Creating Agents with Tunable Behaviours using Multi-Objective Deep Reinforcement Learning

David O'Callaghan (19233706)
School of Computer Science
National University of Ireland, Galway

Supervisor
Dr. Patrick Mannion

In partial fulfillment of the requirements for the degree of
MSc in Computer Science (Artificial Intelligence)
August 25, 2020

DECLARATION I, David O'Callaghan, do hereby declare that this thesis entitled "Creating Agents with Tunable Behaviours using Multi-Objective Deep Reinforcement Learning" is a bonafide record of research work done by me for the award of MSc in Computer Science (Artificial Intelligence) from National University of Ireland, Galway. It has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification.

Signature: David O'Callaghan

Acknowledgements

I would like to take this opportunity to sincerely thank my project supervisor, Dr. Patrick Mannion, for his support and guidance over the past few months. Without him, this work would not have been possible.

I would also like to thank Dr. Michael Schukat and Dr. Enda Howley for their hard work in successfully coordinating the thesis module.

Abstract

In this study, a framework is presented to train agents whose behaviours can be tuned during run-time in a multi-agent environment. The training method uses techniques from Multi-Objective Reinforcement Learning; during training, the agents are presented with a range of objective preferences that correspond to different types of behaviours. As a result, the agents learn multiple behaviours simultaneously and, during the execution stage, an agent's preferences over the objectives can be tuned to dynamically change their behaviour, without the need for retraining. Through the experimentation presented in this thesis, it is empirically shown that a single deep neural network model can represent agent behaviours to a varying degree of cooperativeness or competitiveness in a multi-agent setting.

Keywords: Reinforcement Learning, Multi-Objective Decisions, Multi-Agent Systems, Artificial Intelligence, Machine Learning, Deep Learning

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence. 3, 4

CNN Convolutional Neural Network. 7, 8, 26

DNN Deep Neural Network. 6

DQN Deep Q-Network. 6, 8, 22, 25–27, 64

DST Deep Sea Treasure. 22, 23

MARL Multi-Agent Reinforcement Learning. 10, 11

MAS Multi-Agent System. 10, 11

MDP Markov Decision Process. 4, 5, 11, 13, 24

MLP Multi-Layer Perceptron. 7

MOMDP Multi-Objective Markov Decision Process. 13, 14, 42

MORL Multi-Objective Reinforcement Learning. 13, 14, 22, 56

NN Neural Network. 6–8, 19, 51

ReLU Rectified Linear Unit. 7, 18

RL Reinforcement Learning. 3–6, 10, 11, 13, 14, 19, 25, 54–56

SG Stochastic Game. 11, 42

Chapter 1

Introduction

1.1 Motivation

The standard approach to developing a Reinforcement Learning agent is to learn some fixed behaviour that will allow the agent to solve a sequential decision making problem. If, however, the developer wants the agent to behave differently, the agent normally has to be partially or completely retrained. A framework has been presented previously by Källström and Heintz (2019b) to train agents whose behaviour can be tuned during run-time using Multi-Objective Reinforcement Learning methods. In this framework, each set of objective preferences (scalarisation weights) corresponds to some different type of agent behaviour and the agent is then trained to learn all desired behaviours simultaneously. After the agent is trained, the weights can be adjusted to dynamically change the agent behaviour, without the need for retraining. The aim of this study is to build on this framework and test if it can be applied to more complex environments with larger state-spaces and multiple agents.

1.2 Research Questions

The research questions addressed by this work are as follows:

1. What are the effects of refactoring the tunable agents framework by Källström and Heintz (2019b) to meet the definition of linear scalarisation from Roijers et al. (2013)? (RQ1)
2. Does this same framework scale to more complex environments? (RQ2)
3. Can agents achieve tunable behaviours in a multi-agent setting? (RQ3)

1.3 Structure of Thesis

This thesis is structured as follows. In Chapter 2, a literature review of the related work and background material in the area is presented. Chapter 3 outlines the common methodology used for the two main studies of this thesis. Benchmarking experiments that were conducted are described in Chapter 4. A replication study of the Gathering experiment conducted by Källström and Heintz (2019b) is presented in Chapter 5 along with some further results and analysis. Chapter 6 describes the Wolfpack experiment, where the framework for designing tunable agents is extended to a more complex environment and a multi-agent setting. The conclusions of this thesis are stated in Chapter 7.

Chapter 2

Background and Literature Review

This chapter outlines the background material and related work carried out previously in this research topic in the format of a literature review.

2.1 Agents and Reinforcement Learning

Jennings (2000, p. 280) defines an agent as a "*computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives*". The concept of an agent has a pivotal role to play in many areas of the field of Artificial Intelligence (AI); they have been used in applications as simple as email filters and as complex as air-traffic control systems (Jennings et al., 1998).

Traditionally, the behaviour of agents was constructed manually by domain experts. However, advances in the field of Reinforcement Learning (RL) has made it possible for agents to learn behaviours through interactions with their environment in applications where it would be otherwise difficult to handcraft a

suitable behaviour.

RL is a type of machine learning that is used to enable an agent to learn how to solve sequential decision-making problems. The agent learns by receiving a reward after performing an action in an environment that represents how good or bad the action was (Kaelbling et al., 1996). RL has led to many great success stories in the field of AI; for example, it has been used to develop agents that can play Atari games at a super-human level (Mnih et al., 2015) and to develop a multi-agent system for efficient control of traffic-light signals (Arel et al., 2010).

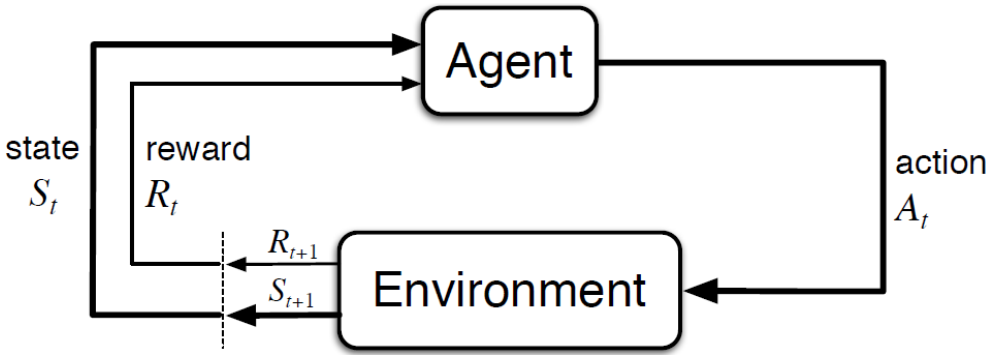

Figure 2.1: Interaction mechanism between the agent and environment for a Markov Decision Process (Sutton and Barto, 2018)

Sequential decision-making problems are most commonly modelled as Markov Decision Processes (MDPs). MDPs consist of a set of environment states S, a set of actions that the agent can make A, a transition function T and a reward function R. An MDP is therefore defined as the tuple $\langle S, A, T, R \rangle$; if $s \in S$ is the current state of the agent, and it takes action $a \in A$, it will transition to state $s' \in S$ with probability $T(s, a, s') \in [0, 1]$ and receive a real-valued reward $r = R(s, a, s')$ (Sutton and Barto, 2018). Figure 2.1 is a depiction of how the agent interacts with the environment in an MDP.

An agent decides which action to take based on its policy π, which is effectively a mapping from environment states to agent actions. The goal of an agent in an

MDP is to find the policy that maximises its expected return at each time-step t; the return is defined as $g_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ where $\gamma \in [0, 1)$ is the discount factor that controls how much the agent values future rewards. Value-based RL algorithms are a common approach for solving MDPs. These methods involve quantifying how good a particular state is using a value function. The value of a state when following policy π is $V^\pi(s) = \mathbb{E}[g_t|\pi, s_t = s]$. Value-based methods therefore try to find the optimal policy $\pi^* = \arg\max_\pi V^\pi(s)$ (Sutton and Barto, 2018).

RL algorithms can be divided into two classes: model-based and model-free methods. In model-based methods, a model of the environment (i.e., the transition function) is required to find an optimal policy. In contrast, model-free methods don't require a model of the environment and rely on learning through experience. Model-free methods require exploration of the environment in order to gain enough knowledge of the state-space; this introduces the exploration-exploitation trade-off: at each time-step, the agent must decide whether to exploit its current knowledge of the environment through a greedy action or to explore the environment with a different action. The most simple approach to this is the ϵ-greedy strategy, where the agent either takes a random action with probability ϵ (exploration) or takes the action with the highest expected return with probability $1 - \epsilon$ (exploitation) (Wiering and Van Otterlo, 2012).

A popular value-based RL method is the Q-learning algorithm, which is a model-free, off-policy learning algorithm that estimates the action-value function $Q(s, a)$ of an MDP by applying the following update rule at time-step t:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right) \tag{2.1}$$

where α is the learning rate and the value function is computed by $V(s) = \max_a Q(s, a)$ (Watkins, 1989). The full Q-learning algorithm is shown in Algo-

rithm 2.1. Q-learning is a tabular method because the action-values for each state-action pair need to be stored in a table known as a Q-table. Tabular methods are normally only useful in very simple problems since the Q-table grows exponentially for every additional state dimension or action (Sutton and Barto, 2018). Therefore, Q-learning becomes infeasible for large state-spaces. To address this issue, methods that approximate the function $Q(s, a)$ are becoming more common as RL is applied to more complex problems.

Algorithm 2.1: Q-learning algorithm from Sutton and Barto (2018)

Initialise the learning rate α and exploration parameter ϵ
Initialise $Q(s, a)$ to random values for all $s \in S, a \in A(s)$ except that $Q(s_{terminal}, a) = 0$
for *episode* $\leftarrow 1$ **to** M **do**
    Get the initial state s from the environment
    while s **is not** *terminal* **do**
        Choose action a from state s using ϵ-greedy policy from Q
        Take action a and observe reward r and next state s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$
        $s \leftarrow s'$
    end
end

2.2 Deep Reinforcement Learning

In more complex RL problems, it is common to use a Neural Network (NN) to approximate the function $Q(s, a)$. The use of Deep Neural Networks (DNNs) for function approximation has shown great success in recent years, leading to a relatively new paradigm of RL known as Deep RL. The example mentioned in Section 2.1 of using RL to train agents that can play Atari games at a super-human level (Mnih et al., 2015) made use of a DNN for function approximation. When a DNN is used to approximate Q-values, the network is referred to as a Deep Q-Network (DQN). Before presenting an algorithm for training DQN agents,

a brief description of NNs is provided below.

2.2.1 Deep Neural Networks

NNs are represented as a graph where the nodes are connected by directed edges with associated weights. Each node computes the weighted sum of the inputs and applies some activation function to compute the output. Two examples of widely used activation functions are the sigmoid function, $f(z) = \frac{1}{1+e^{-z}}$, and the Rectified Linear Unit (ReLU) function, $f(z) = \max(0, z)$. NNs are typically arranged in layers where each layer contains nodes that take the outputs of the nodes of the previous layer as their input. The most basic type of NN is called an Multi-Layer Perceptron (MLP). An MLP consists of at least three layers of nodes: an input layer (simply the data that is passed into the network and uses no activation function), one or more hidden layers and an output layer (Russell and Norvig, 2002).

NNs are widely used in many areas of machine learning as they can represent arbitrarily complex functions. The weights at each edge of the network need to be tuned (trained) to find the appropriate function. NNs are trained by minimising some loss function using an algorithm called backpropagation (Rumelhart et al., 1985).

Since MLPs are fully-connected networks, they typically do not perform well in perception tasks where spatial information is relevant. For example, for an image converted to a 1-dimensional vector, an MLP wouldn't treat two neighbouring pixels as being any more relevant than two pixels that are at opposite sides of the image. Convolutional Neural Networks (CNNs) are commonly used in these situations as they take advantage of the spatial structure of images. Convolutional layers are the most important building block of CNNs; in a convolutional layer, feature maps are computed by performing the convolution operation between the

input data and filters (one filter for each feature map). The convolution operation involves computing the inner product between the filter and an overlapping area of the input data repeatedly by shifting the filter across the image. The elements of each filter are the weights that need to be learned and this can be done by backpropagation. The convolutional layers can extract spatial features from the input data and these are then passed through a fully-connected layers for prediction (Goodfellow et al., 2016).

For a more detailed description of NNs, the reader can refer to Russell and Norvig (2002) and Goodfellow et al. (2016).

2.2.2 Deep Q-learning

The algorithm presented by Mnih et al. (2015) to train a DQN agent is called Deep Q-learning (see Algorithm 2.2). In this study, a CNN was trained to predict the Q-values for each possible action given only the raw image pixels of the corresponding Atari game as input. For training a DQN agent, the target Q-values are not known; they need to be approximated using the network itself. This means that as the weights of the network change, the target values also change, leading to a loss function that is dependent on the iteration of training. From Mnih et al. (2015), the loss function at iteration i is:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}\left[y|s,a\right] - Q(s,a;\theta_i)\right)^2\right] \tag{2.2}$$

where $Q(s,a;\theta_i)$ is the predicted Q-value for action a in state s for the network with weights θ_i and $y = r + \gamma \max_{a'} Q(s',a';\theta_i^-)$ is the target. Note that θ_i^- are the weights of the network at some previous iteration.

