action-parameters versus scaling to the range $[-1, 1]$.



(a) Learning curves



(b) Evaluation scores

Figure C.2: Graphs of the learning curves and evaluation scores of Q-PAMDP with unscaled action-parameters versus scaling to the range $[-1, 1]$.

Appendix D

Initial Action-Parameter Policies for Platform and Robot Soccer Goal

Masson *et al.* [2016] initialise the action-parameter policy for Q-PAMDP on their Platform and Robot Soccer Goal domains to make it easier to learn rather than starting from scratch. The following information is from the source code generously released by Masson *et al.* [2016].[1][2] The initial policy used for Platform is simple, with each action-parameter being initialised to a constant:

$$\begin{aligned} x_{\text{run}} &= 3, \\ x_{\text{hop}} &= 10, \\ x_{\text{leap}} &= 400. \end{aligned}$$

This initialisation causes the leap distance to start close to the maximum but still be less than the gap between platforms. The hop and run distances are initialised just above the minimum of 0 so the agent has some forward motion to start with, otherwise those actions would leave the agent at the same position. For Robot Soccer Goal, a combination of state features is used for the initial action-parameter policy:

$$\begin{aligned} x_{\text{kick-to}} &= \begin{pmatrix} 2.5 + x_b + \left(\frac{50}{\text{PL}}\right)\frac{(x_b - x_k)}{\|\mathbf{p}_b - \mathbf{p}_k\|} \\ 1 + \left(1 - \frac{5}{\text{PW}}\right) y_b + \left(\frac{50}{\text{PW}}\right)\frac{(y_b - y_k)}{\|\mathbf{p}_b - \mathbf{p}_k\|} \end{pmatrix}, \\ x_{\text{shoot-goal-left}} &= \frac{\text{GW}}{2} - 1, \\ x_{\text{shoot-goal-right}} &= -\frac{\text{GW}}{2} + 1, \end{aligned}$$

where PL $=$ 40 is the pitch length, PW $=$ 30 is the pitch width, and GW $=$ 14.02 is the goal width. The normalised distance between the keeper and ball along each axis, ${(x_b - x_k)}/{\|\mathbf{p}_b - \mathbf{p}_k\|}$ and ${(y_b - y_k)}/{\|\mathbf{p}_b - \mathbf{p}_k\|}$, is added to the feature space for all methods, where $\mathbf{p}_b = (x_b, y_b)$ and $\mathbf{p}_k = (x_k, y_k)$ are the positions of the ball and keeper respectively. The rest of the features are explained in Section 2.3.2. Since $x_{\text{kick-to}}$ dictates a position to kick the ball towards, it is represented as a 2-dimensional vector giving the x and y position. This policy causes the kick-to action to start off always kicking the ball away from the keeper towards the bottom of the field. The shoot-goal-left and shoot-goal-right actions kick the ball towards the edges of the goals, rather than the centre. Learning is far more difficult without this initialisation as the kick-to action would start off always kicking the ball to the centre of the pitch, and

[1]`https://github.com/WarwickMasson/aaai-platformer`
[2]`https://github.com/WarwickMasson/aaai-goal`

the shoot-goal actions would kick towards the centre of the goals, almost always resulting in the keeper catching the ball.

D.1 Passthrough Layer

One can mimic this initialisation in P-DQN and PA-DDPG by adding the linear policy to the output of the actor network externally. However, action-parameter bounding techniques require this to be done during the internal update step too. We present a convenient way of incorporating policy initialisation into the architecture of actor networks by adding a fully-connected *passthrough layer* to the network

$$\mathbf{W}s + \mathbf{b},$$

where **W** are the layer weights and **b** is the bias term. This layer uses only the state features as input and is added directly to the policy output, as illustrated in Figure D.1. Assuming the weights of the original output layer are set close to zero, this allows any linear combination of state variables to be used as an initial policy. The benefit of this method is that action boundaries are respected and include the initial policy, as bounding methods are applied after the addition step on the output of the whole network. Note that this only works with inverting gradients or similar bounding techniques where no activation function, or one that is effectively the identity function within the valid range, is used on the output. Non-linear activation functions such as $\tanh$ would skew the linear combination function.

