Variational Autoencoders for Opponent Modelling in Multi-Agent Systems

Georgios Papoudakis

Master of Science by Research

School of Informatics

University of Edinburgh

2019

Abstract

Multi-agent systems exhibit complex behaviors that comes from the interactions of multiple agents in a shared environment. Therefore, modelling the behavior of other agents is a key requirement in order to better understand the local perspective of each agent in the system. In this thesis, taking advantage of recent advances in unsupervised learning, we propose an approach to model opponents using variational autoencoders. Additionally, most existing methods in the literature make the assumption that models of opponents require access to opponents' observations and actions both during training and execution. To this end, we propose an inference method that tries to identify the underling opponent model, using only local information, such as the observation, action and reward sequence of our agent. We provide a thorough experimental evaluation of the proposed methods, both for reinforcement learning as well as unsupervised learning. The experiments indicate that our opponent modelling method achieves equal of better performance in reinforcement learning task than another recent modelling method and other baselines.

Acknowledgements

I would like to thank my supervisor Dr. Stefano Albrecht for the research directions and the feedback in the drafts of this thesis that he provided, which helped me to better understand my research topic. I would also like to thank my fellow students in the Autonomous Agents Research Group for the endless discussions about fascinating research topics, which helped me to get a better understanding our my research goals and directions. I would like to thank Dr. Michael Herrmann for his feedback in RRR and for answering questions regarding the thesis structure. Additionally, I would like to thank everyone involved in the RAS CDT for providing a competitive environment in order to improve myself. Finally, I would like thank Eva, my family and my friends for the continuous support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Georgios Papoudakis*)

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 2

1.3 Disclaimer Regarding the RRR and RRP 3

2 Background **4**

2.1 Deep Learning 4

2.1.1 Structure of Artificial Neural Networks 4

2.1.2 Activation Functions 5

2.1.3 Loss Functions 7

2.1.4 Gradient-Based Optimization 8

2.1.5 Back-Propagation 9

2.1.6 Fully Connected Layers 10

2.1.7 Recurrent Layers 10

2.2 Reinforcement Learning 11

2.2.1 Markov Decision Processes 11

2.2.2 Optimality 12

2.2.3 Exploration-Exploitation Dilemma 13

2.2.4 On-Policy and Off-Policy Methods 13

2.2.5 Model-Free and Model-Based Methods 14

2.2.6 Temporal Difference Methods 14

2.2.7 Policy Gradient Methods 16

2.2.8 Deep Reinforcement Learning 19

2.3 Multi-Agent Reinforcement Learning 21

2.3.1 Markov Game 22

2.3.2 Centralized and Decentralized Learning 22

2.3.3 Multi-Agent Deep Reinforcement Learning 23

2.4 Variational Inference 23
2.4.1 Variational Autoencoder 24
2.4.2 β-VAE 26

3 Related Work **27**
3.1 Learning Opponent Models 27
3.2 Representation Learning 29

4 Methodology **31**
4.1 Problem Formulation 31
4.2 Variational Opponent Modelling 31
4.3 Policy Learning 33
4.4 Inference Model 35
4.5 Training Method 35

5 Experiments **39**
5.1 Experimental Framework 39
5.1.1 Speaker-Listener Environment 39
5.1.2 Double Speaker-Listener Environment 40
5.1.3 Predator-Prey environment 40
5.1.4 Spread Environment 41
5.2 Implementation Details 42
5.2.1 Algorithm Details 42
5.2.2 Hyperparameter Optimization 43
5.3 Evaluation 45
5.3.1 Embedding visualization 45
5.3.2 Opponent Modelling Reinforcement Learning Performance . 47
5.3.3 Inference Network Reinforcement Learning Performance . . . 49
5.3.4 Inference Embeddings Evaluation 50
5.3.5 Ablation Study 51

6 Conclusion **53**
6.1 Discussion 53
6.2 Future Work 54

Bibliography **56**

List of Figures

2.1 A biological and an artificial neuron 5
2.2 An artificial neural network 5
2.3 Commonly used activation functions 7

4.1 The learning architecture of our method. The red arrows show the gradient flow between the networks. 37

5.1 Instances of different MPE environments 42
5.2 Visualization of the embedding space 46
5.3 Performance of our modelling method compared to Grover et al. [2018] 48
5.4 Performance of our method and the two baselines 50
5.5 Performance of our method for different combinations of inputs in the inference network 52

List of Tables

5.1 Possible hyperparameter values 44
5.2 Final hyperparameter values 45
5.3 Intra-Inter clustering ration 47
5.4 Agent prediction accuracy of the inference network. 51

List of Algorithms

1 Pseudocode of the learning opponent models algorithm 34
2 Pseudocode of our method 38

Chapter 1

Introduction

1.1 Motivation

Deep learning has revolutionized the development of agents that can make intelligent decisions in complex environments. Traditional reinforcement learning methods, using tabular representations or linear function approximators, could only be used in simple settings, such as small grid worlds or environments that provided a high-level state representation. Mnih et al. [2015] proposed a successful way to combine neural networks with Q-learning [Watkins and Dayan, 1992]. The combination of deep learning with the existing reinforcement learning algorithms enabled the development of agents that can act in environments with larger state space, such as images. In recent years several promising deep reinforcement learning algorithms have been proposed that can handle large state environments both with continuous and discrete action space [Schulman et al., 2015, Mnih et al., 2016, Schulman et al., 2017].

Despite this success, deep reinforcement learning still did not manage to adapt to multi-agent settings where multiple agents interact and learn concurrently in a shared environment. The alternation of the policy of each agent leads to an endless cycle of continual adaptation of each agent to the changed policies of the other agents, which significantly hinders the learning procedure. This problem is called non-stationarity and is the central issue in multi-agent learning. Many real-world environments can be characterized as multi-agent systems, where multiple agents interact under cooperative, competitive, or mixed settings.

In this thesis, we consider the problem, where multiple sets of pretrained opponents are provided, and we train one agent to interact with all of them successfully. A straightforward approach is to provide an identity to each set of opponents and learn

a policy conditioned on this identity. The issue of this approach is that access to the opponent's identity is required both for training and testing. A solution to this limitation is opponent modelling. By accurately modelling the opponents, we can train a policy conditioned on the model to solve the problem.

Inspired by the recent developments in deep unsupervised learning, we propose an opponent modelling method based on the variational autoencoder framework [Kingma and Welling, 2013]. Our method successfully manages to model opponents as well as achieve state-of-the-art results in several environments. Additionally, we identify that opponent modelling methods are based on the assumption that access to opponents' observations and actions is available. While we do not consider this assumption problematic during training, it is unrealistic during execution. More precisely, assuming access to opponents' information during execution, especially when there is no communication between the agents, is not valid in most real-world scenarios. For this reason, we also propose a method for inferring the opponents' model using only locally available information, such as the observation, action, and reward of the agent that we control.

1.2 Thesis Outline

After motivating our goals, we will now provide the outline of this thesis. In the following Chapter 2, we make a brief introduction to the theoretical background that this work is built on. We first refer to artificial neural networks and to basic deep learning and optimization methods. Following this, we refer to the reinforcement learning concepts, such as Markov decision processes, temporal difference learning and policy gradient methods briefly, and we also describe the most important recent works on deep reinforcement learning. In Chapter 3, we discuss recent deep learning-based opponent modelling methods and recent works on unsupervised learning in reinforcement learning problems. Following this, we present our method in Chapter 4, and we experimentally evaluate its performance in Chapter 5. Finally, we conclude this thesis by discussing the experimental results and proposing directions for future research in Chapter 6.

1.3 Disclaimer Regarding the RRR and RRP

Parts of this thesis have been previously submitted for achieving credits in the courses RRR and RRP. Additionally, this survey Papoudakis et al. [2019] is an extended version of the RRR, where Filippos Christianos (RAS CDT), Arrasy Rahman (UoE) and Dr. Stefano Albrecht (UoE) have participated in its writing. All parts of the survey Papoudakis et al. [2019] that are in this thesis are written by me only. The following sections are heavily influenced from the RRR, RRP and the survey: 2.2.1, 2.3, 3.1, 5.1. The following section existed in the RRR, RRP or the survey, but they have been significantly extended and altered in this thesis: 2.2.6, 2.2.7, 2.2.8.1, 2.4.1, 4.4, 4.5.

Chapter 2

Background

2.1 Deep Learning

Deep learning is a sub-field of machine learning that studies how artificial neural networks can be exploited to solve the existing machine learning problems. A property of neural networks is that they can approximate complex non-linear functions in high dimensional spaces. Neural networks have been successfully applied in all different machine learning domains, such as supervised, unsupervised, and reinforcement learning.

2.1.1 Structure of Artificial Neural Networks

Artificial neural networks are inspired by the neural networks in the brain of humans and animals. The most important component of a neural network is the neuron or node. A node takes as input a vector and outputs the cumulative value of its elements. The nodes are organized in larger sets, which are called layers. Each layer consists of several neurons, and it receives an input from the previous layer and outputs a vector that is input to the next layer. Each node in a layer connects to the nodes of the previous and the next layer through synapses. In the artificial neural networks, we consider that each synapse has a weight that is multiplied with the output of the previous node before being fed to the next node.

A neural network consists of multiple consecutive layers. The first layer that is the input to the network is called the input layer, while the last layer is called the output layer. All the intermediate layers are called hidden layers. The output of each hidden layer is usually fed in a non-linear function in order to force the neural network to

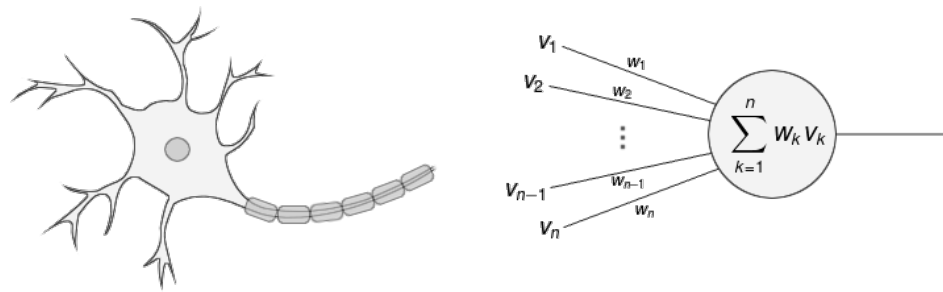


Figure 2.1: A biological and an artificial neuron
Source: `https://hackernoon.com/how-do-artificial-neural-network-recognize-images-c3699af0f553`

exhibit non-linear behavior. The goal in a neural network is to compute the weights that result in the best possible approximation of the target function. It is important to note that there is no method for learning the weights with mathematical optimality guarantees.

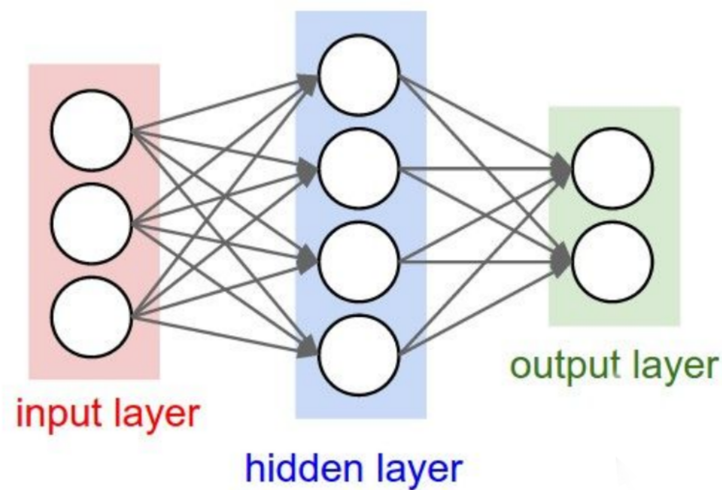


Figure 2.2: An artificial neural network
Source:
`http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture04.pdf`

2.1.2 Activation Functions

An important property of neural networks is that they can approximate non-linear functions. An activation function is used after each layer in order to force the network

to exhibit non-linearity. Another possible utility is to constrain the output of a neural network to a certain value range. Below, the most commonly used activation functions are presented.

- **Linear function.** This is a linear-function that in reinforcement learning is commonly used for the approximation of the unbounded value functions.

$$f(x) = x \tag{2.1}$$

- **Sigmoid function.** This is a non-linear function, which takes values between 0 and 1.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

- **Hyperbolic tangent function (tanh).** This is a non-linear function, which takes values between -1 and 1.

