Senior Thesis - Extending Classical Deep Reinforcement Learning Techniques for use in Multi-Agent Settings

Ollie Matthews

May 20, 2020

Peter Ramadge Yuxin Chen

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering Department of Electrical Engineering Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

Ollie Matthews

Acknowledgements

I thank Professor Ramadge for agreeing to supervise and mark my project and for taking time every two weeks to meet with me to discuss ideas. I also thank Jonathan Spencer for putting up with my questions twice as frequently, and for keeping me up to date on all of the fascinating things happening in reinforcement learning. Finally, I thank Professor Chen for agreeing to be my second reader and Jean Bausmith for organising the whole course.

Abstract

Multi-agent reinforcement learning is an attractive and challenging prospect. Multi-agent settings are non-stationary by nature, and multi-agent contributions to reward introduce ambiguity not present in single-agent learning. In this paper, we start by looking at Policy Gradient based reinforcement learning techniques in a single-agent setting. We then show why these techniques fail in a multi-agent setting and suggest improvements to the algorithms to address these problems. Experiments are done on simulations of cyclists, with the aim of agents learning to work together to cycle efficiently as a pack. The improvements suggested include: adding a varying normaliser to deal with state distribution drift; using a value function which takes as input the actions of other agents to deal with non-stationarity; and an automatic tuning algorithm for learning hyperparameters. We find that our techniques show improvements on the traditional algorithms, in particular when more agents are added. We suggest improvements on our algorithm, and discuss other directions for future work.

Contents

1 Introduction **1**
- 1.1 Single-Agent Reinforcement Learning . . . 1
  - 1.1.1 The Reinforcement Learning Problem . . . 1
  - 1.1.2 Agent . . . 2
  - 1.1.3 Environment . . . 2
  - 1.1.4 Value Functions . . . 3
  - 1.1.5 Challenges in the Single-Agent Setting . . . 4
- 1.2 Multi-Agent Reinforcement Learning . . . 4
  - 1.2.1 Optimal Multi-Agent Policies . . . 4
  - 1.2.2 Challenges in a Multi-Agent Setting . . . 5

2 Classical Methods **7**
- 2.1 Policy Gradients . . . 7
- 2.2 Implementing Policy Gradients . . . 8
- 2.3 Normalising Rewards . . . 10
- 2.4 Actor Critic . . . 11
- 2.5 Exploration . . . 12
  - 2.5.1 Measuring Exploration . . . 12
  - 2.5.2 Entropy Reward . . . 12
- 2.6 Other Methods . . . 15
  - 2.6.1 Q-Learning . . . 15
  - 2.6.2 Trust Region Policy Optimization (TRPO) . . . 15
  - 2.6.3 Other Variants on Policy Gradients . . . 15

3 Testing Classical Techniques on a Single Agent Environment **17**
- 3.1 The Environment . . . 17
- 3.2 REINFORCE . . . 18
  - 3.2.1 Implementation Notes . . . 18
  - 3.2.2 Results . . . 19
- 3.3 Actor Critic . . . 21
  - 3.3.1 Implementation Notes . . . 21
  - 3.3.2 Results . . . 21
- 3.4 Policies . . . 24
  - 3.4.1 Stochasticity . . . 24
  - 3.4.2 Complexity . . . 25
- 3.5 Conclusions . . . 26

4 Classical Techniques in a Multi-Agent Setting **27**
- 4.1 The Environment . . . 27
- 4.2 Results . . . 27
  - 4.2.1 Conclusions . . . 29

5 Extensions of Policy Gradient Methods for Multi-Agent Systems **30**
5.1 Introduction . . . 30
5.2 Problem Simplifications . . . 30
5.3 State Distribution Drift . . . 31
5.3.1 Problem Description . . . 31
5.3.2 Approach . . . 32
5.3.3 Results . . . 33
5.4 Non-Stationarity . . . 34
5.4.1 Problem Description . . . 34
5.4.2 Approach . . . 34
5.4.3 Results . . . 35
5.5 Automatic Hyperparameter Tuning . . . 37
5.5.1 Problem Description . . . 37
5.5.2 Approach . . . 37
5.5.3 Results for Single Hyperparameter Tuning . . . 40
5.5.4 Extension to Other Hyperparameters . . . 41
5.5.5 Reducing the Sample Complexity with Off-Policy Learning . . . 43
5.5.6 Reintroducing More Agents . . . 46

6 Conclusion and Further Work **48**
6.1 Conclusion . . . 48
6.1.1 Use of Coach in MARL . . . 48
6.1.2 Bias in Our Setting . . . 49
6.1.3 Extensions to Other Settings . . . 49
6.2 Further Work . . . 49
6.2.1 Testing on Other Environments . . . 49
6.2.2 Improvements on the Algorithm . . . 50
6.2.3 Computational Efficiency . . . 50

A Proof of Lemmas **52**

List of Figures

1.1 The single-agent problem setup. . . . 1
1.2 An MDP. . . . 2
1.3 The multi-agent problem setup. . . . 4
1.4 A multi-agent problem from the perspective of a single agent. . . . 5

2.1 An illustration of the Actor Critic algorithm. . . . 12

3.1 The effect of baselining on the performance of REINFORCE. . . . 19
3.2 The effect of batch size on the performance of REINFORCE. . . . 20
3.3 The effects of different orders of polynomial feature maps to represent the value function in the Actor Critic algorithm. . . . 21
3.4 The value functions and the value function labels, projected onto "Time" and "Energy". 22
3.5 The effect of batch size on the Actor Critic algorithm. . . . 22
3.6 A comparison of REINFORCE with the Actor Critic algorithm . . . 23
3.7 A deterministic implementation of an optimal policy, showing the velocity and probabilities of taking each action at each time step. . . . 24
3.8 Convergence of reward, mean entropy and mean velocity for Actor Critic. . . . 25
3.9 Comparison of the performance of a policy modelled by a logistic regression and a 2-hidden-layer neural network under REINFORCE. . . . 26

4.1 Variation in drag coefficient. . . . 27
4.2 A comparison of the three key algorithms on a multi-agent system. . . . 28
4.3 REINFORCE applied to a 4 agent system with individual rewards for one seed. . . . 28

5.1 A demonstration of changing feature magnitudes due to state distrubtion drift. . . . 31
5.2 The effects of diminishing feature magnitudes on the performance of the Actor Critic algorithm. . . . 32
5.3 An adaptive normaliser compared to a large and small fixed normaliser. . . . 33
5.4 The effect of tracking a non-stationary value function on the learning of individual random seeds. . . . 36
5.5 Learning our defined U-function instead of a V-function. . . . 36
5.6 The framework considered for hyperparameter learning. . . . 37
5.7 A demonstration of our performance metric, which factors in the potential of a policy as well as current reward. . . . 38
5.8 The optimal exploitation rate for a run over a single seed. . . . 39
5.9 Comparison of learning with automatic hyperparameter tuning to using the optimal parameters. . . . 41
5.10 Comparing an automatically tuned exploitation rate and no exploration. . . . 41
5.11 Tuning the distance penalty and exploitation parameter simultaneously. . . . 42
5.12 A comparison of all of the automatic tuning techniques. . . . 44
5.13 Two different seeds' trajectories through hyperparameter space. . . . 45
5.14 A comparison of the techniques considered in the multi-agent setting, with a varying normaliser applied. . . . 46
5.15 A demonstration of the effectiveness of our approach when the number of agents in the environment is increased. . . . 47

Nomenclature

Abreviations

RL	Reinforcement Learning.
MARL	Multi-Agent Reinforcement Learning.
MDP	Markov Decision Process.
MG	Markov Game.
PAC	Probably Approximately Correct.
PG	Policy Gradients.
AC	Actor Critic.
SAC	Soft Actor Critic.
TRPO	Trust Region Policy Optimization.
A3C	Asynchronous Advantage Actor Critic.
PWM	Pulse Width Modulation.
DP	Distance Penalty.
GPI	Gaussian Position Initialisation.
OPL	Off-Policy Learning.
CNN	Convolutional Neural Network.

Sets

$\mathcal{A}$	Action space.
$\mathcal{S}$	State space.
$\mathcal{T}$	Set of possible trajectories.

Variables

$s \in \mathcal{S}$	State.
$a \in \mathcal{A}$	Action.
$t \in [T]$	Time.
$T \in \mathbb{Z}$	Time Horizon.
$\tau \in \mathcal{T}$	Trajectory.
$\theta \in \Theta$	Policy parameters.
$\phi \in \Phi$	Value function parameters.
$\psi \in \Psi$	Hyperparameters.
$b \in \mathbb{R}^T$	Baseline.
$\sigma \in \mathbb{R}$	Standard deviation.
$m \in [M]$	Number of episodes.
$n \in \mathbb{Z}$	Number of agents.
$\alpha \in [0, 1]$	Averaging parameter.
$\eta \in \mathbb{R}$	Learning rate.
$\lambda \in [0, 1]$	Exploitation rate.
$F \in \mathbb{R}^{m \times n}$	Fisher information matrix.
$x \in \mathbb{R}$	Position.
$v \in \mathbb{R}$	Velocity.
$E \in \mathbb{R}$	Energy.
$f \in \mathbb{R}$	Feature.
$c_D \in [0, 1]$	Drag coefficient.
$W \in \mathbb{R}^{m \times n}$	Weight matrix.
$h \in \mathbb{R}^n$	Network layer.
$P \in \mathbb{R}$	Performance score.
$e \in [E]$	Number of episodes.
$E \in \mathbb{Z}$	Total number of episodes.

Functions

$\mathbb{P} : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathcal{S}}$	State transition matrix.
$\pi_\theta : \mathcal{S} \to \Delta_{\mathcal{A}}$	Policy parameterised by θ.
$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	Reward function.
$h : \Delta_{\mathcal{A}} \to \mathbb{R}$	Entropy of a policy.
$V^\pi : \mathcal{S} \to \mathbb{R}$	State value function under policy π.
$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	State action value function under policy π.
$U^\pi : \mathcal{S} \times \mathcal{A}^{n-1} \to \mathbb{R}$	Multi-agent state value function under policy π.
$A : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	Advantage of a state-action pair.
$R : \mathcal{T} \to \mathbb{R}$	Reward of a trajectory.
$H : \mathcal{T} \to \mathbb{R}$	Entropy of a trajectory.
$p_\theta : \mathcal{T} \to \Delta_{\mathcal{T}}$	Probability of a trajectory given a policy parameterised by θ.
$\mathcal{L}_V : \Phi \to \mathbb{R}$	Loss function for the value approximator.

Chapter 1

Introduction

Reinforcement learning (RL) has been successfully applied to a wide range of problems over the last decade, perhaps most famously in AlphaGo, where it was used to defeat a 9-dan professional at the Chinese game Go. Such successes have shown reinforcement learning to be a powerful way of solving highly complex problems where the ultimate goal can be expressed as a reward function.

Go is an example of a zero-sum game. This means that each player is competing against the other, and if one agent wins then the other must lose. In this way, the total utility of the players is always zero. Most successes in RL revolve around two agent zero-sum games, or single-agent simulations like Tetris where one agent's actions are controlled to maximise reward within the fixed dynamics of its environment .

Extending these ideas to cooperative and mixed competitive-cooperative multi-agent reinforcement learning (MARL) problems is an appealing yet challenging prospect. In these settings, the behaviour of multiple agents needs to be learnt so that they can work towards a common goal. This framework is well suited to most real world situations, where agents are seldom alone, and need to consider the actions of those around them. Introducing more interdependent agents who all are trying to learn at the same time, however, adds significant complexity.

The aim of this paper is to investigate how single-agent reinforcement approaches can be applied to multi-agent systems. We look at the problems they face, and redesign the algorithms to address these. The investigation is framed in simulations of cycle races, where complex team strategies emerge from the need to cycle close together to reduce drag. The aim is a system whereby agents can learn strategic behaviour in order to benefit the team.

1.1 Single-Agent Reinforcement Learning

1.1.1 The Reinforcement Learning Problem

Figure 1.1 shows the classic reinforcement learning problem for a single-agent. The problem is defined in terms of an agent and the environment. The agent can be thought of as the part of a system whose behaviour can be altered, while the environment represents everything else. In the case of a single cyclist the agent is the cyclist, while the environment is the cyclist's surroundings.

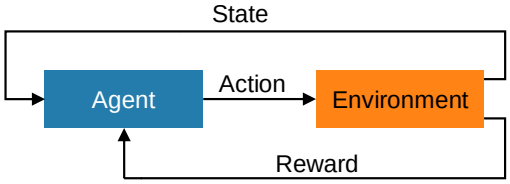


Figure 1.1: The single-agent problem setup.

At each time step, the agent takes an action depending on an observation of the system state. This action alters the system state, and the agent receives a new observation of the state as well as a reward after the action. In the context of the cyclist, the action could be to push harder on the pedals, which would result in the cyclist

changing position and velocity depending on factors like friction and air resistance. The reward could be the distance travelled by the cyclist in that step. The aim of reinforcement learning is to find an optimal policy which chooses the best action based on the state in order to maximise overall reward.

1.1.2 Agent

State observation

In the single-agent setting, we consider a fully observable system, where the agent observes the entire system state at a given time t. The state is denoted $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space. It is a full description of the context of the system - in the case of our cyclist it could be the cyclist's position and velocity as well as myriad other factors, like the time of day and the rustiness of the bicycle.

In reality, it is often impossible for an agent to have access to all of this information and agents are in a partially observable system in which there is a distinction between the system state and the agent's observations of the state. We do not consider this setting here, so refer interchangeably to the state and the state observation.

The state spaces we consider here are discrete, but extremely large.

Policy

The behaviour of an agent is described by its policy, which decides which actions the agent takes. At every time step, the agent takes an action $a_t \in \mathcal{A}$, where $\mathcal{A}$ is a set of possible actions. We consider small, discrete action spaces in this paper, where the cyclist can choose to either slow down, speed up, or stay at the same speed. The policy is the function which maps the observation of state to an action, or in our case a distribution over possible actions.

Here, we consider stochastic policies of the form $\pi_\theta : \mathcal{S} \to \Delta_{\mathcal{A}}$, where $\Delta_{\mathcal{A}}$ is a probability simplex over the action space. The policy is defined by some parameters $\theta \in \mathbb{R}^d$, which are usually the parameters in a neural network. The action taken at time t is distributed according to the policy, as a function of state:

$$a_t \sim \pi_\theta(a_t|s_t). \tag{1.1}$$

1.1.3 Environment

The environment decides what happens when an agent takes an action in a given state. It is defined by a state transition matrix and a reward function. The state transition matrix defines the dynamics of the system, while the reward function describes the goals of the agent.

State transition matrix

The state transition matrix $\mathbb{P} : \mathcal{S} \times \mathcal{A} \to \Delta_{\mathcal{S}}$ maps the current state and the agent's action to a distribution over states. The underlying assumption is that of a Markov Decision Processes (MDPs), illustrated in Figure 1.2. This means that a subsequent state depends only on the previous state-action pair. The probability distribution for the next state is then a function only of current state and action:

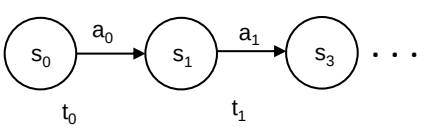


Figure 1.2: An MDP.

$$s_{t+1} \sim \mathbb{P}(s_{t+1}|s_t, a_t). \tag{1.2}$$

Reward

The reward drives learning in reinforcement learning. It is returned by the environment to the agent after the agent takes an action, indicating whether that action was good or bad in the current state. In this paper, we only consider deterministic reward functions $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The aim of agents is to maximise the total reward, R, defined as:

$$R = \sum_{t=1}^{T} r_t, \tag{1.3}$$

where T is the end of an episode. The state at time T is known as the terminal state, and all actions taken here or at a time after T lead to a reward of 0. Problems formulated in this way are known as finite-horizon problems, where an episode always has an end point. If an episode finishes before T, we can consider all subsequent rewards as being 0 until T.

The ultimate goal of the agent can be restated in terms of trajectories. A trajectory $\tau \in \mathcal{T}$ can be defined as a series of states and actions - $\tau = \{s_0, a_0, s_1, a_1, \ldots, s_T\}$. Here, we have defined $\mathcal{T} = (\mathcal{S} \times \mathcal{A})^T \times \mathcal{S}$. Noting that in an MDP with the policies considered here, actions depend only on the current state and states depend only on the previous state-action pair, the probability of a trajectory given a policy is then:

$$p(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t)\mathbb{P}(s_{t+1}|s_t, a_t). \tag{1.4}$$

For brevity, $p(\tau|\theta)$ is denoted $p_\theta(\tau)$. If $R(\tau)$ denotes the reward for a trajectory (calculated using Equation (1.3)), the task of the agent is to learn an optimal policy, $\pi_{\theta^\star}$, which will maximise the expected total reward:

$$\theta^\star = \arg\max_\theta \quad \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)]. \tag{1.5}$$

1.1.4 Value Functions

In reinforcement learning, value is defined as the potential reward that can be achieved. With this in mind, we can define both what is referred to in RL literature as a V-function, $V : \mathcal{S} \to \mathbb{R}$, which maps a state to a value, and a Q-function, $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which represents the value of a state-action pair. They are defined as shown below:

$$V^\pi(s) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t'=t}^{T} r_{t'} \middle| s_t = s \right], \tag{1.6}$$

$$Q^\pi(s, a) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t'=t}^{T} r_{t'} \middle| s_t = s, a_t = a \right]. \tag{1.7}$$

We note that these are policy dependent since they depend on the actions taken at each step. As we learn better policies, the values of states tend to increase. The value functions are expressed recursively in the Bellman equations, in terms of the immediate expected reward and the value of the expected subsequent state. For a deterministic reward function, the Bellman equations are:

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(a|s)} \left[r(s, a) + \mathbb{E}_{s' \sim \mathbb{P}(s'|s,a)} V^\pi(s') \right], \tag{1.8}$$

$$Q^\pi(s, a) = r(s, a) + \mathbb{E}_{s' \sim \mathbb{P}(s'|s,a)} V^\pi(s'), \tag{1.9}$$

$$= r(s, a) + \mathbb{E}_{s' \sim \mathbb{P}(s'|s,a), a' \sim \pi(a'|s')} Q^\pi(s', a'). \tag{1.10}$$

1.1.5 Challenges in the Single-Agent Setting

In tabular settings, where policies and state transitions can be formulated in a table, the Probably Approximately Correct (PAC) lower bound to achieve an ϵ-optimal policy is linear in $|\mathcal{S}|$ [1]. This means for a tabular formulation we pay a penalty proportional to the number of states on sample complexity, which is very problematic for the large state-space problems considered here. This is why we use function approximation methods, where the policy and any value functions are approximated by a neural network with d parameters. In doing this we make the problem feasible even for infinite state problems, but we still incur other problems.

- *Misspecification* - The optimal policy and any value functions may not be parameterisable by θ.
- *Sparsity* - We will never be able to observe all of the states in the system.
- *Exploration Vs. Exploitation* - We want to explore and visit as many states as possible to try to discover new policies, but doing so incurs the opportunity cost of exploiting the perceived best policy to maximise reward.

These problems are ubiquitous in reinforcement learning, and especially the third of these is central to algorithmic decisions in this project.

1.2 Multi-Agent Reinforcement Learning

In a multi-agent setting, we can change our assumption of an MDP to that of a Markov Game (MG) [2], as show in Figure 1.3. At each time step, every agent makes their own observation of the state (which in a fully observable system is the same for all agents) and subsequently takes an action. The transition matrix is now larger, as it maps a state and the actions of all of the agents to a new state. For a system with n agents, we denote this matrix $\mathbb{P}_G : \mathcal{S} \times \mathcal{A}^n \to \mathcal{S}$. The agents also each get an individual reward. This framework introduces more challenges on top of those faced in a single-agent setting.

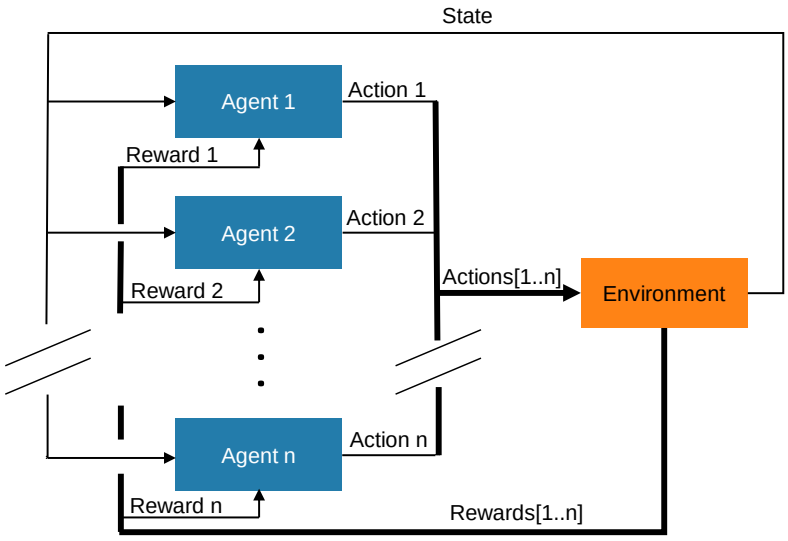


Figure 1.3: The multi-agent problem setup.

1.2.1 Optimal Multi-Agent Policies

The concept of optimality in a multi-agent setting is less well defined than in a single-agent setting since different agents can have different goals. The way the optimal strategy is often defined, in particular in zero-sum games, is using the concept of a Nash equilibrium. This is where any one player cannot gain any more reward by changing their own policy, assuming the other players' strategies are fixed.