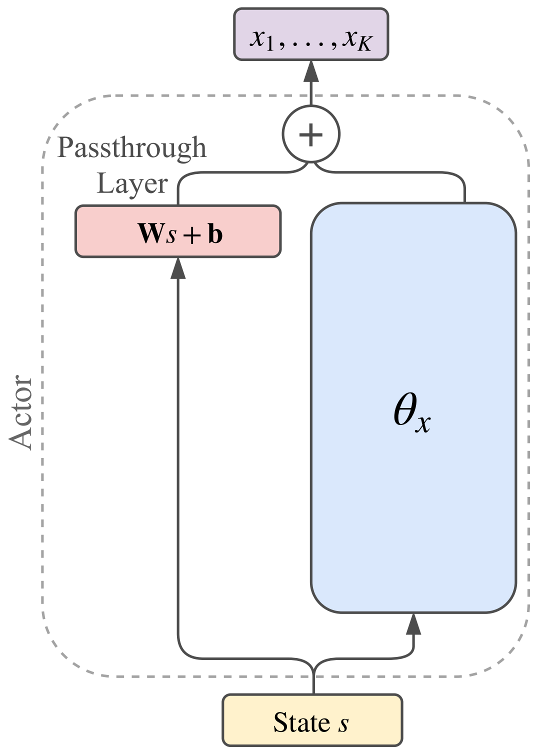


Figure D.1: Illustration of an actor network with the proposed passthrough layer

Since the passthrough layer is a direct linear combination of the input variables, performing updates on the passthrough layer weights using the same learning rate as the rest of the neural network, which typically has multiple hidden layers, can have a greater impact on the policy. This can lead to oscillation or divergence during learning. While one could use a separate learning rate for the passthrough layer, we instead choose to keep the passthrough layer weights static. This avoids introducing another hyperparameter and any problems relating to updating the passthrough layer. Note that this does not reduce the robustness of the action policy as the rest of the network simply learns to compensate, so the full action range is still available.

We perform experiments on Platform and Robot Soccer Goal to compare the performance of P-DQN with initialised action-parameters using a static passthrough layer versus a dynamic passthrough layer that is updated with the rest of the network. Results over 30 seeded-random runs are presented in Table D.1 and Figure D.2, which show that sample efficiency is largely unaffected but on average more agents with a dynamic passthrough layer fail to solve the Platform domain by the end of training. No agents appear to diverge outright during learning; however, this is likely because gradient clipping is employed, which limits large changes to the policy and keeps the scale of parameters in check. A slight increase in the average agent evaluation score is noted on Robot Soccer Goal when using dynamic passthrough layers but we argue the negative effects observed on the Platform domain outweigh this. Thus we use a static passthrough layer throughout our experiments.

Platform

Passthrough Layer	Mean	Median	Area Under Curve
Static	**0.964 ± 0.068**	**0.997**	64 705
Dynamic	0.935 ± 0.121	0.988	**66 020**

Robot Soccer Goal

Passthrough Layer	Mean	Median	Area Under Curve
Static	0.668 ± 0.062	0.656	**61 445**
Dynamic	**0.695 ± 0.071**	**0.714**	**61 408**

Table D.1: Results of using a static versus a dynamic passthrough layer for P-DQN on Platform and Robot Soccer Goal.