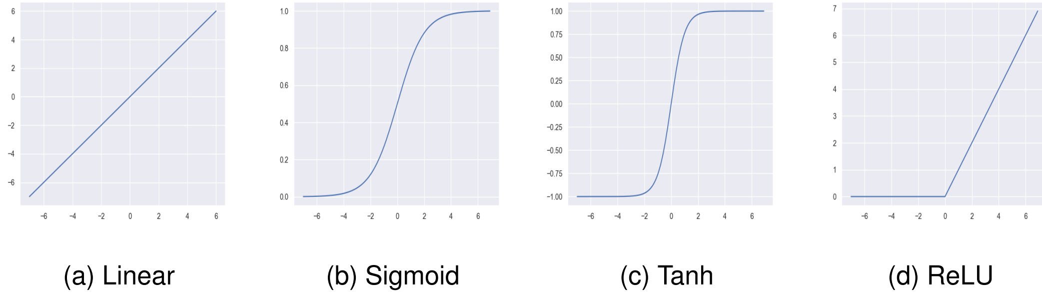
$$f(x) = \frac{e^{x}+e^{-x}}{e^{x}+e^{-x}} \tag{2.3}$$

- **Rectifier Linear Unit (ReLU).** One common problem of the two previous functions is that their gradient saturates to 0 when the input value is not close to 0 [Figure 2.3]. For training neural networks,gradient-based optimization is used as we will present in the following paragraphs. As a result, it is expected that the gradient should not be 0 at any point. ReLU [Maas et al., 2013] does not have this problem, and it is the most common choice of activation function for the hidden layers.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2.4}$$

- **Softmax function.** This is a non-linear function, which is mostly used in the output layer of a neural network. It forces two constraints in the output. Firstly, all the values are greater or equal to 0, and secondly, the cumulative value of the outputs is 1. It is commonly used when the output of a neural network has to represents a discrete probability distribution. Given $x_1, x_2, ..., x_n$ is the output of the layer.

$$f(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} \tag{2.5}$$



(a) Linear (b) Sigmoid (c) Tanh (d) ReLU

Figure 2.3: Commonly used activation functions

2.1.3 Loss Functions

A neural network is initialized with random parameters at the beginning of the training. Training refers to the procedure that the neural network's parameters are modified in order to represent the required behavior. The goal is the output of the neural network to approximate some target function. Therefore, a straightforward approach is to define a loss function, which is called objective, between the output of the neural network and the target. The parameters are fitted in order to minimize the value of the loss. There is a large number of the loss function; however, in this thesis, the followings will be used:

- **Mean squared error**. This is the average of the squared residual between the target value t_i and the output of the neural network y_i. So, the mean squared error is defined as:

$$L = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - t_i)^2 \tag{2.6}$$

- **Kullback–Leibler divergence**. The Kullback–Leibler (KL) divergence measures the difference between two distributions:

$$\begin{aligned} &KL(p(x)||q(x)) = \int p(x) \log \frac{p(x)}{q(x)} dx = \\ &\mathbf{E}_{x \sim p(x)}[\log p(x)] - \mathbf{E}_{x \sim p(x)}[\log q(x)] = \\ &CE(p(x)||q(x)) - H(p(x)) \end{aligned} \tag{2.7}$$

  The KL divergence is the cross entropy between the two distributions minus the entropy of the first distribution. At this point, we have to mention that the KL divergence is not a distance metric, because it is not symmetric $KL(p(x)||q(x)) \neq KL(q(x)||p(x))$ and that the KL divergence is non-negative $KL(p(x)||q(x)) \geq 0$. The KL divergence is zero, only when the two distributions are identical.

Note that under the reinforcement learning setting that we will study in this thesis, we will define different objective functions to minimize.

2.1.4 Gradient-Based Optimization

After defining a loss function, which is called the objective function in the optimization theory, between the output of the network and the target behavior, we have to compute the parameters of the neural network that minimize the average loss of overall training samples. The most common optimization method that is used in this case is gradient-based optimization. Given a differentiable loss function $L(w)$ which depends on the parameters w, the gradient $\nabla_w L(w)$ provides the direction in the parameters space that L has the maximum ascent. Therefore, the negative gradient $-\nabla_w L(w)$ provides the direction of maximum descent. As a result, by iteratively following the direction of maximum descent, we can compute the parameters that minimize the loss function.

$$w \leftarrow w - \eta \nabla_w \frac{1}{N} \sum_{i \in T} L_i(w) \tag{2.8}$$

where T is the set of training samples and η is a small value called learning rate. This algorithm is called gradient descent.

Practically, performing gradient descent to the whole dataset simultaneously requires a lot of computational power and memory, making gradient descent inefficient for most practical problems. The simple and efficient solution is mini-batch gradient descent. Instead of computing the gradient on all the samples in the dataset, a batch with size n, where $n \ll N$, is sampled from the dataset $b \sim T$. Only the samples of the batch are used in a single gradient update.

$$w \leftarrow w - \eta \nabla_w \frac{1}{n} \sum_{i \in b} L_i(w) \tag{2.9}$$

Mini-batch gradient based optimization is the most common choice for optimization in deep learning.

A well-known problem is the selection of the learning rate in gradient descent. A small learning rate may result in slow convergence, while a large learning rate will probably lead to fluctuation and instability of the objective function. Additionally, Sutton [1986], Ruder [2016] reason that gradient descent suffers from convergence when the steepness of the objective function significantly differs between different dimensions. A common solution to this problem is the computation of an adaptive learning rate in each dimension. There is a large number of different works in this

direction; however, in this thesis, we focus on the most popular optimization method in deep learning, the Adaptive Moment Estimation (Adam) optimizer [Kingma and Ba, 2014]. Adam optimizer keeps a record of the running mean and the running variance of the optimization procedure.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_w L(w) \\ u_t &= \beta_2 u_{t-1} + (1-\beta_2)(\nabla_w L(w))^2 \end{aligned} \tag{2.10}$$

where the typical values for the parameters are $\beta = 0.9$ and $\beta_2 = 0.999$ and the initial values m_0 and u_0 are initialized to 0. Due to this initialization the estimations are biased and as a result, Kingma and Ba proposed the following modification

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{u}_t &= \frac{u_t}{1-\beta_2^t} \end{aligned} \tag{2.11}$$

Finally, at each time step the parameters are updated as:

$$w \leftarrow w - \frac{\eta}{\sqrt{\hat{u}_t} + \varepsilon}\hat{m}_t \tag{2.12}$$

where $\varepsilon = 10^{-8}$ to prevent the denominator to reach zero. Adam optimizer resulted in improved performance in many benchmarks and is the most commonly used optimization method in reinforcement learning problems.

2.1.5 Back-Propagation

Previously, we presented optimization methods for minimizing the loss of a neural network and trains its parameters. These methods require the gradient of the loss with respect to the parameters of the neural network. These can be efficiently computed using the Back-Propagation algorithm. Back-propagation is based on the chain rule of differentiation, and it is applied backward, from the output to the input. Starting from the loss L, we first estimate the gradient of the loss with respect to the output of the neural network y. Consider a simple case where $y = a(w^T x)$ where w are the weights, a is the activation function and x the input. We can now compute the gradient of the loss with respect to the product $w^T x$ using the chain rule as:

$$\frac{\partial L}{\partial w^T x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w^T x} = \frac{\partial L}{\partial y}a' \tag{2.13}$$

Following this, we can easily compute the gradient of the loss with respect to the parameters w as:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial w^T x}\frac{\partial w^T x}{\partial w} = \frac{\partial L}{\partial w^T x}x \tag{2.14}$$

Following this rule, we can compute all the required gradients for performing gradient-based optimization. Note that the back-propagation rule is an efficient algorithm having the same computation complexity as the forward computation of the neural network.

2.1.6 Fully Connected Layers

The most common type of layers in neural networks is the fully connected layers. Consider that the input to the layer is a batch X with dimensions $M \times 1$, where M is the input size. Given that the weights of the layer W is a matrix $M \times K$ and the bias b is a vector $K \times 1$, where K is the dimension that the data will be transformed to. The output of the layer is:

$$Y = W^T x + b \tag{2.15}$$

2.1.7 Recurrent Layers

A drawback of fully connected layers is the lack of the ability to model time-related data, such as time-series. A solution to this problem is the use of recurrent neural networks (RNN). RNNs are an extension of the fully connected layer in order to model time dependencies. The simplest variant of RNN, the vanilla RNN, in a fully connected layer that at each time step computes the new output based on its previous output and the output of the previous layer.

$$y_t = W^T x + U^T y_{t-1} + b \tag{2.16}$$

Where W and U are weight matrices. During training, the gradient computation of back-propagation, the gradient flows thought the recurrent gate to previous time steps. This results in the multiplication of the same number of matrices as the unrolling steps of the RNN. Due to the fact that the gradient matrices are unconstrained, a large number of multiplications can lead either the gradient to zero or vast numbers. This is called the vanishing or exploding gradient effect. When the gradients vanish, the gradient-based optimization does not work, while when the gradients explode the gradient-based optimization results in instability.

A more sophisticated variant of a recurrent network that addresses this issue is the Long Short Term Memory (LSTM) layer [Schmidhuber and Hochreiter, 1997]. An LSTM has three gates; the forget gate, the input gate, and the output gate, that help to

address this problem. The activation of the gates are computed as

$$\begin{aligned} f_t &= sigmoid(W_f^T x + U_f^T h_{t-1} + b_f) \\ i_t &= sigmoid(W_i^T x + U_i^T h_{t-1} + b_i) \\ o_t &= sigmoid(W_o^T x + U_o^T h_{t-1} + b_o) \end{aligned} \tag{2.17}$$

where x is the input and h_{t-1} the previous output of the LSTM. In the LSTM, there is also an internal variable, which is called cell state c and the hidden state or output of the LSTM h. They are computed as:

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot tanh(W_c^T x + U_c^T h_{t-1} + b_c) \\ h_t &= o_t \odot tanh(c_t) \end{aligned} \tag{2.18}$$

This gating mechanism partially addresses the vanishing or exploding gradient problem in recurrent network. Note that many times LSTMs suffer from these issues and other tricks are used, such as clipping the norm of the gradient, in order to stabilize training. Additionally, the advantage of LSTMs comes with an increased number of parameters, four times more compared to a vanilla RNN, leading to increased computational cost and difficulty in the optimization.

2.2 Reinforcement Learning

2.2.1 Markov Decision Processes

We can mathematically model a decision making problem using Markov Decision Processes (MDP). An MDP is defined as (S, A, r, T), where S is the set of possible states, A is the set of available actions, $r : S \times A \times S \to \mathbf{R}$ is the reward function and $T : S \times A \times S \to [0,1]$ is the probability distribution over next states given the current state and action. Additionally, in every MDP we define either a stochastic policy function $\pi : S \times A \to [0,1]$ or a deterministic policy function $\mu : S \to \mathbf{R}^{|A|}$, which selects an action for the current state. The return from a current state s_t at time step t is defined as the sum of discounted rewards from t until the horizon H:

$$G_t = \sum_{i=t}^{H} \gamma^{i-t} r_i \tag{2.19}$$

where γ is the discount factor, taking values in $[0,1]$ and r_i is the reward after i time steps. Given a policy π, an MDP has two value functions:

- The state value function $V : S \rightarrow \mathbf{R}$, which is the expected return from a given state under the policy π.

$$V^{\pi}(s_t) = \mathbf{E}_{\pi}[G_t|s_t] \tag{2.20}$$

- The state-action value function $Q : S \times A \rightarrow \mathbf{R}$ which is the expected return from a given state and action under the policy π.

$$Q^{\pi}(s_t, a_t) = \mathbf{E}_{\pi}[G_t|s_t, a_t] \tag{2.21}$$

Using Bellman equations, the value functions can be expressed in an iterative form. That is, an expression of the value function at the current time step using the next time step:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} T(s,a,s')[r(s,a,s') + \gamma V^{\pi}(s')] \tag{2.22}$$

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')[r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a')] \tag{2.23}$$

In order to solve an MDP, we have to compute the optimal policy. That is the policy that maximizes the expected discounted sum of rewards $V^*(s) = \max_{\pi} V^{\pi}(s)$. If we have access to the reward and transition function, this problem can be solved by iterating the Bellman equations and using dynamic programming [Bertsekas, 2005]. However, we do not know these two functions in most scenarios. In this case, we use reinforcement learning [Sutton and Barto, 1998] to compute the optimal policy.

2.2.2 Optimality

As previously mentioned, the goal of an MDP is to compute the policy that maximizes the expected return. In other words, we have to compute the policy that maximizes the state value function. Denoting the optimal policy as π^*, we have that $\pi^*(s) = \arg\max_{\pi} V^{\pi}(s)$. The notion of the optimal policies is also extended to the value functions. Given an optimal policy π^*, the optimal state value function V^* and the optimal state-action value function Q^* are:

$$V^* = \max_{\pi} V^{\pi} \tag{2.24}$$

$$Q^* = \max_{\pi} Q^{\pi} \tag{2.25}$$

Additionally, under an optimal policy, the value function can be computed from the state-action value function as

$$V^*(s) = \max_a Q^*(s,a) \tag{2.26}$$

Finally, the Bellman equations that connect the value functions of two consecutive steps can also be expressed under the assumption of optimal policy as:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[r(s,a,s') + \gamma V^*(s')] \tag{2.27}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[r(s,a,s') + \gamma \max_{a'} Q^*(s',a')] \tag{2.28}$$

2.2.3 Exploration-Exploitation Dilemma

The exploitation-exploration dilemma refers to the trade-off between exploiting the existing belief of the agent in the action selection or exploring the action space that may lead to higher returns. By exploiting, the agent uses the prior training, and at each state, the action that has the highest expected return is selected. Therefore, if the estimation of the agent about the expected returns of each action is accurate, exploiting will result in high reward returns. On the other hand, in the early stages of the training that the agent's estimation of the returns is inaccurate, following the optimal action, according to the agent's belief, will result in sub-optimal returns. Additionally, the agent will never be able to explore states and actions that would result in higher rewards. As a result, during training, the agent should explore all different actions randomly to discover states that have higher returns. The process of balancing between exploring and exploiting during the training procedure is called the exploration-exploitation dilemma, and it is crucial for fast and efficient learning. In the following paragraphs, we will refer to the most common exploration techniques depending on the reinforcement learning method.

2.2.4 On-Policy and Off-Policy Methods

The reinforcement learning methods that will be described in the following paragraphs can be categorized into two classes: on-policy methods and off-policy methods. In contrast to other machine learning areas, such as supervised and unsupervised learning, the training data is not available before training. This means that the agent has to follow different trajectories, and these trajectories will be used for training. In the on-policy methods, the policy is updated using the same policy that the trajectory was generated

from. On the other hand, in off-policy methods, the policy is updated using a different policy from the one that the trajectory was generated from.

A significant advantage of on-policy methods is that they tend to be more stable with respect to initial conditions. However, due to the fact that the on-policy methods use the same policy for trajectory generation and learning, the trajectories that were generated in previous episodes cannot be used anymore. Using trajectories that were generated with a different policy would introduce a bias in the learning algorithm. On the other hand, in the off-policy methods, the previously generated trajectories can be reused since a different policy is used for updating the equations. As a result, previous experience can be reused for training leading to significantly more sample-efficient reinforcement learning algorithms.