In this paper, we avoid this complication by considering the mean reward achieved by the agents as a metric for performance. The settings we consider are all cooperative, where agents can achieve high rewards without compromising the ability of other agents to achieve high rewards, so in these cases it is a reasonable metric.

We do, however, define the system such that each agent can receive an individual reward, which would allow it to be set for more competitive settings. In such cases, a measure of sub-optimality would be needed, which can be defined using the notion of best response [3].

1.2.2 Challenges in a Multi-Agent Setting

The curse of dimensionality

Adding more agents rapidly increases the complexity of the problem, as seen in the size of the state transition matrix. The state space and action space increase exponentially with the number of agents making tabular methods impossible for more agents, even with very simple single-agent state spaces. With function approximation methods, as used in this project, each agent requires a policy and the number of parameters can grow prohibitively large for many agents.

Non-stationarity and reward ambiguity

The most significant challenge is that from the perspective of a single-agent, the setting is non-stationary. This is shown in Figure 1.4, where from the perspective of a given agent, since the behaviour of the other agents cannot be controlled they effectively form part of its environment. Their behaviour changes over time as they learn different policies, which means the environment from the perspective of a particular agent is non-stationary.

Single-agent reinforcement learning algorithms usually operate on stationary processes, whereas here the optimal policy they are searching for varies over time. This can in theory be dealt with by the algorithms considered in this project, but introduces problems.

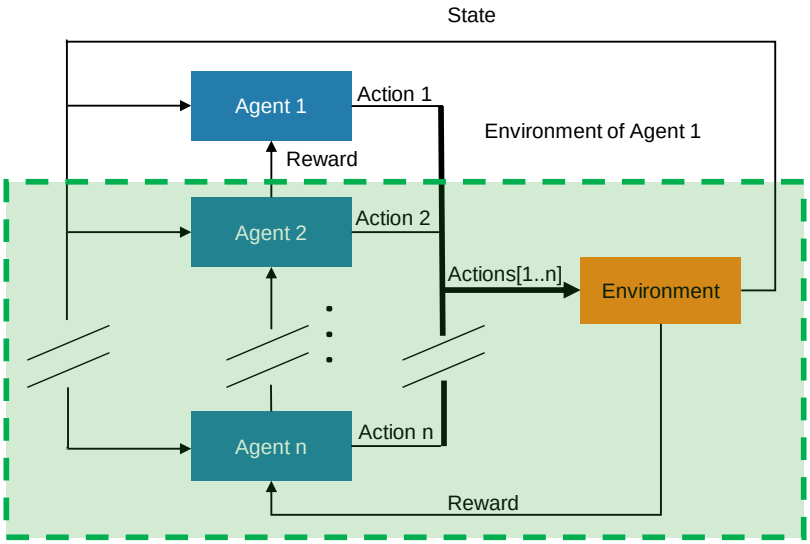


Figure 1.4: A multi-agent problem from the perspective of a single agent.

- An agent can easily fall into a local maximum, which was the optimal policy for certain behaviour from other agents, but which is not optimal for the new behaviour of the agents.
- The rewards from the simulation can swing dramatically if all of the agents are changing their policy at once. It is easy to imagine an under-damped system where multiple agents simultaneously changing their behaviour makes the system unstable.

This single-agent perspective can also introduce ambiguity in rewards. Many problems are naturally framed in terms of a group reward attributed to the agents. For example in a cycle race the team's performance can be more important than that of the individual cyclists. In these situations, agents do not necessarily know whether a high group reward is due to their actions or due to the actions of the other cyclists. This can cause problems where for example one agent could learn a good policy which helps the team perform well, and encourages the other agents' behaviour even if this behaviour is sub-optimal.

Even with an individual reward returned to each agent, their reward could be heavily influenced by the actions of the other agents. An agent might do better in one round even with a worse policy if another agent helps it in some way. This can encourage the agent to act in ways which are not actually beneficial.

This non-stationarity is the chief reason attempts to naively apply single-agent algorithms to a multi-agent setting often fail.

Chapter 2

Classical Methods

In this section, we review some of the classical reinforcement learning methods employed in single-agent settings. We focus on Policy Gradient (PG) methods (as opposed to Q-learning), which can be extended quite easily to a multi-agent setting.

2.1 Policy Gradients

The following derivation is based on that found in Peters and Schaal [4].

The idea is to solve the optimisation problem Equation (1.5) through gradient assent. The gradient of the objective function can be neatly expressed as shown below:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] &= \nabla_\theta \int_{\tau\in\mathcal{T}} R(\tau) p_\theta(\tau)\,\mathrm{d}\tau, && (2.1)\\
&= \int_{\tau\in\mathcal{T}} R(\tau)\nabla_\theta p_\theta(\tau)\,\mathrm{d}\tau, && (2.2)\\
&= \int_{\tau\in\mathcal{T}} R(\tau)\nabla_\theta[\log(p_\theta(\tau))] p_\theta(\tau)\,\mathrm{d}\tau, && (2.3)\\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla_\theta[\log(p_\theta(\tau))]]. && (2.4)
\end{aligned}$$

In Equation (2.2), Leibniz's rule has been applied for constant integration bounds to move the gradient through the integral, and it is noted that the reward is independent of θ. At Equation (2.3), the following identity has been used:

$$\nabla_\theta[\log(p_\theta(\tau))] p_\theta(\tau) = \nabla_\theta p_\theta(\tau), \tag{2.5}$$

which follows from applying the chain rule to $\nabla_\theta[\log(p_\theta(\tau))]$ and rearranging.

The second term in the expectation can also be simplified further, by taking the derivative of the logarithm of Equation (1.4):

$$\begin{aligned}
\nabla_\theta[\log(p_\theta(\tau))] &= \nabla_\theta\left[\log p(s_0) + \sum_{t=0}^{T-1}\log\pi_\theta(a_t|s_t) + \sum_{t=0}^{T-1}\log\mathbb{P}(s_{t+1}|s_t,a_t)\right], && (2.6)\\
&= \left[\sum_{t=0}^{T-1}\nabla_\theta\log\pi_\theta(a_t|s_t)\right], && (2.7)
\end{aligned}$$

since the first and third terms in Equation (2.6) are independent of θ. Substituting Equations (2.7) and (1.3) into Equation (2.4) leads to a final gradient:

$$\nabla_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=0}^{T-1} r_t \right) \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \right]. \tag{2.8}$$

An intuitive interpretation

The gradient is made up of two parts - the expected total reward and a gradient term. The gradient has been expanded out from $\nabla_\theta \log(p_\theta(\tau))$ in Equation (2.4), which is the gradient of the log-probability of a trajectory τ given a policy parameterised by θ. The final gradient in Equation (2.8) is $\nabla_\theta \log \prod \pi_\theta(a_t|s_t)$, the gradient of the log-probability of a set of actions being taken given a trajectory. This is the gradient for a maximum likelihood estimator, where the posterior is the probability of getting a trajectory given a policy.

Policy gradients assume an uninformative prior, not making any assumptions about the dynamics of the environment. Instead, steps are taken in the direction $\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)$ to maximise the probability of taking a given trajectory.

The gradient term alone would do this indiscriminately, making all trajectories more probable. In the policy gradients equation, it is modulated by total reward, which means larger gradient steps are taken towards trajectories which lead to more reward. Finally, an expectation is taken over all possible trajectories, so that over time the agent will arrive at the policy which leads to the best trajectory.

2.2 Implementing Policy Gradients

The gradient updates in Equation (2.8) are not practical to implement in reality - some changes must be made before the method can be used.

First of all, the equation for the gradient takes an expectation over all possible trajectories, which is impossible to calculate without knowledge of the environment dynamics. The solution is to estimate the expectation by finding the mean over a set of M sample trajectories which are gathered from experience. This means the gradient steps become:

$$\nabla_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)}[R(\tau)] \approx \frac{1}{M} \sum_{m=1}^{M} \left(\sum_t r_{i,t} \right) \left(\sum_t \nabla_\theta \log \pi_\theta(a_{m,t}|s_{m,t}) \right). \tag{2.9}$$

This effectively changes the gradient descent method to mini-batch gradient descent, where the gradient is evaluated over M trajectories. Increasing M means the gradient is closer to the true gradient, but gradient updates happen at a lower frequency.

Using Equation (2.9) allows the gradient to be evaluated, and leads to a usable algorithm. However, the algorithm does not perform well in practice due to high variance in the reward term. This is countered in two main ways - appealing to causality and using baselining. Both make use of the following lemma and its corollary, which are proved in Appendix A:

Lemma 1 *For a function $f : \mathcal{S} \to \mathbb{R}$, $\mathbb{E}_{a \sim \pi(a|s)}[\nabla_\theta \log \pi(a|s) f(s)] = 0$.*

Corollary 1.1 *In an MDP, with some function $f : \mathcal{S} \to \mathbb{R}$ which is independent of action at t, $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s) f(s)] = 0$.*

It was noted above that the algorithm works by taking steps to make experienced trajectories more likely, with bigger steps for trajectories that lead to more reward. The gradient used in Equation (2.8) breaks this down into individual actions within the trajectory, finding the gradient for each action and multiplying it by the total reward received by following that trajectory. Distributing the summation in Equation (2.8) reflects this:

$$\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=0} r_{t'}\right)\right]. \tag{2.10}$$

Causality implies that past rewards cannot be influenced by future actions. The above can then be simplified by employing Corollary 1.1:

$$\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=0}^{t-1} r_{t'} + \sum_{t'=t}^{T} r_{t'}\right)\right], \tag{2.11}$$

$$= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} r_{t'}\right)\right], \tag{2.12}$$

$$\tag{2.13}$$

This means that the gradient for each action is only multiplied by the reward to come, since that action did not affect previous rewards. The resulting terms are smaller, so the variance is reduced.

The second way of reducing the variance in the reward term is with baselining. In the implementation above, the gradient steps are taken in the direction of all trajectories experienced, provided the total reward is not 0. In reality, some trajectories will be bad trajectories, from which the policy should step away, not towards. This can be implemented by modulating the gradient with an advantage instead of the total reward. The advantage is a measure of how good an action was compared to how good the expected action at that state would be. A simple measure of advantage is:

$$A(s,a) = \sum_t r(s_t,a_t) - b_t, \tag{2.14}$$

where b_t is a temporally varying baseline, which indicates how well we expect to do at time t. Since b_t does not depend on the action taken, Corollary 1.1 implies that:

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(s,a) b_t\right] = 0, \tag{2.15}$$

and the gradient can be expressed as:

$$\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t} r_{t'} - b_t\right)\right]. \tag{2.16}$$

This general form of algorithm was called REINFORCE by Williams and Ronald in 1992 [5]. Peters and Schaal [4] derive a closed form for the baseline which minimises variance, but it is more simple to use the average rewards at time t as a baseline, which performs almost as well. This means that any action which is better than average is encouraged, while actions which return lower reward than average are discouraged. Algorithm 1 shows an implementation of this form of REINFORCE, where the next baseline is calculated as a weighted average:

$$b_{t,m} = \alpha_b r_{t,m-1} + (1-\alpha_b) b_{t,(m-1)}, \tag{2.17}$$

where α_b is a parameter representing the size of the averaging window and m represents a trajectory. b is a T-dimensional vector which is updated at the end of every episode.

This is the update policy suggested by Sutton and Barto [6] for tracking a non-stationary problem. $\frac{1}{\alpha}$ can be thought of as a mass - high values of α mean the baseline is more affected by the last few samples, while low values of α give the baseline more inertia, so that it varies less over time. Most implementations of REINFORCE use the true mean, which is equivalent to setting $\alpha = \frac{1}{m}$ in the equation above.

Algorithm 1 A psuedo code implementation of the REINFORCE method.

for a number of batches **do**
  for N episodes **do**
    Run an episode with policy π_θ
    Update the baseline: $b_t \leftarrow \alpha r_t + (1-\alpha)b_t, \forall t \in [T]$
  end for
  $\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] \leftarrow \frac{1}{M}\sum_m \sum_t \nabla_\theta \log \pi_\theta(a_{m,t}|s_{m,t}) \left(\sum_{t'=t} r_{s,t'} - b_t\right)$
  $\theta \leftarrow \theta + \eta \nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)]$ ▷ η is the learning rate
end for

2.3 Normalising Rewards

An additional variation on the standard REINFORCE algorithm was also tested. A natural progression from subtracting the mean from the rewards is to also divide the rewards by the average standard deviation before feeding them into the actor. The advantages of this are twofold.

- Normalising the inputs makes it easier to set a learning rate which can be used different environments. The normalised rewards will mostly lie in the range $[-1, 1]$, so similar learning rates should work well in different tasks. This also helps when other terms are added to the learning objective (see Section 2.5).
- As the scores increase, the variation in scores usually increases - with longer episodes, the effect of a good or bad policy is compounded over more steps. This means the changes in the model will be greater for larger rewards, which is counter-intuitive if the policy is supposed to converge over time. By dividing by a running standard deviation, this effect is negated as the impact of trajectories will only be large if the advantage is very different to the expected advantage.

The temptation is to implement this in the same way as with the baseline, by dividing each expected reward by the expected variance at that step. This would not be correct, however, as it would skew the weighting towards certain steps in a trajectory - in general,

$$\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t} r_{t'} - b_t\right)\right], \tag{2.18}$$

$$\neq \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)\left(\frac{\sum_{t'=t} r_{t'} - b_t}{\sigma_{t'}}\right)\right]. \tag{2.19}$$

Instead, the normalisation is done for each gradient step, so that the gradient step length is effectively reduced for trajectories where the expected variance in reward is high. The update rule becomes:

$$\theta \leftarrow \theta + \eta \frac{\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)]}{\sigma_r}, \tag{2.20}$$

where σ_r is the estimated standard deviation in rewards. This is calculated in the same way as the baseline in REINFORCE, with a weighted average of the standard deviations. Noting the deviation from the mean is the same as the advantage, at each gradient step within an episode the standard deviation is updated by:

$$\sigma_r^2 \leftarrow \alpha_\sigma \sum_t (A_t)^2 + (1-\alpha_\sigma)\sigma_r^2. \tag{2.21}$$

2.4 Actor Critic

The baseline in REINFORCE is a function of the time step only. This only works as a baseline if the agent always starts in a position where they have the same potential reward. It can also cause problems if the agent takes a bad action and ends up at some bad state along its trajectory - it could take all the best possible actions after this state but these will all be discouraged because of that one bad action.

This leads to the idea of having a baseline which is a function of state instead of time step. The formulation for this has already been discussed - we just need to learn a V-function. Since the value function is a function of state alone, the baseline in REINFORCE can be replaced with the V-function to give an unbiased estimate of the gradient:

$$\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)] = \mathbb{E}_{\tau\sim p_\theta(\tau)} \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t} r_{t'} - V^\pi(s_t) \right), \tag{2.22}$$

$$\approx \frac{1}{M} \sum_m \sum_t \nabla_\theta \log \pi_\theta(a_{m,t}|s_{m,t}) \left(\sum_{t'=t} r_{m,t'} - V^\pi(s_{m,t}) \right). \tag{2.23}$$

This will provide a better advantage estimate since it does not assume the value of a state is the same at a given time step for different episodes. The task remains of finding the value function, however. Estimating the expected reward from a state from only one experience of the state would be very noisy, and for a continuous state space it can be impossible to sample a state more than once. A solution to this is to learn an approximate value function, parameterised by ϕ, through bootstrapping. The value function can be evaluated as shown below:

$$V^\pi(s) = \mathbb{E}_{\tau\sim p_\theta(\tau)} \left[r_t + r_{t+1} + r_{t+2} + \ldots \mid s_t = s \right], \tag{2.24}$$

$$= \mathbb{E}_{\tau\sim p_\theta(\tau)}[r_t] + V^\pi(s_{t+1}), \tag{2.25}$$

$$\approx r_t + V^\pi_\phi(s_{t+1}), \tag{2.26}$$

where the final term is a biased estimator for the value since it uses an approximation to the value of the next state. However, by training the value function according to an l_2 loss function, the approximations improve over time and can converge to the true value function. The loss function is then given by the l_2 norm of this:

$$\mathcal{L}_V = \| r_t + V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t) \|_2^2. \tag{2.27}$$

We note that this is simply the l_2 norm of the estimated advantage of the state action pair, where the estimated advantage is defined as:

$$\hat{A}_\phi(s_t, a_t, s_{t+1}) = r(a_t, s_t) + V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t). \tag{2.28}$$

This scheme, shown in Figure 2.1, is called an Actor Critic method, since the actor learns a policy which is critiqued by the value function. The advantage drives learning in both the actor and the critic. The critic learns to minimise the advantage so that the value function is a closer representation of the value of a state. The actor uses the advantage to inform which trajectories should be incentivised and which should be discouraged.

Algorithm 2 shows how the method is implemented in practice. After a batch of steps, one gradient step is taken to improve the value function and the policy. This is an example of policy iteration, where alternate steps are taken to improve the value function so that it better represents state values for a given policy, and to improve the policy relative to the value function. Over time, the policy should converge to the optimal policy, and the value function should converge to a good approximation for the values of the policy.

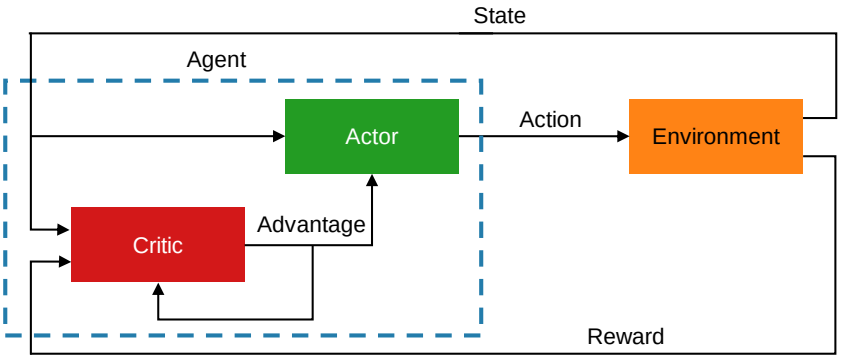


Figure 2.1: An illustration of the Actor Critic algorithm.

Another key advantage of Actor Critic methods is that they rely on Temporal Difference instead of Monte Carlo Learning. This means that they can make use of information from state transitions midway through episodes, while REINFORCE can only do parameter updates at the end of episodes. With more policy updates per episode, Actor Critic methods tend to converge faster than REINFORCE provided a good value function can be approximated.

Algorithm 2 A psuedo code implementation of the Actor Critic method.

for a number of episodes **do**
  Take n_{batch} steps
  Update critic: $\phi \leftarrow \phi - \eta_{actor} \nabla_\phi \| r_t + V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t) \|^2_2$
  Update actor: $\theta \leftarrow \theta + \eta_{critic} \sum_t \nabla_\theta \log \pi_\theta(s_{s,t}, a_{s,t}) \hat{A}(s_t, a_t, s_{t+1})$
end for

2.5 Exploration

One of the key considerations in reinforcement learning is the trade-off between exploration and exploitation. By exploiting a good policy, the agent can get higher rewards, but they could be doing this at the expense of finding a different, better policy. It is important to strike a balance between the two. In this paper, this is approached by introducing an exploration reward term, and handling it in much the same way as the classic reward term.

2.5.1 Measuring Exploration

One way of ensuring exploration is to encourage stochasticity in policies. A random policy will ensure the agent does not always act the same way, so will encounter more states. While does not ensure the agent explores all states - one can imagine a family of states which can only be encountered by a specific series of steps which are unlikely to be explored by a random policy - it is a relatively simple way to ensure the policy does not converge too quickly to a deterministic policy.

A measure of stochasticity of the policy is the entropy of the probabilities of taking each action at a time step. Entropy is a measure of uncertainty, with a maximum value for a completely uniform distribution. By introducing an entropy reward, the agent can be encouraged to follow policies which are more random, and keep exploring.