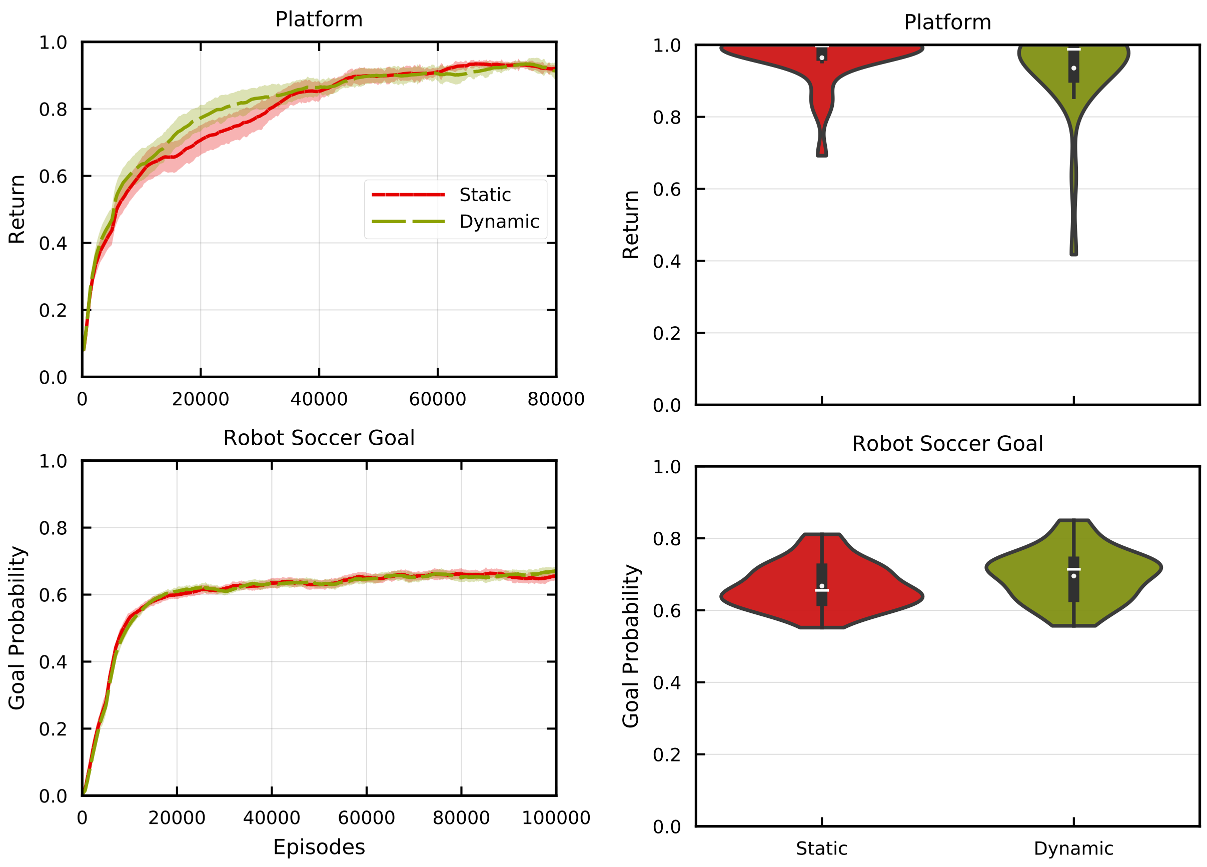


Figure D.2: Results of using a static versus a dynamic passthrough layer for P-DQN on Platform and Robot Soccer Goal. There is little difference between the learning curves of the two approaches in (a).

D.2 Comparison to Uninitialised Policies

To examine how much of an impact the initial policies used by Masson *et al.* [2016] have, we test the performance of P-DQN and Q-PAMDP with and without action-parameter initialisation on Platform and Robot Soccer Goal. We scale the action-parameters to $[-1, 1]$ for P-DQN but keep them unscaled for Q-PAMDP in this experiment, as the original algorithm did not use scaling. This means that the uninitialised, or rather zero-initialised, P-DQN agents start with policies in the middle of the range while uninitialised Q-PAMDP agents start at the minimum. For the uninitialised version of P-DQN, we use ϵ-greedy with uniform-random action-parameter exploration. This was chosen over the Ornstein-Uhlenbeck noise used for the scaled version since more exploration is required with an uninitialised policy. We present the results in Table D.2, with graphs for P-DQN in Figure D.3 and for Q-PAMDP in Figure D.4.

We observe that P-DQN without initialisation quickly learns to solve Platform within $10\,000$ episodes. This may be due to the agent requiring fewer updates to increase the run action-parameter since it starts in the middle of the range, giving it enough forward momentum to traverse over gaps, although the leap action-parameter starts off lower. On Robot Soccer Goal, however, the average performance is significantly reduced. Uninitialised agents take much longer to first score goals and on average scoring only 35% of the time by the end of training, compared to the 66% average with an initial policy. Q-PAMDP agents without action-parameter initialisation also fail to solve Robot Soccer Goal and show little improvement over the course of training. On Platform, uninitialised Q-PAMDP agents take longer to learn than the initialised agents. This is because we do not use scaling for Q-PAMDP in this experiment, so agents start off barely moving forward at all.

Platform

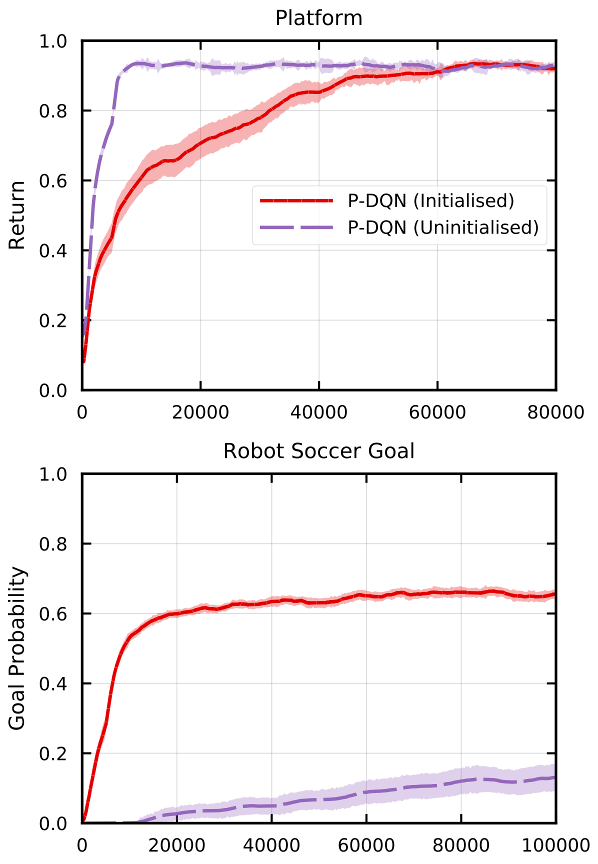
Algorithm	Initialised	Mean	Median	Area Under Curve
P-DQN	Yes	**0.964 ± 0.069**	**0.997**	64 705
	No	**0.953 ± 0.137**	**0.994**	**73 392**
Q-PAMDP	Yes	**0.608 ± 0.175**	**0.552**	**39 520**
	No	0.356 ± 0.252	0.273	16 641