2.2.5 Model-Free and Model-Based Methods

Another categorization that can be used in the context of reinforcement learning is model-free and model-based methods. Model-based methods refer to the algorithms that try to estimate properties of the given MDP using the generated trajectories and then calculate the optimal policy conditioned on the estimated properties. One example is to try to learn the transition function T, and the reward function r and then use value iteration or policy iteration to compute the optimal policy. Practically, in MDP with large state space, it is difficult to learn such descriptive functions. On the other hand, model-free methods estimate the optimal policy without explicitly computing a model of the MDP. In reinforcement learning, this is the most common practice for policy optimization. In the following paragraphs, we will describe the most common model-free algorithms. Note that the model-free algorithms estimate the policy given the state, $\pi(a|s)$ in the stochastic case and $\mu(s)$ in the deterministic case.

2.2.6 Temporal Difference Methods

A big category of model-free reinforcement learning algorithms is the temporal difference (TD) methods. These methods compute the value functions (and mostly the state-action value function) by interacting with the environment. The value functions are computed by minimizing the squared temporal difference error. The TD error is the residual between the state value (or the state-action value) at the current states and the TD target, which is defined in the form of:

$$target = r + \gamma \cdot K \tag{2.29}$$

where K is some expression of the value of the next state, either with respect to the state or the state-action value.

The simplest TD algorithm is TD(0). TD(0) can be used for estimating the state value function V. The target in the TD(0) algorithm is:

$$target = r + \gamma \cdot V(s') \tag{2.30}$$

Assuming that the state value function (Q-values) can be expressed in the tabular form (save a different value for every state) or can be approximated using a parameterized function, we can compute it by minimizing the following loss function:

$$L = \frac{1}{2}\mathbf{E}[(r + \gamma \cdot V(s') - v(s))^2] \tag{2.31}$$

This loss can be minimized using a gradient optimizer, either with respect to the value function (in the tabular case) or with respect to the parameters. In the tabular case, the expression is:

$$V(s) \leftarrow V - \eta(r + \gamma \cdot V(s') - V(s)) \tag{2.32}$$

Using function approximation with parameters w:

$$w \leftarrow w - (r + \gamma \cdot V(s') - V(s))\nabla_w V(s) \tag{2.33}$$

There are various TD methods, and we are not going to refer to all of them. The most famous method is Q-learning [Watkins and Dayan, 1992]. Similarly to TD(0), assuming that the state-action value function (Q-values) can be expressed in the tabular form (save a different value for every state and action) or can be approximated using a parameterized function, we can compute it by minimizing the following loss function:

$$L = \frac{1}{2}\mathbf{E}[(r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a))^2] \tag{2.34}$$

From this equation the update equations for the tabular case and the function approximation case can be derived:

$$Q(s,a) \leftarrow Q(s,a) - \eta(r + \gamma \cdot \max_{a'} Q(s', a') - Q(s,a)) \tag{2.35}$$

$$w \leftarrow w - (r + \gamma \cdot \max_{a'} Q(s', a') - Q(s,a))\nabla_w Q(s,a) \tag{2.36}$$

Exploration in Q-learning is implemented through epsilon-greedy policies. Assuming ε is a variable taking values in $[0,1]$, the epsilon-greedy policy follows the optimal action, $a = \arg\max_a Q(s,a)$, with probability $1 - \varepsilon$ or a random action with probability

ε. There are various strategies for selecting the value of ε. One typical strategy is to start with $\varepsilon = 1$ and decrease that throughout the training to 0. During the evaluation, randomness in action selection is unwanted, and as a result, the value of ε is usually set to 0. Additionally, Q-learning is an off-policy algorithm. Note that the trajectories are gathered using epsilon-greedy, while the TD target is calculated using the greedy policy; that is the policy that maximizes the value of the next state.

2.2.7 Policy Gradient Methods

Another model-free alternative to TD algorithms is the policy gradient methods. Considering a parameterized by parameters θ, our goal is to compute the policy that maximizes the expected discounted sum of rewards from the first state. Therefore, the objective function is $J = V^{\pi}(s_0)$. Using gradient ascent, we follow the direction in the parameter space that maximizes our objective:

$$\theta \leftarrow \theta + \eta \nabla_{\theta} V^{\pi}(s_0) \tag{2.37}$$

where η is the learning rate.

The missing element is the computation of the gradient of the true state-value, which is unknown with respect to the parameters of the policy. This can be computed using the policy gradient theorem. Below we present its two variances for the stochastic and the deterministic policy respectively.

2.2.7.1 Stochastic Policy Gradient

Consider a stochastic policy π_{θ} parameterized by parameters θ. Using the policy gradient theorem [Sutton et al., 2000] we can compute the gradient of the objective with respect to the parameters of the policy and use a gradient optimizer to maximize our objective:

$$\nabla_{\theta} J \propto \sum_{s} d(s) \sum_{a} \nabla_{\theta} \pi(a|s) Q(s,a) \tag{2.38}$$

where $d(s)$ is called on-policy distribution, which is the distribution over the visited states of the MDP, when the policy π is followed. In the expression, the summation operators can be replaced by expectation. As the result the policy gradient can be rewritten as:

$$\nabla_{\theta} J = \mathbf{E}_{s,a}[Q(S_t, A_t) \nabla_{\theta} \log \pi_{\theta}(A_t|S_t)] \tag{2.39}$$

where S_t is sampled from the on-policy distribution and the action A_t is sampled from the policy $\pi(a|S_t)$. In this equation, the state-action value function $Q(s,a)$ is unknown. One way to estimate the Q value is using the return in the specific state $Q(S_t,A_t) = \mathbf{E}[G_t|S_t,A_t]$. This algorithm is called REINFORCE [Williams, 1992] and the value of the policy gradient is estimated as:

$$\nabla_\theta J = \mathbf{E}_{s,a}[G_t \nabla_\theta \log \pi_\theta(A_t|S_t)] \tag{2.40}$$

The above expectation is over all possible actions and states of the environment. In environments with large state and action space it is practically impossible to compute this expectation. For this reason a Monte Carlo approximation is used:

$$\nabla_\theta J = \mathbf{E}_{s,a}[G_t \nabla_\theta \log \pi_\theta(A_t|S_t)] = \frac{1}{N}\sum_{t=0}^{N-1} G_t \nabla_\theta \log \pi_\theta(A_t|S_t) \tag{2.41}$$

While this gradient computation is unbiased, it suffers from high variance. In order to reduce the variance, a baseline function is subtracted from the return G_t. If the baseline function b is not a function of the action a, then the estimation remains unbiased, because:

$$\begin{aligned}
&\mathbf{E}_{s,a}[b(s)\nabla_\theta \pi_\theta(A_t|S_t)] = \mathbf{E}_s[\sum_a b(s)\nabla_\theta \pi_\theta(a|S_t)] = \\
&\mathbf{E}_s[b(s)\nabla_\theta \sum_a \pi_\theta(a|S_t)] = \mathbf{E}_s[b(s)\nabla_\theta 1] = 0
\end{aligned} \tag{2.42}$$

A baseline that satisfy this requirement, and results in minimal variance in the estimation is the state value function V. Since, we do not know the state value function we will approximate with a function that has parameters w, using the TD(0) algorithm that minimizes the following error. This algorithm is called actor-critic [Konda and Tsitsiklis, 2000].

$$L = \frac{1}{2}\mathbf{E}[(r + \gamma V(s') - V(s)^2] \tag{2.43}$$

The actor is responsible for computing the policy using the policy gradient equation, while the critic is responsible for estimating the state value function. The original actor-critic algorithm is on-policy. However, an off-policy version of this method has also been proposed by Wang et al. [2017]. In this algorithm, the exploration during training is implemented using soft-policies. Due to the fact that the policy is a probability distribution over the actions, during training, the exploration is performed by sampling this distribution, instead of choosing the most probable action.

2.2.7.2 Deterministic Policy Gradient

Silver et al. [2014] extended the policy gradient methods to deterministic policies. Consider a deterministic policy μ_θ parameterized by parameters θ. Using the deterministic policy gradient theorem, Silver et al. [2014] proved that the gradient of the value function for a deterministic policy is:

$$\nabla_\theta J = \mathbf{E}[\nabla_\theta \mu(s) \nabla_a Q(s,a)|_{a=\mu(s)}] \tag{2.44}$$

The state-action value function is unknown and it is computed using Q-learning. Therefore, the following error is minimized for the estimation of the parameters of the Q-values.

$$L = \frac{1}{2}\mathbf{E}[(r + \gamma Q(s', \mu(s')) - Q(s, \mu(s))^2] \tag{2.45}$$

At this point, note that in the above equation, there is not the max operator over the actions of the next state. That is because, in the deterministic policy gradient methods, the action that maximizes the state-action value is directly outputted from the actor. This algorithm is off-policy. Since the action selection is deterministic, it is not obvious how we can use sampling (such as epsilon-greedy or soft-policies) for exploration. Lillicrap et al. [2015] proposed to add Ornstein-Uhlenbeck noise in the output of the policy, while Fujimoto et al. [2018] used noise from a normal distribution.

Additionally, in equation 2.44, there is a term that computes the gradient of the state-action value function with respect to actions. As a result, it is required the action space to be continuous. An extension of the deterministic policy gradient method can be achieved using the Gumbel-Softmax distribution [Jang et al., 2016, Maddison et al., 2016]. The Gumbel-Softmax process is defined as:

$$y = Softmax(x+q), \;\; q = -log(-log(u)) \;\; u \sim Uniform[0,1] \tag{2.46}$$

Jang et al. [2016], Maddison et al. [2016] proposed a trick to get differentiable samples from a categorical distribution. Given the deterministic policy for a discrete action space $\mu(s)$, by computing $a = \arg\max_a \mu(s)$, the gradient cannot flow through the argmax operation, because it is not differentiable. For this reason, the relaxed differentiable version of max, Softmax, is used. The differentiable sample is computed as:

$$x_{diff} = (x-y).stop_gradient() + y \tag{2.47}$$

Therefore, action computation is unbiased, because $x_{diff} = x$ and the gradient is only computed through the differentiable part y. This results in low biased differentiable

samples from the discrete distribution. The Gumbel process, that is the additional noise, serves the purpose of exploration.

2.2.8 Deep Reinforcement Learning

All the previously mentioned algorithms can be used with deep networks for approximation. Using deep networks has enabled the development of agents that are able to efficiently act in complex environments with high dimensional state and action space. Significant, milestones for the deep reinforcement learning community are the development of agents with superhuman performance in the Atari games using the visual input [Mnih et al., 2015] and the development of an agent that beat the world champion in the game of Go [Silver et al., 2017].

2.2.8.1 Issues in Deep Reinforcement Learning

A naive implementation of deep reinforcement learning can be implemented by directly approximate any function, such as the policy or the value functions using neural networks. However, in most cases, this attempt is unsuccessful for two main reasons that we describe below. Note that the following problems always existed in reinforcement learning, but they are mostly highlighted when deep networks are used.

- The training dataset in reinforcement learning is not provided before training. Therefore, the agent has to explore different trajectories and use them for training. The problem is that all these consecutive samples are usually highly correlated, leading in updates with high variance and instability.

  Two leading solutions have been proposed for this issue. [Mnih et al., 2015] proposed the use of an experience replay. This is a buffer that all samples that are generated during exploration are stored. During training, the agent uniformly samples a batch from the experience replay and uses it for training. A core issue of this method is that it cannot be combined with on-policy methods, because the previous experience stored in the buffer has not to be generated from the current training policy of the agent, resulting in a bias in these algorithms. The second solution is the use of a parallel framework. Two distinct versions of this solution have been proposed; an asynchronous one [Mnih et al., 2016] and a synchronous one [Schulman et al., 2015, 2017]. In the asynchronous method, there is a global network, and multiple copies are created, called the worker.

Each worker interacts independently with a different copy of the environment and asynchronously updates the global network. Similarly, in the synchronous version, the generated trajectories of each worker are gathered, and the update of the global network is performed in a synchronous fashion. The parallel framework can be used both with on-policy and off-policy methods. At this point, we have to note that the experience replay significantly improves the sample efficiency of the training, because the generated samples can be reused, and it is mostly used when this option is available. As a result, off-policy methods tend to be more sample efficient compared to their on-policy counterparts in the deep reinforcement learning settings.

- The second issue is the fast-moving bootstrapping target in the TD methods. Consider for example, the algorithm Q-learning, where both the value for a state-action pair $Q(s,a)$ and the training target $r+\gamma \max_{a'} Q(s',a')$ is generated from the same neural network. As the optimization changes the parameters of the neural network, the target in the Q-learning drifts. This results in noisy updates and instability. The proposed solution is the use of a second network for the target computation, which changes slowly, called target network [Mnih et al., 2015]. The target network is a copy of the original network, that remains constant or changes slowly in order to stabilize the target. There are two proposed solutions for updating the target network. It can either copy the parameter of the target network every a constant large (compared to the training frequency) number of steps [Mnih et al., 2015], or slowly update the parameters at every training step using the Polyak averaging [Lillicrap et al., 2015], $w_{target} \leftarrow (1-\tau)w_{target} + \tau w,$, where $\tau \ll 1$.

2.2.8.2 Important Works

Using the two previous solutions, the combination of deep networks with reinforcement learning is straightforward. The recent year a large number of works have been proposed that use deep networks for policy and value approximation. The first work in deep reinforcement learning was the Deep Q-Network (DQN) [Mnih et al., 2016], which uses a neural network for the approximation of the state-action value in Q-learning. DQN successfully solved a plethora of Atari games using only the visual input. Incremental improvements of DQN is the prioritized experience replay of Schaul et al. [2015], which changes the sampling method from the experience replay and the double DQN

[Van Hasselt et al., 2016] which uses two target networks in order to reduce the overestimation of Q-learning.