2.5.2 Entropy Reward

Introducing an entropy reward gives a new learning objective:

$$\theta^\star = \arg\max_\theta \quad \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\lambda \sum_t r_t + (1-\lambda)\sum_t h_t\right], \tag{2.29}$$

with h_t denoting the entropy of the policy at time t, and $\lambda \in [0,1]$, the exploitation parameter, representing the trade-off between exploitation and exploration. This can be broken down into two objective functions: the exploitation objective, which is the same as before, and the exploration objective:

$$\begin{aligned}
\theta^\star &= \arg\max_\theta \quad J(\theta), && (2.30)\\
&= \arg\max_\theta \quad \lambda\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[R(\tau)\right] + (1-\lambda)\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[H(\tau)\right], && (2.31)
\end{aligned}$$

where $H(\tau)$ is defined analogously to $R(\tau)$ as:

$$H(\tau) = \sum_t h_t. \tag{2.32}$$

The gradient of the objective function is then the weighted sum of the gradient of the total expected reward and the total expected entropy. The entropy gradient can be derived similarly to the reward gradient:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}[H(\tau)] &= \nabla_\theta \int_{\tau\in\mathcal{T}} H(\tau)p_\theta(\tau)\,\mathrm{d}\tau, && (2.33)\\
&= \int_{\tau\in\mathcal{T}} H(\tau)\nabla_\theta p_\theta(\tau)\,\mathrm{d}\tau + \nabla_\theta H(\tau)p_\theta(\tau), && (2.34)\\
&= \int_{\tau\in\mathcal{T}} \left(H(\tau)\nabla_\theta \log(p_\theta(\tau)) + \nabla_\theta H(\tau)\right)p_\theta(\tau)\,\mathrm{d}\tau, && (2.35)\\
&= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[H(\tau)\nabla_\theta \log(p_\theta(\tau)) + \nabla_\theta H(\tau)\right]. && (2.36)
\end{aligned}$$

The gradient of the learning objective is then given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\lambda R(\tau) + (1-\lambda)H(\tau)\right)\nabla_\theta \log(p_\theta(\tau)) + (1-\lambda)\nabla_\theta H(\tau)\right]. \tag{2.37}$$

There are two additional entropy terms. The first term acts like the reward gradient, making actions which yield higher entropies more likely. The last term in the equation encourages decisions to be more random along the trajectories taken, and its gradient can be evaluated as:

$$\begin{aligned}
\nabla_\theta H(\tau) &= \nabla_\theta \sum_t h_t, && (2.38)\\
&= -\sum_t \left(\sum_{a_i\in\mathcal{A}} \nabla_\theta(\pi(a_i|s_t)\log\pi(a_i|s_t))\right), && (2.39)\\
&= -\sum_t \left(\sum_{a_i\in\mathcal{A}} (1+\log\pi(a_i|s_t))\nabla_\theta\pi(a_i|s_t)\right). && (2.40)
\end{aligned}$$

The same tricks can be used as earlier, considering an advantage instead of the total reward and entropy, leading to a final gradient of:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[A_\lambda(\tau)\nabla_\theta \log(p_\theta(\tau)) + (1-\lambda)\nabla_\theta H(\tau)\right]. \tag{2.41}$$

where the combined advantage, paramaterised by λ is given by:

$$A_\lambda(\tau) = \lambda\left(\sum_t r_t - b_t^r\right) + (1-\lambda)\left(\sum_t h_t - b_t^h\right), \tag{2.42}$$

and the reward and entropy baselines can be functions of time step or state. In this paper, we set the entropy baseline as the maximum possible entropy ($\log |\mathcal{A}|$). This means that the entropy advantage is always negative, so that the policy takes steps away from the action taken of magnitude inversely proportional to the entropy of the action.

In practice, we apply normalisation to the entropy and reward gradients in the way discussed in Section 3.2.1. This ensures that the reward and entropy advantages are of similar magnitude, so that the exploitation parameter can explicitly define the ratio of exploration to exploitation. To keep the gradients unbiased, we also divide the entropy gradient by this term, leading to the gradient updates:

$$\theta \leftarrow \eta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[A'_\lambda(\tau) \nabla_\theta \log(p_\theta(\tau)) + \frac{(1-\lambda)}{\sigma_h} \nabla_\theta H(\tau) \right], \tag{2.43}$$

$$A'_\lambda(\tau) = \frac{\lambda}{\sigma_r} \left(\sum_t r_t - b^r_t \right) + \frac{(1-\lambda)}{\sigma_h} \left(\sum_t h_t - b^h_t \right). \tag{2.44}$$

The σ estimates are allowed to change slowly, by setting a low α_σ parameter value. Adding the standard deviations changes the objective in Equation (2.29) such that setting the exploitation parameter at 0.8, for example, ensures the objective is 80% exploitation and 20% exploration.

Comparison to other methods of entropy regularisation

Using entropy regularisation is not a new concept, but we have implemented it differently here from how it is often implemented. The Soft Actor Critic (SAC) algorithm [7], for example, uses a soft value function which accounts for entropy as well as reward. Here, we maintain a traditional value function which deals with reward value alone, and use a constant entropy baseline.

The primary reason for this is that we want to encourage absolute entropy, rather than entropy relative to previous policies. Having a varying entropy value function means that over time, as the policies become more deterministic, the entropy baseline will decrease. Then, any state where the policy entropy was higher than this lower baseline is encouraged, while more deterministic policies are discouraged less than they would have been when the entropy baseline was higher. Our decision to penalise entropy relative to the maximum possible value means that at any stage in the learning, the policy is encouraged to act as randomly as possible.

The SAC formulates the objective function with only one temperature parameter term:

$$\theta^\star = \arg\max_\theta \quad \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r_t + \lambda \sum_t h_t \right]. \tag{2.45}$$

In their formulation, it is more difficult to define the ratio of exploration to exploitation. Setting $\lambda = 0$ means there is no entropy encouragement, but there is no clear answer to how high λ needs to be to encourage a certain amount of exploration. Whereas with our method it is easy to define an exploration profile, for example explore 20% to start and then fully exploit towards the end, the SAC algorithm is very sensitive to its temperature parameter and setting the parameter is not intuitive.

Haarnoja et al. [7] suggest an extension to SAC where the temperature parameter is automatically tuned by adding the constraint that the policy entropy should be kept above a certain value, and then solving the dual of this constrained optimisation problem. While this does help to make the exploration objective clearer, the minimum entropy parameter is still non-trivial to set. In Section 5.5 we discuss how the exploitation hyperparameter can be automatically tuned in our algorithm in order to achieve a high final reward without needing to set an entropy requirement or solve the dual problem.

2.6 Other Methods

REINFORCE and Actor Critic are among the most basic Policy Gradient methods. State-of-the-art PG based algorithms tend to use a variation on the Actor Critic. In this section, we discuss Q-learning - an alternative to PG methods - and some extensions to the Actor Critic method which were not implemented here with justifications for our choices in methods.

2.6.1 Q-Learning

Q-learning is the most common alternative to PG-based methods. In it, the agent learns a Q-function, and then takes the action at each step which maximises the Q-function:

$$\pi(s) = \arg\max_a Q(s, a). \tag{2.46}$$

The update rule for the Q-function is then given by:

$$Q(s, a) \leftarrow r(s, a) + \max_{a'} Q(s', a'). \tag{2.47}$$

We focus on PG methods in this paper for a few key reasons:

- The technique used to deal with non-stationarity can only work with a PG approach (see Section 5.4).
- Q-learning leads to deterministic policies. This means they inherently encourage less state exploration. While there are methods to do this such as taking an ϵ-greedy approach, they are often less effective than having a stochastic policy.
- Some environments can only be solved with stochastic policies. A classic example is rock, paper, scissors, where if there is an intelligent opponent, the optimal solution is to pick randomly between the possible actions.
- By directly optimising the policy instead of a value function, PG methods can often achieve better sample complexity.

2.6.2 Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization [8] marks a significant improvement in PG methods, and can even guarantee monotonic improvements over policy changes. In TRPO, a constraint is added that the KL-divergence between the current policy and the next policy should not be too large. The logic behind this is that taking a gradient step assumes linearity in the policy around the current parameters, which is only valid for policies which are not very different.

In the TRPO algorithm, a second order approximation is made for the KL-divergence:

$$\mathrm{KL}(\pi_{\theta_t} \| \pi_{\theta_{t+1}}) \approx \frac{1}{2}(\theta_{t+1} - \theta_t)^T F_\theta (\theta_{t+1} - \theta_t), \tag{2.48}$$

where F is the Fisher information matrix, defined as $F = \mathbb{E}_s \left[(\nabla_\theta \log \pi_\theta(s))(\nabla_\theta \log \pi_\theta(s))^T \right]$. While this method is certainly an improvement on not including the KL term, it is not implemented in this project. The reason for this is that estimating the Fisher information matrix can be complex. While the techniques developed in the later sections of this paper could benefit from some variation on TRPO, the aim of this project is to show improvement on traditional methods, not to show state-of-the-art performance.

2.6.3 Other Variants on Policy Gradients

There are many other variants on PG methods, some of which inspired techniques used later in the paper.

The Asynchronous Advantage Actor Critic (A3C) algorithm [9] is an algorithm developed by DeepMind, which extends the Actor Critic algorithm to having multiple workers asynchronously collecting experience. This means the algorithm is faster than the standard Actor Critic, and by learning from other workers who experience completely different trajectories, each worker gets a wider range of experiences. We use a similar technique of parellelising learning in Section 5.5.

The Soft Actor Critic algorithm [7] adds an entropy regularisation term, but also uses off-policy learning with an experience replay buffer to reduce sample complexity. Both techniques are used in this paper, although in different ways - the entropy regularisation as discussed above, and the off-policy learning is introduced later to learn from parallel simulations where different hyperparameters were used.

Chapter 3

Testing Classical Techniques on a Single Agent Environment

3.1 The Environment

For initial tests of single-agent methods, we use a single cyclist environment. This environment is designed to simulate a single cyclist travelling along a one dimensional track. For these simulations, the track is effectively infinitely long, so that the cyclist could be travelling around a circular track.

The state space

The state space for this environment is given by:

- Position, x, an integer (since velocities only take integer values) which ranges from 0 to the maximum achievable distance.
- Velocity, v, an integer, which ranges from v_{min} to v_{max}. These were set such that $v \in \{0, 1, 2, 3, 4\}$.
- Energy, E, a real number which ranges from 0 to 100, to represent the energy the cyclist has left.
- Time, t, an integer which ranges from 0 to a time limit T, set at 200.

The action space

To keep policies relatively simple, a small, discrete action space was used. The action space is given by $\mathcal{A} = \{\delta v \in \{-1, 0, 1\}\}$, where δv represents a change in velocity.

Environment dynamics

For the purpose of these simulations, it was assumed that the cyclist would instantaneously change velocity at the beginning of each time step, according to their action. For a more accurate representation, this could be changed so that they change velocity over a small time step, but our approximation was enough to allow the desired emergent behaviours to emerge. This leads to the following equations for change of state:

$$v_{t+1} = v_t + \delta v_t, \tag{3.1}$$

$$x_{t+1} = x_t + v_{t+1}. \tag{3.2}$$

The second equation is counter-intuitive, and could be could be conceivably replaced with $x_{t+1} = x_t + v_t$, so the change in velocity happens at the end of the step. The problem with this is that the action does not have an effect on the distance travelled until the next state. This would abstract the action from the reward, so

that $r_t = r(a_{t-1}, s_t)$, which violates the first-order Markovian assumptions of the model. For this reason, the acceleration was left at the beginning of the step.

The "energy" state was added to represent a penalty for cycling too quickly. The change in energy over a step was modelled on air resistance. Air resistance is proportional to velocity squared, so the power required to cycle varies with velocity cubed. Energy is therefore updated according to:

$$E_{t+1} = E_t - c_D v_{t+1}^3, \tag{3.3}$$

where c_D is a drag coefficient.

The reward

The aim for initial simulations was to encourage a cyclist to travel as far as possible within the time limit. The reward was then the total distance travelled at the end of the race. This is a sparse reward, and to make learning easier to start, the reward was made dense by letting the reward at each step be the distance travelled by the cyclist during that step i.e. their velocity at the beginning of the step:

$$r(s_t, a_t) = v_t + \delta v_t. \tag{3.4}$$

3.2 REINFORCE

3.2.1 Implementation Notes

The code used was originally based on Andrej Karpathy's implementation of REINFORCE [10]. Apart from this all code, including the code for the environment, was written for this project[1].

Input data normalisation

The different state features have very different ranges, so they were normalised before they were input into the policy, as shown below:

$$f_{normalised} = \frac{2(f - f_{mean})}{f_{range}}, \tag{3.5}$$

where f denotes a feature (i.e. x, v, E or t).

Normalising the data significantly improved results. One might expect a simple policy to deal well with different input magnitudes by scaling the parameters, but in practice features with different magnitudes and means can damage the learning process.

Features which take on a different range of values can make it easy to initialise parameters incorrectly, and allow the solution to fall into a local minimum. With policy gradient methods, an additional problem is that if one feature has larger values than the others, the gradients for parameters corresponding to that feature can be very large. Particularly at the beginning of learning where the value function or baseline can be inaccurate, this can easily lead to policies taking large steps in the wrong direction and falling into locally - but not globally - optimum policies.

Subtracting the mean also made a big difference. The problem here was that if all inputs are of the same sign, the gradients can only push the parameters one way for all of the steps. Similarly to adding a baseline, subtracting the input means meant that the parameters could have negative gradients from some time steps, and positive gradients from others.

[1]Code base can be found at `https://github.com/olliematthews/multi-agent-rl-thesis`.

The importance of normalising features is revisited in Section 5.3.

The policy

For this simple task, a simple multinomial logistic regression was used for the policy. This means that the distribution of probabilities of taking each action is given by:

$$\pi(s) = \text{softmax}(Ws), \tag{3.6}$$

where $W \in \mathbb{R}^{|\mathcal{A}|\times|\mathcal{S}|}$ is a weight matrix and the softmax function maps real numbers to probabilities.

The values of W were initialised with a truncated normal distribution of standard deviation 0.1. The decision to use a simple model is revisited in Section 3.4.2.

The task is to travel as far as possible with time and energy restraints. There is an optimal speed to travel at for the agent to maximise its reward, which lies between 1 and 2 for the environment parameters used here. Since the agent's velocity is discretised to integer values, the way it can achieve this is with a strategy similar to pulse width modulation (PWM) control. The optimal policy should have the agent travel some number of steps at velocity 1 and some steps at 2.

Randomness

In order to make the results repeatable, they were run with a random seed at each time. For each simulation, the seed defined which actions the agent would take as well as how the system was initialised. Results were then averaged over multiple seeds.

3.2.2 Results

For this task, the maximum achievable score is 243, corresponding to an average velocity of 1.22 (43 steps at a velocity of 2, and 157 steps at 1). In most cases, the agent quickly learns not to go too fast as it will quickly run out of energy. It also learns that not moving at all results in no reward, and tends to move at some intermediate velocity. Different methods lead to different rates of convergence of the reward, and different final average rewards. Some key parameters in the REINFORCE algorithm are explored below.

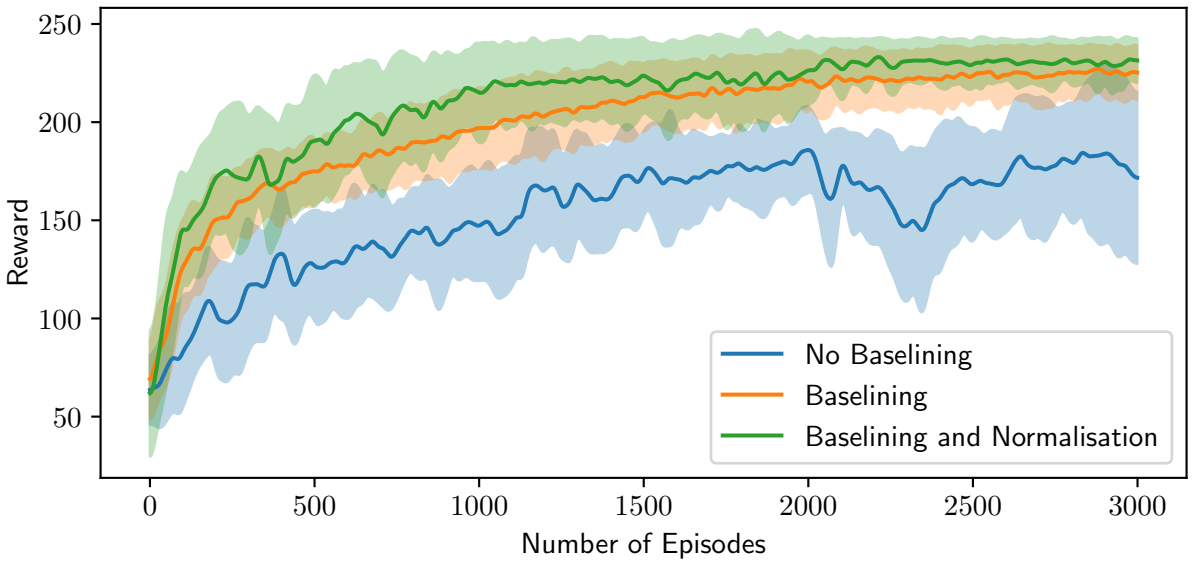


Figure 3.1: The effect of baselining on the performance of REINFORCE.

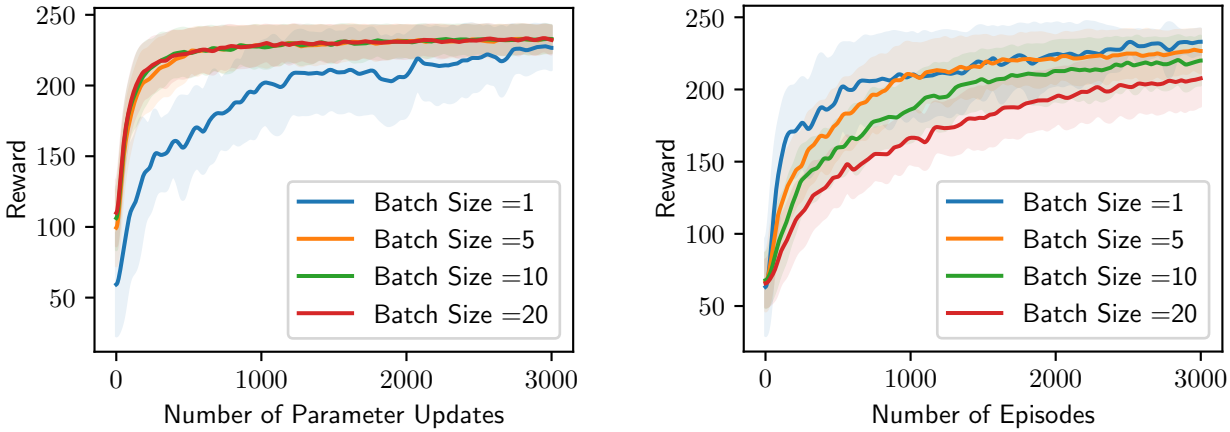
Baselining

Figure 3.1 shows the three most basic implementations of policy gradient methods described in Section 2.1. The first line shows policy gradients with no baselining, the second shows a baseline calculated by a true average, and the third shows policy gradients implemented with normalised advantages. The learning rate was optimised for each of the schemes (this is done for all graphs from here onwards). The mean rewards over 10 seeds are plotted in solid over the shaded area which represents the standard deviation in the rewards.

All three methods show evidence of learning, but without a baseline, the policies do not get close to the maximum value of 243. The rewards are notably erratic, with the high-variance reward function causing the policy to jump around. The baseline helps significantly, with the solution converging much more quickly and smoothly. Normalising the advantages improves convergence speed further, with the final rewards higher than without the normalisation.

Batch Size

The graphs below show the effect of changing the batch size for the REINFORCE algorithm. In 3.2a, the number of parameter updates is shown on the x-axis, while in 3.2b, the number of simulated episodes is shown.



(a) Policy convergence in number of parameter updates.

(b) Policy convergence in number of episodes.

Figure 3.2: The effect of batch size on the performance of REINFORCE.

The algorithm needs to perform significantly less parameter updates to converge with the larger batches, where the approximation for the expectation is better. With 5 or more batches, learning is close to monotonic, and the marginal gain from larger batches after this is minimal.

Figure 3.2b is a rescaled version of Figure 3.2a, for a situation where limiting the number of simulations is more critical than limiting the number of updates. Here, the advantage of larger batches stops after a batch size of around 2, with the number of episodes required for a parameter update outweighing the improved accuracy of gradients.

The choice of batch size is situation dependent. In some cases, with very large policy networks, it can be more expensive to update the network than to run simulations. In this case, simulations take significantly longer than evaluating the gradients for the simple policy, so a small batch size of 3 was optimal.

3.3 Actor Critic

3.3.1 Implementation Notes

Episode termination

Since a batch does not necessarily end on the end of an episode, care needs to be taken when dealing with episode termination. To deal with this, the label for the value at a state after the termination state was explicitly set to 0. This helped to anchor the function, so that the parameters could be learnt more quickly. Actions at such states also given an advantage of 0 to avoid the final action getting a large advantage when the state resets.

Value function approximator

A linear regression was used for the value function approximator, but without the non-linearity supplied by the discrete actions, it was not enough to adequately model the value. To deal with this a polynomial feature map of order p was applied to the state space, allowing a polynomial value function to be mapped.

The parameters in the value approximator were again initialised with a truncated normal distribution, with a standard deviation of 0.1 and mean of 0.

3.3.2 Results

Polynomial order

The Actor Critic algorithm introduces a new network, and with it more parameters to tune. The polynomial feature map was introduced when it became clear a linear regression would not be able to fit a good value function. This is the problem of misspecification - the optimal value function is not parameterisable under a linear regression. A crude approximation of the value function can make the algorithm unstable, since the advantage is effectively the residual error of the approximation. With an overly simplistic model, advantages reflect the errors in the model instead of potential improvements in a policy.

Figure 3.3 shows the effect of different orders of polynomial on learning. It is clear that the algorithm significantly under-performs with a linear polynomial, with the rewards jumping around erratically to reflect the large error signals from the value function. The results of a quadratic feature map are dramatic - the agents learn significantly more quickly and achieve better final policies.