Robot Soccer Goal

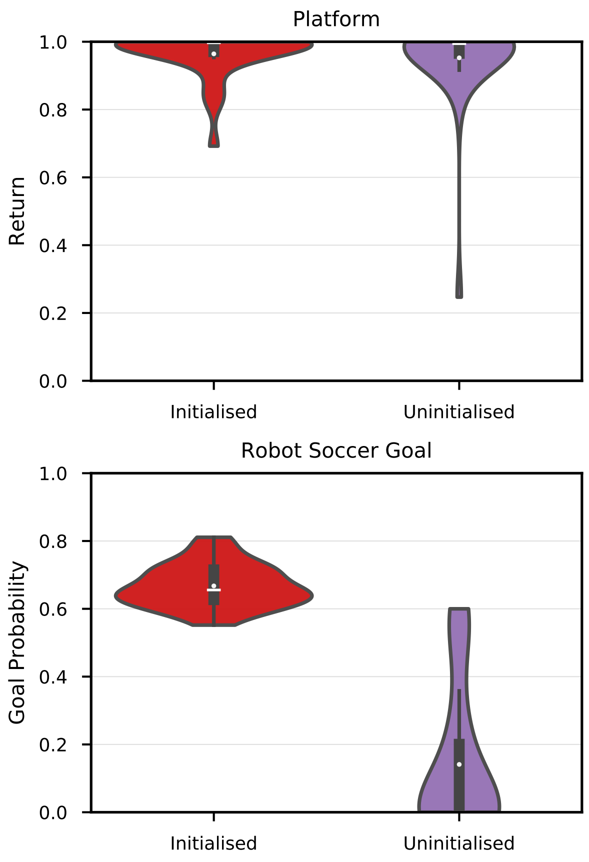
Algorithm	Initialised	Mean	Median	Area Under Curve
P-DQN	Yes	**0.668 ± 0.062**	**0.656**	**61 445**
	No	0.141 ± 0.211	0.000	7145
Q-PAMDP	Yes	**0.428 ± 0.084**	**0.445**	**38 724**
	No	0.011 ± 0.037	0.000	509

Table D.2: Results of uninitialised (zero-initialised) action-parameters versus using initial policies matching that of Masson *et al.* [2016] on Platform and Robot Soccer Goal for P-DQN and Q-PAMDP.

Given the massive impact on learning performance these initial action-parameter policies have, authors using Platform and Robot Soccer Goal as benchmark domains should explicitly state whether or not they use the initialisation strategy from Masson *et al.* [2016] and how their action-parameters are normalised.