Regarding the policy gradient methods, Mnih et al. [2016] proposed the asynchronous framework and applied that to the actor-critic algorithm, resulting in the asynchronous advantage actor-critic (A3C). A3C outperformed all the previous approaches in the Atari domain both in performance and in training time. Schulman et al. [2015] proposed the trust region policy optimization (TRPO) algorithm. It is a policy gradient algorithm that guarantees that every optimization step improves the policy, given that the new resulting policy is within a certain region from the old policy. In order to satisfy this constraint with gradient-based optimization, it is required the computation of second-order derivatives, which significantly increases the computational cost. Schulman et al. [2017] proposed proximal policy optimization (PPO) relaxation of the constraint, which requires first-order optimization. PPO achieved the same or better performance compared to TRPO while requiring significantly less computational power. All these algorithms are on-policy and require a parallel framework in order to perform efficiently.

On the other hand, a number of off-policy methods proposed. Lillicrap et al. [2015] combined the deterministic policy gradient algorithm with deep networks (DDPG). DDPG performs well in a number of continuous robotic tasks. Fujimoto et al. [2018] proposed the twin delay DDPG (TD3) a variant of DDPG, which uses two critic networks for the state-action value function approximation, which results in the reduction of the overestimation error in the critic. TD3 outperformed all the previous approaches, such as DDPG and PPO, in a large number of continuous tasks. Finally, Haarnoja et al. [2018] proposed the soft actor-critic algorithm, an off-policy variant of the original actor-critic algorithm, which optimizes an entropy regularized objective. These algorithms are off-policy and present considerably better sample efficiency compared to the on-policy variants.

2.3 Multi-Agent Reinforcement Learning

During this thesis, we are interested in multi-agent environments. For this reason, in this section, a brief introduction to multi-agent reinforcement learning is provided.

2.3.1 Markov Game

A Markov Game or Stochastic Game [Littman, 1994] is a generalization of an MDP in multiple agents. A Markov Game is defined as:

- The set of agents $I = \{1, 2, ...N\}$.
- The set of states S.
- The set of action for each agent $A = A_1 \times A_2 \times ...A_N$.
- The transition probability $T : S \times A \times S \rightarrow [0, 1]$.
- The reward function $r : S \times A \times S \rightarrow \mathbf{R}^N$. The reward is defined per agent $r = (r_1, r_2, ..., r_N)$

The goal of each agent i is to maximize its own expected discounted cumulative reward $R_i = \mathbf{E}[\sum_{t=0} \gamma^t r_{i,t}]$.

2.3.2 Centralized and Decentralized Learning

There are two major architectures that can be used for training in multi-agent reinforcement learning; the centralized and the decentralized. In the centralized architecture, all the agents are modeled as one entity, and a single agent reinforcement learning algorithm is used. The input to the learning algorithm is the joint state of the agents, and the output is the joint action. The joint state consists of the concatenation of the observations of all agents, and the joint action consists of the combination of the actions of all agents. Therefore, the input grows linearly, and the output exponentially with respect to the number of agents. This significantly hinders the training procedure when the number of agents increases or when the state and action spaces are high-dimensional. An important work in this direction is the Joint Action Learners (JAL) algorithm [Bowling and Veloso, 2002].

On the other hand, in the decentralized architecture, each agent is trained independently from the others. Despite the fact that this method does not suffer from scalability issues, many different problems arise as well. Some of these issues are non-stationarity of the environment due to multiple agents changing their policy at the time, the credit assignment problem, and lack of explicit coordination. The research community has identified that the decentralized methods are more promising and has focused on solving the previously mentioned problem. Multiple works have been proposed [Papoudakis

et al., 2019], such as independent Q-learning [Tan, 1993], Wolf-PHC [Bowling and Veloso, 2001] and the AWESOME algorithm [Conitzer and Sandholm, 2007].

2.3.3 Multi-Agent Deep Reinforcement Learning

Despite the success of deep reinforcement learning, it has only partially adapted in multi-agent settings. The most successful architecture in multi-agent deep reinforcement learning is the centralized training and the decentralized execution. For this architecture, the actor-critic algorithm is used, which consists of two components. The critic is trained centralized and has access to the observations and actions of all agents, while the actor is trained decentralized. Since the actor computes the policy, we only use this component during testing, and therefore, we have a fully decentralized execution.

Foerster et al. [2018] used the actor-critic algorithm with stochastic policy to train agents in Starcraft. The authors proposed a single centralized critic for all the agents, and a different actor for each agent. Additionally, they proposed a modification in the advantage estimation of the actor critic.

$$A(s,a) = Q(s,a) - \sum_{a'_i} \pi_i(a'_i|o_i) Q(s,(a_{-i},a'_i)) \tag{2.48}$$

This modification serves two purposes. First of all, the advantage estimation, that is used in the policy gradient is conditioned on the observations and actions of all the agents. Since each agent has access to the observations and actions of all the other agents, they manage to deal with non-stationarity. Additionally, this counterfactual advantage can be interpreted as the value of action a compared to all the other action values. As a result, this advantage is used in order to address the credit assignment problem in the multi-agent system, and this is the core contribution of the paper discussed.

On the other hand, Lowe et al. [2017] proposed a multi-agent architecture using the deterministic policy gradient [Silver et al., 2014] algorithm (MADDPG). In this method, each agent has its own centralized critic and decentralized actor. Since the training of each agent is conditioned on the observation and action of all the other agents, each agent perceives the environment as stationary.

2.4 Variational Inference

Variation inference is a collection of methods for the approximation of intractable distributions. Consider a posterior distribution $q(z|x)$ which has a complex functional

form and sampling from this distribution is computationally inefficient. The goal is to approximate this distribution using a known distribution p_θ with parameters θ. This is called variational distribution, and a common choice is the Gaussian, where $\theta = (\mu, \Sigma)$. We want to minimize the following term:

$$\begin{aligned} & KL(p(z|x)||q(z|x)) = \int p(z|x) \log \frac{p(z|x)}{q(z|x)} dz = \\ & \int p(z|x) \log p(z|x) dz - \int p(z|x) \log q(z|x) dz = \\ & \mathbf{E}_{z \sim p(z|x)}[\log p(z|x)] - \mathbf{E}_{z \sim p(z|x)}[\log q(z|x)] \end{aligned} \tag{2.49}$$

From the Bayer rule, the posterior $q(z|x)$ can be rewritten as:

$$q(z|x) = \frac{q(x|z)q(z)}{q(x)} \tag{2.50}$$

By substituting this to equation 2.49, we have that:

$$\begin{aligned} & KL(p(z|x)||q(z||x)) = \mathbf{E}_{z \sim p(z|x)}[\log p(z|x)] - \\ & \mathbf{E}_{z \sim p(z|x)}[\log q(x|z)] - \mathbf{E}_{z \sim p(z|x)}[\log q(z)] + \log p(x) \end{aligned} \tag{2.51}$$

The term $p(x)$ does not depend on z and therefore the expectation is removed. As previously mentioned, the KL divergence is always greater or equal to 0. As a result:

$$\mathbf{E}_{z \sim p(z|x)}[\log p(z|x)] - \mathbf{E}_{z \sim p(z|x)}[\log q(x|z)] - \mathbf{E}_{z \sim p(z|x)}[\log q(z)] + \log p(x) \geq 0 \tag{2.52}$$

Note that

$$KL(p(z|x)||q(z)) = \mathbf{E}_{z \sim p(z|x)}[\log p(z|x)] - \mathbf{E}_{z \sim p(z|x)}[\log q(z)]$$

Therefore, the inequality 2.52 can be rewritten as:

$$\log p(x) \geq \mathbf{E}_{z \sim p(z|x)}[\log q(x|z)] - KL(p(z|x)||q(z)) \tag{2.53}$$

The quantity $p(x)$ is called model evidence, and we want this to be maximum. Inequality 2.53 provides a lower bound in the evidence, which is called Evidence Lower Bound (ELBO). Since there is no closed-form equation for the quantity $p(x)$, it is possible to maximize it, by maximizing the ELBO.

2.4.1 Variational Autoencoder

Variational autoencoders [Kingma and Welling, 2013] is a family of unsupervised learning methods and generative models that learn a low dimensional probabilistic

representation of the data. In terms of architecture, they look similar to classic autoencoders. Variational autoencoders use the ELBO to learn a low dimension probabilistic representation of the data. It consists of two parts, which are usually implemented using two neural networks. The first neural network, the encoder, represents the variational distribution $p_\theta(z|x)$. The second neural network, the decoder, is used for the reconstruction of the input, which is the first term of the ELBO, $q(x|z)$. Consider the prior in the latent variables is a Gaussian, with zero mean and unit variant $q(z) = \mathcal{N}(z;0,\mathbf{I})$. Note that while a Gaussian is the most commonly used prior, other distribution can also be used.

Variational autoencoders are commonly used in representation learning and as generative models. It is each to generate new data that resembles the original data. The first step is to get a latent sample from the prior $z \sim \mathcal{N}(0,\mathbf{I})$. Then, using this latent variable, the decoder can generate a sample that could belong in the original dataset. Variational autoencoder has been successfully used in a number of different applications, such as computer vision [Walker et al., 2016] and natural language processing [Bowman et al., 2015].

The KL divergence between two Gaussian distributions $\mathcal{N}(\mu_1,\Sigma1)$ and $\mathcal{N}(\mu_2,\Sigma2)$ can be computed as:

$$KL = \frac{1}{2}[\log\frac{D_2}{D_1} - d + tr\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)] \tag{2.54}$$

where D_1, D_2 are the determinants of matrix Σ_1 and Σ_2 respectively, and d is the dimension of the samples of the distribution. In the case of a variational autoencoder where the prior is a Gaussian with zero mean and unit variance, we can compute the second term of the ELBO as:

$$KL(p(z|x)||q(z)) = \frac{1}{2}[-\log D_1 - d + tr\{\Sigma_1\} + \mu_1^T\mu_1] \tag{2.55}$$

After feeding an input, the encoder outputs a value for the mean and the variance of the representation of the specific input. The second part of the ELBO tries to make this representation to come from a Gaussian, like the prior. The decoder requires as input a sample from the outputted Gaussian distribution $z \sim \mathcal{N}(\mu_1(x),\Sigma_1(x))$ of the encoder in order to reconstruct the input. The problem, in this case, is that the gradient of the reconstruction loss cannot flow through the encoder, due to the fact that the sample generation is not differentiable. A common solution to this issue is the use of the reparameterization trick to create differentiable samples by observing that:

$$z \sim \mathcal{N}(\mu_1(x),\Sigma_1(x)) \iff y \sim \mathcal{N}(0,\mathbf{I}), z = Ay + \mu_1(x) \tag{2.56}$$

where $\Sigma_1 = AA^T$. Using the reparameterization trick, we can create differentiable samples of any Gaussian distribution, with respect to the parameters of the distribution. Therefore, the gradient of the reconstruction loss flows through the samples to the parameters of the encoder, which are trained using gradient based optimization.

2.4.2 β-VAE

β-VAE Higgins et al. [2017] is a modification of the standard VAE loss that controls the importance of the KL divergence in the ELBO by using a parameter β. Higgins et al. [2017] reason that by maximizing the following equation for $\beta > 0$ achieves similar results as in VAE:

$$\max_{v,\varphi} \mathbf{E}_{z \sim p(z|x)}[\log q_v(x|z)] - \beta KL(p_\varphi(z|x) || \mathcal{N}(z; 0, \mathbf{I})) \tag{2.57}$$

It is evident that if $\beta < 1$, the maximizer places more emphasis on maximizing the reconstruction term, and as a result, better quality samples are generated. Additionally, the smaller the value of β, the less the posterior variational distribution will look like the prior Gaussian distribution. If $\beta > 1$, then the variational distribution is forced to match the prior, and this results in independence between the different latent variables. This leads to disentanglement in the latent variables, which is an essential property for many machine learning applications as well as provides better human interpretation. However, in this thesis, we are interested in the reconstruction loss, and we will evaluate values $\beta < 1$.

Chapter 3

Related Work

After introducing the required theoretical background, in this chapter, we will describe important works. Our work can be considered as an intersection of opponent modelling, and representation learning and single-agent reinforcement learning in multi-agent systems.

3.1 Learning Opponent Models

Opponent modelling is a common method for learning low-dimensional information about opponents. There is a large number of works [Albrecht and Stone, 2018] focused on this topic. However, in this work, we will focus only on recent works that are concerned with deep reinforcement learning.

He et al. [2016] proposed two learning architectures for modelling opponents. The core idea behind both architectures is to use features of the observations of the opponents in order to compute the Q-values of the agent. The first architecture, that the authors proposed, is called DRON-concat. In this architecture, they use two networks. The first network has access to the observation of the agent, and it is responsible for approximating the state-action values. The second network has access to the observation of the opponent. In the agent's network, they also feed the activation of the hidden layer of the opponent's network. In this way, the action-value network conditions its values on the opponents modelling, which renders a stable training procedure. Both networks use the temporal difference target of the current agent as supervision in order to shape their parameters.

He et al., additionally, proposed a more sophisticated architecture by observing that the Q-value of an agent can be expressed as $Q^{\pi|\pi_o}(s,a) = \sum_{a_{-i}} \pi(a_{-i}|s) Q^{\pi}(s,a,a_{-i})$.

Therefore, they suggested the use of a Mixture-of-Experts network (DRON-MOE). They trained many experts to predict different Q-values, and they combine them using a weighted average. The weights of each Q-value are generated using a gating network with Softmax output. Both networks are trained using the TD target like DRON-concat.