Training of DQN agents can be unstable due to the target values constantly changing while tuning the network weights; two measures were taken by Mnih

et al. (2015) to address this. The first measure is the idea of experience replay; at each time-step the experience tuple (s, a, r, s') is stored in replay memory. Then, instead of only updating the network based on the current experience, a minibatch is randomly sampled from the replay memory and a training pass is done with those experiences. This results in greater data efficiency (as the same experience can be used multiple times in training) and less variance in the weight updates. The replay memory is restricted to a maximum length and the oldest experiences are removed first. The second measure to stabilise training is the addition of a dedicated network for computing the target values known as a target network. The target network is cloned from the online network every C time-steps. This measure reduces the risk of policy divergence and oscillations.

Algorithm 2.2: Deep Q-learning algorithm with experience replay adapted from Mnih et al. (2015)

Initialise the online network Q with random weights θ
Initialise target network $\hat{Q}$ with weights $\theta^- = \theta$
Initialise the experience replay memory D to maximum capacity N
Set the ϵ for ϵ-greedy exploration to 1.0
for *episode* $\leftarrow 1$ **to** M **do**
    Get the initial state s from the environment
    while s **is not** *terminal* **do**
        Choose action a from state s using ϵ-greedy policy
        Take action a and observe reward r and next state s'
        Store experience tuple (s, a, r, s') in D
        Sample a random minibatch from D
        foreach *experience* i **in** *minibatch* **do**
            if s' **is** *terminal* **then** $y \leftarrow r$;
            else $y \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a'; \theta^-)$;
        end
        Compute loss $L \leftarrow (y - Q(s, a; \theta))^2$
        Update weights θ of online network Q by minimising L using gradient descent
        Every C steps set $\theta^- \leftarrow \theta$
        Decay ϵ
    end
end

2.3 Multi-Agent Reinforcement Learning

Environments containing more than one agent are known as Multi-Agent Systems (MASs). MASs are useful for solving problems that require more scalable and robust solutions as they are distributed by nature. Agents in an MAS may work cooperatively and/or competitively to solve a task (Wooldridge, 2009). Multi-Agent Reinforcement Learning (MARL) is an area of research in RL that is concerned with solving problems using RL techniques in multi-agent environments.

2.3.1 Stochastic Games

The MDP framework presented in Section 2.1 is only applicable to single-agent systems. A Stochastic Game (SG) is a generalisation of the MDP framework that enables the use of multiple agents. In an SG, multiple agents perform actions at each time-step and the next state of the environment along with the reward for each agent are dependent on the joint actions of all of the agents (Busoniu et al., 2008).

An SG between n agents, consists of a set of states S, a set of actions for each agent $A_1, A_2, ..., A_n$, a transition function T and a reward function for each agent $R_1, R_2, ..., R_n$ (Busoniu et al., 2008). Note that when $n = 1$, the definition of the SG is the same as an MDP. When agents have their own local state observation o_i, a degree of uncertainty over the environment state s is introduced and the SG is referred to as partially observable. An observation function O is added to the definition of the SG, which is used to compute the local state for each agent i as $o_i = O(s, i)$ (Emery-Montemerlo et al., 2004).

One approach to solving SGs is to train each agent in the MAS using single-agent RL techniques (such as Deep Q-learning) and treat other agents as part of the environment (Mannion et al., 2017).

2.3.2 Social Dilemmas

This section provides a brief description of payoff matrices and social dilemmas. Although it is not central to the main research of this thesis, the theory is required for interpreting some results that are presented in Chapter 6.

Payoff matrices are a useful tool from the field of Game Theory that can be used to analyse encounters in MASs and MARL. The payoff (reward) for an agent playing a certain strategy (action), given the strategies of all other agents can be extracted from a payoff matrix. The general form of a payoff matrix for a

two-agent matrix game where each agent must either play strategy S_1 or strategy S_2 is shown in Figure 2.2a. In this figure, ${P_{ij}}^{(n)}$ is the payoff for agent n when it plays S_i and its opponent plays S_j.

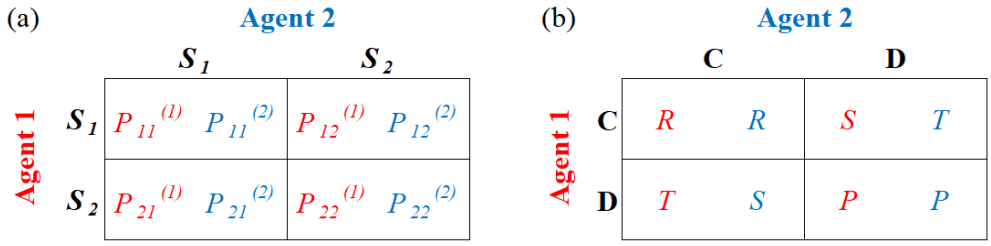

Figure 2.2: Payoff matrices for two-agent games. (a) General form (b) Social Dilemma

Payoff matrices are often used to analyse social dilemmas between agents. Consider the payoff matrix shown in Figure 2.2b. This is a two-agent game where each agent can either Cooperate (C) or Defect (D). R is the *Reward* payoff for mutual cooperation, P is the *Punishment* payoff for mutual defection, S is the *Sucker* payoff that a cooperating agent receives when its opponent defects and T is the *Temptation* payoff that a defecting agent receives when its opponent cooperates. This game is a social dilemma under the following four conditions (Macy and Flache, 2002):

1. $R > P$: Agents prefer mutual cooperation to mutual defection
2. $R > S$: Agents prefer mutual cooperation to being taken advantage of by a defecting agent
3. $2R > T + S$: Agents prefer mutual cooperation to receiving the *Temptation* payoff or *Sucker* payoff with equal likelihood
4. **either** $T > R$ (*greed*): Agents prefer taking advantage of a cooperating agent to mutual cooperation

or $P > S$ (*fear*): Agents prefer mutual defection to being taken advantage of by a defector

2.4 Multi-Objective Reinforcement Learning

Standard RL methods operate under the assumption that the problem can be solved by optimising a single objective; this is captured by the scalar reward received at each time-step. However, many real-world problems are multi-objective by nature and these objectives can be in conflict with each other (Vamplew et al., 2018) . For example, if designing an agent for a self-driving car, driving fast and keeping fuel consumption low would be conflicting objectives. Developing agents for such problems can be done with Multi-Objective Reinforcement Learning (MORL).

In order to solve multi-objective sequential decision making problems, the MDP framework needs to be extended to a Multi-Objective Markov Decision Process (MOMDP), where at each time-step, the agent receives a vector of real-valued rewards $\mathbf{r}_t$ (one reward for each objective). The vectorised value function is therefore defined as $\mathbf{V}^{\pi}(s) = \mathbb{E}[\mathbf{g}_t|\pi, s_t = s]$ where $\mathbf{g}_t = \sum_{k=0}^{\infty} \gamma^k \mathbf{r}_{t+k+1}$ (Roijers et al., 2013). Note the bold font signifying vectors.

Solution methods for MOMDPs can be divided into two categories: single policy methods and multiple-policy methods (Vamplew et al., 2011). There is no single optimal policy to be found for MOMDPs, in the same way that there is no single solution to a multi-objective optimisation problem. Instead, the solution is the set of points that form the Pareto front (Deb, 2014). However, a scalarisation function can be used to convert the vectorised value function of the MOMDP into a function that yields a scalar, making it possible to find a single policy. The general form of a scalarisation formula is $V^{\pi}_{\mathbf{w}}(s) = f(\mathbf{V}^{\pi}(s), \mathbf{w})$ where $\mathbf{w}$ is a

vector of weights corresponding to the preferences between objectives. A common form of scalarisation applied in MORL is linear scalarisation: $V^{\pi}_{\mathbf{w}}(s) = \mathbf{w} \cdot \mathbf{V}^{\pi}(s)$ (Roijers et al., 2013). Although this method is widely used, it has the drawback that it cannot find any policy in a concave region of the Pareto front (Vamplew et al., 2008). If $\mathbf{w}$ is unknown prior to the learning stage, a set of policies can be learned using a range of values for $\mathbf{w}$ (Roijers et al., 2013). Of course, this also increases the complexity of the problem.

2.5 Agents with Tunable Behaviours

Agents are most commonly designed to achieve some fixed goal behaviour. If a different behaviour is desired, the agent may need to be partially or completely redesigned or, in the case of an RL agent, retrained. The ability to tune the behaviour of an agent would be beneficial for many applications; for example, a stock trading agent could be tuned to take more risks or video game playing agents could be tuned to be more aggressive or more defensive.

One approach to this is to train an agent using different reward functions to yield several desired behaviours (policies) and then switch between the policies during run-time to change the behaviour of the agent. An approach similar to this was adopted by Klinkhammer (2018) who studied how to select the best policy to follow at a given time based on aligning sub-rewards with a global reward.

Another approach to designing agents capable of dynamic behaviours is to use concepts from MORL. A set of weights in the scalarisation could be used for each type of desired behaviour and after training an agent using the MOMDP framework, the weights could be adjusted during run-time to switch between the behaviours. This idea was introduced by Källström and Heintz (2019b).

In the multi-objective framework used in this study, an agent would receive a

linearly scalarised reward based on objective preferences (weights) at each time-step. An adapted version of the Deep Q-learning algorithm (Algorithm 2.2) was used to train the agent; an important addition however was that the objective preferences of the agent were fed into the network along with the current state of the environment. This meant that the agent could use the knowledge of its preferences when taking an action. A block diagram of this framework is shown in Figure 2.3. Two gridworld experiments were conducted to evaluate the framework and it was shown empirically that the behaviours of agents could be tuned during run-time to behave similarly to agents trained to reach fixed behaviours. The types of behaviours analysed included cooperation, competitiveness and risk-aversion.

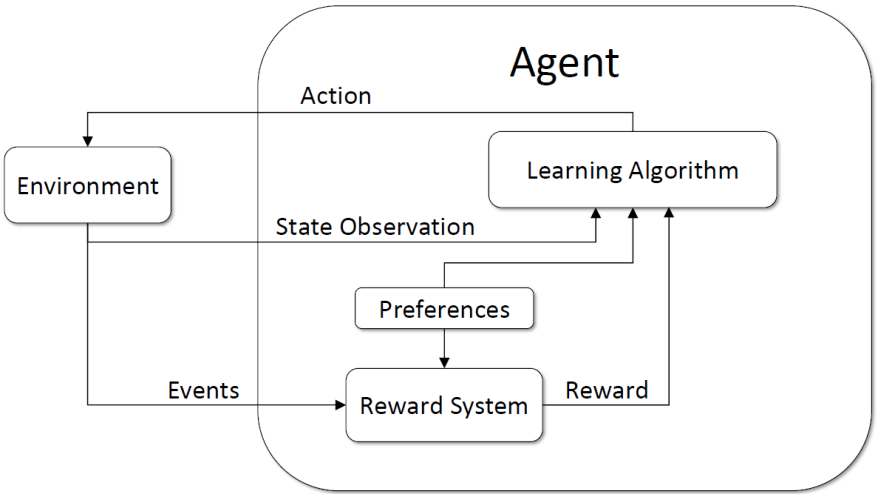

Figure 2.3: Architecture for training agents with tunable behaviours. Image taken from Källström and Heintz (2019b)

A further study was conducted using this same framework in a different environment by Källström and Heintz (2019a). One of the experiments in this study focused on designing an agent for fighter-pilot training simulations whose behaviour could be tuned to balance whether to favour safety or time-to-destination more. This work also showed that tunable agents could reach similar behaviours

to fixed preference agents. The state-space for the environment in this experiment was based on metrics such as distance and direction whereas the state-spaces in the gridworld experiments in Källström and Heintz (2019b) were image based. Therefore, it is difficult to compare the complexity of the problems for training tunable agents.

Chapter 3

Methodology

In this chapter, the method for training tunable agents and the architecture of the neural network used for function approximation are presented.

3.1 Neural Network Architecture

The two multi-objective environments that are the basis for the main research conducted for this thesis are the Gathering environment (see Chapter 5) and the Wolfpack environment (see Chapter 6). A state for each of these gridworld environments is represented as a 3-channel RGB image of the associated grid. Since the environments are multi-objective, the reward returned after each time-step is a 1-dimensional vector of a fixed length. An agent expresses its preferences over the elements in the reward vector through an objective preference weight vector (i.e., the scalarisation weights). Since the goal is to train an agent with tunable behaviours, each action needs to depend on both the current state of the environment and the objective preference weight vector. The neural network architecture used is therefore as follows.

The environment state observed by the agent at each time-step is actually the

previous three frames of the environment stacked together. This in effect means that the *current state* of the environment is a 9-channel image as opposed to a 3-channel one. Note that for the first two time-steps of each episode, this stack of frames includes duplicates to keep the stack as a fixed size.