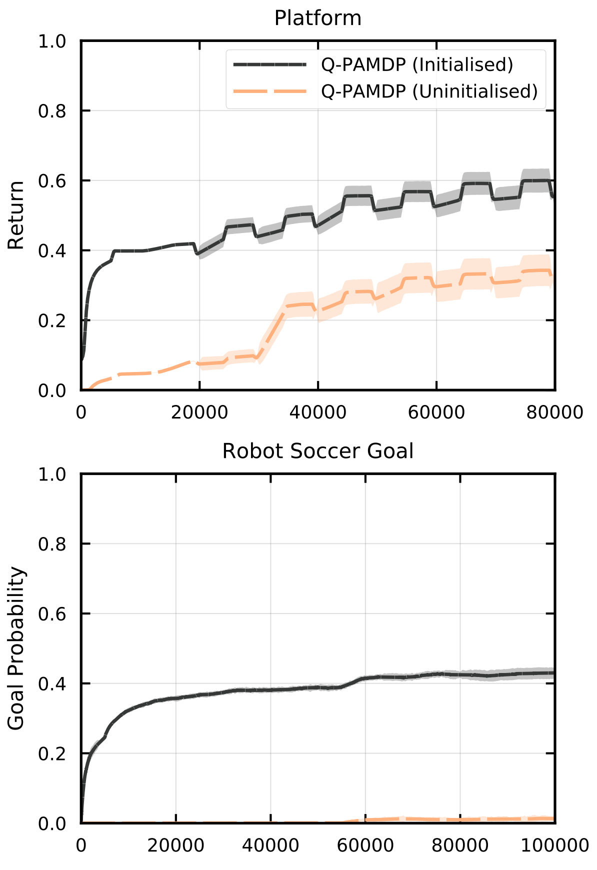


(a) Learning curves

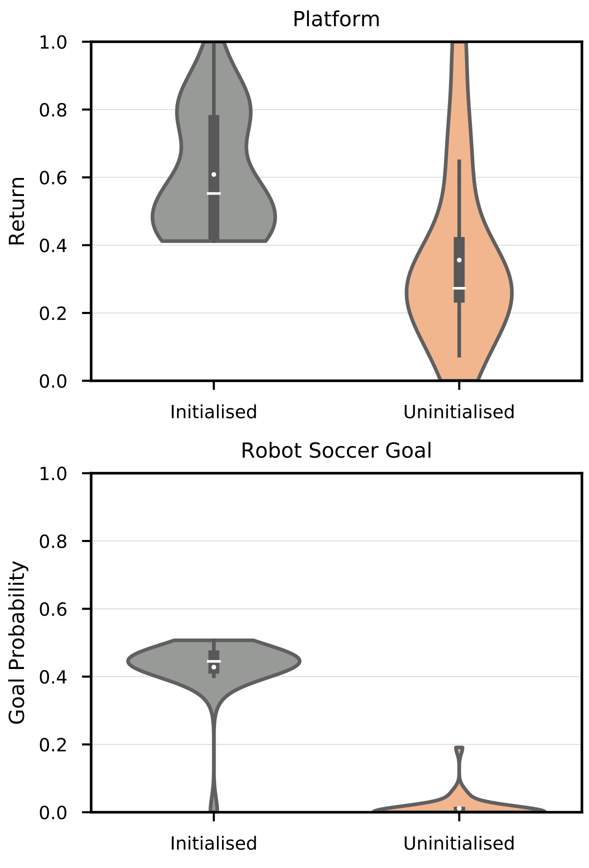


(b) Evaluation scores

Figure D.3: Graphs of the learning curves and evaluation scores of P-DQN with uninitialised (zero-initialised) action-parameters versus using initial policies matching that of Masson *et al.* [2016] on Platform and Robot Soccer Goal.



(a) Learning curves



(b) Evaluation scores

Figure D.4: Graphs of the learning curves and evaluation scores of Q-PAMDP with uninitialised (zero-initialised) action-parameters versus using initial policies matching that of Masson *et al.* [2016] on Platform and Robot Soccer Goal.

Appendix E

Effects of β Parameter for Mixed n-Step Targets

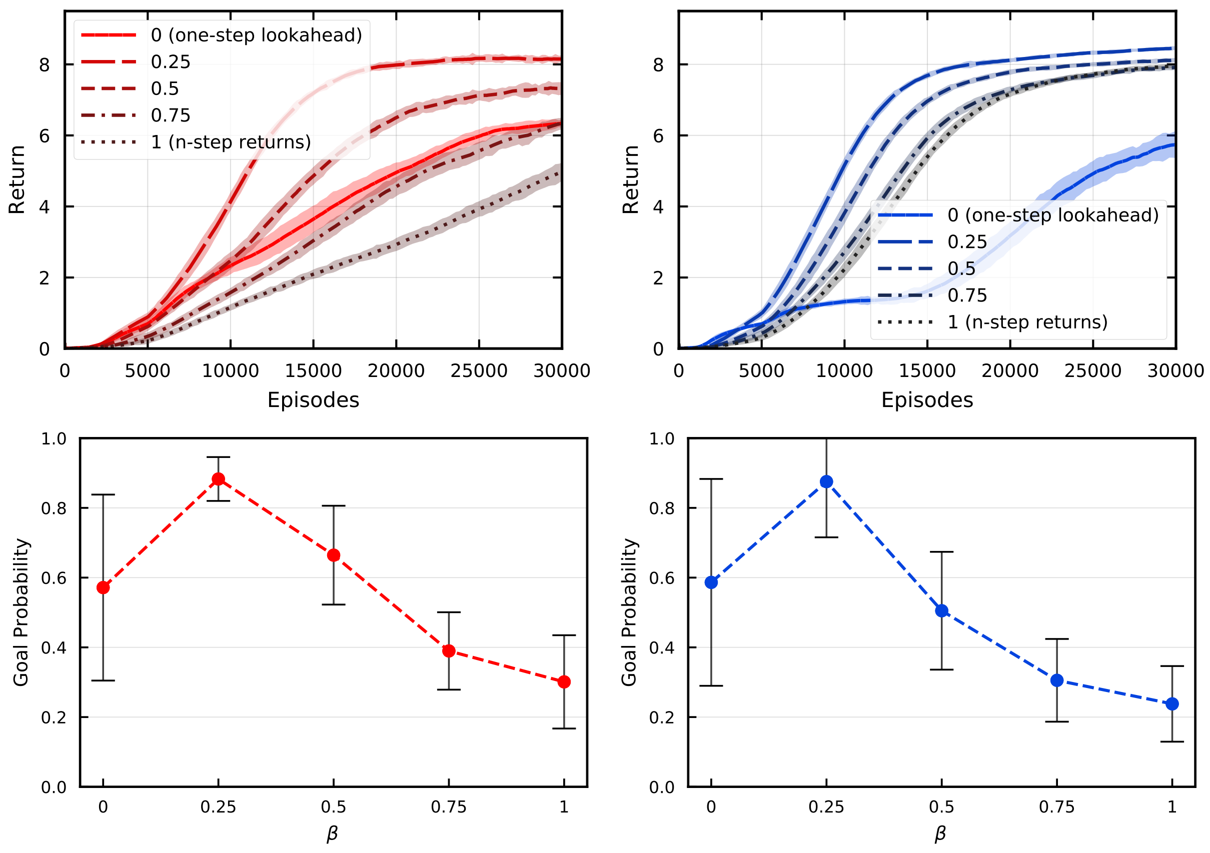
Hausknecht and Stone [2016b] successfully employ mixed n-step return update targets to improve the performance of PA-DDPG on HFO, and of DQN on certain Atari games but decreasing performance on others, suggesting that the effect of mixing n-step returns may be highly domain and algorithm dependent. There is no definitive method on how to choose β, however the authors show that smaller values—roughly below 0.5 for both DQN and PA-DDPG—appear to perform better.