Raileanu et al. [2018] proposed Self Other-Modelling (SOM), an agent that predicts the goals of the opponent using its policy. The agent is trained using an actor-critic architecture:

$$\begin{pmatrix} \pi \\ V \end{pmatrix} = f_{self}(s_{self}, z_{self}, z_{other}) \tag{3.1}$$

The agent is using the same network for computing the goal of the other agent z_{other} by changing the argument position of z_{other} and z_{self}. The goal of the other agent z_{other} is a vector of trainable parameters which is trained using gradient-based optimization. The authors denote the network that is used to infer the goal of the opponent agent as f_{other}, even though it has the same parameters as f_{self}. The parameters of the network are trained using the A3C algorithm [Mnih et al., 2016]. The z_{other} is trained by minimizing the cross-entropy between the real action of the agent and the policy that is produced by the f_{other} network. The SOM was applied to a variety of environments and outperformed the existing baselines, such as a modified version of the work of He et al..

Another approach in order to learn representations of an opponent's policy is the work of Grover et al. [2018]. The authors trained a plethora of agents in the Robosumo [Al-Shedivat et al., 2018], and Cooperative Communication [Mordatch and Abbeel, 2017] environment and created a dataset of episodes of different agents. Using the episodes of the dataset, they trained an encoder-decoder architecture. The encoder is responsible for learning embedding representations of the opponents, and those representations should both be informative about the policy of each agent and discriminative between the embedding of different agents. The decoder is trained to reconstruct the policy of each agent, and it is trained using imitation learning. Overall, in order to ensure discrimination between the embedding of different agents, the authors minimized the following objective:

$$L(e_+, e_-, e_*) = \frac{1}{(1 + e^{|f(e_)* - f(e_-)|_2 - |f(e_*) - f(e_+)|_2})^2} \tag{3.2}$$

where e_*, e_+ are episodes of one agent and e_- is the episode of another agent. The imitation learning objective is the standard cross-entropy function

$$L = \mathbb{E}_{o,a \sim e_+}[-\log \pi(a|s, e_*)] \tag{3.3}$$

The authors used the pretrained encoder in different machine learning problems. They used the learned representations in supervised tasks in order to predict the Robosumo episode outcome (win, draw, or loss). Additionally, they used the embeddings for clustering different agents' policies in the embedding space. Finally, the most important use of the encoder was in reinforcement learning. They used the encoder to produce embedding of the opponent's policy and conditioned a policy network on them. The policy network was able to learn faster and more stable policies against several different opponents.

Similarly, Rabinowitz et al. [2018] proposed the Theory of Mind Net (TomNet) to learn agent representations in multi-agent systems. Given a pretrained set of opponents' trajectories, TomNet learns two embedding representations for each agent an encoder-decoder architecture. The decoder is responsible for reconstructing the policy of the current agent using imitation learning as well as predicting other properties of the given MDP. Rabinowitz et al. [2018] provided experimental evidence that TomNet can accomplish a variety of tasks, such as modelling deep reinforcement learning agents and inferring goals of the opponents.

3.2 Representation Learning

Learning opponent models using deep learning has only recently proposed in the context of multi-agent deep reinforcement learning. However, there are multiple recent works on learning representations of various tasks and use them to learn more efficient policies.

One of the early works in this direction is the Unsupervised Reinforcement and Auxiliary Learning (UNREAL) [Jaderberg et al., 2016]. Jaderberg et al. [2016] proposed an architecture based on A3C [Mnih et al., 2016], which alongside the reinforcement learning, it also learns some unsupervised tasks. A plethora of different unsupervised tasks were proposed, such as the prediction of the reward sign based on the previous frames and methods for maximizing the absolute difference between consecutive frames. Jaderberg et al. [2016] reasoned that combining all the auxiliary losses would result in better shaping of the features that are extracted from the deep network. UNREAL outperformed A3C in a variety of benchmarks both in terms of performance and in terms of sample efficiency.

Hausman et al. [2018] introduced a lower bound in an entropy regularized reinforcement learning objective. The goal of this work is to learn a low-dimensional representation of different tasks. In this way, a policy network conditioned on this

representation can learn a different policy based on different latent samples. Additionally, Hausman et al. [2018] reasoned that it is possible to interpolate the latent representations in order to solve more complex tasks. The authors propose the training of an embedding network that takes as input a one-hot vector representing the identity of the task. Because the task identity will not be available during testing, the authors train an inference network based only on the current observation and action to infer the identity of the task. The inference network is trained by minimizing the cross-entropy between the embedding and the inference network. This work can be considered that it is closer to our method as we will describe in the following Chapter 4.

Other recent works rely on the variational autoencoder framework [Kingma and Welling, 2013] for learning unsupervised representations of MDPs and condition the policy on the latent representations. Ha and Schmidhuber [2018] proposed World Models a VAE based method that learns to encode the visual observations in a low-dimensional latent space. A policy network was then trained on the top of the latent representation for learning the policy of the agent. An important observation is that the policy network consists of almost only one thousand parameters, which are significantly fewer compared to most reinforcement learning algorithms that deal with visual inputs that require five to ten million parameters. Two recent approaches that also are based on ELBO was proposed by Rakelly et al. [2019] and Zintgraf et al. [2019], which proposed a VAE-based model to learn latent representations about different MDPs.

Another work in representation learning and reinforcement learning is the Model Agnostic Exploration with Structured Noise (MAESN) Gupta et al. [2018]. MAESN is used for learning to act against many different tasks as well as generalize against previously unseen tasks. Gupta et al. [2018] propose a model that consists of latent representations of different tasks, each one represented by a different Gaussian. The mean and the variance of each Gaussian consist of parameters that will be learned during training. The policy network is trained conditioned on samples from the Gaussian and the observation of each timestep. The architecture is trained using Model Agnostic Meta-Learning (MAML) [Finn et al., 2017] in order to be able to adapt to new tasks quickly.

Chapter 4

Methodology

4.1 Problem Formulation

We consider a modified Markov Game, which consists of N agents $I = \{1,2,...,N\}$, the set of states, the set of actions $A = A_1 \times A_{-1}$, where A_{-1} is the set of action of all N agents except agent 1 as well as the transition $T : S \times A \times S \rightarrow \mathbf{R}$ and the reward $r : S \times A \times S \rightarrow \mathbf{R}^N$ function. We consider partially observable settings, where each agent i has access only to its local observation o_i and reward r_i. Additionally, a set of pretrained opponents is provided $PA = \{\{2,3,...,N\}_1, \{2,3,...,N\}_2, ..., \{2,3,...,N\}_M\}$. Note that with the word opponent we refer to other, one or more, agents in the environment both in cooperative and adversarial settings. At the beginning of each episode, we sample a pretrained opponent from the set PA. Our goal is to train the agent 1 using reinforcement learning, to efficiently act against all possible opponents from the set PA.

$$\max \mathbf{E}_{PA}[\sum_t \gamma^t r_{1,t}] \tag{4.1}$$

4.2 Variational Opponent Modelling

The first step to solve the previously described problem is to learn a model of the opponents in a latent space. By learning a model in an unsupervised fashion, we can separate different agents in the latent space based on their behavior. Therefore, the policy network will be conditioned on the latent variable and will develop a different policy for each different region of the latent space. We incrementally extend the idea of Grover et al. [2018] in order to reason about uncertainty about the opponents' model. Grover et al. [2018] proposed a point-based representation for opponent modelling. We

consider this approach inefficient for training because the policy network is only trained on some points of the subspace instead of the whole subspace. Therefore, we propose a variational autoencoder based architecture for learning opponent models. A probabilistic representation that uses sampling for creating embedding vectors will result in a more robust policy network, trained on a large number of different representations.

Consider that an opponent is represented by its trajectory of observations and actions $\tau = (o_1, a_1, o_2, a_2, ..., o_H, a_H)$, where H is the horizon of the episode, and if there is more than one opponent agent in the environment o_i and a_i represent the concatenated observations and actions respectively. Additionally, consider a dataset of trajectories of all the pretrained possible opponents $E = \{E_1, E_2, ...E_M\}$, where $E_i = \{\tau_1, \tau_2, ..., \tau_K\}$. Our goal is to learn a representation latent variable z based on the trajectory of the opponent τ, which is the probability $q(z|\tau)$. At this point, note that this posterior is unknown and probably intractable. For this reason, we will use the variational inference framework that was described in Section 2.4. Therefore, we minimize the KL divergence between the unknown posterior and the variational distribution that we assume is a Gaussian. Zintgraf et al. [2019] also proposed a similar approach for modelling MDPs for meta-learning in single-agent settings. Our approach is incremental to the work of Zintgraf et al. [2019] and to the best of our knowledge, the first work that utilizes variational autoencoders to learn representations about opponents. Similarly, to Section 2.4, we can learn a latent space z of the opponent models by maximizing the β-VAE [Higgins et al., 2017] objective.

$$\max_{v,\varphi} \log p_v(\tau_{:H}|z) - \beta KL(p_\varphi(z|\tau_{:H})||\mathcal{N}(z; 0, \mathbf{I}))] \tag{4.2}$$

This architecture consists of a recurrent encoder with parameters φ, which receives sequentially as input the observation action pair of the opponent and projects it in the latent space z. Note that at each time the latent variable z depends on all the previous observation and action pairs of the specific trajectory due to the recurrent nature of the encoder. Regarding the reconstruction loss, and given that the current observation and action of the opponent depends on the previous pair of observation and action, the reconstruction loss can be expressed as:

$$\log p_v(\tau_{:H}|z) = \log p_v(a_1|z_1, o_1) p_v(o_1|z_1) + \sum_{i=2}^{H} \log p_v(a_i|o_i, z_i) p_v(o_i|z_i, o_{i-1}, a_{i-1}) \tag{4.3}$$

Therefore, the decoder consists of two fully-connected networks. The first network receives as input the sampled latent variable and the previous observation action pair

and outputs the reconstructed observation of the given time-step. The second network receives as input the current observation and a sample from the latent space and outputs the current action. Our models learn the transition model of the environments as well as representations about the opponent's policy.

Furthermore, we want to increase the power of this model in order to create better representations. One particular property that we are interested in is that the embeddings of each agent should cluster together and be separated from the embeddings of the other agents. Therefore, we use the inverse squared Softmax objective that was proposed by Grover et al. [2018]. Consider, two agents where two trajectories from the first agent and one trajectory of the second agent are available, and let z_*, z_+ and z_- represent their embeddings respectively. It is expected that the embeddings of the first agent should be close in the embedding space, while they are far from the embedding of the second agent. This can be forced by minimizing the following objective:

$$d(z_+, z_-, z_*) = \frac{1}{(1 + e^{|z_* - z_-|_2 - |z_* - z_+|_2})^2} \tag{4.4}$$

From, this equation it is clear that $d(z_+, z_-, z_*) \geq 0$. Therefore, we can derive the following less tight lower bound in the evidence:

$$\begin{aligned} &\frac{1}{H}\sum_{i=1}^{H}[\log p_v(\tau_i|z_{*,i}) - \beta KL(p_\varphi(z_{*,i}|\tau_{:i})||\mathcal{N}(z_{*,i};0,\mathbf{I}))] \geq \\ &\frac{1}{H}\sum_{i=1}^{H}[\log p_v(\tau_i|z_{*,i}) - \beta KL(p_\varphi(z_{*,i}|\tau_{:i})||\mathcal{N}(z_{*,i};0,\mathbf{I})) - d(z_{+,i}, z_{-,i}, z_{*,i})] \end{aligned} \tag{4.5}$$

Below, the pseudocode (1) of the proposed opponent modelling method based on variational autoencoders is provided.

4.3 Policy Learning

After learning a low-dimensional representation of the opponents, the policy network can be conditioned on this. Consider the augmented state space $O' = O \times Z$, where O is the original observation space of the agent 1 in the Markov game, and Z is the representation space of the opponent models. The advantage of learning the policy on O' compared to O is that the policy can be different based on different $z \in Z$. Therefore, our agent develops a different behavior based on its opponent, which results in higher returns, assuming that the opponent models are successfully represented in the embedding space.

Algorithm 1 Pseudocode of the learning opponent models algorithm

```
for i = 1 : M do
    Sample a positive episode z+ ← sample(E_i)
    Sample a reference episode z* ← sample(E_i)
    Compute reconstruction loss from 4.3
    for j = 1 : M do
        if i == j then
            continue
        Sample a negative episode z− ← sample(E_j)
        Compute the discrimination loss disc_loss = d_θ(z*, z+, z−)
        Compute the KL divergence
        Minimize the total loss L = recon_loss + β · KL + λ · disc_loss
```

We use the twin delayed deep deterministic policy gradient algorithm (TD3) [Fujimoto et al., 2018] for policy learning. TD3 is a sample efficient off-policy learning algorithm that resulted in state-of-the-art performance in a plethora of continuous benchmarks. Our experimental environments have discrete action space, and as a result, we use the Gumbel-Softmax trick to derive differentiable samples from the policy distribution. TD3 is an actor-critic algorithm and a variant of the DDPG [Lillicrap et al., 2015] algorithm. It consists of a critic, which has two networks for learning the Q-values. Given the two critic networks Q_{ϕ_1} and Q_{ϕ_2}, their parameters ϕ_1 and ϕ_2 are trained by minimizing the following loss:

$$target = \min(Q_{target,\phi_1}(o',a',z'), Q_{target,\phi_2}(o',a',z')) \tag{4.6}$$

$$\min_{\phi_1,\phi_2} \frac{1}{2}[(r+\gamma \cdot target - Q_{\phi_1}(o,a,z))^2 + (r+\gamma \cdot target - Q_{\phi_1}(o,a,z))^2] \tag{4.7}$$

Note that there are two target network Q_{target,ϕ_1} and Q_{target,ϕ_2} which are updated from Q_{ϕ_1} and Q_{ϕ_2} respectively using the Polyak average. Another modification of the TD3 algorithm compared to DDPG is that the action of the next state is computed using the exploration policy $a' = \mu(s') + \mathcal{N}$, where $\mathcal{N}$ is the exploration noise. Finally, Fujimoto et al. [2018] proposed to update the actor less frequent than the critic, but we do not use this in any of the experiments in this thesis.