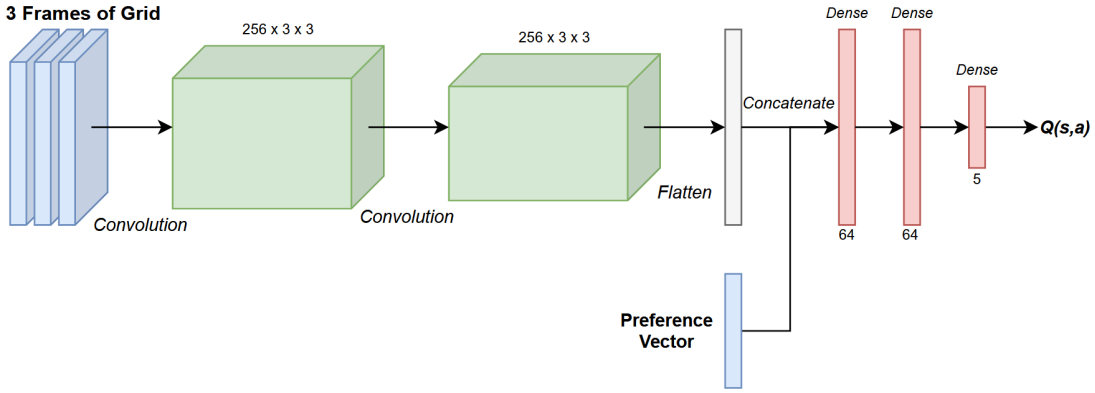

Figure 3.1: Architecture of the Deep Neural Network created for training tunable agents in both the Gathering experiment and the Wolfpack experiment

The image data is rescaled between 0 and 1 before being processed by two convolutional layers to extract features from the grid image. No form of pooling operation is used after each convolutional layer due to the image already being of relatively low dimensionality ($8 \times 8 \times 3$ per frame for Gathering and $16 \times 16 \times 3$ per frame for Wolfpack). The output of the second convolutional layer is flattened to a 1-dimensional vector and concatenated with the preference weight vector, which is also rescaled between 0 and 1. The concatenated 1-dimensional vector is then passed through two fully-connected (i.e., dense) hidden layers and finally a dense output layer.

A block diagram of this architecture is shown in Figure 3.1. Note that the dimensions of the convolutional layers shown refers to 256 filters of size 3×3. The ReLU activation function is used after all convolutional layers and dense layers except for the output layer, which has a linear activation function since predicting Q-values is a regression problem.

Although the exact architecture used by Källström and Heintz (2019b) is not provided in their paper, the architecture described above was chosen to fit the description they provide as closely as possible. Specifications such as how to scale the data, the number of units in each layer and the filter sizes were found through experimentation in this work.

Note that block diagrams of the NN architectures for the models used in the Gathering and Wolfpack experiments outlining the input and output dimensions of each layer are shown in Appendix A.

3.2 Tunable Agent Algorithm

Algorithm 3.1 outlines the method used for training the neural network described in Section 3.1. The Deep Q-learning algorithm (see Algorithm 2.2) is a central component to this training scheme. Källström and Heintz (2019b) presented a high-level method for training tunable agents; Algorithm 3.1 is a lower-level version of this method with steps specific to using the Deep Q-learning algorithm as the base RL algorithm.

At the beginning of each episode, the agent samples a set of objective preference weights from the preference sample space; this forms the objective preference weight vector $\mathbf{w}$. The action at each time-step is chosen based on $\mathbf{w}$ and the current state of the environment s (the stack of the last three frames of the grid) using the network described in Section 3.1 with an ϵ-greedy policy. After an action a is executed, the next state of the environment s' and reward vector $\mathbf{r}$ is received from the environment. A scalar reward r is then computed using linear scalarisation between $\mathbf{r}$ and $\mathbf{w}$. The agent then stores the experience tuple $(s, a, r, s', \mathbf{w})$ in its replay memory. After each episode (excluding the first T episodes to allow the replay memory to grow initially), the agent is trained on a

minibatch from the replay memory.

A key difference to highlight between the single-objective Deep Q-learning algorithm and Algorithm 3.1 is that the network is conditioned on the objective preference weight vector so $\mathbf{w}$ needs to also be stored in the experience replay memory. The training frequency is another difference to highlight. Here, the weights of the network are updated at the end of every episode, while in the Deep Q-learning algorithm, they are updated after every time-step. The training method described in Källström and Heintz (2019b) appears to involve a training step even less frequently (every n episodes).

Algorithm 3.1: Training algorithm for tunable agent

Initialise the online network Q with random weights θ
Initialise target network $\hat{Q}$ with weights $\theta^- = \theta$
Initialise the experience replay memory D to maximum capacity N
Set the ϵ for ϵ-greedy exploration to 1.0
Set T to the number of episodes to start training after
for *episode* $\leftarrow 1$ **to** M **do**
    Get the initial state s from the environment
    Sample a preference weight vector $\mathbf{w}$ from the preference weight sample space
    while s **is not** *terminal* **do**
        Choose action a using network Q based on s and $\mathbf{w}$ using an ϵ-greedy policy
        Take action a and observe reward vector $\mathbf{r}$ and next state s'
        Compute the scalarised reward $r \leftarrow \mathbf{r} \cdot \mathbf{w}$
        Store experience tuple $(s, a, r, s', \mathbf{w})$ in D
    end
    if *episode* $> T$ **then**
        Sample a random minibatch from D
        foreach *experience* i **in** *minibatch* **do**
            if s'_i **is** *terminal* **then** $y_i \leftarrow r_i$;
            else $y_i \leftarrow r_i + \gamma \max_{a'} \hat{Q}(s'_i, a', \mathbf{w}_i; \theta^-)$;
        end
        Compute loss $L \leftarrow (y - Q(s, a, \mathbf{w}; \theta))^2$
        Update weights θ of online network Q by minimising L using gradient descent
    end
    Decay ϵ
    Every C episodes update $\hat{Q}$: set $\theta^- \leftarrow \theta$
end

Chapter 4

Benchmarking Experiments

In this chapter, the experimentation conducted for initial benchmarking for a simple MORL problem and a single-objective DQN agent are described. This experimentation formed the foundation for building on for the main research of this thesis.

4.1 Deep Sea Treasure

This section describes a benchmarking experiment carried out to design a framework for applying linear scalarisation to a simple multi-objective sequential decision making problem. The environment used in this experiment was the Deep Sea Treasure (DST) environment, which was first introduced by Vamplew et al. (2008) as a benchmarking test with a known Pareto front for assessing the performance of MORL algorithms.

Figure 4.1 shows a visual representation of the environment for the DST problem. The environment is an 11×10 grid that the agent, represented by the submarine icon, can navigate around by selecting one of four possible actions (left, right, up or down) at each time-step. The agent can only observe its current lo-

cation (x and y coordinates) in the grid at each time-step. After each action, the agent receives two reward signals: a time-penalty of -1 and a treasure reward. There are 10 treasure states in the environment, each of a different value that is dependent on the depth (shown in yellow). The episode ends if the agent reaches a treasure state or if 1,000 time-steps is reached. If the agent attempts to navigate outside the grid or into the sea-bed (shown in blue), the state remains unchanged and a time-penalty is still received. The multi-objective task for the agent is to minimise the amount of steps taken and maximise the treasure collected. Since these objectives are in conflict with each other, there is no single correct solution; instead, the solution is the set of points that form the Pareto front. A variation of the benchmark test with a globally convex Pareto front from Mannion et al. (2017) was used so that linear scalarisation of the rewards could be applied to find the Pareto front.

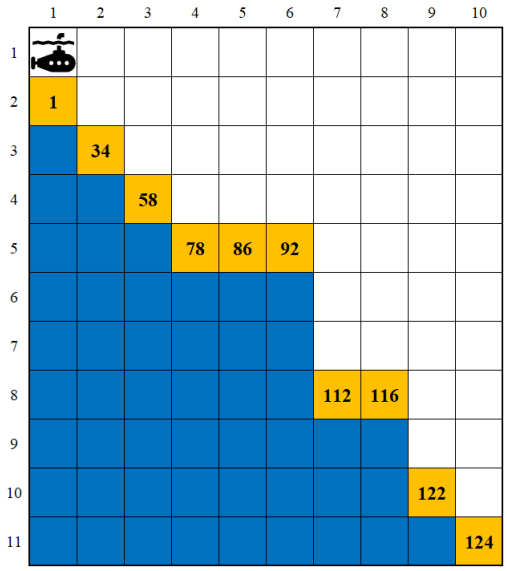

Figure 4.1: Deep Sea Treasure environment with convex Pareto front

An outer-loop approach using the Q-learning algorithm (see Algorithm 2.1) was used to find the Pareto front of this version of the DST problem. This meant

solving a series of single-objective MDPs, each with a different set of scalarisation weights applied to the two reward signals. 101 evenly spaced scalarisation weight combinations were used, which meant storing 101 different Q-tables. A learning rate (α) of 0.1 and a discount factor (γ) of 1.0 were used. The exploration factor (ϵ) was set to 0.998^n where n is the current episode number. The training progress plots over 4,000 episodes for a subset of the scalarisation weights are shown in Figure 4.2. Note that the associated scalarisation weights are shown in the legend of each subplot where the first element corresponds to the time-penalty weighting and the second element corresponds to the treasure reward weighting.

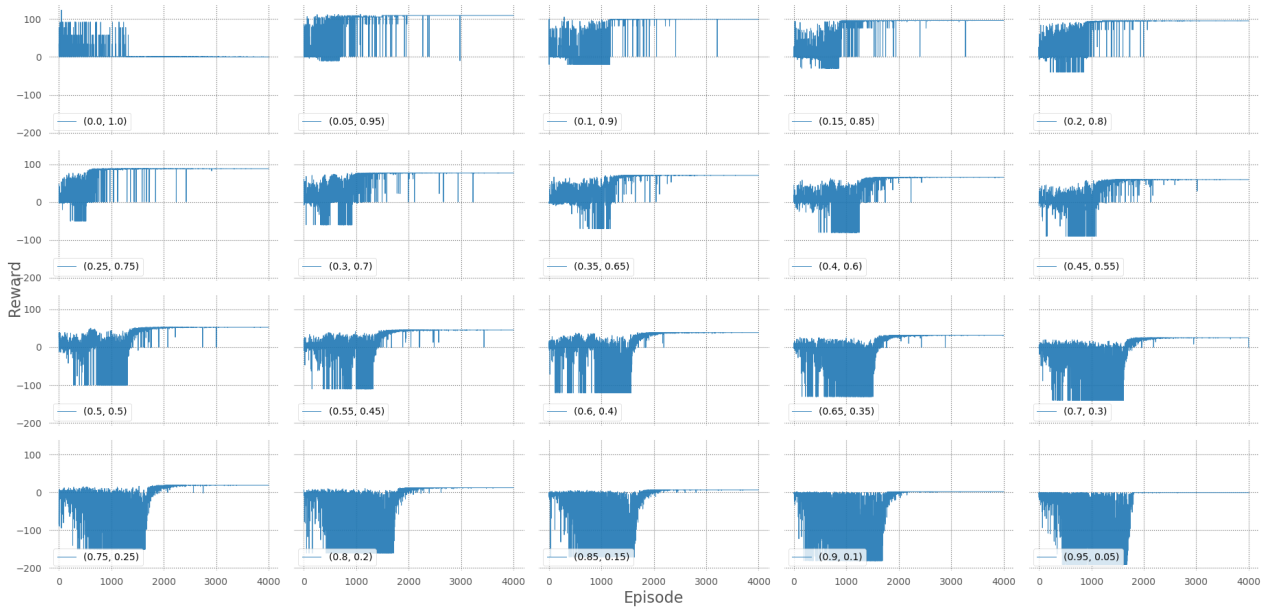

Figure 4.2: Scalarised Q-learning training progress for a subset of scalarisation weights

The true Pareto front for this problem in is shown in Figure 4.3. Each point on the Pareto front is easily calculated as the negative of the Manhattan distance from the agent start state to a given treasure state (the x-axis value) and the treasure value at that state (the y-axis value). The estimated Pareto front found

by scalarised Q-learning was then computed by simulating an episode using each set of scalarisation weights and the associated Q-table using a greedy policy and computing the total episode reward in vector form. The Pareto front of the 101 resulting points was then calculated and the result was found match the true Pareto front (see Figure 4.3).

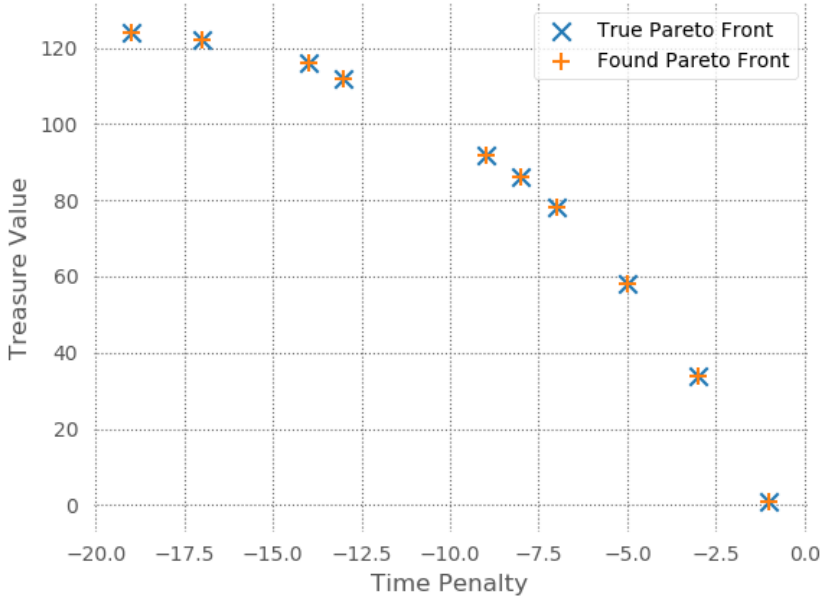

Figure 4.3: The true Pareto front for the convex Deep Sea Treasure environment and the Pareto front found using scalarised Q-learning

4.2 Single-Objective Gathering

Since the Deep Q-learning algorithm (Algorithm 2.2) was going to be a central component to the framework for training tunable agents, a second benchmarking experiment was carried out to implement a DQN agent on a single-objective task. A variation of the Gathering environment from Källström and Heintz (2019b) was used as the single-objective RL problem. The environment is an 8×8 grid where

an agent can navigate left, right, up or down to collect eight items randomly placed in the centre 4×4 area of the grid. The agent always starts in the bottom left corner of the grid at the beginning of an episode. This environment is the same as shown in Figure 5.1 except the second agent (shown in pink) is not included. Although the collectable items vary in colour, the reward for picking up each item is the same. For each step the agent takes in the environment, a scalar reward of -0.01 is received and for each item collected a scalar reward of 1 is received. An episode ends if all items are collected by the agent or if 50 steps are taken.