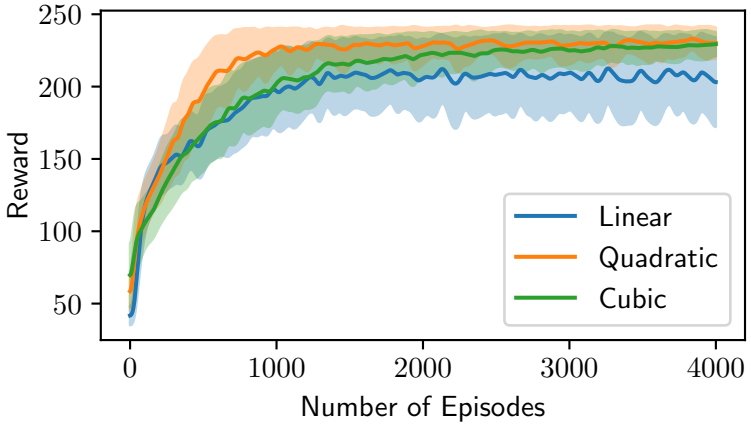
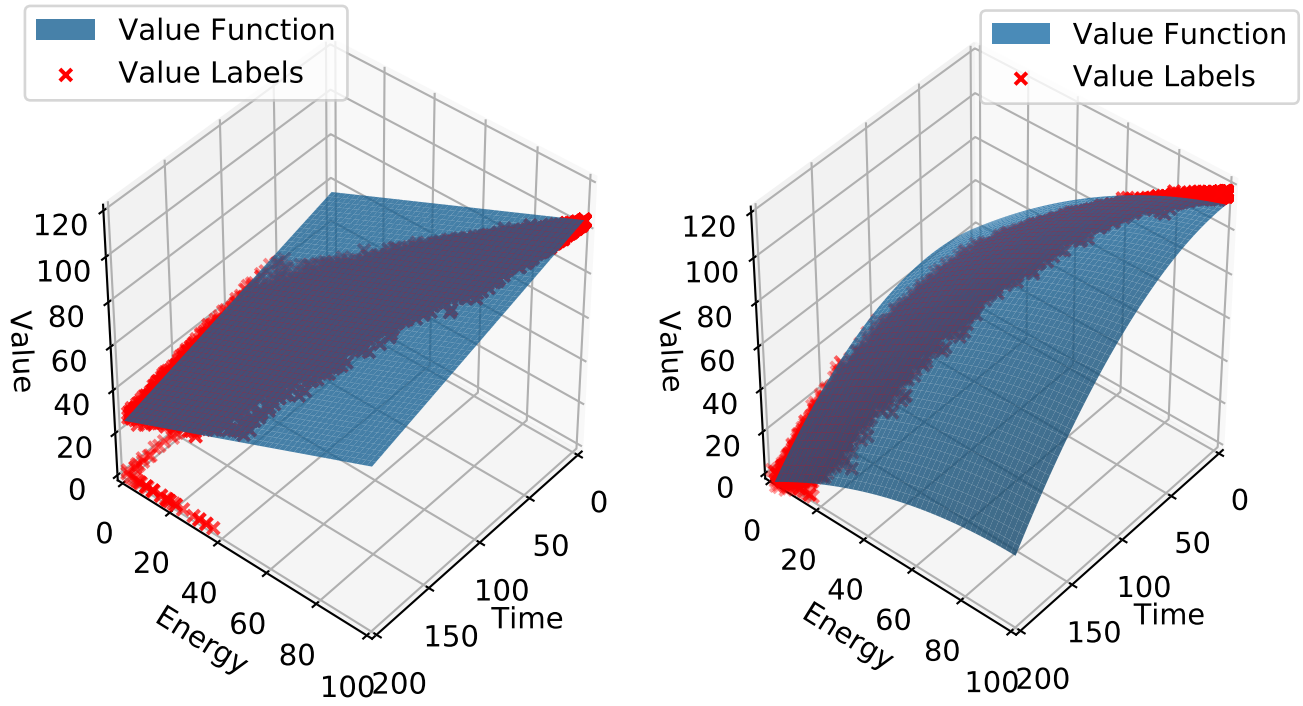


Figure 3.3: The effects of different orders of polynomial feature maps to represent the value function in the Actor Critic algorithm.

In this case, a quadratic map seems complex enough to represent the value function, and the cubic map actually slows learning. This is because a lower actor learning rate is required in the cubic map (the optimal rate was 0.003 compared to 0.005 for the quadratic). With a higher learning rate, the policy was prone to fall into local maxima while the more complex value function is still being fitted.

Figure 3.4 shows the value functions and the value labels, projected onto the two biggest contributors to state value - time and energy. Both functions are able to show decreasing value as the agent gets closer to the maximum time and as their energy drops, but the linear function is unable to represent the value dropping to



(a) Linear map

(b) Quadratic map

Figure 3.4: The value functions and the value function labels, projected onto "Time" and "Energy".

0 at the terminal states. This means the value function is unable to properly model the value, which explains it cannot encourage the policy as well as the higher order polynomial value functions.

Batch size

In this case, the batch size is the number of steps (as opposed to the number of episodes) taken before a policy and value function update. Figure 3.5 shows the effects of different batch sizes on learning. The results are again averaged over 10 seeds, with a quadratic feature map.

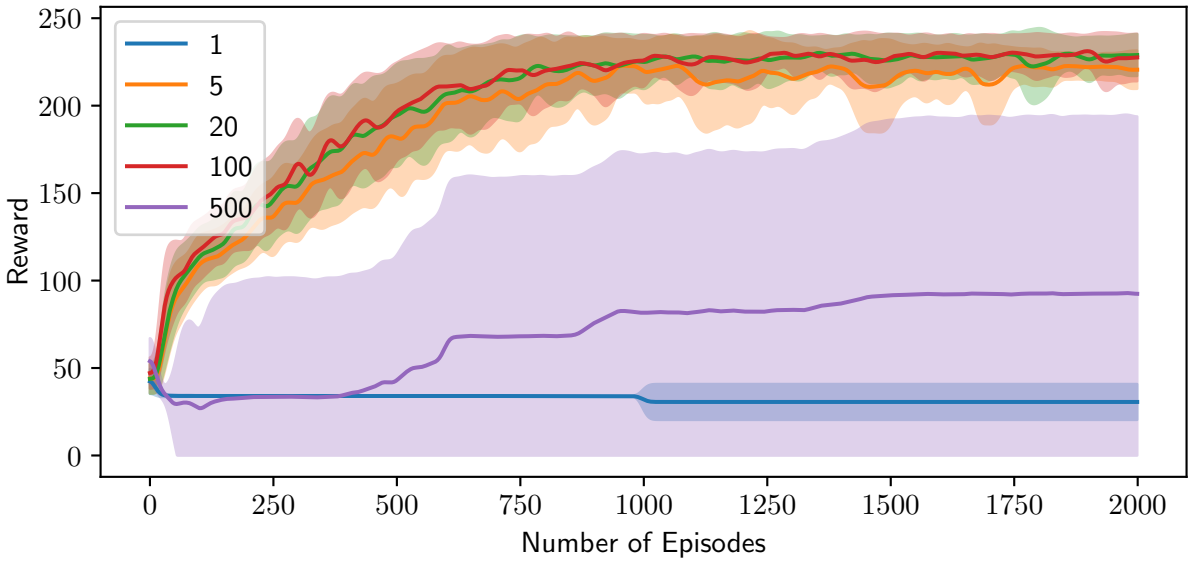


Figure 3.5: The effect of batch size on the Actor Critic algorithm.

Unsurprisingly the results are poor for a batch size of 1 where the gradients are too noisy to learn at all,

and the rewards never increase. The results with this batch size have 0 variance for many episodes, with a reward of 34. This is a local maximum where the agent accelerates every time, quickly running out of energy. The reason the variance increases circa 1000 episodes is because the agent in one of the seeds learns to do the opposite, never accelerating to get a reward of 0. Policies such as these are easy to get stuck in, and without accurate gradients, the algorithm cannot escape.

There is a dramatic change once the batch size is increased to 5. This can be attributed to the agent being able to escape the aforementioned local maxima with this batch size. It is key to note that before a whole episode is run, since the critic is learning from a biased estimator (see Equation (2.26)) the critic policy can be very inaccurate. Since this is what informs the learning of the agent, the policy can quickly converge to one which optimises value under this inaccurate value function, especially with frequent updates. If the policy converges quickly to a deterministic one under an inaccurate value function, the agent may then never see enough other trajectories to re-inform the value function, and can get stuck in this sub-optimal policy.

At the end of an episode, the value of a terminal state is labelled 0, which roots the value function and helps the critic to learn realistic parameters. With a batch size of 5, the policy can then learn effectively, while with single batch updates the agent has already fallen into a poor policy by this point.

After this, subsequent increases in batch size to 100 improve learning, but not as dramatically. The trade-off is between the noise of gradient estimates (which manifests as slower learning when averaged over 10 seeds), and the frequency of updates. When the batch size is increased to 500, such that batch updates happen at a frequency of a few episodes, learning slows dramatically.

Comparison to REINFORCE

Figure 3.6 shows a comparison of REINFORCE, the Actor Critic algorithm, and the Actor Critic with normalised rewards as discussed in Section 2.3. REINFORCE has a batch size of 3 (which is optimal for rate of learning per episode) and has gradient normalisation applied. The Actor Critic algorithms have a batch size of 100, and a quadratic feature map.

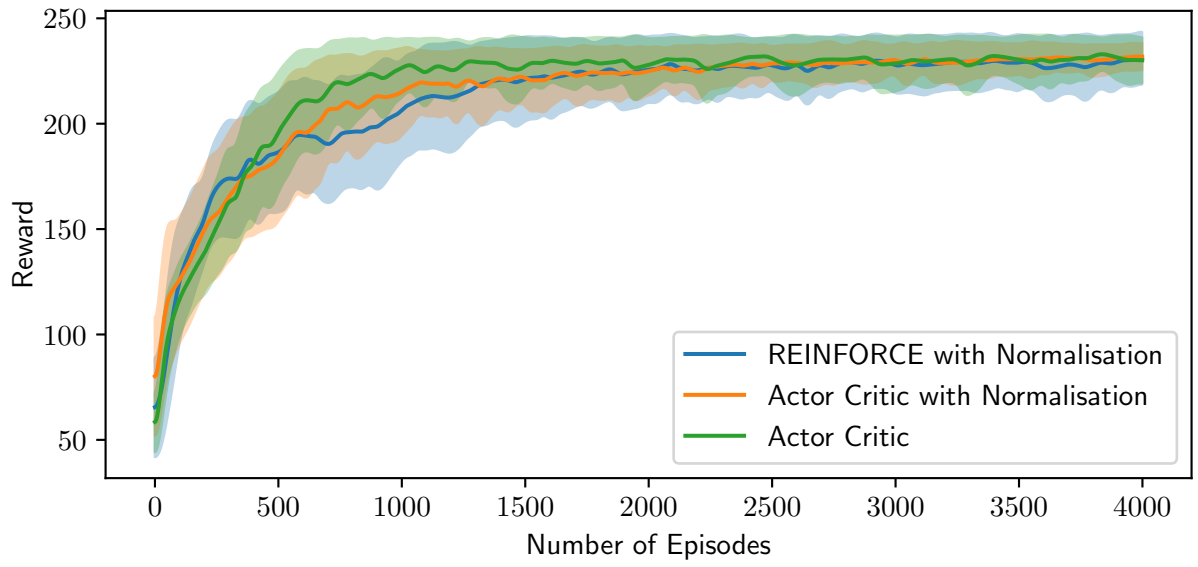


Figure 3.6: A comparison of REINFORCE with the Actor Critic algorithm

The Actor Critic algorithm outperforms REINFORCE in learning speed, although they ultimately converge to similar rewards. For this relatively simple task, where starting states are always the same, it is not surprising that they perform similarly over time. As the policy becomes more deterministic, the trajectory varies less so

that the states visited are all similar, and an estimate of expected reward based on the time step can be as accurate as a state value function. The problem with REINFORCE occurs more with complex games, where it can fail to ever reach a good deterministic policy without a good value function, or the initial state is not always the same.

Unlike with REINFORCE, normalising the states by their standard deviation actually slowed learning. This can be attributed to the fact that the standard deviation had to be initialised at a high value to avoid initial gradients being too large while the agent was still learning a good value function. Normalising the rewards can still be useful when considering exploration, so that the exploration advantage and reward advantage are of similar magnitude when combining them to form a total advantage (see Section 2.5.2).

3.4 Policies

The policies used here were all stochastic and the result of a logistic regression applied to the states. It is worth checking that these are appropriate for this sort of problem, and whether a more complex or a deterministic policy would be better.

3.4.1 Stochasticity

Figure 3.7 shows the highest scoring policy from the Actor Critic algorithm in action. The graphs demonstrate what happens when the policy is followed deterministically - always choosing the action with the highest probability. The top graph shows the velocity the cyclist takes, and the bottom graph shows the probability of each action at each time step.

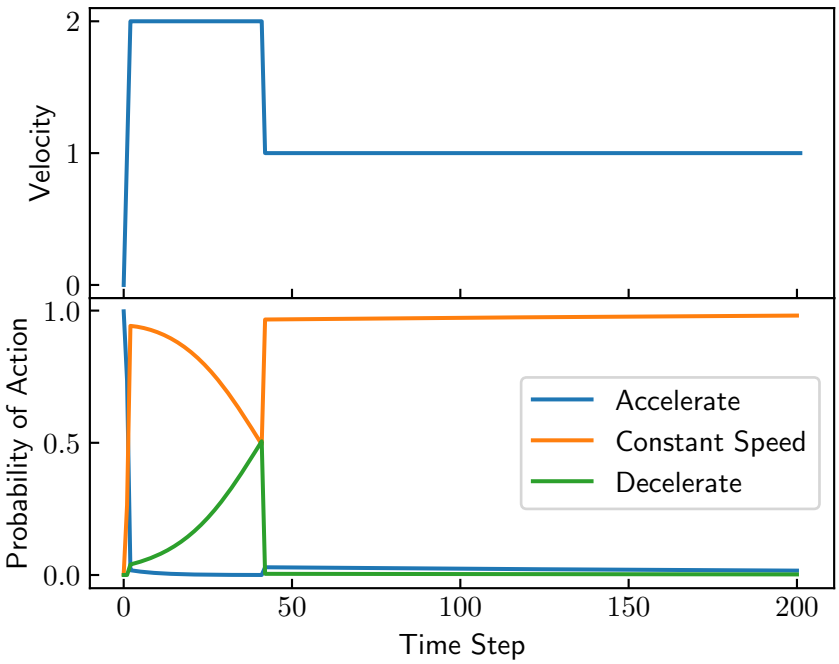


Figure 3.7: A deterministic implementation of an optimal policy, showing the velocity and probabilities of taking each action at each time step.

The result is a score of 240, which is close to the maximum score of 243. The agent achieves this in a way similar to the way PWM works. The optimal velocity lies between 1 and 2, but the agent can only travel at integer speeds. To achieve the optimal velocity on average, the cyclist travels faster at first, and then slows down at a certain point.

At a velocity of 0, the cyclist always accelerates, as this state has no value. The cyclist accelerates to a velocity of 2, where the probability of decelerating increases until the cyclist slows down, after which they stay at the same speed for the rest of the race.

As the policy converges to this one, it becomes more deterministic, with the probabilities lying closer to 0 and 1 at each step. This can be seen in Figure 3.8, where the reward, average entropy of the policy and average velocity of the agent are plotted for the Actor Critic algorithm. Over time, the entropy of the policies drops as the agent learns good behaviour and follows it more. At the same time, the velocity converges to a value, with the variance dropping as the agent learns to travel at the optimal speed.

This is one of the strengths of a stochastic policy - it can act stochastically at first while the agent tries out new states, and then converge to a more deterministic policy over time. This means that in this case, the agent does not need any specific incentive to explore, as it is able to explore enough states with the stochastic policy alone.

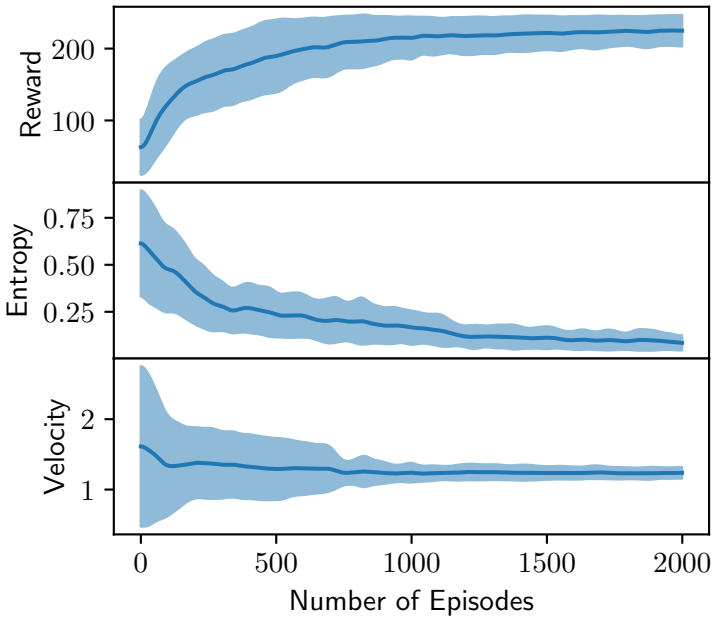


Figure 3.8: Convergence of reward, mean entropy and mean velocity for Actor Critic.

This also highlights a potential problem in more complex games if the agent converges to a deterministic policy too soon. The problem is especially pertinent in the non-stationary environments involved in multi-agent learning. If an agent converges quickly to a deterministic policy, it will not be able to explore later as other agents act differently, preventing it from finding an optimal collaborative policy. For this reason it can be useful to include an entropy term in the reward, as discussed in Section 2.5, so that the agent is explicitly encouraged to keep to explorative stochastic policies.

3.4.2 Complexity

A simple function class was used for the policies for ease of implementation and to maximise simulation speeds. The problem is quite simple, and a close-to-optimal policy is possible with a logistic regression, as shown in the previous section. However the question can be asked of whether better performance can be achieved with a more expressive policy function class.

The previous section showed that the policy does not converge completely to a deterministic one in the end. Since the environment dynamics are stationary, there must be an optimal deterministic policy. This is intuitively true because if the agent has a stochastic policy and achieves some maximum score once, it could do better by having the deterministic policy which always takes the actions leading to that score. That policy may not, however, be parameterisable under a logistic regression.

For this reason, the REINFORCE algorithm was tested again, but with a neural network instead of a logistic regression. Figure 3.9 shows the results, where REINFORCE with normalisation was used and the neural network had two hidden layers with 5 neurons and ReLU activation functions.

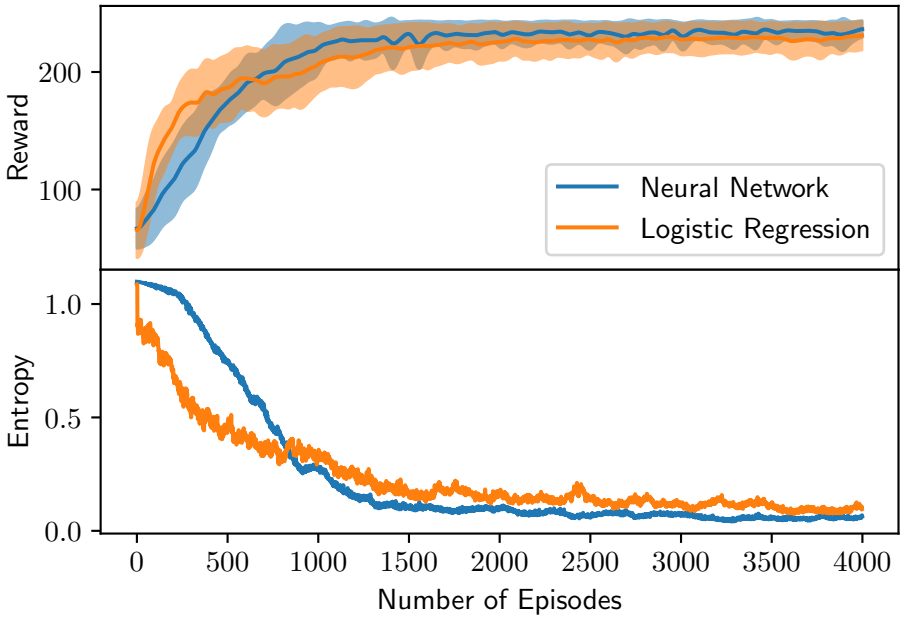


Figure 3.9: Comparison of the performance of a policy modelled by a logistic regression and a 2-hidden-layer neural network under REINFORCE.

The graph shows that the neural network outperforms the logistic regression. It trains more slowly at first as it has more parameters to tune, but later it appears that the additional degrees of freedom offered by the deeper network allow it to converge more quickly. With the better results come a more deterministic policy also. The simpler logistic regression uses stochasticity to allow it to perform well on average, even if it cannot model the optimal deterministic policy. The neural network has more degrees of freedom, and can get closer to the optimal deterministic policy.

For the multi-agent section of this project,we only consider policies under a logistic regression. This is due to computational constraints, but also as these policies serve well enough to provide a proof of concept for the techniques used. It should be noted from this section, however, that further improvements could be achieved with a richer policy function class. This is especially true as we start to consider increasingly complex multi-agent environments.

3.5 Conclusions

In this section, we have shown the ability of classical algorithms to find near-optimal solutions to a simple problem. The learning algorithms allowed the agents to learn policies which dealt cleverly with the problem of discrete velocities using PWM-like behaviour. The Actor Critic algorithm has been shown to be more effective than REINFORCE, although REINFORCE does also achieve good results. The problem of misspecification was encountered both in value function approximation and in policy approximation, with the solution in both cases to use a more expressive function class for approximation.

In the next section, we investigate the efficacy of these classical RL techniques in a multi-agent environment.

Chapter 4

Classical Techniques in a Multi-Agent Setting

4.1 The Environment

The environment for the multi-agent setting is the same as the single-agent environment with added inter-agent interaction. This occurs through the drag experienced by each agent. The drag coefficient is modelled as a function of distance from the rider in front, as shown in Figure 4.1. There is a large well in the drag coefficient in order to encourage cyclists to cycle close to the rider in front.

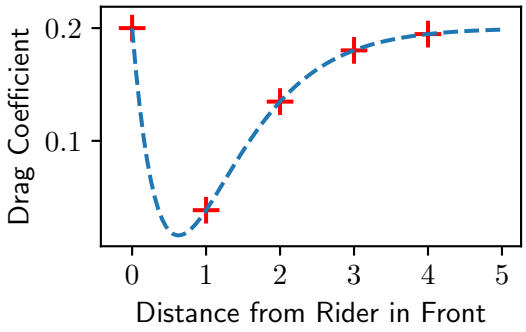


Figure 4.1: Variation in drag coefficient.