Additionally, the policy network μ_θ, with parameters θ is trained by maximizing the

Q-value of the first network of the critic.

$$\max_{\theta} Q_{\phi_1}(o, \mu_\theta(o,z), z) \tag{4.8}$$

4.4 Inference Model

The limitation of the existing opponent modelling methods as well as the VAE-based method that we proposed, as previously discussed in Section 3.1, is that in order to evaluate the embedding space of the encoder, access to the observations and actions of the opponents is required. While this assumption is not problematic during centralized training, it poses a problem for decentralized execution where agents do not share observations. As a result, an alternative way to access the embedding space during testing is required. For this purpose, we will train a different network, called inference network. The inference network will be trained to infer the opponent model, using only the locally available information of our agent, such as its observations, actions, and rewards. The idea behind the recurrent network that receives as input the observation, action, reward triplet has been extensively studied in the single-agent meta reinforcement learning literature [Wang et al., 2016, Duan et al., 2016]. For training the inference network, KL divergence between the inference and the embedding network is minimized. Given the embedding distribution p that takes as input the sequence of the opponents observations and actions (o_{opp}, a_{opp}) and the inference distribution q_ψ, with parameters ψ, that takes as input the observation action and reward sequence (o,a,r) of our agent, we minimize the following loss with respect to the parameters ψ:

$$\min_{\psi} KL(q_\psi(o,a,r) || p_\varphi(o_{opp}, a_{opp}))$$

We use this mode-seeking version of the KL divergence to force the inference network to learn to identify different modes of the opponent distribution, instead of learning an inference distribution that all opponent models are likely as the minimization of the mean-seeking KL divergence would result in. The work of Hausman et al. [2018] is the most similar to our method, where an inference network was used to infer the the identity of different tasks.

4.5 Training Method

To summarize, we propose a learning architecture for efficiently identifying opponents at the test time, without accessing their observations and actions. For the choice of the

policy network, we can use different learning algorithms. Since we use deep networks for approximation, we will use an experience replay to break the correlation between consecutive samples. It is important to note that, since LSTMs are used, we require to have sequences of samples in order to train. For this reason, we use a modified version of the experience replay that enables storing and sampling of episode sequences. Similar experience replay was used by other works that deal with batches that consist of sequences, such as Hausknecht and Stone [2015]. This replay is essential in order to break the correlation between consecutive sequences and stabilize the training procedure. Finally, during exploration, we use samples from the inference network and not the embedding network, because this experimentally led to higher returns.

After defining our method, in this paragraph, we will describe our learning architecture. We need an algorithm that will allow learning the four networks concurrently. We assume that we have a pretrained encoder available, such as the one that we discussed previously. Figure 4.1 presents the diagram of the learning architecture, and the pseudocode of the proposed method is provided below (2). Additionally, all networks are trained based on batches that are sampled from the experience replay. A batch B consists of episodes that were generated following the exploration policy $\{(o, o_{opp}, a, a_{opp}, r, r_{opp}, o', o'_{opp})_i\} \in B$. When we perform experiments to evaluate our opponent modelling method or the method of Grover et al. [2018], we do not use the inference network, and as a result, both actor and critic are conditioned on the representations generated by the embedding network. The four networks that we have to train and the pseudocode of our method are:

- The **state-action value networks** $Q_{\phi_1}(o,a,z)$ and $Q_{\phi_2}(o,a,z)$ with parameters ϕ_1 and ϕ_2 respectively, that take as input the observation, actions pair of our agent and a sample from the inference network and output an approximation of the true state-action value. We train these networks by using Q-learning [Watkins and Dayan, 1992]:

$$\min_{\phi_1,\phi_2} \frac{1}{2}\mathbf{E}_B[(r+\gamma \cdot target - Q_{\phi_1}(o,a,z))^2 + (r+\gamma \cdot target - Q_{\phi_2}(o,a,z))^2] \quad (4.9)$$

  We note that similar to every deep value-based learning algorithm, target networks are required to compute the target values. Thus, there is one target network for computing the next action, one target network for computing the target inference embedding and two target networks for computing the next Q-value.

- The **policy network** $\mu_\theta(o,z)$, with parameters θ, that takes as input the observation of our agent and a sample from the embedding network and outputs an action.

This network is trained by minimizing the standard deterministic policy gradient objective Silver et al. [2014]:

$$L(\theta) = -\mathbf{E}_B[Q_{\phi_1}(o, \mu_\theta(o,z), z)] \tag{4.10}$$

We use a sample from the embedding network, which is created using the opponents' fixed policies in order to perform the actor updates. Similar approach has been used by Hausman et al. [2018] as well, where different embedding networks were used during training and execution. This is used because the current policy is significantly different from the policy that the trajectories were generated from in the experience replay.

- The **inference network** $q(z|o,a,r)$, which is a recurrent network, with parameters ψ, that takes as input the observation, action and reward sequence of the agent and outputs the mean and the logarithmic variance (in order to be able to perform unconstrained optimization) of the generated Gaussian distribution. This network is trained by minimizing the KL divergence for the length of the episode, between the embedding and the inference distribution:

$$L(\psi) = \mathbf{E}_B[KL(q_\psi(o,a,r)||p(o_{opp}, a_{opp}))] \tag{4.11}$$

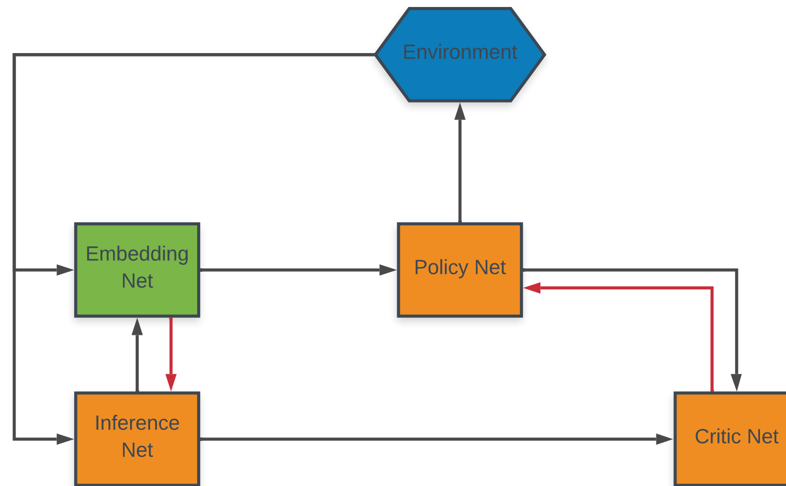


Figure 4.1: The learning architecture of our method. The red arrows show the gradient flow between the networks.

Algorithm 2 Pseudocode of our method

for $e = 1 : M$ episodes **do**
  $opp \leftarrow sample(opponents)$
  for $t = 1 : num_steps$ **do**
    Get the observation of our agent o and the observation of the opponent o_{opp}
    Compute the action of the opponent a_{opp}
    Get a sample of the inference network $z \leftarrow q(z|o, a_{t-1}, r_{t-1})$
    Compute the action a of the agent using exploration
    Perform the actions in the environment and get new observations and rewards
  Store the sequences of both agents in the experience replay
  Sample a batch of sequences from the experience replay
  Update the critic networks by minimizing the loss from 4.9
  Update the actor by minimizing the loss from 4.10
  Update the inference network by minimizing the loss from 4.11
  Update the target value network $\varphi' \leftarrow \tau\varphi + (1 - \tau)\varphi'$
  Update the target policy network $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$
  Update the target inference network $\psi' \leftarrow \tau\psi + (1 - \tau)\psi'$

Chapter 5

Experiments

5.1 Experimental Framework

After describing the proposed methodology, in this chapter, a thorough evaluation will be provided. For this purpose, the Multi-agent Particle Environment (MPE) [Mordatch and Abbeel, 2017] will be used. This framework provides several different multi-agent environments. This is the best possible open-source choice available because it consists of low dimensional environments that do not require much computational power to train. Despite the low dimensionality, these tasks are hard enough and exhibit all the issues that multi-agent systems suffer from, such as non-stationarity, credit assignment, and lack of coordination. The environments have continue observation and discrete action space. Below, we provide an analytic description of the different environments that we will use to evaluate our method.

5.1.1 Speaker-Listener Environment

The speaker-listener environment consists of two agents, called speaker and listener as well as three designated landmarks that each one has a different color, red green or blue. At the beginning of the episode, the listener is assigned a color, which can be red, green, or blue. The task of the listener is to navigate to the landmark that has the same color. However, the color of the listener can only be observed by the speaker. So the speaker has to learn to communicate the correct color to the listener. The listener should be able to understand the generated message of the speaker and to navigate to the correct color. The observation space of the listener has 13 dimensions, which consists of the position of the listener, the positions of the three landmarks in the 2D environment

and the 5-dimensional communication message of the speaker and its action space has five dimensions. The speaker has a 3-dimensional observation space, which is a vector assigned to the color of the listener, and 5-dimensional action space. We use our method to train the listener to be able to understand a set of different speakers, pretrained speakers that use different communication messages. Both speaker and listener share the same reward, which is the negative Euclidean distance between the listener and the correct landmark. In Figure 5.4a, an instance from the speaker-listener environment is presented.

5.1.2 Double Speaker-Listener Environment

The double speaker-listener environment consists of two agents and three designated landmarks that each one has a different color, red green or blue, similarly to the speaker-listener environment. The only difference is that both agents are simultaneously both speakers and listeners. Therefore, at the beginning of the episode, each agent has a color that can only be observed by the other agent. Each agent must learn both to communicate a message to the other agent as well as navigate to the correct landmark. The observation space of each agent has 16 dimensions, where 13 of them are the same as the listener's in the previous environment and the other three is the vector assigned to the color of the opponent, while the action space is 5-dimensional for the navigation actions and 5-dimensional for the communication message as well. The reward is the average of the negative Euclidean distances between each agent and the correct landmark. This environment is significantly more difficult compared to the speaker-listener because our agent has to infer both the color that the other agent observes as well as communicate the correct message that the opponent expects. In Figure 5.4b, an instance from the double speaker-listener environment is presented.

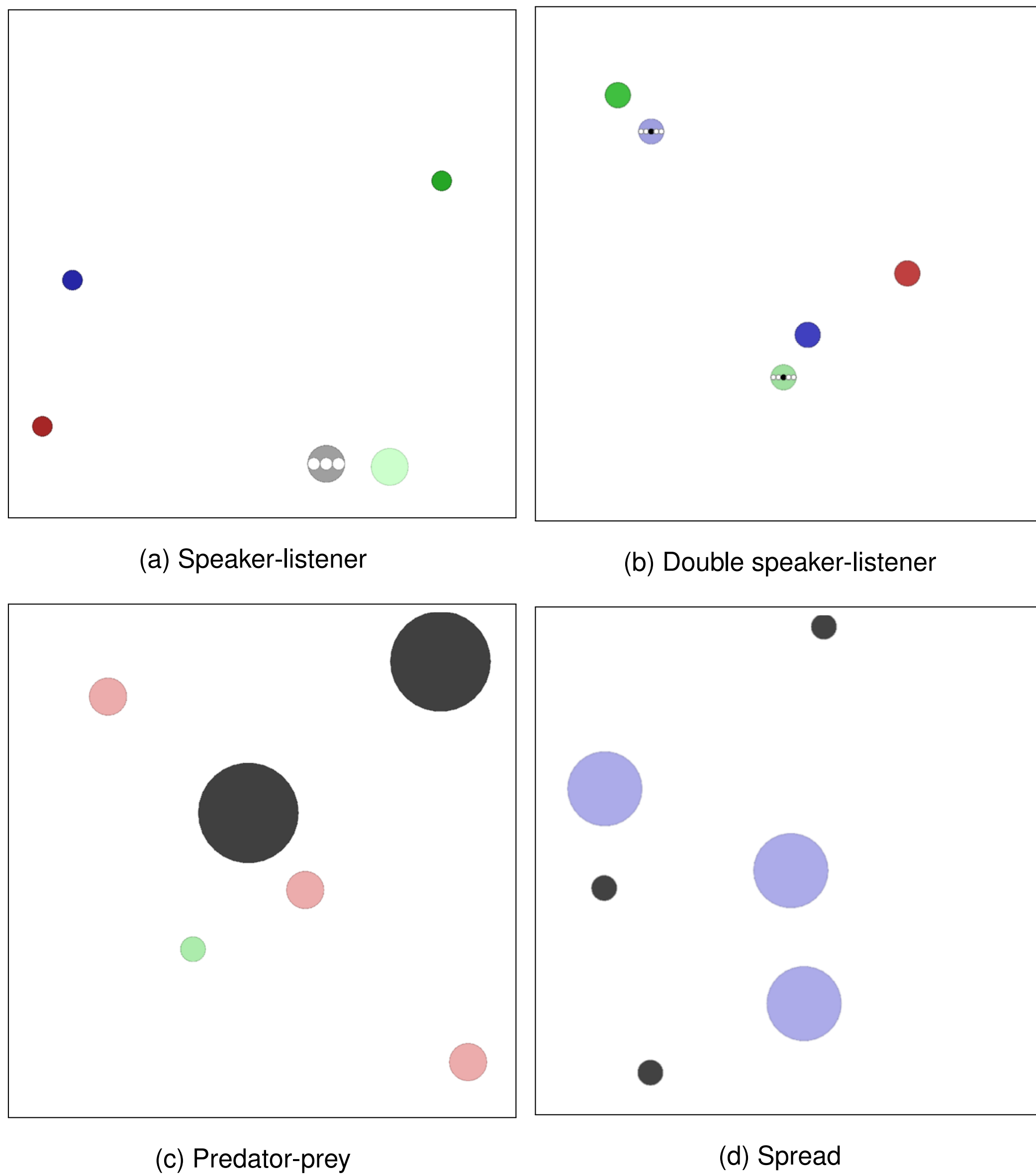
5.1.3 Predator-Prey environment

This environment consists of one prey agent and three predator agents. At the beginning of the episode, the prey and the predators are randomly placed in a 2D map. The goal of the prey is to avoid being caught by the predators, while the predators' goal is to touch the prey. In the environment, there are additionally two large black obstacles in order to block the agents. The advantage of the prey compared to the three predators is that it can move faster compared to the three adversaries. This environment, compared to the previous two, is competitive. We deliberately chose this environment in order to prove

that our method is agnostic to the environment setting. In our work, we will apply the proposed algorithm in the prey agent in order to examine whether it can avoid a large number of different pretrained predator agents. The observation of each agent has 14 dimensions representing the position of all the other agents as well as the obstacles in the 2D space, while their action space consists of 5 actions. In Figure 5.4c, an instance from the predator-prey environment is presented.