Hyperparameter	**Value**
Loss Function	Mean Squared Error
Optimizer	Adam
Learning Rate	0.001
Discount Factor	0.95
Initial Epsilon	1
Epsilon Decay	1/5000
Final Epsilon	0.05
Replay Memory Size	3,000
Minibatch Size	64
Start training model after (episodes)	50
Copy to target every (steps)	1,000
Number of Training Episodes	10,000

Table 4.1: Hyperparameters for training the DQN agent in the single-objective Gathering environment

A CNN was used to learn the policy for the DQN agent. The input to the network is a 3-channel RGB image of the grid, which gets processed by two convolutional layers. After flattening the output of the second convolutional layer, the data is passed through two dense layers. The full architecture of the network, including the dimensions input and output of each layer, is shown in Figure C.1.

An alteration was made to the Deep Q-learning algorithm for training the

agent; instead of sampling a minibatch from the replay memory and updating the weights of the DQN after every time-step, as seen on line 17 of Algorithm 2.2, it is done after every episode. A similar approach was taken by Källström and Heintz (2019b) in their training algorithm for tunable agents.

The agent was trained for 10,000 episodes using the hyperparameters in Table 4.1. The *mean* reward per 100 episodes over the course of training is shown in Figure 4.4. Note that the reward per episode is shown as background data and the smoothened data is highlighted; from this point forward, only smoothened versions of the reward per episode are shown and the standard deviation is shown as background data in plots of this nature.

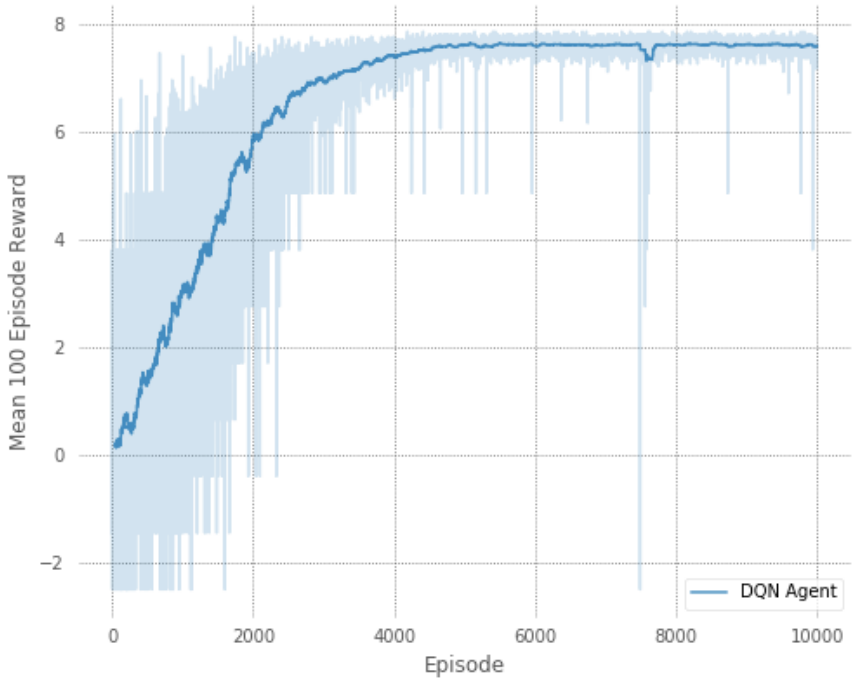

Figure 4.4: Training curve for the DQN agent in the single-objective Gathering environment

As can be seen in Figure 4.4, the agent converges to a near optimal policy for collecting all eight items in the environment in the smallest number of steps. This was an ideal starting point for the replication study discussed in Chapter 5.

Chapter 5

Gathering: A Replication Study

This chapter describes a replication study of the Gathering experiment by Källström and Heintz (2019b). This work was carried out to establish the framework for training agents with tunable behaviours and conduct some further analysis to address research question 1 (see Section 1.2). The development for this experiment is an extension of the work discussed in Section 4.2 of this thesis.

5.1 Simulation Methods

The environment for this experiment (example shown in Figure 5.1) consists of an 8×8 grid with eight coloured items randomly placed in the centre 4×4 area every time the environment is reset. There are always three green items, three red items and two yellow items. A deterministic agent starts in the top right corner of the grid at the beginning of each episode (shown in pink). This agent always attempts to collect the red items in order of which is closest its current position. Note that the deterministic agent also collects other items that are in its path. A trainable agent starts in the bottom left corner of the grid at the beginning of each episode (shown in blue). Each agent can move either left, right, up or down

or remain in its current position at each time-step (5 actions).

After each time-step, the environment returns a vector of *events*, where each element of the vector is a binary variable signifying whether or not the corresponding event occurred during the time-step. An agent can be trained to have different preferences over these events. The six events are:

1. *step*: The trainable agent took a step (always 1)
2. *wall*: The trainable agent hit a wall (attempted to move outside the grid)
3. *green*: The trainable agent collected a green item
4. *red*: The trainable agent collected a red item
5. *yellow*: The trainable agent collected a yellow item
6. *other-agent-red*: The deterministic agent collected a red item

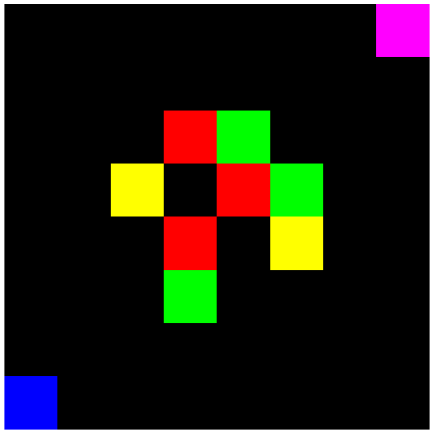

Figure 5.1: Environment for the multi-objective Gathering experiment from (Källström and Heintz, 2019b). The blue box represents the agent being trained, the pink box represents the deterministic agent and the green, red and yellow boxes are the items that the agents can collect.

The environment reaches a terminal state when the trainable agent has collected all items that it has a positive preference for or when 30 time-steps have occurred (whichever condition is satisfied first).

The preference weight sample space that defined the possible preferences for the events was any multiple of 5 between -20 and 20 for events 3-6 above. The agent was given a fixed preference of -1 for *step* and -5 for *wall* because there was no reason to have tunable preferences over these events. An example preference weight vector sampled could therefore be $\begin{bmatrix}-1 & -5 & 10 & -5 & 0 & 20\end{bmatrix}$.

When Källström and Heintz (2019b) carried out this experiment, they compared the performance of a tunable agent trained on a range of preferences from the preference weight sample space against agents trained with a fixed objective preference weight vector. The four different sets of preference weights (i.e., behaviours) that were analysed were (note that the preference weights for *step* and *wall* are omitted):

1. *Competitive* : $\begin{bmatrix}10 & 20 & 10 & -20\end{bmatrix}$ - A highly negative preference for *other-agent-red* would cause the trainable agent to be heavily penalised whenever the deterministic agent picks up a red item. Training with this preference weight vector should cause the trainable agent to be highly competitive with the deterministic agent.

2. *Cooperative* : $\begin{bmatrix}10 & 20 & 10 & 20\end{bmatrix}$ - An equally high positive preference for *red* and *other-agent-red* would cause the trainable agent to receive a high reward when either agent picks up a red item. Training with this preference weight vector should cause the trainable agent to be cooperative with the deterministic agent.

3. *Fair* : $\begin{bmatrix}20 & 15 & 20 & 20\end{bmatrix}$ - A slightly lower positive preference for *red* than *other-agent-red* should result in an agent that leaves some red items for the deterministic agent to collect.

4. *Generous* : $\begin{bmatrix}20 & 0 & 20 & 20\end{bmatrix}$ - Zero preference for *red* and a high positive preference for *other-agent-red* should result in an agent that leaves all red

items for the deterministic agent.

Using Algorithm 3.1 with the hyperparameters listed in Table 5.1, a tunable agent was trained in the Gathering environment for 200,000 episodes. The full model architecture is shown in Figure C.2. During training, a 20% dropout was also used in each convolutional and dense layer to help prevent overfitting. Four fixed agents were also trained for the behaviour types listed above. The training algorithm and neural network architecture was exactly the same for training the fixed agents except that the objective preference weight vector was kept constant for the entire training time, instead of being randomly sampled at the beginning of each episode.

Hyperparameter	**Value**
Loss Function	Mean Squared Error
Optimizer	Adam
Learning Rate	0.0001
Discount Factor	0.99
Initial Epsilon	1.0
Epsilon Decay	1/50,000
Final Epsilon	0.01
Replay Memory Size	6,000
Minibatch Size	32
Start training model after (episodes)	50
Copy to target every (steps)	1,000
Number of Training Episodes	200,000

Table 5.1: Hyperparameters for training the tunable and fixed agents in the Gathering environment

A minor adjustment to Algorithm 3.1 was that the *step* and *wall* preference weights were not used as inputs to the network since there were always constant; the scalarisation was computed with all six preference weights but the network was only conditioned on the *green*, *red*, *yellow* and *other-agent-red* preferences. The weights at the input to the network were also scaled between -0.5 and 0.5

by dividing each weight by 40. The training progress plots for the five types of agents are shown in Figure 5.2.

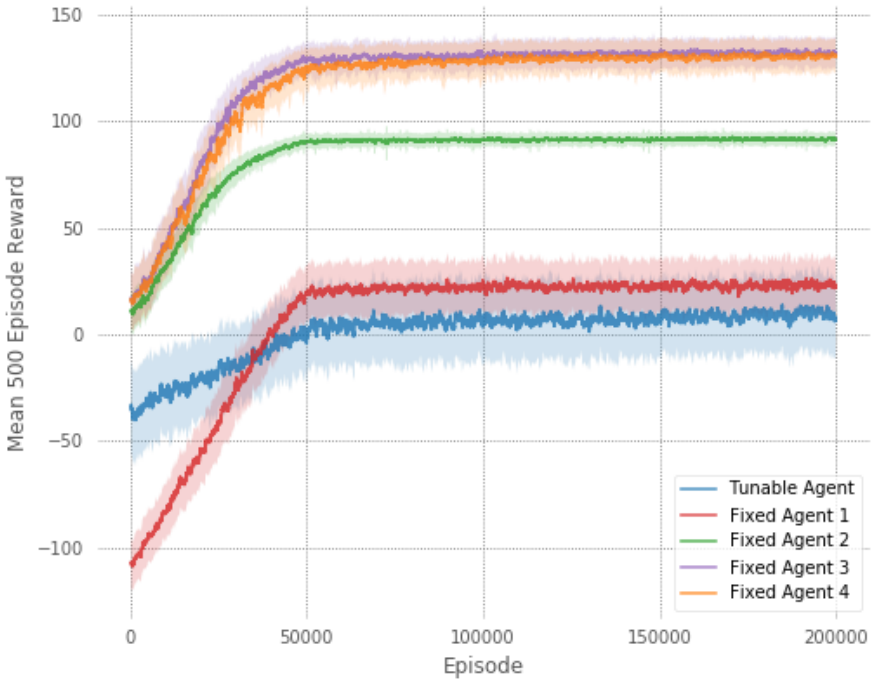

Figure 5.2: Training progress for the tunable agent and four different fixed agents in the Gathering environment using Scalarisation Method 1. The shaded regions represent the error that is computed as the standard deviation.

The approach to scalarisation described above that was presented by Källström and Heintz (2019b) does not meet the definition of linear scalarisation given by Roijers et al. (2013, p. 77): "*Each element of* $\mathbf{w}$ *specifies how much one unit of value for the corresponding objective contributes to the scalarized value. The elements of the weight vector* $\mathbf{w}$ *are all positive real numbers and constrained to sum to 1.*". The preference weight vector that gets sampled at the beginning of each episode is used to compute the scalar reward and the elements of the vector are neither all positive real numbers nor constrained to sum to 1.