Each rider's state observation is also augmented with an observation of the nearest rider's relative position and velocity. If one rider runs out of energy, it drops out of the race. If a rider is the only rider left in the race, the states corresponding to the nearest rider's position and velocity are set to 0.

The reward is kept the same as in the single-agent case. Each agent is rewarded for the distance they move in a time step, and they only care about their own reward. The agents still benefit from collaborative policies, however, since moving in a pack improves all of their total scores. While this avoids the ambiguity introduced by group rewards, the complexity of a changing environment makes this problem more complex than the single case.

4.2 Results

Three of the algorithms were initially applied to this system - REINFORCE, the Actor Critic, and the Actor Critic with added entropy regularisation. The result is shown in Figure 4.2.

In this multi-agent case, REINFORCE significantly under-performs compared to the Actor Critic methods. This is to be expected - having the expected reward as a function of time step is even less accurate in a multi-agent framework. With the added variance introduced by more agents, the probability of a agent following similar trajectories drops. Estimating the value of a state by looking at rewards at that time step on past trajectories then becomes increasingly inaccurate.

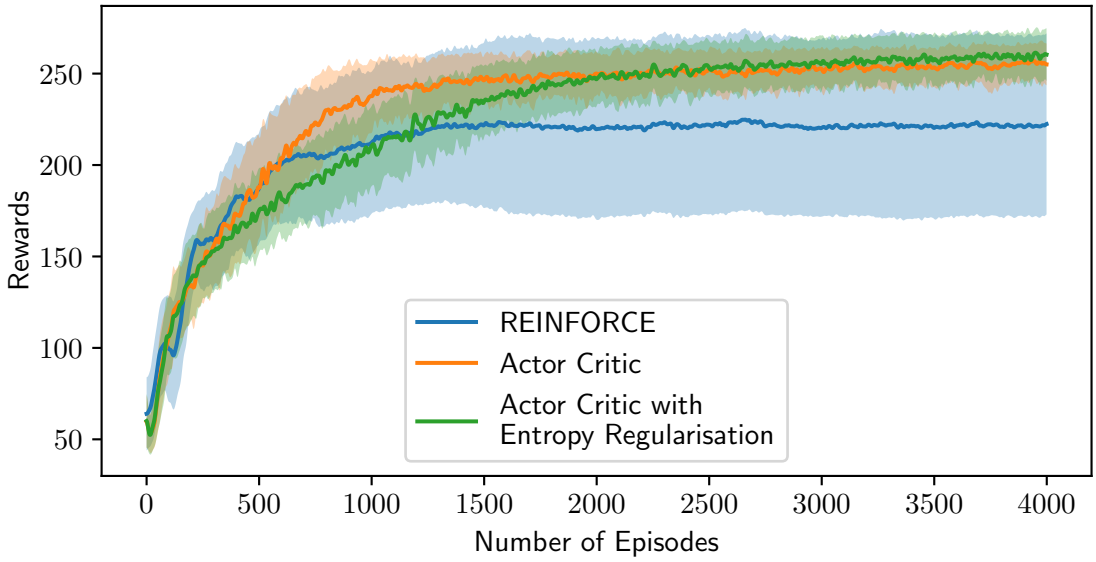


Figure 4.2: A comparison of the three key algorithms on a multi-agent system.

Figure 4.3 shows the four agents' performance for one of the seeds when REINFORCE is applied. Each colour represents a different agent. This demonstrates some of the key difficulties introduced by the non-stationary environment. Rider 0 can be seen to settle for a sub-optimal local maximum, out of which it cannot escape. All of the policies are very unstable, with Rider 3's score oscillating dramatically. Overall, the agents do less well than in the single-agent case due to the non-stationary environment.

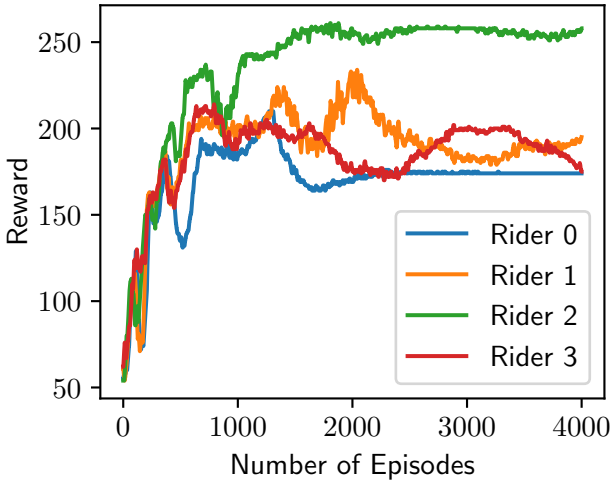


Figure 4.3: REINFORCE applied to a 4 agent system with individual rewards for one seed.

The Actor-Critic methods work considerably better, converging to an average reward of over 250 for each rider. The variances for these methods is also significantly lower than in REINFORCE, since in all of the seeds the agents learn at a similar rate. In REINFORCE, learning is more random, with rewards oscillating due to a less accurate performance baseline.

Adding an entropy regularisation term also helps in this case, albeit only slightly. The agents learn more slowly, since there is less emphasis on exploiting good policies, but ultimately learn better policies on average. This can be attributed to the fact that they are less likely to fall into sub-optimal policies for lack of experience of more optimal ones.

The agents do not, however, perform significantly better than in the single-agent case on average. While in the single-agent case, policies could achieve an average reward of 239, in this setting the highest average reward achieved was 251. The maximum possible reward on this environment is 368, which is well above the achieved rewards. While this would require complex policies implementing position switching in the pack as well as PWM velocity control, Figure 4.3 shows agents not being able to get near to the single-agent solutions achieved before. The complexity introduced by the non-stationary environment makes the agents incapable of maximising their own performance.

4.2.1 Conclusions

Naively implementing single-agent algorithms in this multi-agent setting leads to poor results, with agents often even failing to learn policies which they were able to learn in a single-agent case. This is in part due to the difficulty of the setting, where more complex policies would be required to achieve the maximum score, but it is also due to the problems discussed in Section 1.2.2.

In the following section, we simplify the environment in order to pinpoint where the difficulties arise, and implement changes in the algorithms to deal with them.

Chapter 5

Extensions of Policy Gradient Methods for Multi-Agent Systems

5.1 Introduction

The previous section demonstrates that classical single-agent reinforcement learning techniques are not sufficient to track a complex multi-agent problem. In this section, we propose extensions to the standard Actor Critic algorithm to deal with some of the problems that come up in multi-agent systems. Most of these approaches are not limited in use to multi-agent settings, and could also be helpful in single-agent environments.

5.2 Problem Simplifications

In this section, the problem is simplified in four key ways.

- The number of cyclists is reduced to 2. Along with the next simplification, this makes it easy to allow each agent to have a fully observable state space, and reduces the stochastic effects introduced by more cyclists.
- The race is changed so that an episode ends when one agent stops (due to lack of energy or the time limit). This means that each agent can have full access to the system state, without needing to deal with what happens when other agents are not there.
- The distance at which the agents experience minimal drag is reduced to 0. At this point the simulation becomes significantly less realistic, but since both agents can benefit from the reduction in drag at the same time, this removes the requirement for complicated policies where the agents need to switch positions in the pack in a coordinated way.
- The velocity granularity is decreased to 0.1. This makes it easier for agents to get to a target velocity without needing to use the PWM-like schemes we say in the single-agent case.

With these changes, each agent is able to have access to the full system state i.e. the positions, velocities, and energy levels of both riders. To allow them to form policies more effectively, the position and velocity of the other agent is given relative to their own. Note that the goal is still for the agents to travel as large a total distance as possible. The optimal reward in this setting is 403.

We also introduce neural networks to represent the value function from here. At this point, a polynomial kernel was not expressive enough to encourage good learning - better results were achieved with a two layer neural network with ten neurons in each layer. The networks were trained using the Adam optimiser [11].

5.3 State Distribution Drift

5.3.1 Problem Description

State distribution drift is a natural process in learning, where the state distributions encountered by the agents change as their policies develop and they take different actions. It can, however, cause problems when the range or mean of values taken by a particular feature vary dramatically. As discussed in Section 3.2.1, it is important for the networks used that the features fed in are normalised and centered. If the normalisation and centering factors are kept fixed during learning, this can quickly be violated.

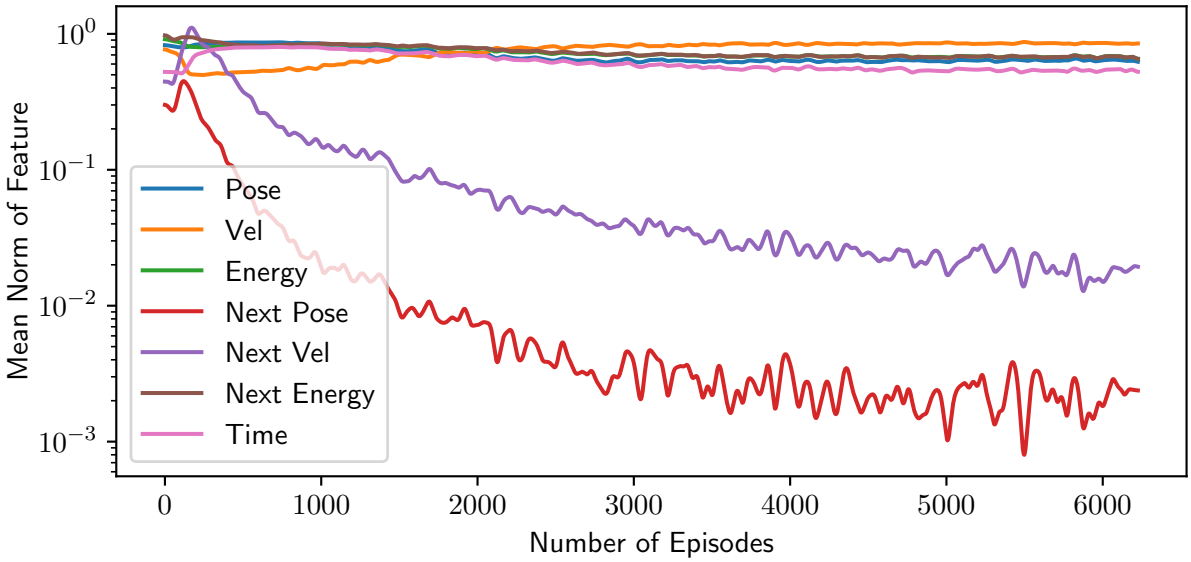


Figure 5.1: A demonstration of changing feature magnitudes due to state distrubtion drift.

To demonstrate state distribution drift, in Figure 5.1 we plot the mean magnitudes of the features over an episode on a logarithmic scale as an agent learns in our simplified setting. While the states used in the single-agent case stay roughly constant in magnitude, the relative velocity (Next Vel) and relative position (Next Pose) of the other rider fall heavily in magnitude as the episodes go on. This is good as it indicates the cyclists learning to stay together, but can cause problems when not dealt with properly.

The effects of the drift on learning are shown in Figure 5.2, where the shaded area now shows the range of values achieved at a given episode over the seeds. The Actor Critic is first applied to the environment with no alterations, then with a distance penalty (DP) and finally with the distance penalty and Gaussian position initialisation of the cyclists (GPI). The distance penalty is an extra term in the reward function which encourages the agents to stay close together by penalising average distance between them. Gaussian position initialisation is where the agents' positions are initialised according to a mean 0, standard deviation 0.5 Gaussian distribution instead of on the same spot.

Learning is slow in all cases. When the distance penalty is added, the agents are able to learn a fairly good policy which achieves a score of around 300 on average, but without it learning is much slower. The agents notably achieve a similar maximum score in all cases of nearly 400, which is close to optimal, but they also all regularly achieve a minimum score of 70. This behaviour is explained by looking at the cases where the agents do well and when they do badly.

When the agents do well, they both accelerate straight away to a velocity of 2 and stay at this velocity. They do badly if one agent slows down at any point, so that they are no longer in a pack, and they do not get the same drag reduction. In this case, they are travelling too fast, and quickly run out of energy.

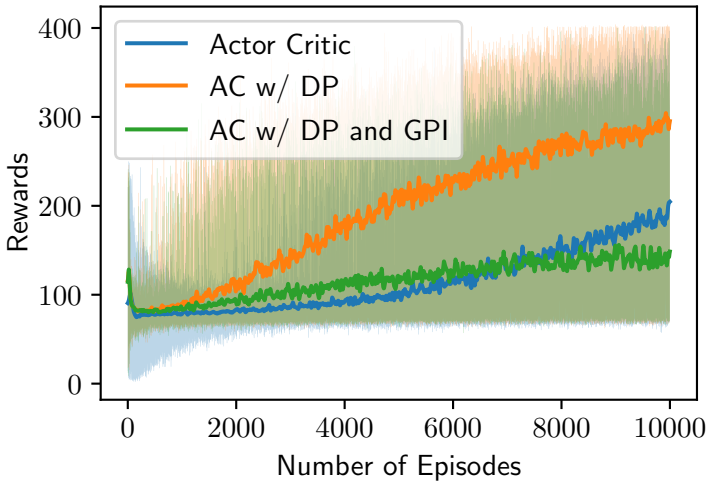


Figure 5.2: The effects of diminishing feature magnitudes on the performance of the Actor Critic algorithm.

This is an example of a non-robust policy. The policy can work in the specific setting where both agents start on the same spot, but if they fall out of position or are initialised randomly, they are unable to perform well at all. The agents are learning to track a single speed, which incidentally keeps them close together instead of explicitly learning to cycle close to one another. Even when they are encouraged to cycle close together with the distance penalty, they struggle to enforce this in their policies when noise is added to their initialisation.

The reason for this is that by the end, the magnitudes of the features required to track the other agent's position and velocity are so small that parameters which are two to three orders of magnitude larger than the others would be required to properly track them.

Although this problem has only come up in the multi-agent setting, it could conceivably come up in any RL problem, as an agent finds themselves in different states as learning goes on. For example in a video game which relies on a Convolutional Neural Network (CNN) as its state input, badly normalised inputs due to a dramatic change of colours between different levels could wreck havoc on the CNN. The problem actually applies to other features in our state space like time and distance, but it has only been a problem now because it applies so significantly to features which are necessary to achieve a good policy.

5.3.2 Approach

The solution used here is to have varying centralisation and normalisation parameters which update to keep the mean of the normalised states close to zero and their standard deviation close to one.

If the raw state is represented by a vector $s \in \mathbb{R}^S$, we denote the normalised state $s_n \in \mathbb{R}^S$, such that:

$$s_n = D(s - c), \tag{5.1}$$

where $c \in \mathbb{R}^S$ is a centering vector, and $D \in \mathbb{R}^{S \times S}$ is a diagonal matrix of normalisation parameters. Up until this point, D and c have been fixed, but now we allow them to change as the episodes go on. The aim is that the mean value of s_n be 0 over an episode, which clearly occurs when $c = \mathbb{E}_{t \in [T]} s_t$. Similarly, for s_n to have a standard deviation of 1, we want D to be the reciprocal of the standard deviations of each feature.

We cannot know the mean and standard deviation of the episode before we run it, so instead we estimate it using a weighted mean of previous episodes:

$$c_n \leftarrow \alpha_{drift} \mathbb{E}_{t \in [T]} s_t + (1 - \alpha_{drift}) c_{n-1}, \tag{5.2}$$

$$D_n \leftarrow \alpha_{drift} \cdot \operatorname{diag}\left(\mathbb{E}_{t \in [T]} [s_t - \mathbb{E}_{t \in [T]} s_t]^2\right)^{-1} + (1 - \alpha_{drift}) D_{n-1}, \tag{5.3}$$

where $\alpha_{drift} \in [0, 1]$ should be kept low here to keep the values from changing too quickly. This is a generalisation of the previous method - setting $\alpha_{drift} = 0$ means the parameters stay at their initial values as before.

To keep these parameter updates independent from the policy and value function updates done by gradient steps, the weights on the input dense layer of the policy are also changed. The first layer, h_1, (note this is the output layer for a logistic regression) is given by:

$$h_1 = W_0 s_n + b_0, \tag{5.4}$$
$$= W_0 D(s - c) + b_0, \tag{5.5}$$

where W_0 and b_0 are the weights and bias on the first layer respectively. If D and c are changed at the end of an episode, then the output of the policy will change even though there has not been a policy update. Therefore every time the centering vector and normalisation matrix are changed, the first layer weights also need to be changed, such that:

$$W_{0,n} = W_{0,n-1} D_{n-1} D_n^{-1}, \tag{5.6}$$
$$b_{0,n} = b_{0,n-1} - W_{0,n-1} D_{n-1} c_{n-1} + W_{0,n} D_n c_n, \tag{5.7}$$
$$= b_{0,n-1} + W_{0,n} D_n (c_n - c_{n-1}). \tag{5.8}$$

This means that the policy and value functions can work under the assumption of normalised states, but the normalisation updates do not affect the policy.

5.3.3 Results

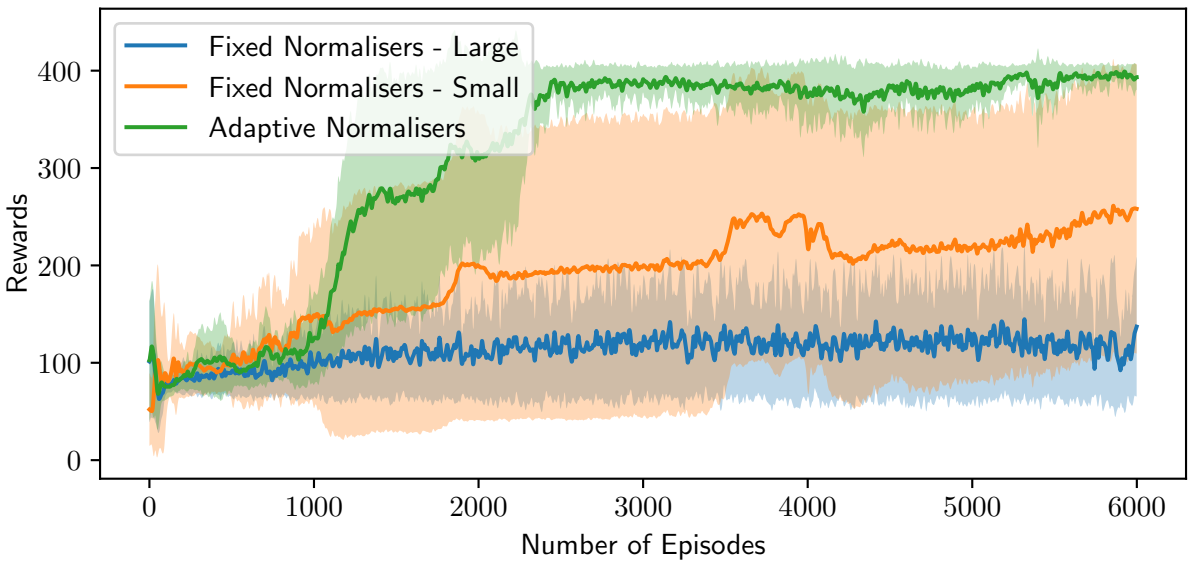


Figure 5.3: An adaptive normaliser compared to a large and small fixed normaliser.

Figure 5.3 shows the effects of varying the centering vector and matrix. A distance penalty has been implemented, as well as Gaussian position initialisation. Note that there are two lines corresponding to a fixed normaliser. In one, the normaliser is initialised such that at the beginning of the run the features are all normalised. This is the example shown in Figure 5.1, where some "normalised" features have disappearingly small

magnitudes at the end. There is also a smaller initialiser shown, such that D is taken as the inverse of the average magnitudes at the end of a run of episodes.

The adaptive normaliser clearly outperforms the fixed normaliser in both cases. With the large normaliser, the actor is never able to learn a good policy, since important the features for collaborative policies (relative distance and velocity of other rider) are too small in magnitude to affect the policy. With the small normaliser the results are better. In some of the seeds, the actors learn good policies eventually, but because some states have such large magnitudes to begin with, in many of the seeds the agents fall into sub-optimal policies before they can reach this stage. The adaptive normaliser successfully keeps the features normalised throughout so that an effective policy is always found.

5.4 Non-Stationarity

5.4.1 Problem Description

Up until now, the problem of non-stationarity in a multi-agent system has not been directly attacked. Rather, single-agent techniques have been applied to the multi-agent setting, with small alterations to see if they can still work in a non-stationary environment. The aim of this section is to provide the basis for a specific multi-agent approach geared towards dealing with this non-stationarity.

The non-stationarity problem can be seen by inspecting the value function from the perspective of a single-agent.