5.1.4 Spread Environment

The spread environment consists of three large agents and three landmarks as well. At the beginning of the episode, the three agents and the three landmarks are spread randomly in the 2D space. The goal of the agents is to navigate to three different landmarks without colliding. The reward is the negative distance of each agent from the landmark. In the case of collision, there is an additional negative reward. The reward is the same for all agents. All the agents have the same observation space, which consists of 18 dimensions, while their action space consists of 5 actions. In Figure 5.4d, an instance from the spread environment, is presented.



(a) Speaker-listener

(b) Double speaker-listener

(c) Predator-prey

(d) Spread

Figure 5.1: Instances of different MPE environments

5.2 Implementation Details

Before providing the experimental evaluation of our proposed methodology, in this section, we will describe the algorithmic and optimization details of our method.

5.2.1 Algorithm Details

It is well known that deep reinforcement learning is very unstable, and several stabilization tricks are required. Below we refer to different tricks and normalizations that were

used:

- Reward normalization. We normalize the rewards that are used in the critic update (equation 4.7) to have zero mean and standard deviation one. This is important in order to prevent the critic loss from getting large values that lead to instability in gradient updates. Given the rewards r that belong in the sampled batch from the experience replay, the normalized rewards are computed as:

$$norm_r = \frac{r - mean(r)}{std(r) + 10^{-5}} \tag{5.1}$$

- Regularization of the actor loss. We add the squared logits of the actor in the loss. This leads the action logits to small values. This is essential because large values of the logits would make the addition of Gumbel noise insignificant, and the agent would stop to explore. Given the logits l that are generated by the actor, the loss can be expressed as:

$$L(\theta) = \mathbf{E}_B[-Q_{\phi_1}(o, \mu_\theta(o,z), z) + reg \cdot l(o,z)^2] \tag{5.2}$$

- Gradient clipping. Another standard technique, especially when recurrent networks are used, is the clipping of the norm of the gradient. In this way, large update steps are prevented, and the training stability improves.

For the implementation of the described algorithms, the Pytorch [Paszke et al., 2017] framework was used. An initial version of MADDPG and the modified MPE that enables better visualization was used from this Github repository `https://github.com/shariqiqbal2810/maddpg-pytorch`. Parts of this repository were also used in all the implementations. The code for the sequential experience replay was co-written with Arrasy Rahman. Codes for scheduling experiments were co-written with Arrasy Rahman and Filippos Christianos.

5.2.2 Hyperparameter Optimization

Hyperparameters in reinforcement learning are critical in order to achieve sufficient performance. Henderson et al. [2018] provided excellent experimental evidence that small variations in the reinforcement learning hyperparameters and the deep learning architecture can significantly affect the algorithm's performance. In order to successfully evaluate of our model, a large number of values of each hyperparameter was tested. In Table 5.1, the different hyperparameters that were used are presented. By computing

the Cartesian product, we end up with 13824 different configurations. Note that we reduced this number by removing experiments that the hyperparameter combination would possibly result in low undiscounted returns, such as the batch size is smaller than the update frequency. For computing the optimal hyperparameters, the speaker-listener, the simplest possible environment, was used. All the experiments were performed in CPUs using the Eddie [1] cluster, which enables the submission of multiple parallel jobs, which resulted in significant improvement of the computation time.

Table 5.1: Possible hyperparameter values

Hyperparameter	**Value**	**Hyperparameter**	**Value**
# layers actor	$3,4$	inference learning rate	$10^{-4}, 10^{-3}$
# layers critic	$3,4$	clipping actor	0.5
# layers inference	$3,4$	clipping critic	0.5
# layers embedding	2	clipping inference	$0.2, 0.5$
Layer size embedding	100	batch size	$50, 100, 200$
Activation function	ReLU	update frequency	$32, 50, 100$
Layer size others	$64, 128, 200,$	τ	$10^{-3}, 10^{-2}$
actor learning rate	$10^{-4}, 10^{-3}$	actor regularizer	$10^{-3}, 10^{-2}$
critic learning rate	$10^{-4}, 10^{-3}, 10^{-2}$	γ	0.99

After running all experiments that were discussed above, below in Table 5.2, the hyperparameters that resulted in optimal performance are presented. Note that the hyperparameters were optimized only for the speaker-listener environment and it is likely that some modifications will result in small variations in performance in the other three environments.

[1] https://www.ed.ac.uk/information-services/research-support/research-computing/ecdf/high-performance-computing

Table 5.2: Final hyperparameter values

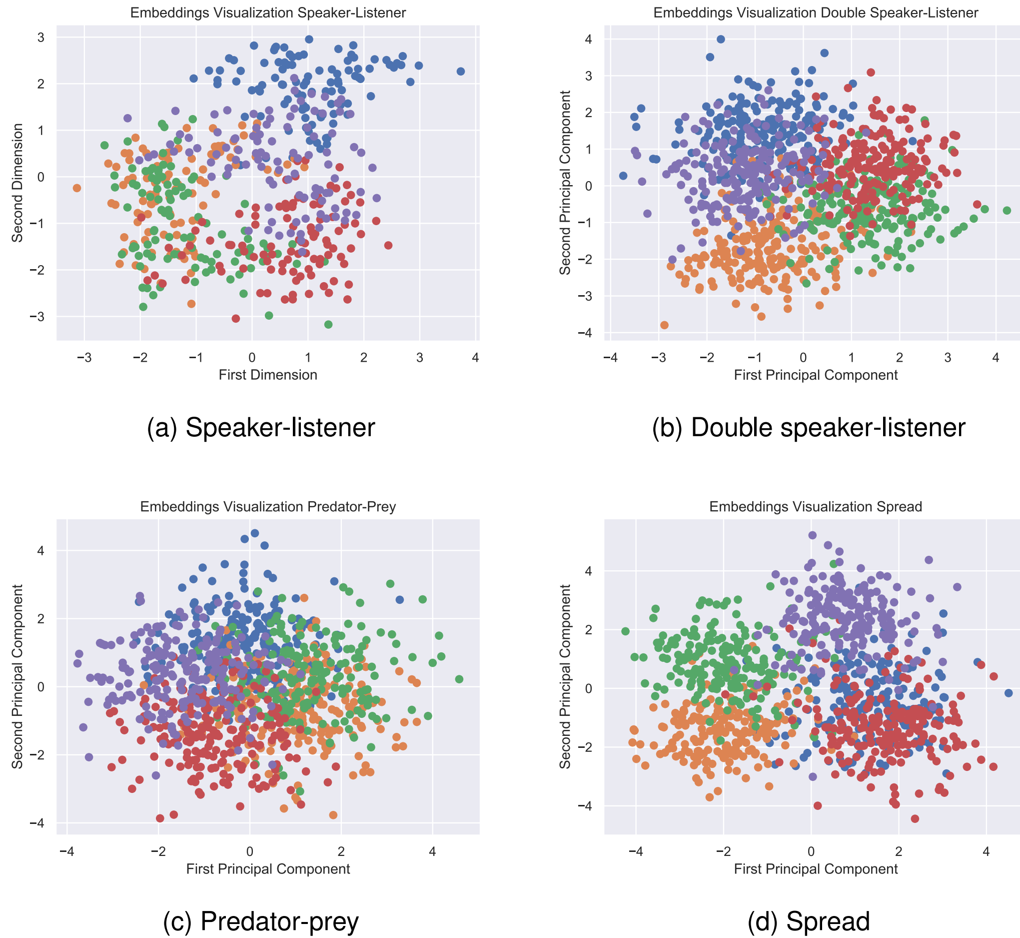
Hyperparameter	Value	Hyperparameter	Value
# layers actor	4	clipping actor	0.5
# layers critic	4	clipping critic	0.5
# layers inference	4	clipping inference	0.2
# layers embedding	2	batch size	100
Layer size embedding	100	update frequency	50
Activation function	ReLU	τ	10^{-2}
Layer size others	200	actor regularizer	10^{-3}
actor learning rate	10^{-4}	γ	0.99
critic learning rate	10^{-3}	β	0.01
inference learning rate	10^{-4}	λ	1

5.3 Evaluation

After describing our experimental framework, in this section, we will provide evaluation and comparison, both for the opponent modelling method as well as the reinforcement learning architecture. We evaluate our method against five opponents in each environment. For the speaker-listener environment, we create different speakers that generate different communication messages for the same input. In the double speaker-listener environment, we create the communication messages of the agents similarly to the speaker-listener environment, and we train the navigation policy of the agents using the MADDPG algorithm Lowe et al. [2017], that was described in Section 2.3.3. For the other two environments, we train the agents using MADDPG and different initial seeds, which leads to different policies.

5.3.1 Embedding visualization

First, we visualize the samples that are generated by the embedding network. In Figure 5.2, a visualization of the embedding space of the five different opponents projected in the first two principal components is provided. Figure 5.2 presents the sample of the last timestep in the episode, and different opponents are represented in different colors.



(a) Speaker-listener

(b) Double speaker-listener

(c) Predator-prey

(d) Spread

Figure 5.2: Visualization of the embedding space

Note that, in all the environments except the speaker-listener, we project the embeddings from the initial ten dimensions to the first two principal components for the sake of visualization. Despite this dimension reduction, it is visible that the embeddings of each agent are gathered together, and they do not spread arbitrarily in the embedding space. The overlapping embeddings could possibly mean that these agents have a similar policy in some cases. To formally evaluate the clustering of the embeddings of different agents, we will use the intra-inter clustering ratio that was used by Grover et al. [2018]. The intra cluster distance is the average Euclidean norm between all the embeddings of an agent, while the inter distance is the average Euclidean norm between all the embeddings of different agents. Given N agents, where there are N_i embeddings available for each agent and $z_{i,k}$ is the kth embedding of agent i, the intra-inter cluster

ratio is defined as:

$$IICR = \frac{\frac{1}{N}\sum_{i=i}^{N}\frac{1}{N_i^2}\sum_k\sum_l ||z_{i,k} - z_{i,l}||_2}{\frac{1}{N(N-1)}\sum_{i=i}^{N}\sum_{j\neq i}^{N}\frac{1}{N_iN_j}\sum_j\sum_k ||z_{i,k} - z_{j,l}||_2} \tag{5.3}$$

According to Grover et al. [2018], IICR value lower than 1 indicates a clear opponent separation in the embedding space. Below, in Table 5.3, the IICR values for the four evaluation environments are provided. Note that, the IICR metric is not evaluated on trajectories that were used for training the opponent model. We generate opponent trajectories against an agent that we train using the methods that were described in Chapter 4.

Table 5.3: Intra-Inter clustering ration

Environment	**IICR**
Speaker-listener	10.492
Double speaker-listener	0.835
Predator-prey	0.889
Spread	0.953

From Table 5.3, we observe that the IICR value is lower than 1 in all the environments, except the speaker-listener. As a result, in these three environments, the different agents are separable in the embedding space. In the speaker-listener environment, the IICR value is considerably greater than 1, indicating a poor separation of the embeddings of different agents. The reason behind this is that many agents generate the same communication message when they observe the same color. As a result, the embeddings of the different agents are generated under the same distribution, leading to a large IICR value.

5.3.2 Opponent Modelling Reinforcement Learning Performance

We evaluate the average episode return that the TD3 [Fujimoto et al., 2018] algorithm achieves in the four environments when it is trained using the embedding network. We evaluate against the modelling method of Grover et al. [2018]. Both our model and the model of Grover et al. [2018] require access to opponent's observation and actions during training as well as execution. During the training, we evaluate our model every 2500 training steps for 100 episodes. During the evaluation episode, we use greedy

policy. The opponents that were used for the evaluation are randomly sampled from the initial set of pretrained opponents. In Figure 5.3, the average and two standard errors of the mean of the episodic returns of the two modelling techniques is provided.