We conduct our own reduced grid search on the β parameter over $\{0, 0.25, 0.5, 0.75, 1\}$ to examine the effects of mixing n-step returns on P-DQN and PA-DDPG on HFO. For each different value, we train 30 independent, randomly-seeded agents over 30 000 episodes. The results in Figure E.1a show that P-DQN is much more sensitive to the value of β than PA-DDPG. While there is a significant increase in training performance from 0 to 0.25, it decreases dramatically as β increases further, similarly with the evaluation performance in Figure E.1a. A similar correlation between training performance and β is observed for PA-DDPG but to a lesser extent. Surprisingly, the evaluation scores for PA-DDPG show a much greater decrease as β increases, contrary to the learning curves which indicate a more minor decrease in learning speed and average episodic return. This can be explained primarily by episodic return not being linearly correlated with goal-scoring probability on HFO, as agents can theoretically achieve a return of up to 6 (out of a maximum of 11) by kicking the ball very close to the goal. The removal of exploration noise during evaluation may also be a factor, leading to worse deterministic action policies than during training.

A β value of 0.25 performs best for both P-DQN and PA-DDPG, although it may not be optimal given the limited scope of our search. P-DQN being detrimentally sensitive to high values of β is consistent with the findings of Hausknecht and Stone [2016b] for regular DQN, although our results for PA-DDPG appear lower than theirs due to a larger sample size of agents being evaluated for each value in our experiments—30 as opposed to 1. Clearly the value of β for mixing n-step returns has a major impact on performance for both algorithms, so picking the right value for the task is important.

Algorithm	β	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN	0	0.571 ± 0.289	0.639	140 ± 34	119 724
	0.25	$\mathbf{0.883 \pm 0.085}$	**0.917**	$\mathbf{111 \pm 11}$	**182 573**
	0.50	0.664 ± 0.164	0.689	131 ± 12	144 473
	0.75	0.390 ± 0.133	0.401	145 ± 18	107 084
	1	0.301 ± 0.156	0.292	151 ± 16	76 525
PA-DDPG	0	0.586 ± 0.319	0.697	127 ± 32	87 350
	0.25	$\mathbf{0.875 \pm 0.182}$	**0.945**	$\mathbf{95 \pm 7}$	**193 687**
	0.50	0.505 ± 0.191	0.471	109 ± 6	176 491
	0.75	0.305 ± 0.141	0.306	120 ± 12	159 715
	1	0.238 ± 0.131	0.227	122 ± 10	153 952

Table E.1: Results of P-DQN and PA-DDPG trained using different values of β on HFO.



(a) P-DQN

(b) PA-DDPG

Figure E.1: Learning curves (top) and trend lines (bottom) showing the mean agent evaluation scores of P-DQN and PA-DDPG over different values of β on HFO. The error bars indicate one standard deviation above and below the mean average score.