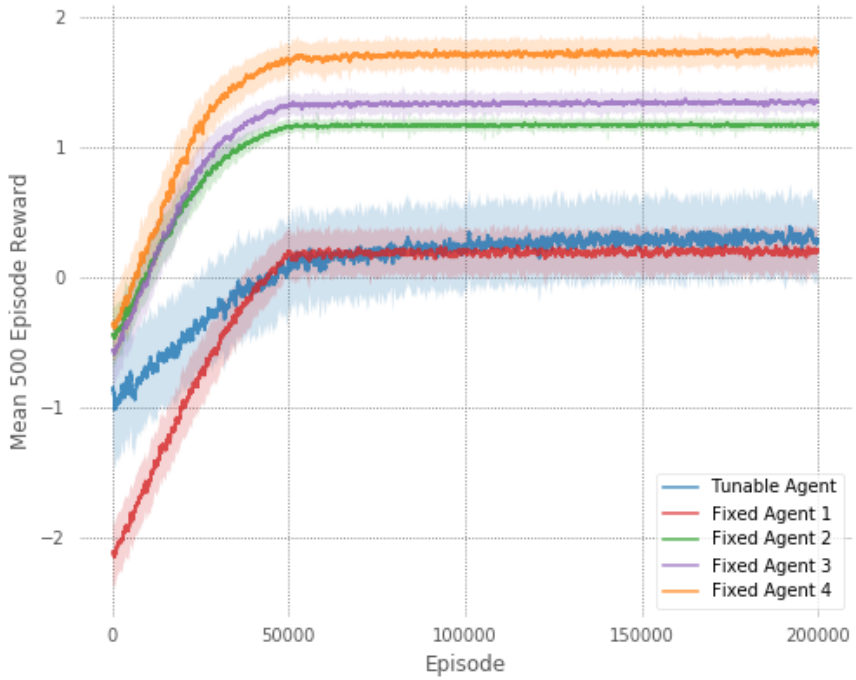

Figure 5.3: Training progress for the tunable agent and four different fixed agents in the Gathering environment using Scalarisation Method 2. The shaded regions represent the error that is computed as the standard deviation.

A second set of agents were subsequently trained in an environment and with a training scheme that could enable this definition of linear scalarisation to be met. The preference weight sample space was left unchanged and the preference weight vector used as input to the neural network was scaled in the same way as described above. Before scalarisation, however, the absolute values of all of the elements in the weight vector were computed and the result was normalised so the elements summed to 1. A consequence of having all positive elements in the weight vector was that it was no longer suitable for each element of the reward vector (i.e., the event vector) returned by the environment to be binary; there needed to be a negative sign introduced so a negative preference could reduce the

scalarised reward. The environment was consequently refactored to return -1 if an event occurred that the agent had a negative preference over.

Using this alternative setup, the same five types of agents were trained for 200,000 episodes with all other steps in the training algorithm left unchanged. The training progress plots for these agents are shown in Figure 5.3.

From this point forward, the scalarisation method used by Källström and Heintz (2019b) will be referred to as Scalarisation Method 1 and the method described above that meets the definition by Roijers et al. (2013) will be referred to as Scalarisation Method 2.

5.2 Results

For each of the four behaviours listed in the Section 5.1, results were collected over 250 simulation episodes in the Gathering environment with the corresponding fixed agent and the tunable agent using the associated objective preference weight vector. The mean and standard deviation of each of the metrics were computed. This was done for the agents trained using Scalarisation Method 1 (see Table 5.2) and the agents trained using Scalarisation Method 2 (see Table 5.3). Results of a similar format were reported by Källström and Heintz (2019b).

Behaviour	Agent Type	steps	green	red	yellow	other-agent-red
1 *Competitive*	*Fixed*	12.98 ± 2.01	2.60 ± 0.59	1.29 ± 0.63	1.78 ± 0.44	1.71 ± 0.63
	Tunable	14.79 ± 3.66	2.52 ± 0.63	1.03 ± 0.68	1.68 ± 0.52	1.97 ± 0.68
2 *Cooperative*	*Fixed*	12.06 ± 1.81	2.68 ± 0.55	0.85 ± 0.67	1.70 ± 0.49	2.15 ± 0.67
	Tunable	17.49 ± 4.34	2.36 ± 0.72	0.34 ± 0.66	1.53 ± 0.61	2.66 ± 0.66
3 *Fair*	*Fixed*	12.26 ± 1.77	2.67 ± 0.53	0.30 ± 0.51	1.72 ± 0.47	2.70 ± 0.51
	Tunable	18.32 ± 4.85	2.21 ± 0.79	0.25 ± 0.59	1.54 ± 0.59	2.75 ± 0.59
4 *Generous*	*Fixed*	13.08 ± 2.03	2.50 ± 0.60	0.02 ± 0.13	1.62 ± 0.54	2.97 ± 0.19
	Tunable	18.21 ± 3.90	2.17 ± 0.78	0.01 ± 0.11	1.47 ± 0.64	2.98 ± 0.14

Table 5.2: Results from simulation of 250 episodes of all trained agents in the Gathering environment using Scalarisation Method 1

Behaviour	Agent Type	steps	green	red	yellow	other-agent-red
1 *Competitive*	*Fixed*	12.75 ± 1.85	2.56 ± 0.59	1.40 ± 0.65	1.77 ± 0.45	1.60 ± 0.65
	Tunable	14.42 ± 3.35	2.53 ± 0.64	1.18 ± 0.64	1.72 ± 0.50	1.82 ± 0.64
2 *Cooperative*	*Fixed*	11.85 ± 1.72	2.64 ± 0.54	0.84 ± 0.69	1.77 ± 0.43	2.16 ± 0.69
	Tunable	13.28 ± 2.56	2.52 ± 0.62	0.93 ± 0.72	1.63 ± 0.52	2.07 ± 0.72
3 *Fair*	*Fixed*	11.93 ± 1.53	2.62 ± 0.53	0.38 ± 0.56	1.74 ± 0.48	2.62 ± 0.56
	Tunable	13.90 ± 3.28	2.40 ± 0.68	0.33 ± 0.53	1.63 ± 0.57	2.67 ± 0.53
4 *Generous*	*Fixed*	12.26 ± 2.00	2.52 ± 0.59	0.00 ± 0.00	1.69 ± 0.48	2.98 ± 0.17
	Tunable	13.22 ± 3.16	2.38 ± 0.67	0.09 ± 0.28	1.60 ± 0.57	2.72 ± 0.47

Table 5.3: Results from simulation of 250 episodes of all trained agents in the Gathering environment using Scalarisation Method 2

Some further analysis was done using both tunable agents in order to gain further insight into the tuning performance. For each combination of *red* and *other-agent-red* between -20 and 20 in multiples of 5 and while keeping the preference weight fixed for *green* and *yellow*, 250 episodes were simulated and the mean number of red items collected in each episode for each behaviour was recorded. The results collected for the agent trained using Scalarisation Method 1 and Scalarisation Method 2 are presented as heatmaps (see Figures 5.4 and 5.5).

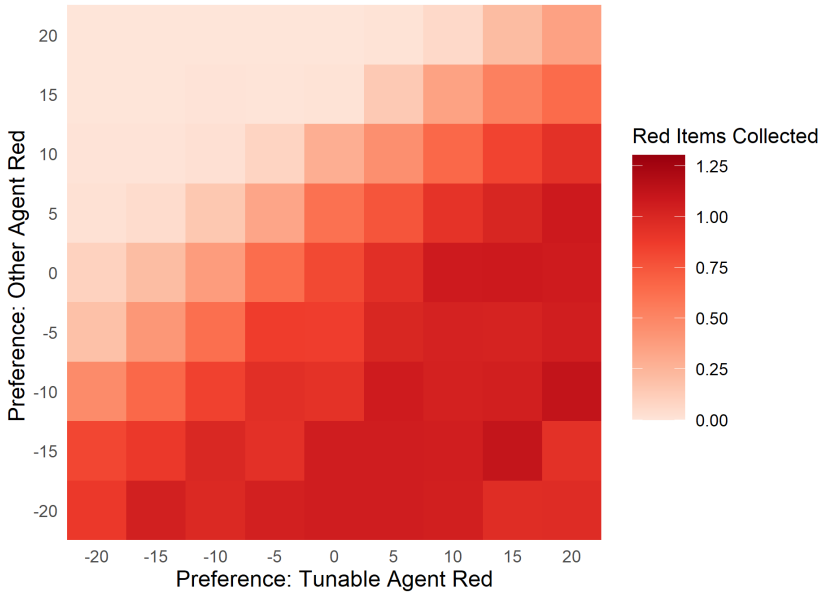

Figure 5.4: Tuning performance for the tunable agent in the Gathering environment using Scalarisation Method 1

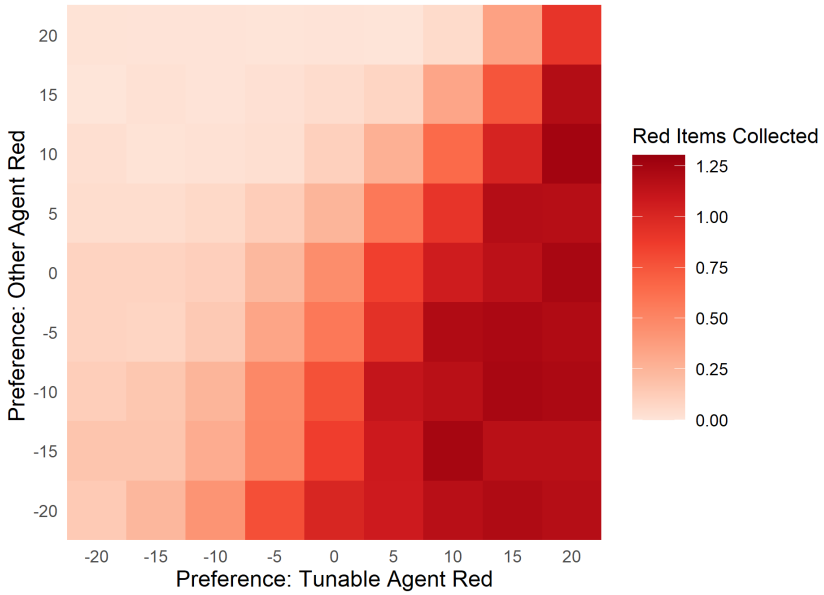

Figure 5.5: Tuning performance for the tunable agent in the Gathering environment using Scalarisation Method 2

5.3 Discussion

Referring to Table 5.2, it can be seen that the expected behaviours (as described in the Section 5.1) are reached by the agents trained using Scalarisation Method 1. For example, in the *Competitive* scenario the deterministic agent picks up the least amount of red items and in the *Generous* scenario the trainable agent (whether fixed or tunable) picks up the least amount of red items. The tunable agent reaches close to the performance of the fixed agent in each case. However, the tunable agent takes significantly more steps on average in each case. For the agents trained using Scalarisation Method 2 (see Table 5.3), the expected behaviours are also reached. However, there are some noticeable differences when comparing the performance of the tunable agents; there is less deviation between the metrics for the fixed and tunable agents in Table 5.3 compared to Table 5.2 and the tunable agent takes less steps on average when trained using Scalarisation Method 2. Note that examples of the four types of behaviours analysed here can be seen in the video recordings linked in Appendix B.

The heatmaps in Figures 5.4 and 5.5 make it abundantly clear that the tunable agents can exhibit a spectrum of different behaviours without any need for re-training. As the *red* preference weight is increased along x-axis, the tunable agent gradually collects *more* red items. As the *other-agent-red* preference weight is increased along the y-axis, the tunable agent can gradually collects *less* red items. Since the colour gradient is on the same scale in both figures, they can be compared easily; the tunable agent trained using Scalarisation Method 2 appears to reach closer to the desired behaviour for each preference weighting. For example, the bottom right of the heatmap in Figure 5.5 is a darker shade than the same area in Figure 5.4, signifying that the tunable agent trained using Scalarisation Method 2 converges to more competitive behaviours.

A contributing factor as to why both scalarisation methods yield different re-

sults is likely to be the smaller search space introduced by normalising the weight vector to sum to 1. For example, Scalarisation Method 1 treats the preference weight vectors $\begin{bmatrix}10 & 10 & 10 & 10\end{bmatrix}$ and $\begin{bmatrix}20 & 20 & 20 & 20\end{bmatrix}$ differently, while Scalarisation Method 2 would treat them both as $\begin{bmatrix}0.25 & 0.25 & 0.25 & 0.25\end{bmatrix}$; note that, once again, the *step* and *wall* preference weights are omitted.

It is important to note that the results reported by Källström and Heintz (2019b) were collected with agents trained for 20 million time-steps. Assuming an average episode length of 13 time-steps, this is a total training time of over 1.5 million episodes, which is significantly more than the 200,000 episode training times used in this study. Perhaps with further training, the behaviours of the tunable agents using the two different scalarisation schemes could be closer.

Chapter 6

Wolfpack: A Multi-Agent Study

This chapter describes an experiment carried out to address research questions 2 and 3 (see Section 1.2). This study focuses on extending the framework for training tunable agents to a multi-agent setting and a more complex environment. The environment is a simplified version of the one used in the Wolfpack experiment carried out by Leibo et al. (2017).