Recall that in the single-agent case, the state and state-action value functions are defined thus:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}\left[\mathbb{E}_r r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^\pi(s')\right], \tag{5.9}$$

$$Q^\pi(s,a) = \mathbb{E}_r r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^\pi(s'), \tag{5.10}$$

$$= \mathbb{E}_r r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a),a'\sim\pi(a'|s')}Q^\pi(s',a'), \tag{5.11}$$

where we have allowed for stochastic rewards. In our multi-agent case, the state value function needs to take into account the actions of all of the agents. Here, we denote $\boldsymbol{a} \in \mathcal{A}^n$ the set of actions, and $\boldsymbol{\pi}$ the set of policies of each agent. We also denote $\bar{\boldsymbol{a}}_i$ the set of all actions except for that of agent i, and similarly for $\bar{\boldsymbol{\pi}}_i$.

$$V^\pi(s) = \mathbb{E}_{\boldsymbol{a}\sim\boldsymbol{\pi}(a|s)}\left[\mathbb{E}_r r(s,\boldsymbol{a}) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,\boldsymbol{a})}V^\pi(s')\right], \tag{5.12}$$

$$= \mathbb{E}_{a_i\sim\pi_i(_i|s)}\left[\mathbb{E}_{\bar{\boldsymbol{a}}_i\sim\bar{\boldsymbol{\pi}}_i(\bar{\boldsymbol{a}}_i|s)}\left[\mathbb{E}_r r(s,\boldsymbol{a}) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,\boldsymbol{a})}V^\pi(s')\right]\right], \tag{5.13}$$

$$= \mathbb{E}_{a_i\sim\pi_i(a_i|s)}\left[\mathbb{E}_{r,\bar{\boldsymbol{a}}_i\sim\bar{\boldsymbol{\pi}}_i(\bar{\boldsymbol{a}}_i|s)}r(s,\boldsymbol{a}) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,\boldsymbol{a}),\bar{\boldsymbol{a}}_i\sim\bar{\boldsymbol{\pi}}_i(\bar{\boldsymbol{a}}_i|s)}V^\pi(s')\right]. \tag{5.14}$$

This takes the same form as Equation (5.9), with both inner expectations taken over the actions of the other agents as well as the environment statistics. However, while the environment statistics (i.e. the transition matrix and the probabilities of getting a certain reward) are stationary, the policies of the other agents are not. This is where the stationarity problem arises - estimating this time-varying value function is difficult.

5.4.2 Approach

This problem can be averted by using a value function valued on the actions of all agents as well as the state. We define a multi-agent Q-function, $Q : \mathcal{S} \times \mathcal{A}^n$, as:

$$Q^\pi(s,\boldsymbol{a}) = \mathbb{E}_r r(s,\boldsymbol{a}) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,\boldsymbol{a}),\boldsymbol{a}'\sim\boldsymbol{\pi}(\boldsymbol{a}'|s')}Q^\pi(s',\boldsymbol{a}'). \tag{5.15}$$

In this value function, the actions of the agents are all passed in as inputs so that no expectation is taken over the non-stationary other agents' policies. This is the approach taken by Gupta et al. [12] and Lowe et al. [13]. The former considers cooperative environments, where the goal of all the agents is the same, while the latter considers mixed cooperative-competitive environments, where some agents can have different goals. In the cooperative setting, one Q-function can be learned so the agents can all share a critic. In Lowe's paper, each agent learns its own Q-function so that they can each pursue different goals.

In this paper, we will use an approach closer to Lowe's but instead of learning a Q-function, each critic learns a function which maps a state and the actions of all other agents to a value. Similarly to how in the single-agent case a critic learns a V-function instead of a Q-function, here we allow the critic to learn a more simple value function, independent of its own agent's actions. We denote this function $U : \mathcal{S} \times \mathcal{A}^{n-1}$, as it is neither a V-function which depends on state alone, or a Q-function which depends on the action of the agent. It is defined as follows:

$$U^{\pi}(s, \bar{\boldsymbol{a}}_i) = \mathbb{E}_{a_i \sim \pi(a_i|s)} \left[\mathbb{E}_r r(s, \boldsymbol{a}) + \mathbb{E}_{s' \sim \mathbb{P}(s'|s,\boldsymbol{a}), \bar{\boldsymbol{a}}'_i \sim \hat{\boldsymbol{\pi}}(\bar{\boldsymbol{a}}'_i|s')} U^{\pi}(s', \bar{\boldsymbol{a}}_i) \right]. \tag{5.16}$$

We note that this looks just like the single-agent V-function shown in Equation 5.9, except for the additional requirement of taking an expectation of the U-function over the other actions at the end. We can do this since the critic has the benefit of hindsight. Although at rollout the agent cannot see the actions of the other agents, during training the critic is able to view all the actions taken along with their probabilities in order to learn the U-function.

With this U-function, the advantage is now defined as:

$$A(s, a_i) = r(s, a) + \mathbb{E}_{\bar{\boldsymbol{a}}'_i \sim \hat{\boldsymbol{\pi}}(\bar{\boldsymbol{a}}'_i|s')} U^{\pi}(s', \bar{\boldsymbol{a}}_i) - U^{\pi}(s, \bar{\boldsymbol{a}}_i). \tag{5.17}$$

This is the reward at the current step plus the value of the next step minus the expected reward at the current step given the actions of the other agents. We note that the baseline is not a function of the action of agent i, so we can still introduce it without violating the requirements for Corollary 1.1.

This sort of approach would not be possible with Q-learning, and this is one of the chief reasons PG algorithms were used here. The Q-function learnt during training could not be applied at rollout if it depended on the actions of other agents. Here, the U-function is only used during training to teach the actor, so this is not a problem.

5.4.3 Results

Figure 5.4 shows a comparison of the rewards of the individual random seeds between when the classic Actor Critic is used, and when a U-function is learnt instead. The rewards are smoothed with a Gaussian filter of standard deviation 50 to improve readability of the graphs. Gaussian position initialisation is also applied, to increase the difficulty of this problem, and is applied to all experiments after this point.

With the V-function many of the seeds get stuck in a sub-optimal policy which, as discussed in Section 1.2.2, is a common problem in non-stationary settings. Learning a U-function instead means that all of the seeds manage to escape this sub-optimal solution, and achieve close to optimum policies of reward around 400. The performance of the agents in these lines also oscillate less, and reach a higher final average reward as a result of this. This improvement in stability comes with a slight cost in speed, with individual agents taking longer to learn the more complex U-function.

Figure 5.5 shows the mean of the ten seeds for the classic and U-function Actor Critic, with an additional line representing the classic Actor Critic with entropy regularisation. In the case of entropy regularisation, the exploitation parameter was varied at a pre-set schedule: $\lambda = 0.9$ until episode 8000, after which $\lambda = 1$.

When entropy regularisation is introduced to the classic Actor Critic, all of the seeds are able to escape the sub-optimal solutions. Although they initially learn slightly more slowly since they are being encouraged to explore more, they quickly catch up, and end up learning faster than the other methods. Since the rewards oscillate more under the non-stationary value function, the final average reward is slightly lower than when the U-function is used.

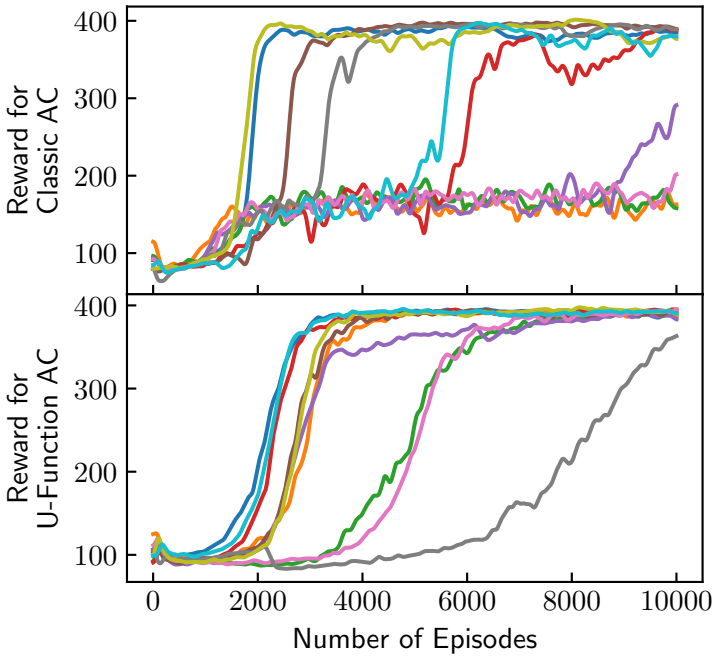


Figure 5.4: The effect of tracking a non-stationary value function on the learning of individual random seeds.

This shows that entropy regularisation is also an effective way to improve learning in a more complex environment, and the natural extension is to use it in conjunction with the non-stationary value function. This proved not to be easy, with the exploitation parameter schedule used for the classic Actor Critic actually slowing down learning with the U-function. There are clearly benefits to using entropy regularisation, but setting the amount of entropy regularisation at different stages in the learning process is non-trivial. We consider this problem in the next section.

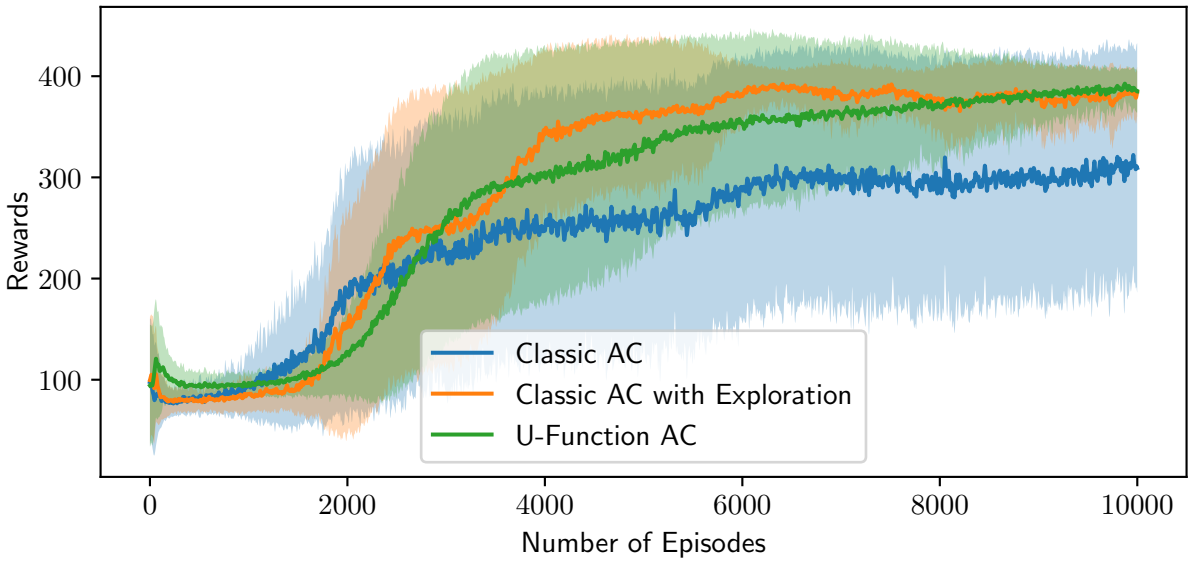


Figure 5.5: Learning our defined U-function instead of a V-function.

5.5 Automatic Hyperparameter Tuning

5.5.1 Problem Description

As the algorithms employed become more complex, more tunable hyperparameters are introduced. As seen in the previous section, their optimal values are not always obvious. In general, learning happens best when there is more exploration to start, and then more exploitation later on. However the exploration could take on an infinite number of different profiles, and a profile that works well for one set of simulations will not necessarily work well for another. We can test a large number of profiles, but this then becomes a tedious process of hyperparameter tuning with the entire learning process needing to be repeated every time. This inspired the idea of automatic hyperparameter tuning.

This is effectively a meta-learning problem. We are trying to learn a set of learning hyperparameters (including the exploitation rate), such that the agents are able to learn optimal policies quickly. We frame the process in terms of a coach here which is effectively another, higher-level, agent tasked learning the hyperparameters.

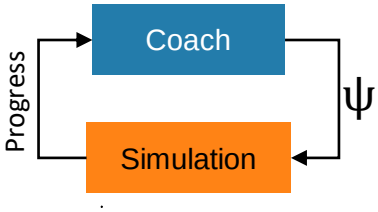


Figure 5.6: The framework considered for hyperparameter learning.

The framework is shown in Figure 5.6. The learning process we have considered up until now is encompassed in the simulator which can run a certain number of episodes of learning and output some performance statistics of the agents. The simulator can be started at a state, which is described by: the value and policy functions of the agents at the start; the set of hyperparameters that are fed to it; and a seed which sets the randomness in the simulation. In this paper, simulations run for a fixed amount of time, either 200 or 400 episodes.

The coach runs a simulation, observes the performance of the agents and then from this observation decides which hyperparameters to feed in next time. We denote the set of hyperparameters $\psi \in \Psi$, which for the moment consist only of the exploitation rate λ. For each simulation run there is some optimal set of hyperparameters $\psi^\star$ which would lead to the most progress. The goal of the coach is to minimise regret, defined here as:

$$\text{Regret} = \sum_{\text{simulations}} \text{Progress}(\psi^\star) - \text{Progress}(\psi), \tag{5.18}$$

while also ensuring that the agents achieve maximum reward by the end. It wants to run as few simulations possible with the best possible hyperparameters while achieving an optimal solution.

5.5.2 Approach

We identify two key difficulties in this problem, and discuss our approaches to each one below.

Quantifying performance

The first difficulty is quantifying performance and progress. Performance should be an indication of how close the agents are to reaching an optimal policy, and progress can be taken as a change in performance. However defining the sub-optimality of a policy is non-trivial.

We could consider the average reward of the agents as an indication of performance but this neglects the effect of exploration. Comparing the average rewards of the classic Actor Critic method with and without exploration in Figure 5.5, which are replicated in the top half of Figure 5.7, we see that applying exploration leads to a period where the reward is lower than that without exploration. This is necessary since the agents are exploring more states in order to achieve higher rewards later, so in reality we can consider the explorative agent to be outperforming the other agent in this period.

To address this, we define a performance score P, which encourages entropy as well as reward. The underlying motivation is that a more random policy which achieves the same average reward as another is better. This is for two reasons related to the fact that a more stochastic policy samples a wider set of trajectories.

Firstly, if the policy sees a wider spread of trajectories, it probably sees a wider spread of rewards, which implies that the maximum reward is likely to be higher than that of the less random policy. An extreme example of this is one policy which only samples one trajectory and another which samples multiple. Unless all the trajectories sampled by the latter yield the same reward, it will have a trajectory which yields a higher than average reward. By encouraging the more random policy to converge to a deterministic policy, it can converge towards a policy which only samples the maximum reward trajectory, and outperforms the other policy.

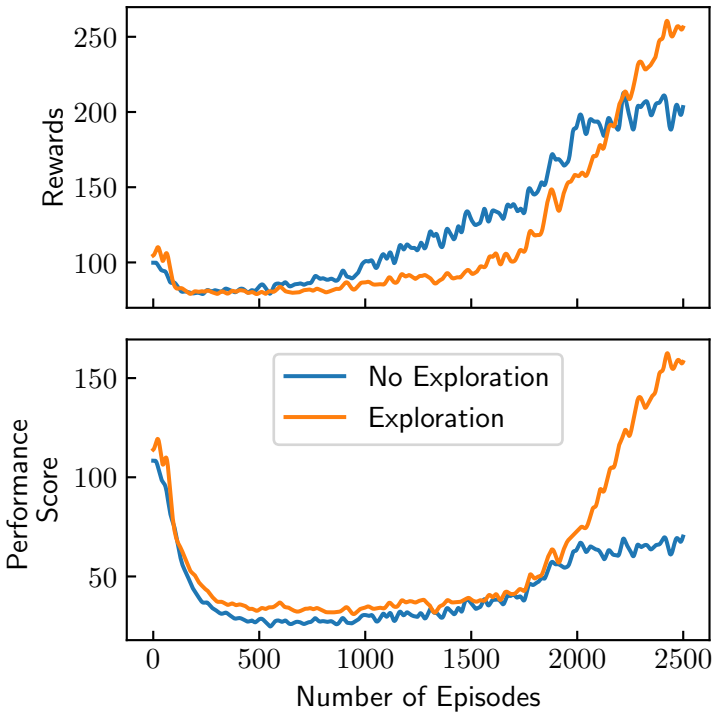


Figure 5.7: A demonstration of our performance metric, which factors in the potential of a policy as well as current reward.

Secondly, a high entropy policy is more able to adapt to changes. Since it experiences more trajectories, if one of those trajectories proves to be better than another due to some change in the environment or a change in the perceived value of states, it is more likely to be able to adapt to this change. A more deterministic policy might never see the trajectories affected by the change, so will be unable to adapt.

With this in mind, we can define P as the product of average reward and entropy. In this paper, we actually add an additional episodic dependency on the H term, such that over time the solution converges to a high reward one instead of an explorative one. The performance score is then given by:

$$P = r \cdot H^{1-\frac{e}{E}}, \tag{5.19}$$

where e is the episode and E is the total number of episodes. Using this parameter, we see in Figure 5.7 that the performance of the more explorative trajectory through training is always higher than that of the less explorative trajectory.

It might seem that this is a convoluted way of redefining the exploitation parameter we defined earlier - we have just stated it in terms of an arbitrary episode entropy dependency parameter instead of the exploitation parameter λ. Our results show that this is not the case. The performance objective is more clear than the exploitation parameter, as we know that by setting the entropy dependency to 0, we will encourage the agents to try to achieve the best policy, which may or may not involve total exploitation. Defining the entropy profile at a higher level allows the coach to do all the work in setting the right exploitation value to achieve the desired trade-off between entropy and reward.

Zero-shot learning

The more difficult problem is that once we have reached the optimal solution, we do not want to run another set of simulations with other hyperparameters to see if we could reach that optimal solution faster. We effectively have a zero-shot learning problem, where we want to achieve the best parameters at each stage without ever having experienced that learning stage. Since the optimal hyperparameters change at every stage of learning, this can seem like an impossible problem.

One way to approach this would be to run a large set of simulations in parallel at each stage of learning, with hyperparameters covering a representative range of values. At the end of each simulation, the performance of each agent can be compared, and the value functions and policies of the best performing simulation can be stored. Then another set of simulations is run in parallel with each agent starting with the optimal policies and value functions from the last simulations, and the process is repeated until an optimal policy is found. This way, if an ϵ-grid is made which covers all possible parameters, the policy at each stage of learning is guaranteed to be within $\frac{\epsilon}{2}$ of the best parameter.

The result of this method is shown in Figure 5.8 for a single seed, where a grid was taken covering λ values from 0 to 1 with a separation of 0.1 between each. The rewards shown are those achieved by the highest scoring worker for each simulation. The problem with this strategy is that most of the simulations will be run with parameters far from the optimal values, incurring large regret, and the number of simulations to run increases exponentially with the number of hyperparameters. Moreover, the learning done in the majority of the simulations is wasted when only the best parameters are taken.

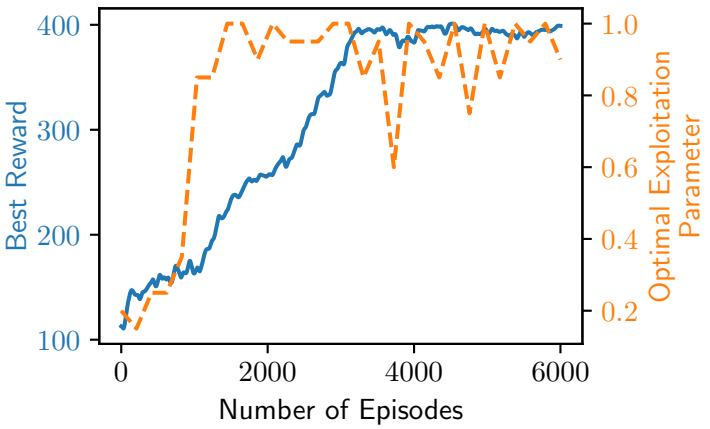


Figure 5.8: The optimal exploitation rate for a run over a single seed.

Despite being clearly inappropriate to use in reality, the results from this experiment give estimates for $\psi^\star$, showing that the optimal parameter value varies over time. The optimal exploitation rate starts low at the start, but then increases rapidly as the agent starts to find a good policy.

The optimal exploitation rate does not appear stationary when the policy has been learnt. We identify two reasons for this. The first is a matter of chance. When a close-to-optimal policy has been learnt the effect of reducing the exploitation rate slightly does not affect the agents' learning heavily. There are then a set of many parameters which yield high performance, and the one which yields the best score is down to which actions the agent happened to take as opposed to the specific learning parameters used.

The other reason represents desired behaviour in our implementation. Once the agents do converge to good policies, the exploitation has a tendency to drop down, since there is still incentive for the coach to encourage high entropy. The effect of this is that the agents search for another policy which yields similar rewards but with more entropy. This sort of behaviour could allow agents to escape from a sub-optimal policy by exploring more when they start to converge to a deterministic policy. By the end, the exploitation rate goes back up since the coach is less incentivised to encourage randomness, and the agents are able to hone in on the best of the policies.

Final approach

Instead of running a grid of parameters at each time step, we only run two. We make the assumption of temporal correlation between the optimal parameters (which seems justified for the exploitation parameter from Figure 5.8), so that we only need to search in the area around the best parameters from the last set of simulations to find the optimal set for the next simulation.

Algorithm 3 shows the hyperparameter tuning algorithm for a single hyperparameter. $\hat{\pi}_i$ denotes the policy of the highest performing simulation run at i, and similarly for V and ψ. $\delta\psi$ is some initial small step size taken for the parameter.

We run two simulations in parallel at each step, with a different hyperparameter value. One takes the best hyperparameter from the last set of simulations, the other takes a different value. The performance of each is compared at the end of the simulations, and then the process is repeated. To choose the other hyperparameter, we do a multiplicative line search. This means that if we found that we travelled in the right direction at the last hyperparameter change, we travel again in the same direction, but twice as far. If we found that the optimal parameter remained the same as the last run, we start to search in the opposite direction. The objective of this is to be able to quickly traverse the hyperparameter space, since the optimal parameter can change dramatically as seen in Figure 5.8.

Algorithm 3 The hyperparameter tuning algorithm for a single hyperparameter.