Appendix F

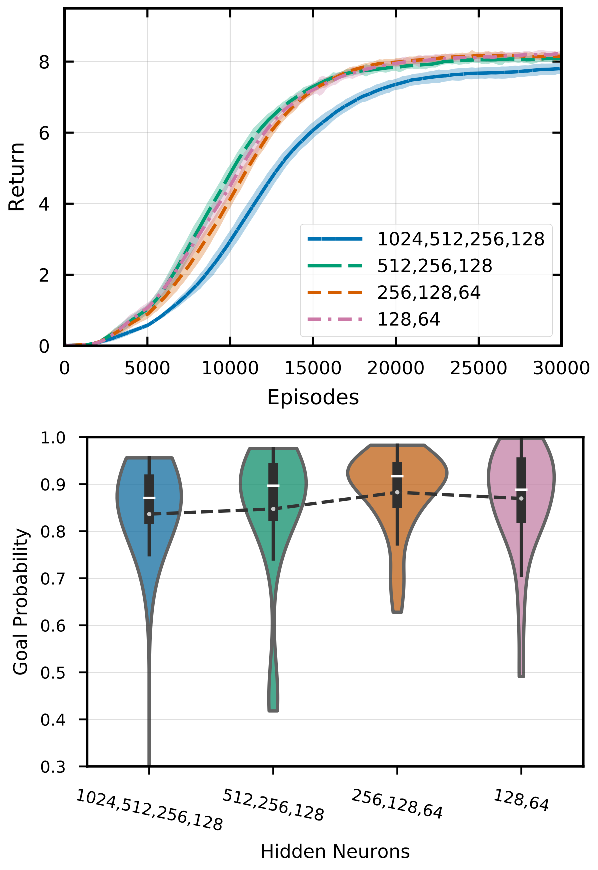
Network Size Sensitivity

Hausknecht and Stone [2016a] report using a relatively large and deep network architecture to train PA-DDPG with parameterised actions on HFO, with the actor and critic networks each consisting of four hidden layers $(1024, 512, 256, 128)$. Xiong *et al.* [2018], on the other hand, use a much smaller network consisting of three hidden layers $(256, 128, 64)$ for P-DQN. To investigate how sensitive each algorithm is to the number of hidden layers and neurons, we use P-DQN and PA-DDPG to train agents on HFO over four network configurations: $(1024, 512, 256, 128)$, $(512, 256, 128)$, $(256, 128, 64)$, and $(128, 64)$. We keep the actor and critic network architectures symmetric as in prior literature, and use the same hyper-parameters across all network sizes per algorithm due to time and resource constraints. So while there is room for fine-tuning, particularly of the learning rates, the results shown in Table F.1 and Figure F.1 still depict interesting trends.

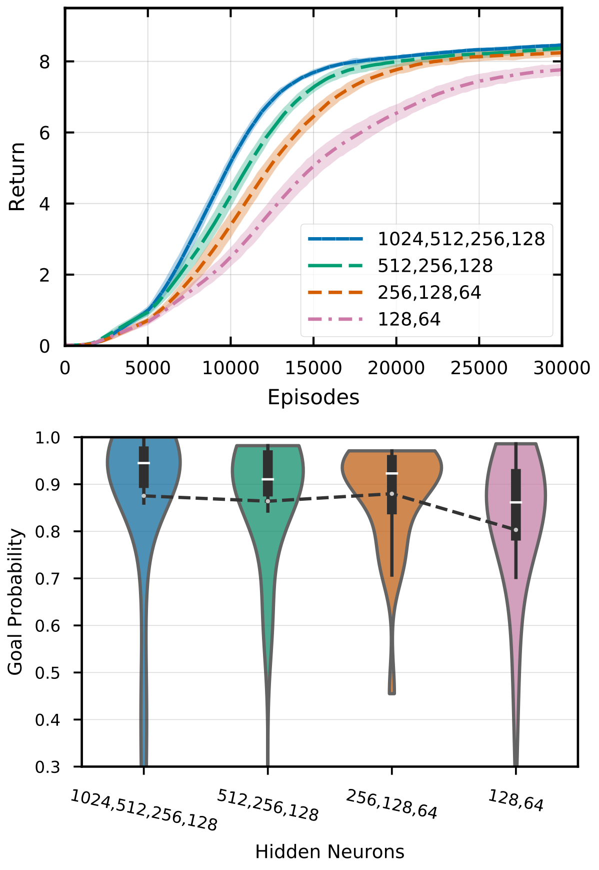
The performance of P-DQN appears to improve as the network size decreases, having the greatest area under the curve when using just two hidden layers $(128, 64)$. However, three hidden layers $(256, 128, 64)$, as used by Xiong *et al.* [2018], produces the highest mean and median evaluation scores for P-DQN. The opposite is true for PA-DDPG, where agents with four hidden layers $(1024, 512, 256, 128)$, as used by Hausknecht and Stone [2016a], have the highest training and evaluation scores while the algorithm suffers a significant decrease in evaluation scores with smaller networks. PA-DDPG appears somewhat more sensitive to the number of hidden layers and neurons than P-DQN, which is evident from Figure F.1 and the difference in mean evaluation score between the best and worst performing network sizes—0.09 for PA-DDPG versus 0.046 for P-DQN. This could be explained due to PA-DDPG using all actions and their action-parameters as input, thus requiring more representational power to learn the relationships between a greater number of variables than P-DQN. Further investigation is needed for conclusive results, however.

Algorithm	Hidden Layers	Mean	Median	Avg. Steps to Goal	Area Under Curve
P-DQN	1024, 512, 256, 128	0.837 ± 0.165	0.871	120 ± 14	162 119
	512, 256, 128	0.848 ± 0.145	0.897	116 ± 18	185 749
	256, 128, 64	**0.883 ± 0.085**	**0.917**	**111 ± 11**	182 573
	128, 64	0.870 ± 0.112	0.889	119 ± 14	**185 836**
PA-DDPG	1024, 512, 256, 128	**0.875 ± 0.181**	**0.945**	**95 ± 7**	**188 782**
	512, 256, 128	0.864 ± 0.181	0.911	100 ± 7	185 072
	256, 128, 64	**0.880 ± 0.111**	0.923	101 ± 5	173 571
	128, 64	0.803 ± 0.198	0.862	111 ± 11	149 836

Table F.1: Results of P-DQN and PA-DDPG trained using different network sizes on HFO.