6.1 Simulation Methods

The Wolfpack environment involves three agents that can navigate around a gridworld-type space. There is one *prey* agent and two *predator/wolf* agents. The predators must navigate around the grid to capture the prey, either as a pack (team) or alone. The grid that the agents can navigate is shown in Figure 6.1; it is of size 16×16 and contains several obstacles (shown in grey) that the agents must move around. The predators are shown in blue and the prey is shown in red. The prey is captured when one of the predators is at the same location as the prey. A team-capture occurs when the prey is captured by one predator while the other predator is within a certain radius, referred to as the

capture-radius. The green area in Figure 6.1 represents the capture-radius, which is only highlighted for the purpose of this figure and is not actually part of the grid image. The capture-radius was set to a Manhattan distance of 3 throughout this research. At each time-step, the agents can select one of five actions (up, down, left, right or stay), the same as the Gathering environment described in Chapter 5.

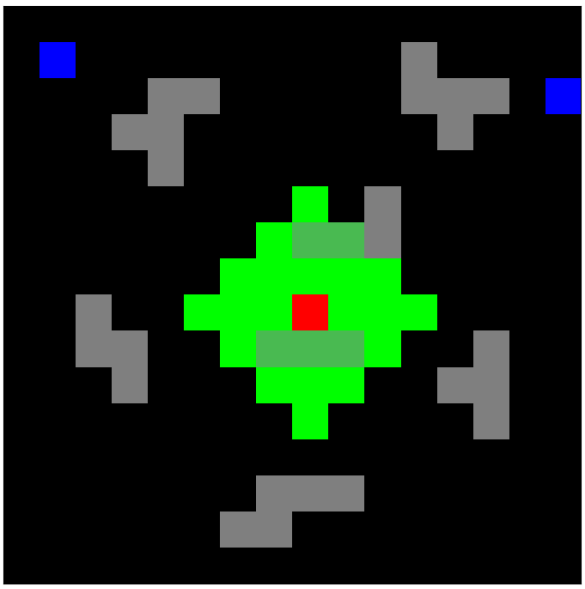

Figure 6.1: Environment for the Wolfpack experiment. Predators are represented by blue boxes, the prey is represented by a red box and the obstacles are the grey boxes. The capture-radius is also included and is shown in green.

The Wolfpack environment used in the original study by Leibo et al. (2017) was of a higher degree of complexity. The grid dimensions were 20×20, the agents had only partial observability of the grid that was dependent on their forward-facing direction and they could rotate as an action to change their direction of view. The environment was simplified in this study to reduce the training time for the agents.

The environment in the original study was also a single-objective environment, with a different scalar reward received for team-captures and lone-captures. It

was adapted to a multi-objective environment for this research by giving four different reward signals at each time-step:

1. *step*: A reward of -1 for each time-step to encourage the agents to complete an episode quickly.
2. *wall*: A reward of -1 any time the associated agent hits the grid boundary or an obstacle.
3. *lone-capture*: A reward of 1 if the associated agent captures the prey with no other predator inside the capture-radius.
4. *team-capture*: A reward of 1 if the associated agent is inside the capture-radius with another predator when the prey is captured.

These signals are equivalent to the event signals for the Gathering environment. At the beginning of each episode, the starting positions of the three agents are set randomly to empty locations in the grid. A reward vector is returned for each agent in the environment after each time-step and an episode ends if the prey gets captured or 150 time-steps take place. A separate state for each agent is also returned after each time-step; in the image of the grid that each predator sees, they are represented as a blue box and the other predator is represented as a green box. This was done to allow the predator to learn to identify its own location in the grid and the other agents are just treated as part of the environment. A similar approach was taken by Leibo et al. (2017). This problem is a multi-objective stochastic game as the MOMDP framework (Section 2.4) and the SG framework (Section 2.3) are merged. Since each agent's observation of the environment is different, the problem could be viewed as a partially observable SG with an observation function.

Hyperparameter	Value
Loss Function	Huber
Optimizer	Adam
Learning Rate	0.0001
Discount Factor	0.99
Initial Epsilon	1.0
Epsilon Decay	1/21,250
Final Epsilon	0.01
Replay Memory Size	6,000
Minibatch Size	64
Start training model after (episodes)	50
Copy to target every (steps)	1,000
Number of Training Episodes	80,000

Table 6.1: Hyperparameters for training the predator agents in the Wolfpack environment

For this study, the prey was not trained to evade the predators and it was just assigned a random-action policy. It is not clear what approach was taken by Leibo et al. (2017) in this regard. The predators were trained as two separate tunable agents using Algorithm 3.1. The *step* preference was kept fixed at 0.005 and the *wall* preference was fixed at 0.025 (similar to what was done in the Gathering experiment in Chapter 5). The *lone-capture* preference was then sampled from 5 evenly spaced values between 0 and 0.97 and the *team-capture* preference was chosen to make the full objective preference weight vector sum to 1. This preference weighting scheme made the scalarisation comply with the definition of linear scalarisation from Roijers et al. (2013).

The hyperparameters used for training both tunable predators are shown in Table 6.1. Since the episodes in this environment were longer than the episodes in the Gathering environment, the agent would have more experience between each update of the model weights. For this reason, the minibatch size was increased from 32 to 64. As the networks for the two predators were being trained simultaneously, the training time was significantly longer than that in the Gathering

experiment. The full architecture of the neural network used is shown in Figure C.3. A 20% dropout was also used in each convolutional layer to help prevent overfitting. The training progress plots for the two tunable predators are shown in Figure 6.2.

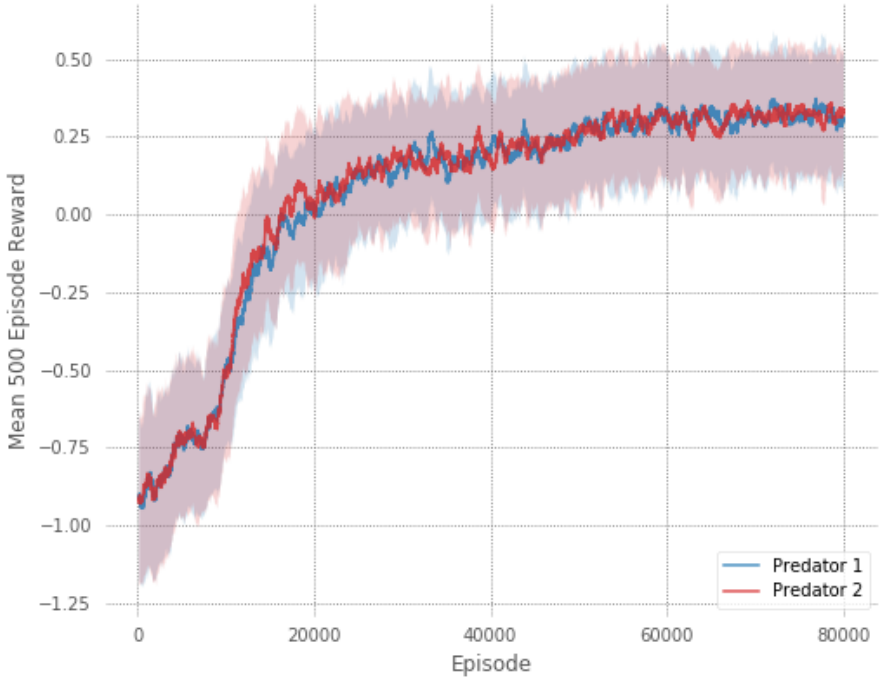

Figure 6.2: Training progress for the two tunable predator agents in the Wolfpack environment. The shaded regions represent the error that is computed as the standard deviation.

Agents with fixed preferences were also trained in the Wolfpack environment. To give the fixed agent experience of games with agents of different behaviours, a tunable agent was instantiated as the second predator during training time. This would allow a fair comparison between tunable and fixed agents. Two types of behaviours were considered for fixed agents:

1. *Cooperative* : $\mathbf{w} = \begin{bmatrix} 0.005 & 0.025 & 0.0 & 0.97 \end{bmatrix}$
2. *Competitive / Defective* : $\mathbf{w} = \begin{bmatrix} 0.005 & 0.025 & 0.97 & 0.0 \end{bmatrix}$

The training progress plots for the two types of fixed agents are shown in Figure 6.3.

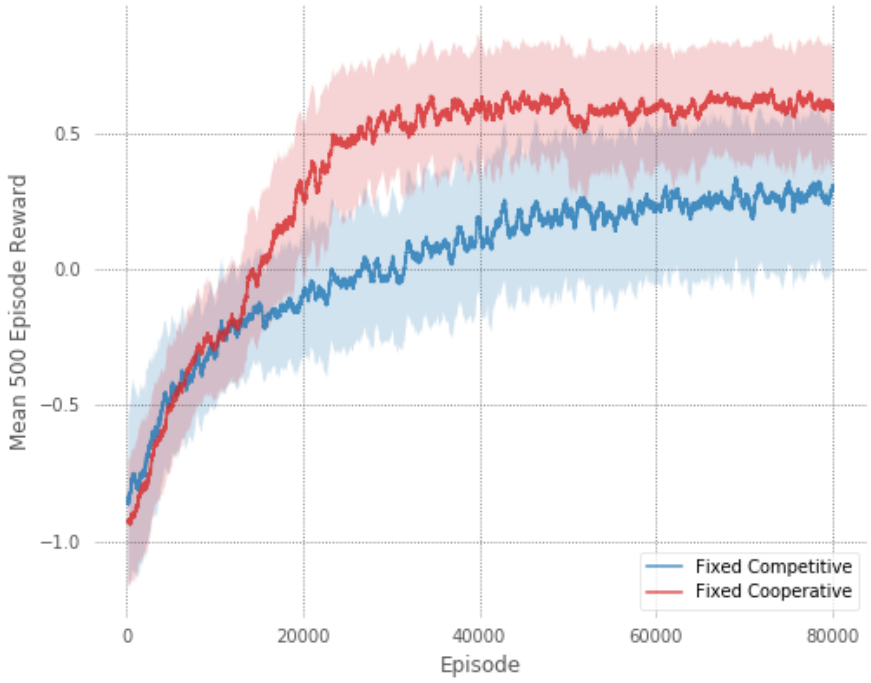

Figure 6.3: Training progress for the two types of fixed behaviour predator agents in the Wolfpack environment. The shaded regions represent the error that is computed as the standard deviation.

6.2 Results

Results were collected to analyse the ability of the tunable agents to achieve competitive and cooperative behaviours of a varying degree. The first test was to

instantiate two tunable predators in the Wolfpack environment and simulate 250 episodes for a range of different preference weights where the two predators had the same preference in any given episode. Figure 6.4 shows how the lone-capture rate and team-capture rates vary with the agents degree of cooperativeness (team-capture preference) or competitiveness (lone-capture preference).

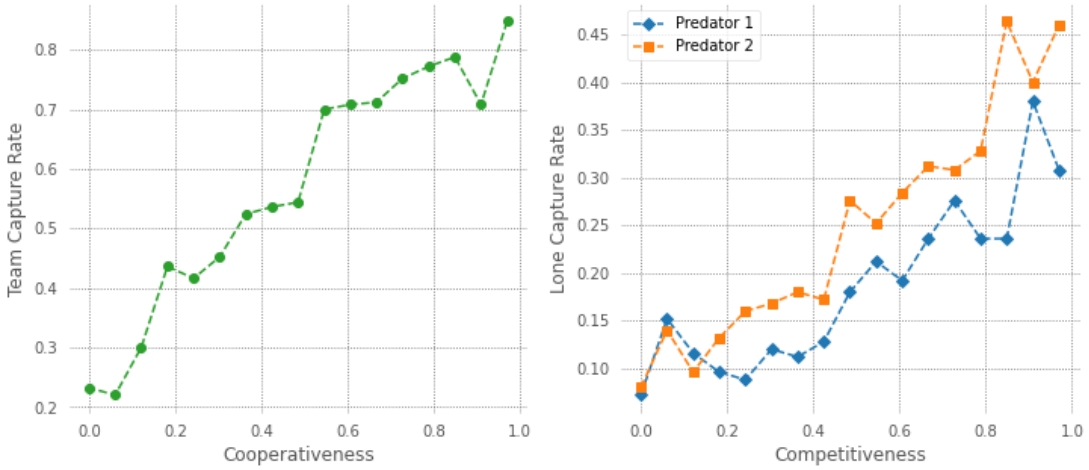

Figure 6.4: Tuning performance for two predator agents with matched preferences

Naturally, the next scenario to simulate was a series of encounters between tunable predators of varying preferences. Once again, 250 episodes were simulated of each game and the two different types of captures were tracked. The results are displayed as a heatmap in Figure 6.5, showing how the team-capture rate varies with the degree of cooperativeness of each predator.