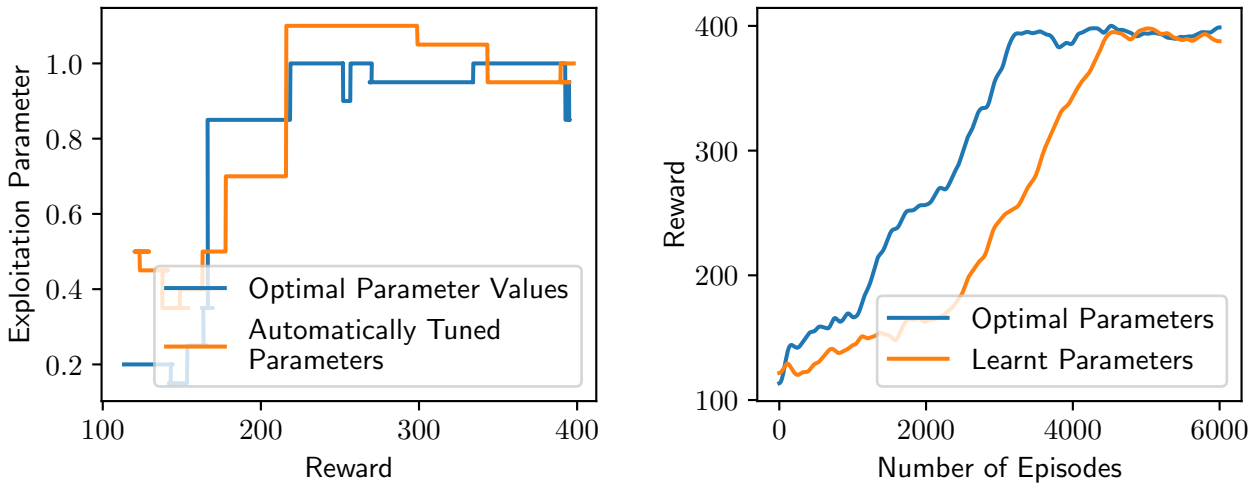
Initialise $\hat{\pi}_0$, $\hat{V}_0$ randomly
Initialise $\hat{\psi}_0$, ψ_{new} to starting values
for i in a number of simulations **do**
  Initialise two simulations with $\pi_i = \hat{\pi}_{i-1}$ and $V_i = \hat{V}_{i-1}$ and the same seed
  Set the hyperparameters of one to $\hat{\psi}_{i-1}$, and the other to ψ_{new}
  Run the simulations, get $\hat{\pi}_i$, $\hat{V}_i$, $\hat{\psi}_i$ from the simulation with the highest value of P
  $\Delta\psi \leftarrow \hat{\psi}_i - \hat{\psi}_{i-1}$
  if $\Delta\psi \neq 0$ **then**
    $\psi_{new} \leftarrow \hat{\psi}_i + 2 \cdot \Delta\psi$ ▷ Multiplicative increase in displacement
  else
    $\psi_{new} \leftarrow \hat{\psi}_i - \text{sign}(\Delta\psi) \cdot \delta\psi$ ▷ Search in the other direction
  end if
end for

5.5.3 Results for Single Hyperparameter Tuning

Figure 5.9 shows the result of running the algorithm for the same seed as that used to find the optimal exploitation rates. In 5.9a, the parameter values are plotted against the reward to emphasise that the best parameters are a function of the progress, not number of episodes passed. The automatic tuning algorithm is run with parameters being updated every 400 episodes.

Despite this slower update rate, the parameters are able to closely track those in the optimal simulation. The exploitation rate is initialised a bit too high, and initially starts to decrease towards the optimal values. When required, the multiplicative step lengths allow it to increase quickly to track the optimal rate. The result is seen in Figure 5.9b, where when the exploitation rate is learnt automatically, learning follows a very similar trajectory to with the optimal hyperparameters with a delay which can be attributed to poor initialisation.

During the tuning process, the exploitation rate set by the coach goes above 1. This was not a value tested in the ϵ-grid used for the 'optimal' parameters as it is technically outside of the allowed bounds for the parameter. A value of $\lambda > 1$ actually discourages entropy, encouraging the policy to converge to a deterministic one. This



(a) Comparing the learnt parameters to the true optimal parameters.

(b) Comparing the reward with learnt parameters to the reward with the optimal parameters.

Figure 5.9: Comparison of learning with automatic hyperparameter tuning to using the optimal parameters.

is an interesting result, which showcases one of reinforcement learning's best strengths - the coach is able to act in a way which would be non-intuitive to a human, achieving higher performance than if it were copying human behaviour.

In Figure 5.10, we show the results of automatically tuning the exploitation rate against having no exploration. By having a greater initial exploration rate, the automatically tuned simulations avoid the heavy dip in performance seen in the results up until this point. They are also able to reach their final value more quickly as they have seen more trajectories leading up to this point. There is a period half way through the learning process where the automatically-tuned simulations appear to lag behind the simulations with no exploration. This can be attributed to our maximum entropy approach in the performance metric - at this period, the coach's aim is not to maximise reward, but to maximise reward and entropy. By the end, when all of the emphasis is on reward, the coach in all of the seeds is able to encourage the agents to converge to approximately optimal policies.

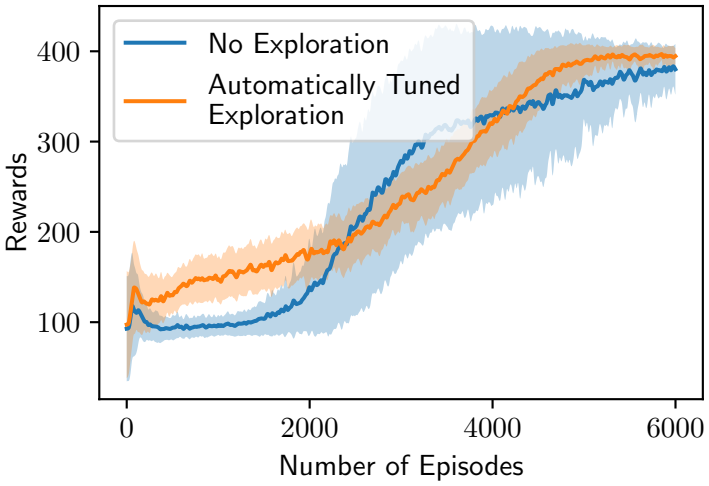


Figure 5.10: Comparing an automatically tuned exploitation rate and no exploration.

5.5.4 Extension to Other Hyperparameters

There is no reason to stop at the exploitation rate when deciding which hyperparameters to tune. Indeed, an appeal of this method in a multi-agent setting is being able to alter the reward function to encourage other high-level behaviour. The example used in this section is encouraging the agents to stay close together in the race with a distance penalty. In earlier sections, we applied this as a tool to make the learning task easier for agents, but here we consider it as another hyperparameter which the coach can learn to tune. The coach can increase or decrease the penalty, which is multiplied by the average distance between the riders and subtracted

Algorithm 4 The hyperparameter tuning algorithm for multiple hyperparameters.

for a number of simulations i **do**
  Pick the next hyperparameter to tune, ψ_j
  $k \leftarrow 1$
  while $k \leq 2$ and $\Delta\psi \neq 0$ **do**
    Run two simulations to test a change in the hyperparameter
    Observe the result
    $\Delta\psi_j \leftarrow \hat{\psi}^{\star}_{j,i} - \hat{\psi}^{\star}_{j,i-1}$
    $k \leftarrow k + 1$
  end while
end for

from each of their rewards.

This way of learning is highly intuitive and relevant to the real world. A coach might judge the performance of a team on a number of observations from games they play, and then will instruct them to act in a specific way to improve performance. In the same way, here we arm the coach with a way of observing performance and a set of instructions they can give, like telling the cyclists to stay close together. The advantage here in comparison with the real world is that the coach is able to run different simulations in parallel with exactly the same control conditions to see which strategies work best.

The problem with introducing more hyperparameters to tune is that it becomes more difficult to compare the performance changes due to changes in each parameter. One way of tuning multiple parameters would be to create a grid, of dimensionality $2^{|\Psi|}$, of parameter values to test. Then every combination of parameters changing and staying the same can be observed in parallel and the best can be chosen. This comes at the cost of running many more simulations in parallel, which we want to avoid.

Instead, we opt to tune the parameters one at a time. A high-level overview of the algorithm is shown in Algorithm 4. The idea is to cycle through the parameters, never staying on one for too long. For every set of simulations, we test a change in the same way seen in Algorithm 3 and save the result. If the change is successful, we try to change this hyperparameter again, but we do not do this more than twice to avoid focusing on one hyperparameter at the expense of others.

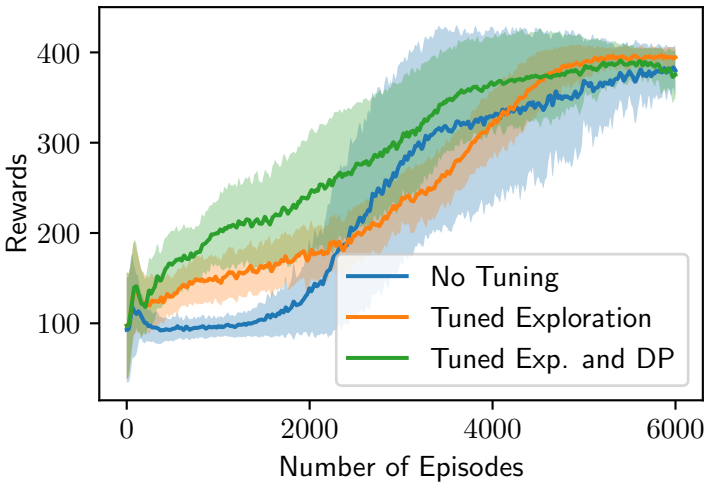


Figure 5.11: Tuning the distance penalty and exploitation parameter simultaneously.

The results of tuning both the distance penalty (DP) and the exploitation parameter are shown in Figure 5.11. In most cases, the ability to tune the distance penalty means that learning happens far more quickly. In one case, it takes a bit longer to tune the hyperparameters which explains the higher variance in the line when compared to only tuning the exploitation parameter. The problem is that with more hyperparameters, it can take a longer time to test changing an important hyperparameter in one direction. In this case it takes at least three sets of simulations for one direction of a hyperparameter to be checked again, and potentially more if some directions are checked twice.

This problem could be averted by opting to tune both hyperparameters at once or to check both directions at once, at the cost of having to run more simulations in parallel. However, even with two sets of simulations running parallel, this method is difficult to justify in terms of sample complexity. Although the tuning methods outperform the method with no tuning in terms of global episodes run, when the fact is taken into account that the tuning methods need to run twice as many simulations to get these results they become less attractive. In the final addition to the algorithm, we attempt to rectify this with off-policy learning.

5.5.5 Reducing the Sample Complexity with Off-Policy Learning

Approach

The sample inefficiency comes from the fact that when one set of agents share their policies and value functions with the other set, all of the learning done by the other set of agents is rendered useless. We can address this with off-policy learning. The general idea is that one set of agents can look at the experiences of the other agents and learn from them under their own learning hyperparameters.

Off policy learning is often used in RL as a way to reduce sample complexity. It is used in algorithms like the Soft Actor Critic on a buffer of past experiences, from which the agent can later learn. Here, we use the same principal but use the unseen set of experiences from the other parallel simulation.

The formulas for off-policy learning are simple to calculate. For the value function, if we want the value labels under our policy π, we want to estimate:

$$V^{\pi}_{label} = \mathbb{E}_{a\sim\pi(a|s)}\left[r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^{\pi}(s')\right], \tag{5.20}$$

but our samples are actually taken under a different policy π'. To go between the two, we say that:

$$\mathbb{E}_{a\sim\pi(a|s)}\left[r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^{\pi}(s')\right] = \sum_{a}\pi(a|s)\left[r(s,a) + bE_{s'\sim\mathbb{P}(s'|s,a)}V^{\pi}(s')\right], \tag{5.21}$$

$$= \sum_{a}\frac{\pi(a|s)}{\pi'(a|s)}\pi'(a|s)\left[r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^{\pi}(s')\right], \tag{5.22}$$

$$= \mathbb{E}_{a\sim\pi'(a|s)}\left[\frac{\pi(a|s)}{\pi'(a|s)}\left(r(s,a) + \mathbb{E}_{s'\sim\mathbb{P}(s'|s,a)}V^{\pi}(s')\right)\right]. \tag{5.23}$$

We call $\frac{\pi(a|s)}{\pi'(a|s)}$ the importance weight - it places more importance on actions which are more likely under the current policy than the sampling policy. To get the value labels, we do the same as before, but multiply by this importance weight:

$$V^{\pi}_{label} = \frac{\pi(a|s)}{\pi'(a|s)}\left(r(s,a) + V^{\pi}(s')\right). \tag{5.24}$$

In the case of policy updates, the equation is similar - we simply need to multiply the advantage with the importance weight for the action. This is derived in Degris et al. [14], where the assumption is made that the gradient of the Q-function with respect to the policy parameters θ is small. As discussed in the paper, we can still guarantee policy improvement with this form.

The same algorithm as earlier can then be applied with an added step. When the optimal policy is decided, the tuples representing the experiences of the other simulation are used to do a batch of off-policy learning on the value function and policy. The batch size used for these is larger so that there are less updates from these experiences. The reasoning for this is that these experiences are less relevant to the optimal simulation's progress, so should have less weight.

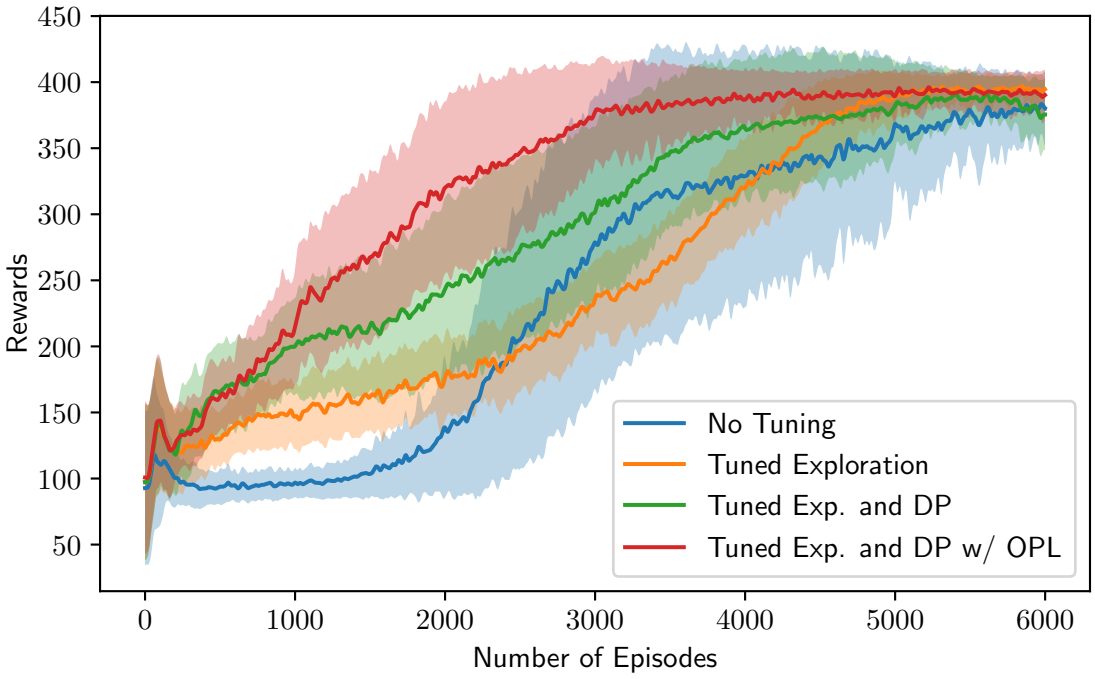


Figure 5.12: A comparison of all of the automatic tuning techniques.

Results

Figure 5.12 shows the results of applying off-policy learning (OPL) to the same problem as above. The learning happens considerably faster than without learning off-policy, and the learning is notably smoother, with the line taking a concave form. The increased speed of learning is as expected since more samples are being utilised. The shape indicates that all of the trial seeds learn around a similar time, so the average shape is closer to their individual learning curves. Without off-policy learning, some of the agents take longer to learn which attributes to the varying curvature of their average. Sampling from another set of trajectories is a form of exploration, so we expect all of the seeds to be faster at finding their way to the optimal solution in this way.

The learning is not, however, twice as fast as without off-policy learning, which would represent a full recovery of the sample complexity lost in running parallel simulations. This indicates that the off-policy samples are not as valuable as the on-policy samples, which is not entirely surprising. The on-policy samples give important information about the states visited by the current policy, while the samples from the off-policy trajectories could not be as relevant.

Figure 5.13 shows the trajectories taken through the hyperparameter space by two different seeds during learning. The solid lines mark changes in parameters while the dashed lines mark changes that were tried but were deemed detrimental. Seed 0 is slower to learn, as seen in the lower part of the figure, while Seed 1 very quickly comes to an optimal solution.

Seed 1's quick learning can be explained by the fact that the coach quickly learns to increase the distance penalty, while in the case of seed 0, the coach decides not to increase the distance penalty to start, and then spends time optimising the exploitation parameter. By the time the decision comes round again to increase the distance penalty, the other seed has made significantly more progress. This demonstrates the risk of tuning multiple parameters at once - the coach can focus its energy on optimising the wrong parameters and miss out on potential improvements from another. The only negative effect of this is that the learning is slowed slightly, however. By the end both coaches have increased the exploitation rate and distance penalty to encourage the agents to reach a good solution.

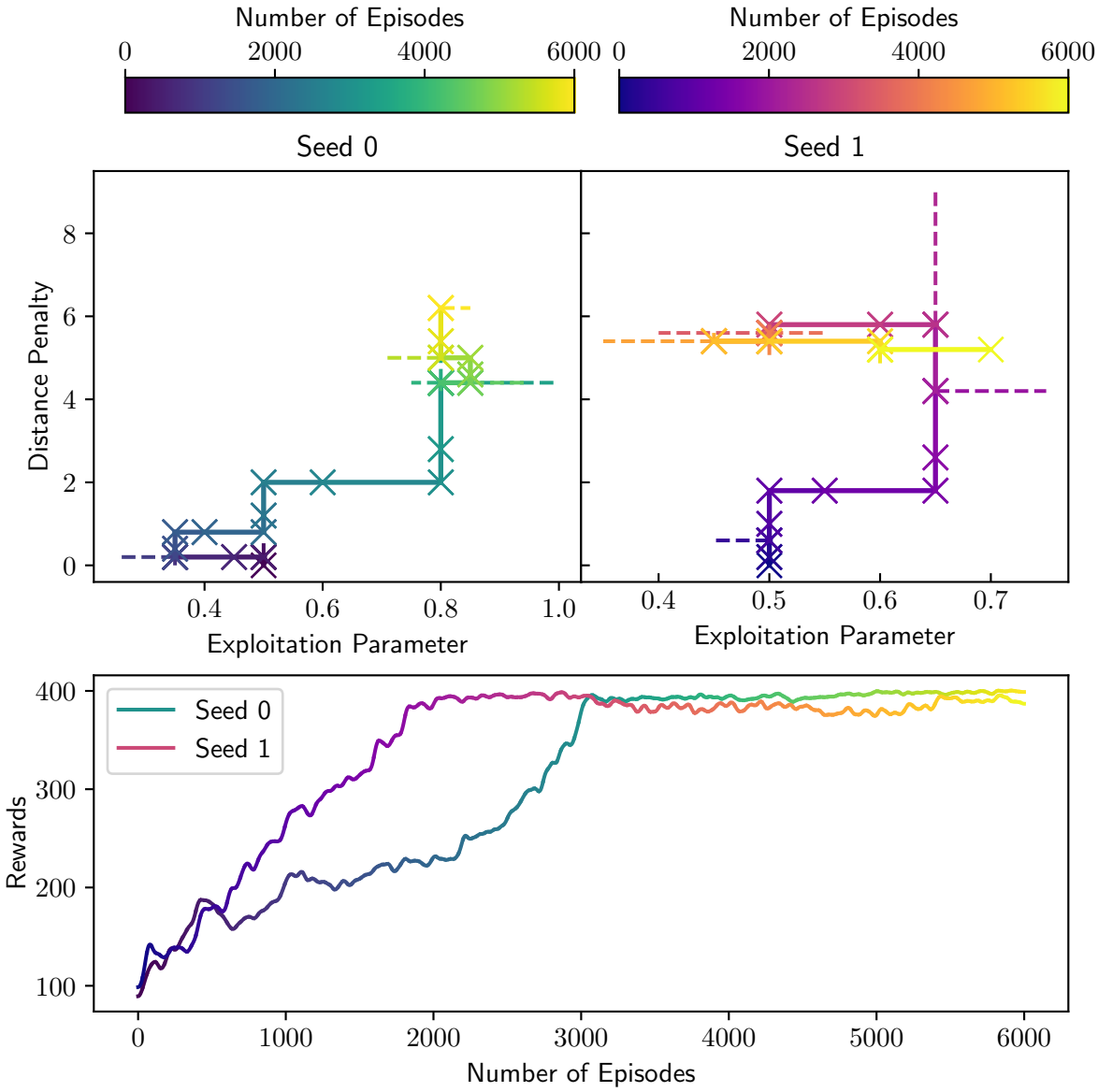


Figure 5.13: Two different seeds' trajectories through hyperparameter space.