(a) P-DQN



(b) PA-DDPG

Figure F.1: Learning curves (top) and violin plots of mean evaluation scores (bottom) for P-DQN and PA-DDPG using decreasing network sizes to train on HFO.

References

[Agarwal 2018] Arpit Agarwal. *Deep Reinforcement Learning with Skill Library: Exploring with Temporal Abstractions and coarse approximate Dynamics Models*. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, July 2018.

[Bertsekas 1987] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Bhatnagar *et al.* 2009] Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471 – 2482, 2009.

[Brockman *et al.* 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *arXiv preprint arXiv:1606.01540*, 2016.

[Deisenroth *et al.* 2013] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142, 2013.

[Fujimoto *et al.* 2018] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1582–1591, 2018.

[Glorot *et al.* 2011] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[Goodfellow *et al.* 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[Haarnoja *et al.* 2018] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1856–1865, 2018.

[Hausknecht and Stone 2016a] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.

[Hausknecht and Stone 2016b] Matthew Hausknecht and Peter Stone. On-policy vs. off-policy updates for deep reinforcement learning. In *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI Workshop*, July 2016.

[He *et al.* 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[Hinton *et al.* 2012] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6a: overview of mini-batch gradient descent. *COURSERA: Neural Networks for Machine Learning*, 2012.

[Hussein *et al.* 2018] Ahmed Hussein, Eyad Elyan, and Chrisina Jayne. Deep imitation learning with memory for robocup soccer simulation. In *Engineering Applications of Neural Networks*, pages 31–43, Cham, 2018. Springer International Publishing.

[Kingma and Ba 2014] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, abs/1412.6980, 2014.

[Kitano *et al.* 1997] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. RoboCup: A challenge problem for AI. *AI magazine*, 18(1):73, 1997.

[Klimek *et al.* 2017] Maciej Klimek, Henryk Michalewski, and Piotr Miłoś. Hierarchical reinforcement learning with parameters. In *Conference on Robot Learning*, pages 301–313, 2017.

[Konidaris *et al.* 2011] George D. Konidaris, Sarah Osentoski, and Philip S. Thomas. Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*, pages 380–385, August 2011.

[Krizhevsky *et al.* 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[Levine *et al.* 2016] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. (1):1334–1373, January 2016.

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

[Masson *et al.* 2016] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1934–1940. AAAI Press, 2016.

[Medsker and Jain 1999] Larry R. Medsker and Lakhmi C. Jain. *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.

[Mnih *et al.* 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, abs/1312.5602, 2013.

[Mnih *et al.* 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[Mnih *et al.* 2016] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 1928–1937, New York, New York, USA, June 2016. PMLR.

[Parr *et al.* 2008] Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, pages 752–759, New York, NY, USA, 2008. ACM.

[Paszke *et al.* 2017] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, 2017.

[Peng and Williams 1996] Jing Peng and Ronald J. Williams. Incremental multi-step Q-learning. *Machine Learning*, 22(1):283–290, March 1996.

[Peters *et al.* 2005] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. *Natural Actor-Critic*, pages 280–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Schaul *et al.* 2015] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, abs/1511.05952, 2015.

[Schulman *et al.* 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, pages 1889–1897. JMLR.org, 2015.

[Schulman *et al.* 2016] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations*, 2016.

[Selvaraju *et al.* 2017] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 618–626, 2017.

[Silver *et al.* 2014] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, number 1, pages 387–395, Bejing, China, June 2014. PMLR.

[Silver *et al.* 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[Spooner *et al.* 2018] Thomas Spooner, John Fearnley, Rahul Savani, and Andreas Koukorinis. Market making via reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 434–442, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[Sutton and Barto 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[Sutton *et al.* 1999] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of* *the 12th International Conference on Neural Information Processing Systems*, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[Todorov *et al.* 2012] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[Tukey 1977] John W Tukey. *Exploratory data analysis*. Addison-Wesley, 1977.

[Uhlenbeck and Ornstein 1930] George E. Uhlenbeck and Leonard S. Ornstein. On the theory of the Brownian motion. *Physical Review*, 36:823–841, September 1930.

[Wang *et al.* 2016] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pages 1995–2003, 2016.

[Watkins and Dayan 1992] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[Watkins 1989] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[Wei *et al.* 2018] Ermo Wei, Drew Wicke, and Sean Luke. Hierarchical approaches for reinforcement learning in parameterized action space. In *AAAI Spring Symposium Series*, 2018.

[White and White 2010] Martha White and Adam White. Interval estimation for reinforcement-learning algorithms in continuous-state domains. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, volume 2, pages 2433–2441, USA, 2010. Curran Associates Inc.

[Xiong *et al.* 2018] Jiechao Xiong, Qing Wang, Zhuoran Yang, Peng Sun, Lei Han, Yang Zheng, Haobo Fu, Tong Zhang, Ji Liu, and Han Liu. Parametrized deep Q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. *arXiv preprint arXiv:1810.06394*, 2018.