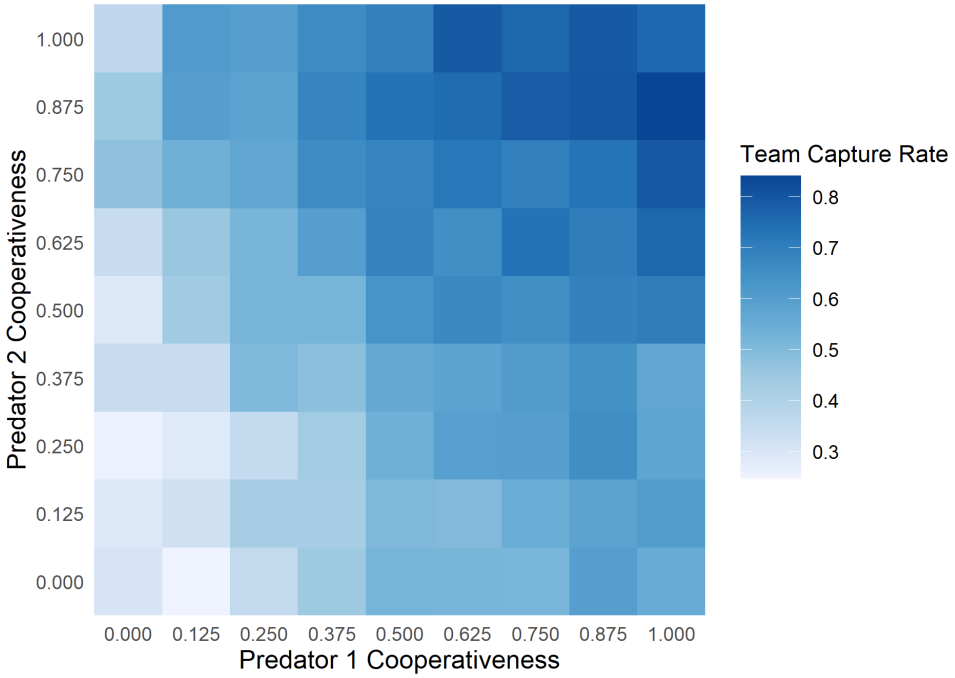

Figure 6.5: Tuning performance for two predator agents with varied preferences

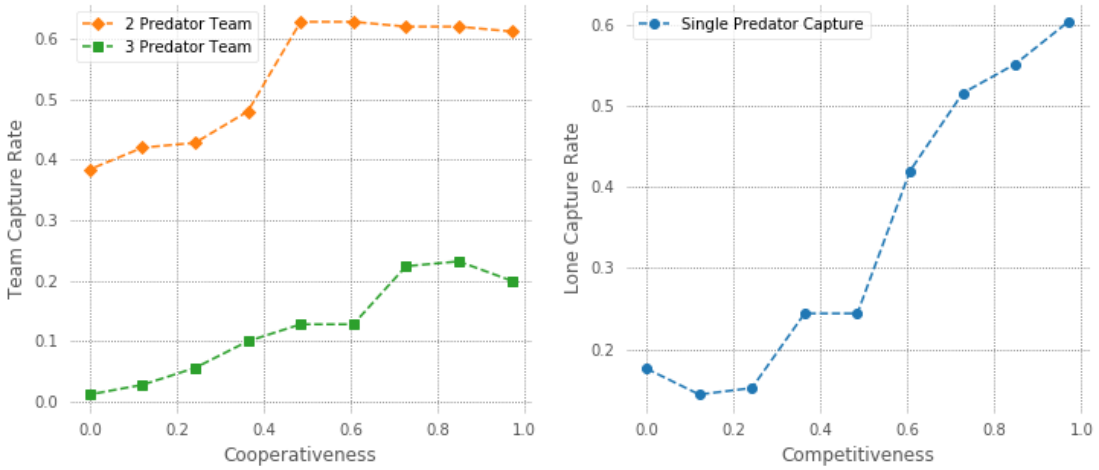

Figure 6.6: Tuning performance for three predator agents with matched preferences

(a)

Pred. 1	Pred. 2: C	Pred. 2: D
C	0.792 0.792	0.592 0.296
D	0.254 0.532	0.306 0.438

(b)

Pred. 1	Pred. 2: C	Pred. 2: D
C	0.540 0.540	0.524 0.372
D	0.360 0.520	0.464 0.348

(c)

Pred. 1	Pred. 2: C	Pred. 2: D
C	0.908 0.908	0.540 0.344
D	0.340 0.600	0.436 0.468

Figure 6.7: Empirical payoff matrices for the Wolfpack experiment. (a) Tunable agents (b) Fixed agents with matched models (c) Fixed agents with unique models.

Using a predator agent that was trained with a different random seed, a third predator was instantiated into the environment. The degree of cooperativeness and competitiveness was varied in simulations of 250 episodes for each setup (all three predators had matched preferences) and the team-capture and lone-capture rates were tracked. The results are shown in Figure 6.6.

Empirical payoff matrices were generated by simulating 250 games (episodes) between fully cooperative agents and fully competitive (defective) agents (i.e., agents with the objective preferences weights used for training the fixed agents). The payoff for the encounter was computed as the capture rate (lone or team, depending on the type of agent) over the 250 episodes. Figure 6.7a shows the empirical payoff matrix constructed for encounters between the two tunable predators whose training progress is shown in Figure 6.2. The equivalent payoff matrix for the two fixed agents is shown in Figure 6.7b; this meant that, for the cooperative versus cooperative and the defective versus defective scenarios, the same model was instantiated twice. Figure 6.7c shows the payoff matrix for fixed agents where two extra fixed agents were trained with different random seeds to avoid the scenario of two equivalent models playing against each other. The method for generating these empirical payoff matrices is based on the method used by Leibo et al. (2017).

6.3 Discussion

When viewing the behaviour of the tunable predators in simulation, the effects of varying the predators' objective preferences were clear. Setting a high preference for lone-captures, made the predators more competitive with each other. Once an episode began, both predators would move as quickly as possible to capture the prey on their own. Conversely, setting a high preference for team-captures, made the predators more cooperative. At the beginning of each episode, both predators would typically move towards each other and approach the prey together, often taking different routes around obstacles to trap the prey. These two types of behaviours can be seen in the video recordings linked in Appendix B.

Referring to Figure 6.4, a strong positive correlation between both tunable predators' level of cooperativeness and the resulting team-capture rate can be seen. The same can be said for both predators' level of competitiveness and their respective lone-capture rates. Note that during training, the tunable predators could only sample from 5 discrete preference weights (as described in Section 6.1); these plots therefore include data-points of simulations with preference weights that are unseen to the predators and the models are still able to generalise to an appropriate level of cooperativeness or competitiveness.

From the simulations between tunable predators with independently varied objective preferences, it can be seen in Figure 6.6 that the trend remains between individual predators' level of cooperativeness and the resulting team-capture rate. This is a very interesting result as it shows that the models don't learn any assumption of how the other predator behaves and that they generalise to encounters of any type of behaviour. This is a direct result of sampling the preference weight vector for each predator individually at the beginning of each episode during training as opposed to using the same preference for each one. Note that the plot also contains preference weights that were not used during training, once

again highlighting the generalisability of the models.

Adding a third predator into the Wolfpack environment was a test to see how well the models can generalise to unseen states. During training, the images of the grid that either of the networks see, contains one red box (for the prey), one blue box (for the predator that the network is being trained for) and only one green box (for the other predator). However, adding a third predator during simulation means that an image would contain two green boxes and that the state would not have been seen by the models before. The plots in Figure 6.6, however, show that the models can generalise to working in a 3-predator team (although less frequently than a 2-predator team) and the cooperativeness can still be increased by tuning the objective preference weights.

Referring to the payoff matrix in Figure 6.7a, it can be seen that the rational behaviour is to cooperate in any encounter since the payoff for cooperating is highest no matter what type of behaviour the opposing predator possesses. Note that there is no element of greed (when the temptation payoff is greater than the reward payoff) or fear (when the punishment payoff is greater than the sucker payoff) in this matrix game. Therefore, this game does not meet the conditions for a social dilemma that were described in Section 2.3.2. In the version of the Wolfpack experiment by Leibo et al. (2017), empirical payoff matrices that were not social dilemmas were also seen for certain reward structures. Payoff matrices could have been generated for agents of different levels of cooperativeness and perhaps different social dynamics would have been seen since this would be analogous to changing the reward structure. However, this was not carried out since the main reason payoff matrices were generated in this study was to compare the performance of tunable and fixed preference agents as opposed to analysing the social dynamics.

The payoff matrix generated for the two fixed agents (Figure 6.7b) yielded

some unexpected results; the payoff for mutual cooperation is significantly lower than it was for the tunable agents. This suggests that the tunable agents reached higher levels of performance over fixed agents, going against what was seen in Källström and Heintz (2019b). However, from viewing simulations of episodes of mutual cooperation, it was discovered that the reason for the low payoff was that both predators frequently met at the same grid location (making only one predator visable) and would then take the same actions until the end of the episode as both used the same NN model. Since only one predator was visible to each predator, it appeared as though there was no other predator present to cooperate with and therefore no incentive to approach the prey. The fixed competitive agents, however, didn't tend to approach each other so this was not an issue for that type of encounter. This was the reason for training two extra fixed agents using different random seeds. The resulting payoff matrix in Figure 6.7c shows this behaviour disappear and dynamics closer to that of Figure 6.7a are seen.

The values in the payoff matrices in Figures 6.7a and 6.7c are indeed quite similar, showing that tunable agents can reach similar behaviours to fixed agents. The fixed agents do however have slightly higher capture rates in all scenarios compared to the tunable agents; this observation is in alignment with what was seen when comparing fixed agents and tunable agents in Chapter 5 and in Källström and Heintz (2019b).

Chapter 7

Conclusions

This chapter states the conclusions of this thesis. Firstly, the proposed answers to the research questions posed in Section 1.2 are stated along with some further contributions of this study. This is followed by a description of the limitations, impact and proposed future work regarding this research.

7.1 Summary of Contributions

The answers to the research questions of this thesis are as follows.

1. What are the effects of refactoring the tunable agents framework by Källström and Heintz (2019b) to meet the definition of linear scalarisation from Roijers et al. (2013)? (RQ1)

   In Chapter 5, an alternative scalarisation method for the Gathering experiment was presented that ensured that the objective preference weight vector contained all real-valued, positive elements and that they would sum to one. The performance of the agents trained using both scalarisation methods were compared and the results showed that the tunable agent trained using the method that met the definition of linear scalarisation from Roijers

et al. (2013) reached behaviours closer to that of the agents trained using fixed objective preferences. It was suggested that this may have been due to the effects that this scalarisation had on reducing the objective preference weight sample space.

2. Does this same framework scale to more complex environments? (RQ2)

   The Wolfpack experiment described in Chapter 6 used a gridworld environment that was of a higher degree of complexity to the environments used in the experimentation conducted by Källström and Heintz (2019b). The Wolfpack environment grid size was 16×16, which contained four times the number of grid cells as the 8×8 grids used by Källström and Heintz (2019b). Another aspect of the Wolfpack environment that made it more complex was the fact that the locations of each agent were randomised at the start of every episode, whereas in the Gathering environment, the agents' start positions were always the same and the locations of the items were only randomised within a small area. The results presented in Chapter 6 therefore prove that the framework can indeed scale to more complex environments.

3. Can agents achieve tunable behaviours in a multi-agent setting? (RQ3)

   Although the Gathering environment in Chapter 5 contained two agents, only one of them could be tuned to exhibit varying preferences over the objectives. The training scheme for agents used for the Wolfpack experiment in Chapter 6 involved training separate networks for tunable predator agents where the objective preference weights were sampled independently by each agent at the beginning of each episode. The results presented for this experiment showed that objective preferences for either agent could be tuned independently to achieve varying degrees of cooperative behaviour.

In addition to the research questions answered above, there are two further

contributions made by this thesis. Firstly, it was shown that the framework for training tunable agents presented by Källström and Heintz (2019b) could be replicated to a high degree of accuracy. The exact hyperparameters used for training the agents in the experiments carried out in this work have been presented in this thesis and the code is publicly available (see Appendix A); this means that results from this thesis are easily reproducible and the framework can be implemented for further research by another party.

The second additional contribution made by this work is that it was empirically shown that the agents trained could generalise well to unseen objective preference weightings as well as unseen environment states. The former was shown by collecting results for objective preference weightings that were not used during the training cycle for the tunable agents (see Figures 6.4 and 6.5). The latter was shown by instantiating a third pre-trained predator agent into the Wolfpack environment during simulation and seeing that tunable cooperativeness for all three predators could still be achieved (see Figure 6.6).

7.2 Impact

The contributions of this work discussed in Section 7.1 open the door for this method of training agents with tunable behaviours to be applied to a huge array of different problems. This framework would be beneficial to any RL problem where there is some degree of uncertainty over the desired type of agent behaviour. If an agent with fixed objective preferences was trained and it was then seen that the behaviour needed to be changed slightly, the agent would need to be retrained with new objective preferences. However, using the method for training tunable agents discussed throughout this thesis, the objective preferences could simply be fine-tuned after training.

7.3 Limitations

The biggest limitation in this work was the very long training times for the agents presented in the experimentation of this thesis. Training the tunable agent in the Gathering environment in Chapter 5 for 200,000 episodes took over 22 hours and training the two tunable predators in the Wolfpack environment simultaneously in Chapter 6 for 80,000 episodes took over 47 hours. Note that an NVIDIA Tesla P100 GPU with 27.4 GB of RAM was used in both cases. Long training times are a common theme in Deep RL and they should always be taken into account. This inherently presents a limitation on the scalability of this work. For many problem domains where this framework for training tunable agents could theoretically be applied, it may not be computationally feasible to do so.