The agents in Seed 1 arrive quickly at the optimal solution, when there is still a heavy emphasis on maximising entropy in the performance score. The coach therefore lowers the exploitation rate for a period. This leads the agents to explore more, at the expense of average reward. This is the same behaviour as seen when finding the optimal parameters in Figure 5.8. The agents are encouraged to explore once they reach a solution, only to settle for the best solution later on in training. While in this case there is no benefit to doing this, in a case where the policies the agents find are sub-optimal, this behaviour could help the agents to escape through exploration.

Figure 5.14 shows a comparison of the methods considered here, but with the horizontal axis representing the true number of episodes run (accounting for the doubling effect in the hyperparameter tuning model). The varying normaliser has been applied to all models since performance is poor without it.

The automatic tuning model learns marginally faster than the others, and tuning the exploitation rate early on allows it to avoid the heavy dip in performance the others experience. The final policies achieve marginally higher reward than those of the other algorithms. It is important to note here that the use of this method is

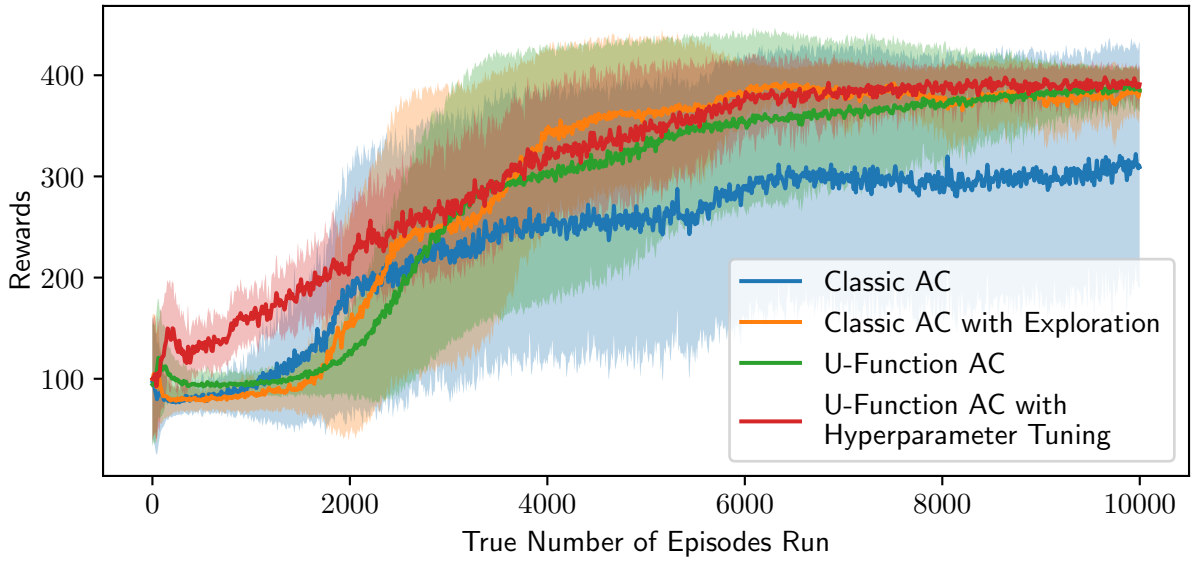


Figure 5.14: A comparison of the techniques considered in the multi-agent setting, with a varying normaliser applied.

not in encouraging fast learning. While we offset it with off-policy learning, the requirement to run two sets of simulations in parallel is heavily detrimental to the sample complexity of the algorithm.

The usefulness is instead in the ability of the coach to guide the agents into learning complex policies by observing high-level statistics over episodes. While this was not necessary in this problem to achieve a close to optimal policy, we see that the variance of the performance with the automatic tuning is far lower than the other cases. Allowing the coach to tune the learning parameters during training can take more samples, but also ensures that all of the seeds manage to find the optimal solution at a predictable rate. In other problems the ability to tune hyperparameters which alter the learning of agents could allow them to reach policies that would be difficult to learn otherwise.

5.5.6 Reintroducing More Agents

As a final experiment, we increase the number of cyclists back up to four, to see how the algorithms deal with a more complex environment. Figure 5.15 shows the results, where again the true number of episodes run is plotted on the x-axis. As well as the classic Actor Critic and U-function Actor Critic with no exploration, we show the U-function Actor Critic with an informed exploration schedule. This means that we used an average of the exploitation parameters set by each of the coaches at each time step for this set of seeds.

While in the more simple two-agent setting, the benefits of applying the automatic hyperparameter tuning algorithm are not clear cut, here they are dramatic. The classic Actor Critic algorithm struggles to escape from a set of sub-optimal policies which yield a reward of around 150. In one of the seeds, the agents' performance actually deteriorates around episode 4000 leading to a drop in performance. Only at the end of learning, around 9000 episodes in, does one of the seeds show evidence of escaping the sub-optimal policy. The same exploration schedule used in the two-agent case actually caused the Actor Critic agents to learn less well, so was not included here.

The U-function Actor Critic leads to the agents learning more slowly, with an even lower average reward. They do learn, but very slowly, with the agents starting to speed up their learning process towards episdoe 9000.

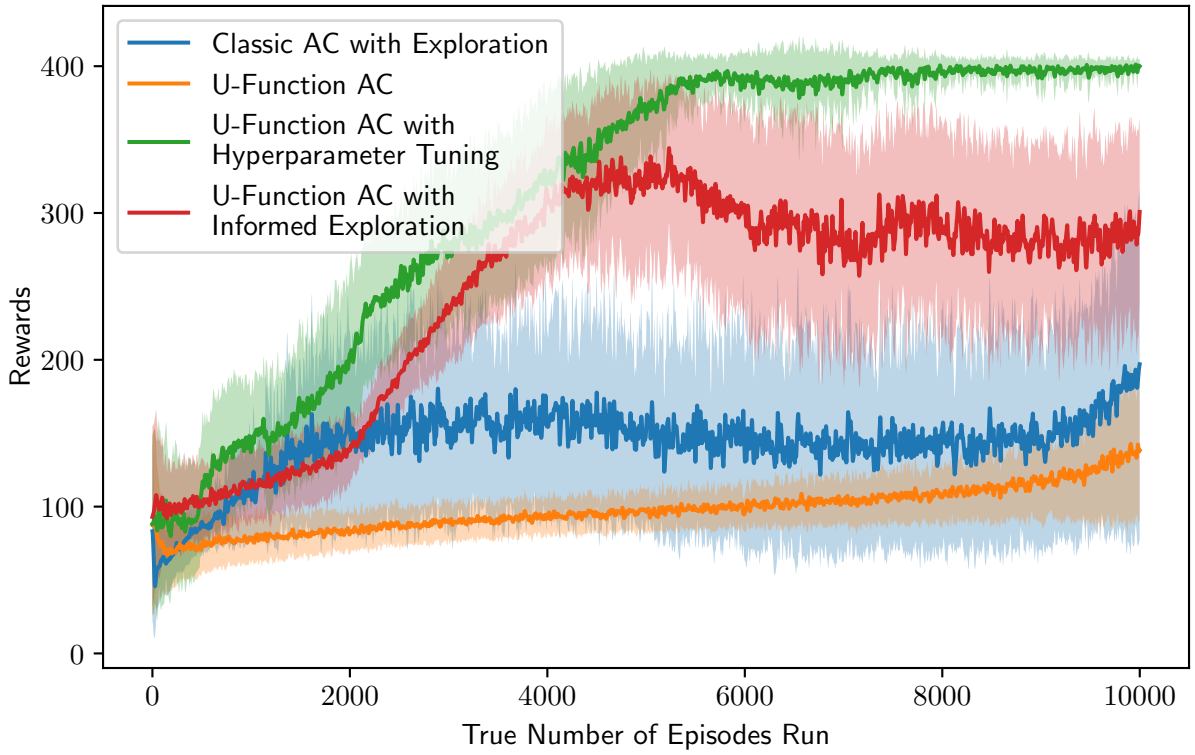


Figure 5.15: A demonstration of the effectiveness of our approach when the number of agents in the environment is increased.

Including automatic hyperparameter adjustment makes the learning significantly faster, with the agents all achieving approximately-optimal policies around 5500 total episodes into learning. After this, the performance drops slightly as the coaches encourage the agents to search for other solutions, but the final average policy reward is 401, which is very close to the optimal value of 403.

The U-function with informed exploration fares better than without any exploration, with the agents learning relatively quickly initially. However, around episode 4000, the average performance stops increasing when the informed exploitation rate starts to increase. This is for two reasons - firstly, at this point the automatically-tuned agents have distance penalties imposed by their coaches which make sure they stay together and without these the informed exploration agents struggle to converge to the same solution. Secondly, since each agent learns at a different rate, using an average exploration schedule does not work well for all of them. This shows the importance of tuning the hyperparameters as a function of current progress in learning, further justifying our approach.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

This project has broadly been split into two parts. The first part examined traditional RL techniques: looking at their strengths in a single-agent setting, and ultimately their failures in a non-stationary multi-agent setting. In the second part, we have tried to find solutions to the problems that come up with the classical techniques, and designed a more robust algorithm designed for multi-agent settings.

The use of a varying normaliser was shown to be crucial in allowing any learning algorithm to deal with state distribution drift. Additionally, the technique used in multi-agent literature to deal with the non-stationarity by learning a value function which takes in the actions of other agents was implemented in a more efficient form by learning a U-function instead of a Q-function. The exploration-exploitation problem was formulated in a novel way, giving the algorithm the capability to automatically tune the parameters in order to track a maximum-entropy-based performance heuristic. The tuning algorithm was shown to be adaptable to different hyperparameters, such that a coach can shape learning by encouraging different behaviour in the agents' reward functions. Finally, the sample complexity of the above approach was reduced with off-policy learning, allowing the algorithm to learn faster than other algorithms despite effectively running half as many episodes of learning.

The final algorithm shows a way in which episode-level statistics, like the distance between cyclists, can be used to inform learning decisions for the agents. This framework is similar to the way in which a coach will observe the performance of their players, and give them instructions on how to improve based on their observations. To our knowledge, this is the first implementation of such a framework.

6.1.1 Use of Coach in MARL

While the coach framework could be used in any RL setting, we believe it is particularly helpful in cooperative and mixed cooperative-competitive MARL settings. In such problems, the framework allows for group behaviour to be directly encouraged by the coach in a more direct way than the normal learning process of the agents.

In the example of our cyclists, the coach can test out different training regimes, encouraging them to stay together or not at different times and see the results. For the agents to learn this behaviour without a coach, they need to explore and discover the value in such strategies, which can take significantly longer and can even be impossible without the correct encouragement. When the cyclists cycle closer together, the fastest need to slow down, and the average reward often decreases. While significant exploration is needed without a coach to learn to do this, the coach has the ability to explicitly offset the drop in reward with a reward bonus for staying together, allowing the agents to more easily see the benefits in the strategy. This means that in our final experiment, the coach is able to guide the agents to policies they are unable to achieve without help.

The coach could be used in a similar way in other multi-agent settings. By varying the learning hyperparameters it can evaluate different strategies which would be difficult for the agents to implement without explicit incentives.

6.1.2 Bias in Our Setting

While the final results are promising, we acknowledge that there is some bias introduced by allowing the coach to apply a distance penalty. We know the desired behaviour is for the agents to stay close together, and in a sense allowing the coach to directly encourage this behaviour gives it an unfair advantage.

Even though this is true, our results still show a proof of concept: that enforcing group behaviour in this way can be very effective. The coach does have the option to discourage the cyclists to stay together (by setting a negative distance penalty), so we have not directly told it the emergent behaviour we want to see. Even when only allowed to tune the exploitation parameter, the coach can encourage significantly faster learning. In other settings there may not be a hyperparameter which has such an immediate effect on learning, but by giving the coach a range of hyperparameters to tune we could allow it to decide for itself if by varying them in a specific way performance can be improved.

6.1.3 Extensions to Other Settings

To extend the approach to other settings, we would need to arm the coach with hyperparameters to tune. Beyond the obvious exploitation parameter, other hyperparameters would need to be chosen specifically for the environment. We envision this could be done in a similar way to with the distance penalty - through information provided during simulations.

OpenAI Gym [15] is a collection of RL environments on which many algorithms are tested. In an OpenAI environment, every step in a simulation returns some information as well as the reward. In the case of our cyclist this information was the average distance between cyclists, and in the same way a bonus term could be added to the agents' reward function in other environments to encourage or discourage certain statistics about the simulation.

A simulation could be made to feed back all manor of information to the coach and through this the coach could encourage or discourage different behaviour. We note that in choosing which information is provided to the coach, we implicitly introduce bias by telling the coach which behaviour it can encourage. For this reason, it would be important to allow the coach to encourage a range of different statistics. By doing so, we give the coach the freedom to choose which behaviour to encourage and can even allow it to act in a way which might not be intuitive to a human, as seen in Figure 5.9a.

6.2 Further Work

6.2.1 Testing on Other Environments

As discussed above, the approaches used here could be extended to other, more complex environments. The approach of automatically tuning hyperparameters was shown to be especially effective in the more complex, four-agent systemso it would be interesting to see how it fares against other state-of-the-art algorithms on difficult RL problems. By testing the approach on different environments against existing baselines, we could could get a better idea of how effective it is.

The coach framework could be tested on both single-agent and multi-agent problems. In particular, although we did not consider mixed cooperative-competitive environments here, by giving each agent its own critic we have reserved the ability for the approaches to be used in settings where agents do not all have the same goals.

Such cases might call for each group of agents working together to have an individual coach and it could be interesting to see if there is a viable way to implement this.

6.2.2 Improvements on the Algorithm

The algorithms used here are not optimised, and we believe their performance could be improved with more careful design. As well as implementation improvements like using a Trust region approach, and using a richer function class for the actor, we identify two key areas for improvement.

The parameter search algorithm

In particular if more hyperparameters were introduced to allow the coach to develop more complex strategies, the parameter search algorithm would need to be made more efficient at deciding which hyperparameters to tune. It could do this with a variable wait time on some parameters, such that if it decides they have little effect they can be ignored for a certain period. The coach could also be allowed to change the number of simulations run at once, going down to one simulation when it believes the hyperparameters are optimal, and increasing the number of simulations to more quickly search the hyperparameter space at other times.

The parameter optimisation could even be attempted over one stream of simulations alone. This could be more viable in complex environments where learning more slowly. By comparing the rate of learning in one batch of episodes to the rate of learning in the last batch, the impact of the parameters could be inferred. This method did not show any success in our environment where the performance of agents could change rapidly, but in an environment where learning follows a more predictable trajectory this approach could deal with the sample inefficiency problem by only ever running one simulation at a time.

Off-policy learning

The off-policy learning was applied in a very simple way here, by feeding all of the experience samples in one go. Other algorithms, like the Soft Actor Critic, sample from an experience buffer whenever they take a normal gradient step, as well as the immediate experience. This could be done here, by simply adding the experiences from the other seed into a large buffer of experiences which can be used together with on-policy experiences. By mixing the two sets of experiences in this way, the algorithm would always be learning from a range of experiences instead of learning from all of the off-policy experiences in one batch.

6.2.3 Computational Efficiency

A topic which has largely been left undiscussed in this project is computational efficiency. It is often not considered in theoretical RL publications, but is an important factor for techniques to be implemented in the real world.

The main limiting factor when increasing the number of agents in simulations was the increasing complexity of taking an expectation over the U-function. To do this, the value function needs to be applied to all of the states for every possible set of actions, of which there are $|\mathcal{A}|^{n-1}$, where n is the number of agents (note that in Lowe et al [13] since a Q-function is used this is $|\mathcal{A}|^n$ instead). This was manageable under the small action space, but as the number of agents increases the problem can quickly become intractable.

The problem is not unique to our approach, and would need to be addressed to apply techniques to systems with larger action spaces or more agents. An area for future work would be to try to find ways around this curse of dimensionality.

One approach would be to only feed through possible sets of actions where the probability of taking those actions is larger than a threshold. Especially when the policies become more deterministic, this could dramatically reduce the number of possibilities which need to be passed through the value function.

Another approach would be to allow the value function to drop its dependence on certain agents. In our example, it is only the nearest cyclists that really have an effect, so the value function could be made to only depend on their actions. There then comes a trade-off between putting more agents' actions into the value function to make the value function more stationary, against removing the dependence on certain agents to reduce computational complexity. This is a non-trivial problem, as agents would also need to be able to decide which other agents' actions are most important to factor in, but it would be an important one to consider for practical MARL problems, in particular ones where the number of agents in the environment can vary.

Appendix A

Proof of Lemmas

The lemma used in the policy gradient derivation is proved below, as well as its corollary.

Lemma 1 *For a function* $f : \mathcal{S} \to \mathbb{R}$, $\mathbb{E}_{a\sim\pi(a|s)}[\nabla_\theta \log \pi(a|s) f(s)] = 0$.

Proof. The expectation can be expressed as

$$\mathbb{E}_{a\sim\pi_\theta(a|s)}[\nabla_\theta \log \pi(a|s) f(s)] = \sum_{a\in\mathcal{A}_t} \nabla_\theta \log \pi(s,a) f(s) \pi(a|s), \tag{A.1}$$

and since f is independent of a_t, it can be moved out of the summation:

$$\mathbb{E}_{a\sim\pi_\theta(a|s)}[\nabla_\theta \log \pi(a|s) f(s)] = f(s) \sum_{a\in\mathcal{A}_t} \nabla_\theta \log \pi(a|s) \pi(a|s). \tag{A.2}$$

Appealing again to identity Equation (2.5), and noting that $\pi(s,a)$ is a probability distribution over actions so must sum to 1, this becomes:

$$\begin{aligned}
\mathbb{E}_{a\sim\pi_\theta(a|s)}[\nabla_\theta \log \pi(a|s) \cdot f] &= f(s) \sum_{a\in\mathcal{A}_t} \nabla_\theta \pi(a|s), && \text{(A.3)}\\
&= f(s) \cdot \nabla_\theta \sum_{a\in\mathcal{A}_t} \pi(a|s), && \text{(A.4)}\\
&= f(s) \cdot \nabla_\theta 1 = 0. && \text{(A.5)}
\end{aligned}$$

□

Corollary 1.1 *In an MDP, with some function* $f : \mathcal{S} \to \mathbb{R}$ *which is independent of action at* t, $\mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s) f(s)] = 0$.

Proof. The expectation can be expressed as

$$\mathbb{E}_{\tau\sim\pi}[\nabla_\theta \log \pi(a|s) f(s)] = \int_\tau \nabla_\theta \log \pi(a|s) f(s) p(\tau)\, \mathrm{d}\tau. \tag{A.6}$$

The probability of a trajectory $p(\tau)$ can be expanded out as shown below:

$$\begin{aligned}
p(\tau) &= p(s_t, a_{t-1}, s_{t-1}, a_{t-2}, \ldots), && \text{(A.7)}\\
&= p(a_{t-1}|s_t, s_{t-1}, a_{t-2}, \ldots) p(s_t, s_{t-1}, a_{t-2}, \ldots), && \text{(A.8)}\\
&= p(a_{t-1}|s_{t-1}) p(\tau'), && \text{(A.9)}\\
&= \pi(a_{t-1}|s_{t-1}) p(\tau'), && \text{(A.10)}
\end{aligned}$$

where in the third line, we have used the fact that the action only depends on the current state, and substituted $\tau' = \{s_t, s_{t-1}, a_{t-2}, \ldots\}$. Substituting this back into Equation (A.6) shows that this is just an extension of Lemma 1:

$$\mathbb{E}_{\tau\sim\pi}[\nabla_\theta \log \pi(s,a) f(s)] = \int_\tau \nabla_\theta \log \pi(s,a) f(s) p(a_{t-1}|s_{t-1}) p(\tau')\, \mathrm{d}\tau, \tag{A.11}$$

$$= \int_{\tau'} \sum_{a_{t-1}} \nabla_\theta \log \pi(s,a) f(s) p(a_{t-1}|s_{t-1}) p(\tau')\, \mathrm{d}\tau', \tag{A.12}$$

$$= \int_{\tau'} p(\tau') \sum_{a_{t-1}} \nabla_\theta \log \pi(s,a) f(s) p(a_{t-1}|s_{t-1})\, \mathrm{d}\tau', \tag{A.13}$$

$$= \int_{\tau'} p(\tau') \mathbb{E}_{a_t \in \mathcal{A}_t}[\nabla_\theta \log \pi(s,a) f(s)]\, \mathrm{d}\tau', \tag{A.14}$$

$$= 0. \tag{A.15}$$

□

Bibliography

[1] C. Dann and E. Brunskill, "Sample complexity of episodic fixed-horizon reinforcement learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 2818–2826.

[2] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine learning proceedings 1994*. Elsevier, 1994, pp. 157–163.

[3] Y. Bai and C. Jin, "Provable self-play algorithms for competitive reinforcement learning," *arXiv preprint arXiv:2002.04017*, 2020.

[4] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural networks*, vol. 21, no. 4, pp. 682–697, 2008.

[5] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[6] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[10] I. Karpathy, "Deep reinforcement learning: Pong from pixels," May 2016, accessed: 12/10/2019. [Online]. Available: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[12] J. K. Gupta, M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning," in *International Conference on Autonomous Agents and Multiagent Systems*. Springer, 2017, pp. 66–83.

[13] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in neural information processing systems*, 2017, pp. 6379–6390.

[14] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *arXiv preprint arXiv:1205.4839*, 2012.

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[16] S. Levine, "Lecture slides for cs 285 deep reinforcement learning," 2019.

[17] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, 1993, pp. 330–337.

[18] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing exploration in reinforcement learning with deep predictive models," *arXiv preprint arXiv:1507.00814*, 2015.

[19] L. Weng, "Policy gradient algorithms," *lilianweng.github.io/lil-log*, 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html