7.4 Future Work

There are many possible directions that could be taken to extend this research in the future. The first suggestion for future work is related to the hyperparameters for training the models in Chapters 5 and 6. The hyperparameters could be optimised to ensure the models can be trained to reach the highest attainable performance. In addition, the effects that the various hyperparameters have on the tuning performance of the models could be analysed. Due to the long training times and time constraints on the project work, it was not feasible to conduct this work here.

Another related suggestion is to research the effect of different exploration strategies on the results. In this study, ϵ-greedy exploration was used and no other strategy was tested. It would be valuable to assess the performance of more sophisticated exploration strategies such as softmax exploration.

The experimentation in the Wolfpack environment could be expanded on sig-

nificantly in the future. The size of the grid could be increased from 16×16 and the agents could be given partial observability of the state-space, as was done in the original study (Leibo et al., 2017). Furthermore, the prey could also be trained to evade the predators, this could possibly introduce different social dynamics between the predators as it would be very difficult to capture the prey alone.

Another suggestion for future work is to apply this tunable agents framework to a more complex multi-agent environment such as the *half field offence* 2D robosoccer simulation environment (Hausknecht et al., 2016). In this problem, agents could be trained to have tunable levels of attacking or defensive behaviours.

The base RL algorithm used for training tunable agents is another aspect of this research that could be investigated further. The training scheme used in this work (see Chapter 3) could easily be adapted to use a base RL algorithm other than Deep Q-learning. One suggestion is to adapt the training scheme to fit with Actor-critic methods as they have been very successful in Deep RL applications in recent years.

A final suggestion for future work is related to scalarisation. In this work, linear scalarisation was used to compute the scalarised rewards when training the tunable agents. This same approach was taken by Källström and Heintz (2019b). It was mentioned in Section 2.4 that the usage of linear scalarisation in MORL has its limitations as it can not find policies in concave regions of the Pareto front. It would therefore be beneficial to investigate the possibility of adapting the training methods used in this study to work with some form of non-linear scalarisation such as non-linear monotonically increasing scalarisation functions discussed in Roijers et al. (2013).

7.5 Final Remarks

This thesis has presented a framework for training agents with tunable behaviours in multi-agent environments. Through empirical evidence, it has been shown that a single neural network model can represent a spectrum of behaviours for an agent and can even generalise to behaviours and environment states that have not been seen during training. There are many possibilities for future research using the methodology outlined in this thesis.

The relevant code all of the environments, training algorithms, simulations and plotting is available at the GitHub repository stated in Appendix A. A link to video recordings of simulations for both the Gathering experiment (Chapter 5) and the Wolfpack experiment (Chapter 6) is provided in Appendix B.

References

Itamar Arel, Cong Liu, Tom Urbanik, and Airton G Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, 2010. 4

Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008. 11

Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014. 13

Rosemary Emery-Montemerlo, Geoff Gordon, Jeff Schneider, and Sebastian Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, pages 136–143. IEEE, 2004. 11

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016. 8

Matthew Hausknecht, Prannoy Mupparaju, Sandeep Subramanian, Shivaram Kalyanakrishnan, and Peter Stone. Half field offense: An environment for mul-

tiagent learning and ad hoc teamwork. In *AAMAS Adaptive Learning Agents (ALA) Workshop*. sn, 2016. 56

Nicholas R Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):277–296, 2000. 3

Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1): 7–38, 1998. 3

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 4

Johan Källström and Fredrik Heintz. Multi-agent multi-objective deep reinforcement learning for efficient and effective pilot training. In *FT2019. Proceedings of the 10th Aerospace Technology Congress, October 8-9, 2019, Stockholm, Sweden*, pages 101–111, 2019a. 15

Johan Källström and Fredrik Heintz. Tunable dynamics in agent-based simulation using multi-objective reinforcement learning. In *Adaptive and Learning Agents Workshop (ALA-19) at AAMAS, Montreal, Canada, May 13-14, 2019*, pages 1–7, 2019b. 1, 2, 14, 15, 16, 19, 20, 25, 27, 29, 30, 31, 33, 35, 39, 51, 52, 53, 54, 56

Eric R Klinkhammer. Learning in complex domains: Leveraging multiple rewards through alignment. In *Adaptive and Learning Agents Workshop (ALA-18) at AAMAS, Stockholm, Sweden, July 14-15, 2018*, pages 1–9, 2018. 14

Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. *arXiv preprint arXiv:1702.03037*, 2017. 40, 41, 42, 43, 48, 50, 56

Michael W Macy and Andreas Flache. Learning dynamics in social dilemmas. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7229–7236, 2002. 12

Patrick Mannion, Sam Devlin, Karl Mason, Jim Duggan, and Enda Howley. Policy invariance under reward transformations for multi-objective reinforcement learning. *Neurocomputing*, 263:60–73, 2017. 11, 23

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 4, 6, 8, 10

Diederik M Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113, 2013. 2, 13, 14, 33, 35, 43, 52, 56

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985. 7

Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, 2002. 7, 8

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 4, 5, 6

Peter Vamplew, John Yearwood, Richard Dazeley, and Adam Berry. On the limitations of scalarisation for multi-objective reinforcement learning of pareto fronts. In *Australasian Joint Conference on Artificial Intelligence*, pages 372–378. Springer, 2008. 14, 22

Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, 84(1-2):51–80, 2011. 13

Peter Vamplew, Richard Dazeley, Cameron Foale, Sally Firmin, and Jane Mummery. Human-aligned artificial intelligence is a multiobjective problem. *Ethics and Information Technology*, 20(1):27–40, 2018. 13

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. 5

Marco Wiering and Martijn Van Otterlo. *Reinforcement learning*, volume 12. Springer, 2012. 5

Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009. 10

Appendix A

GitHub Repository

All of the code written for this work is publicly available at the following GitHub Repository: `https://github.com/docallaghan/tunable-agents`

Appendix B

Simulation Videos

Video recordings of simulations in for both the Gathering environment and the Wolfpack environment are available at the following YouTube link: `https://www.youtube.com/playlist?list=PLwuLqnGt8K-Glorp01rs_TTbIQpOOqkwj`

Appendix C

Model Architectures

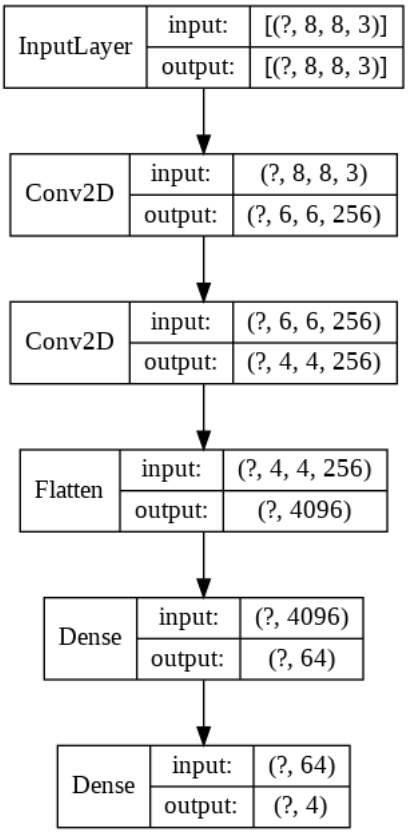

Figure C.1: Model architecture used for the DQN agent in the single-objective Gathering experiment. Image generated using `tf.keras.utils.plot_model`.

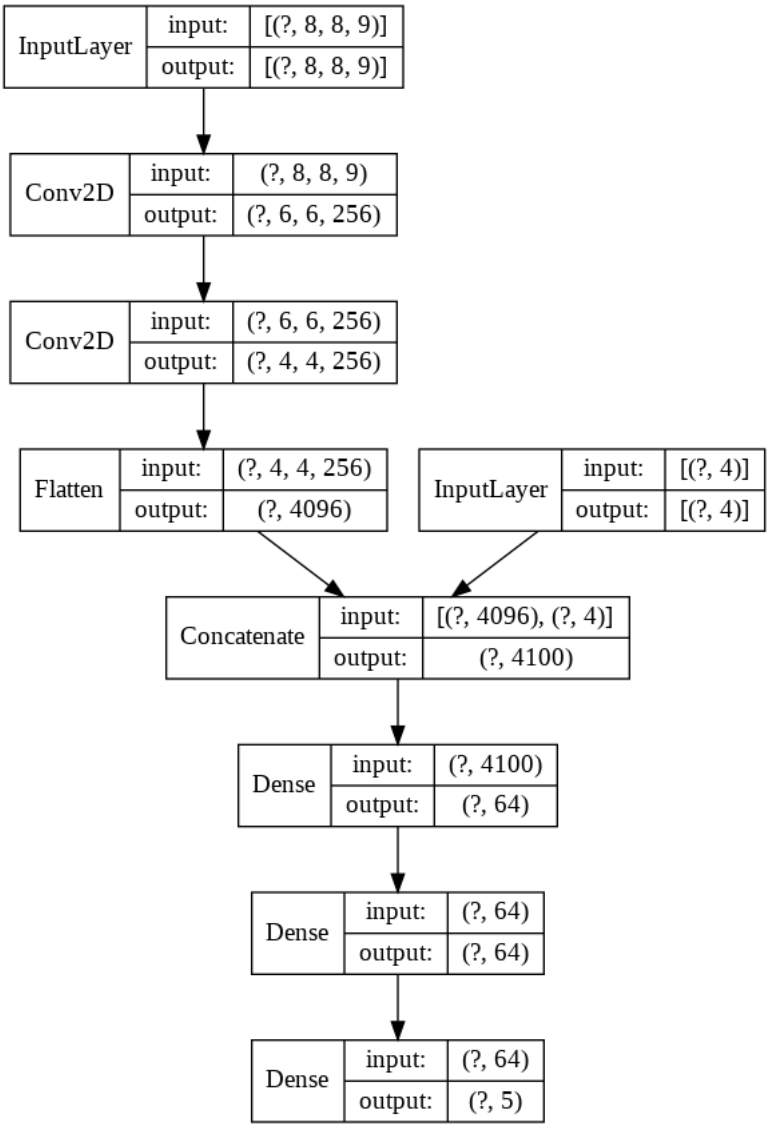

Figure C.2: Model architecture used for tunable agent in the multi-objective Gathering experiment. Note that the dropout layers after each convolutional layer and dense layer are not shown. Image generated using `tf.keras.utils.plot_model`.

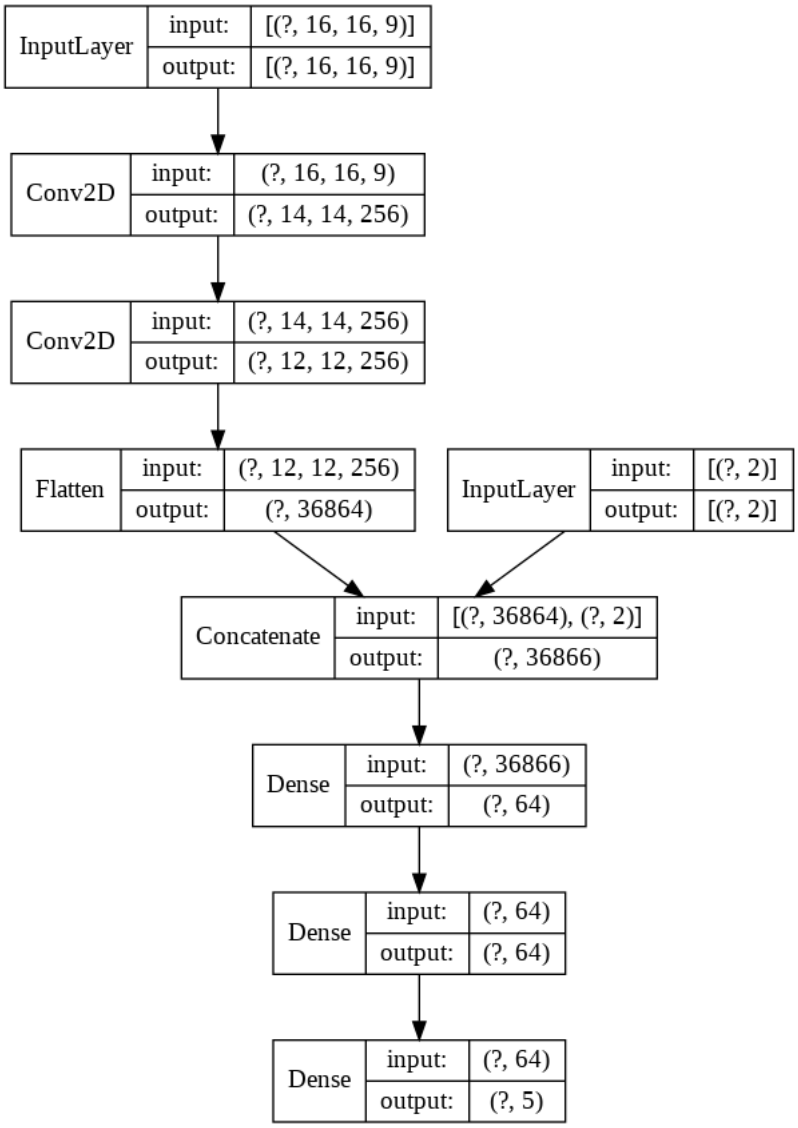

Figure C.3: Model architecture used for the tunable agents in the Wolfpack experiment. Note that the dropout layers after each convolutional layer are not shown. Image generated using `tf.keras.utils.plot_model`.