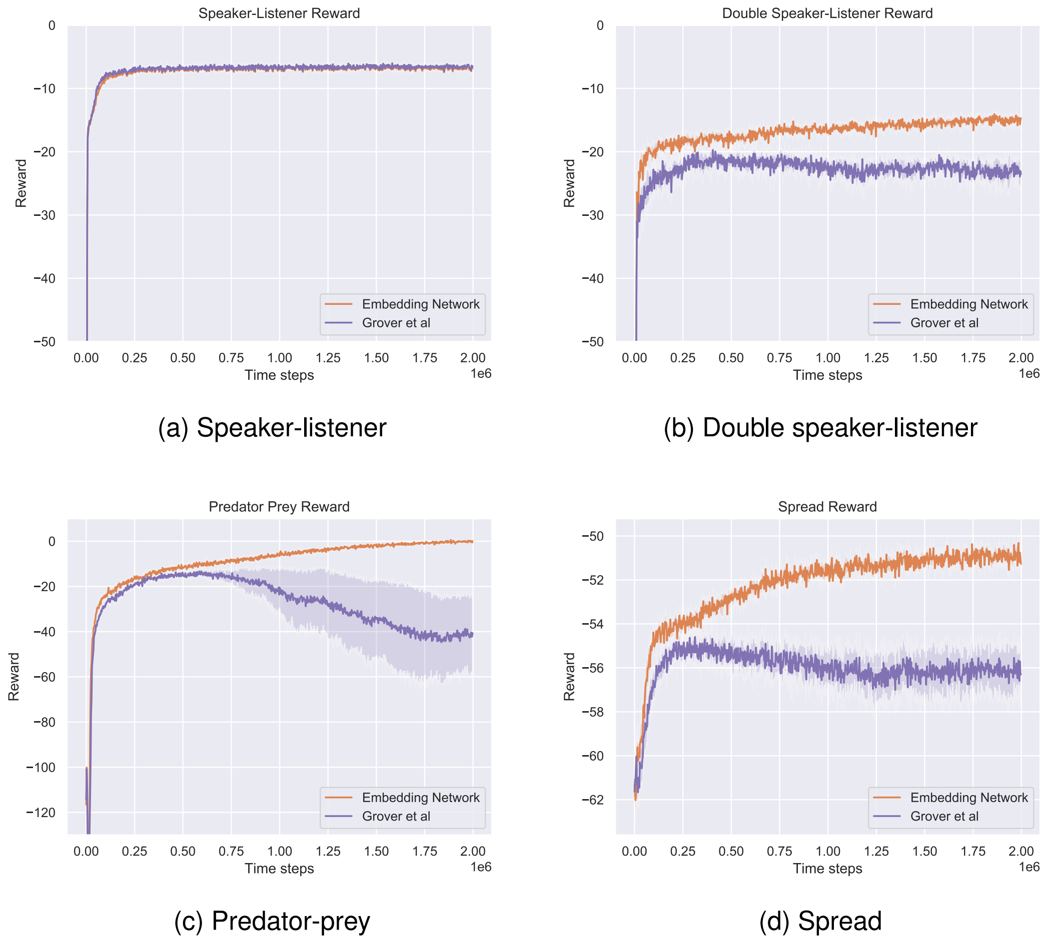


Figure 5.3: Performance of our modelling method compared to Grover et al. [2018]

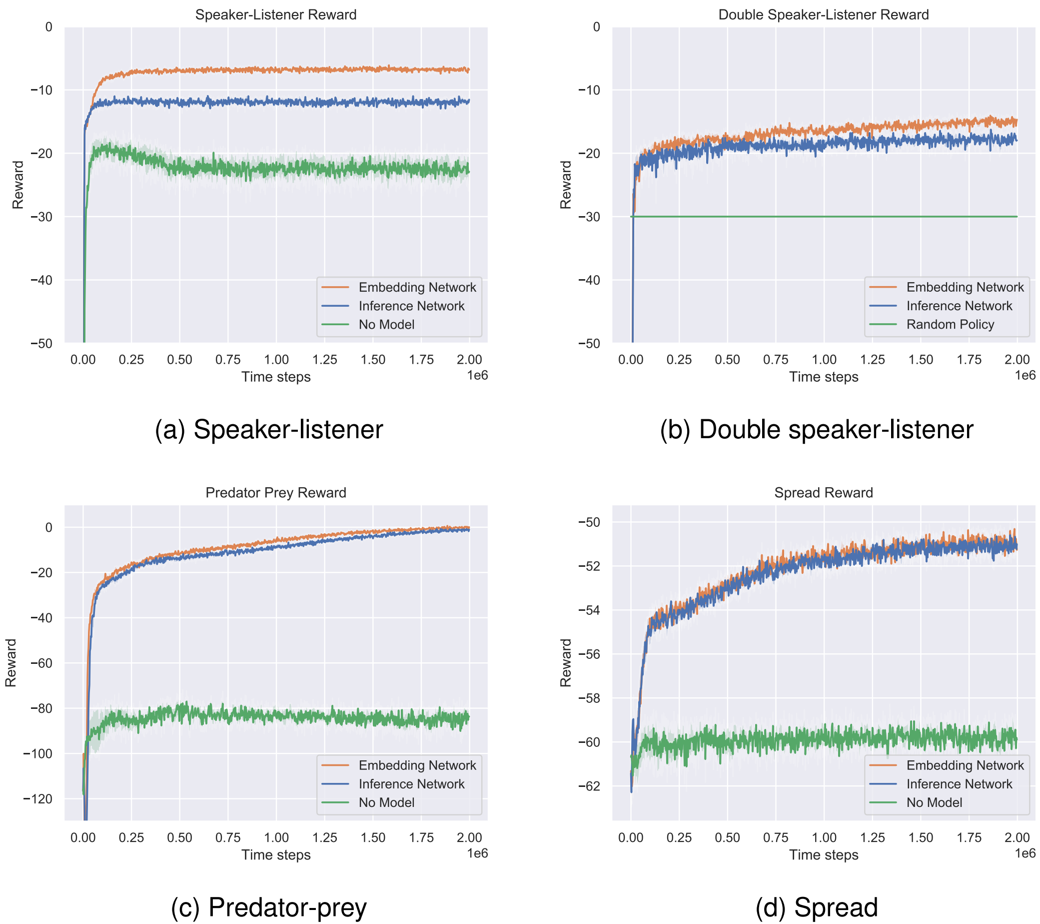
From Figure 5.3, it is clear that our method performs significantly better compared to the method of Grover et al. [2018] in all environments except the speaker-listener. We believe that our method performs better due to the modelling decision, using probability distribution instead of point-based representations. The Grover et al. [2018] method, as well as our method, is trained only on opponent trajectories that were generated against optimal or near-optimal agents. When the agent is not fully trained yet, the actions that it performs lead the opponent to perceive different observations from what the model is trained on. Our model is robust to this phenomenon because the probabilistic modelling covers a larger embedding space. On the other hand, the Grover et al. [2018] method only learns to create embedding in a small subspace of the embedding space. Another

fact to support our hypothesis is that the only environment where both models get the same performance is the speaker-listener, where the observations of the opponent, the color of the listener, is completely independent of the action of our agent, the listener. Therefore, the actions of our agent will not affect the model's output throughout the training procedure.

5.3.3 Inference Network Reinforcement Learning Performance

After evaluating our opponent modelling method, we evaluate the execution performance of our inference method in the four MPE environments. During the training, we evaluate our model every 2500 training steps for 100 episodes, similarly to the method that was described in the previous section. We compare the performance of our method with two baselines:

- **True Opponent Model (TOM)**: We train the TD3 algorithm using only the embedding network. Therefore, the agent has access to the opponent model that we proposed. This baseline requires access to an opponent's observations and actions during execution. TOM is an upper bound on the performance since access to the opponent's information is provided. The goal is our method's performance to be as close as possible to the TOM's performance.
- **No Opponent Model (NOM)**: We also present a lower bound baseline. We train TD3 without using any opponent modelling technique. This method is trained using as input only the local observation available. Therefore, the agent has no information about the opponents that it has to interact with. As a result, it is expected a lower performance compared to our proposed method.
- **Random**: In some environments, such as the double speaker-listener environment we could not successfully train TD3 against all five opponents. For this reason, we provide the experimental results from a random policy, which results in better returns compared to the trained TD3 without an opponent model.



(a) Speaker-listener

(b) Double speaker-listener

(c) Predator-prey

(d) Spread

Figure 5.4: Performance of our method and the two baselines

In Figure 5.4, we can see that the performance using the inference network is close to the performance using the embedding network. Therefore, the inference network can infer the opponent model successfully. On the other hand, the lower baselines, NOM, or random policy, result is significantly lower returns.

5.3.4 Inference Embeddings Evaluation

In addition to the evaluation of the average return, that the inference network achieves, we are also interested in evaluating how often the inference network identifies the correct opponent model. For this reason, we make one further evaluation. We run 1000 evaluation episodes, and we generate embeddings both from the inference network and from the opponent model. In the end, we use the k-means algorithm to cluster the embeddings that were generated from the opponent model. Finally, we evaluate the percentage of the inference embeddings that belong in the same cluster that the true

embeddings belong to. Given that there are 5 clusters, the average random accuracy is 20%.

Table 5.4: Agent prediction accuracy of the inference network.

Environment	Accuracy
Speaker-listener	$34.9\% \pm 2.1\%$
Double speaker-listener	$33.3\% \pm 1.5\%$
Predator-prey	$51.0\% \pm 1.4\%$
Spread	$41.2\% \pm 2.2\%$

From Table 5.4, we observe that the inference network is able to identify the correct embedding half of the episodes in the predator-prey environments, 41% of the episodes in the spread environment, while only one out of three times in the other two environments. There are two explanations for this behavior. The first is that many times, the agent does not require to know the correct embedding to achieve the performance that is presented in Figure 5.4. In order to verify this assumption, we perform an ablation study in Section 5.3.5. Secondly, many times, the correct embedding identification is impossible. Consider the scenario in the speaker-listener environment, where two of the landmarks are close to each other, and the agent has to navigate to one them. By being close to those two landmarks, the reward, that the agent receives is optimal, while the inference network cannot clearly understand which is the correct landmark in order to generate the correct embedding.

5.3.5 Ablation Study

We will now perform an ablation study to understand the performance requirements of the inference model better. Our proposed method utilizes observation, action, and reward sequences to infer the opponent's model. We will use different combinations of these elements in the inference network, and we will compare the average returns. The environment that will be used for the ablation study is the speaker-listener environment. In Figure 5.5, the average episode return is presented for four different cases; the observation, action, and reward are provided, the observation and the action are provided, only the observation is provided, and the action and reward are provided.

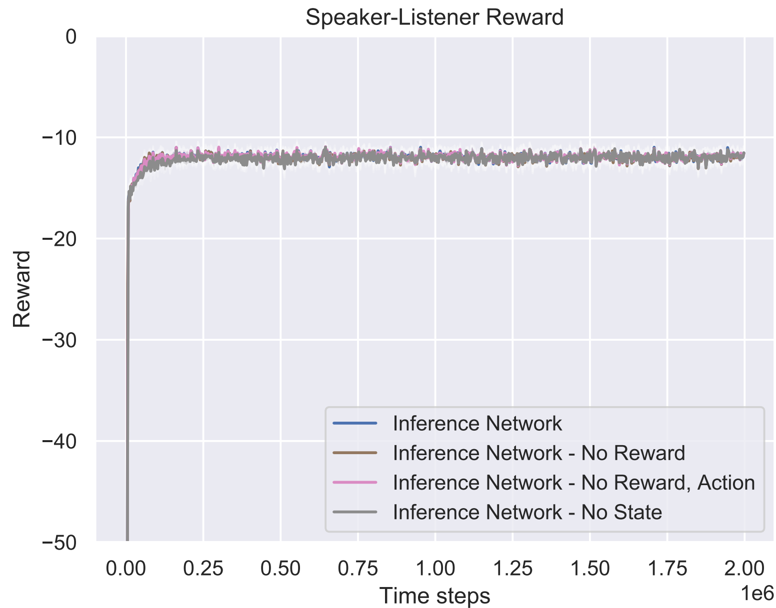


Figure 5.5: Performance of our method for different combinations of inputs in the inference network

From Figure 5.5, it is clear that the performance is the same for all different combinations of input. Therefore, the policy network probably tends to ignore the model and learns a good policy, based only on the observations. Additionally, this comes to a contradiction with the results of Figure 5.4, where the lack of the model led to low returns. This improvement of the average return, when the model is used may come from the sequential form of our learning architecture, such as the LSTM and the sequential experience replay. Another possible explanation is that the addition of noise in the learning architecture, during the sampling of the inference embeddings, leads to better exploration strategies as well as increase the robustness of the system. Finally, the last explanation behind the results of Figure 5.5 is related to the policy learning method. As mentioned above, the actor's loss is conditioned on the embedding network and not the inference. As a result, there is no gradient flow from the actor's error to the parameters of the inference network. Therefore, the parameters of the inference network are not shaped to maximize the episode's return and take advantage of different combination of inputs.

Chapter 6

Conclusion

To conclude this work, we will now present a summary of our contribution and present a brief review of the results of Chapter 5. Following this, we provide future research directions that this work could be extended to.

6.1 Discussion

To summarize our contribution, in this thesis:

- We proposed an opponent modelling method based on variational autoencoders, which is agnostic to the type of interactions of the agents (cooperative, competitive, mixed) in the environment.
- We extended the method of Grover et al. [2018] as well as our method in order to model more than one agent.
- We identified the limitation of the modelling methods, such as that access to opponents' information is required during execution, and we proposed a simple method to overcome this.

In Section 5.3.1, we provided visualization of the embedding space of our model. It is evident, both from Figure 5.2 and the Table 5.3 that different opponents are separable in the embedding space, in all the environment except the speaker-listener. Following this, in Section 5.3.2, we evaluated the average episode return that TD3 Fujimoto et al. [2018] achieves when conditioned on the representation that is created from our model against the representation that is created from the model proposed by Grover et al. [2018]. Our model outperforms the model of Grover et al. [2018] in all the

environments except the speaker-listener, where both achieve the same results. Finally, a possible explanation of this behavior is discussed.

After the analytical evaluation of our opponent modelling method, we also evaluated the inference method that we proposed. To this end, in Section 5.3.3, we evaluated the average returns that our inference method achieves against two baselines. The first baseline, which is an upper baseline as well, is the performance of our opponent model and the second baseline is against the algorithm TD3 Fujimoto et al. [2018] without any model, which means that it only has access to the observations of our agent. Using the inference network, the performance is close to the upper baseline and significantly better from the second baseline. Additionally, in Section 5.3.4, we present the opponent identification accuracy of the inference network. While the performance is relatively low, we reason that this is a natural consequence of the experimental framework. Finally, in Section 5.3.5, we perform an ablation study to understand the performance of the inference network. We conclude that the increased performance of the inference network results from design choices of the architecture, such as the sequential form of the training as well as the addition of noise in the training procedure.

6.2 Future Work

To finalize this thesis, below, we propose a number of promising research directions.

- As a first step, we would like to improve the idea of the inference network. During our evaluation, we concluded that the inference network does not result in better performance. Therefore, more work is required to achieve a better model that successfully infers opponents and contributes to performance improvement. Additionally, it would be fruitful to evaluate other environments in the future, where the inference model could result in significant improvement.

- In this thesis, we perform variational inference in two steps. First, we learn a model using the variational inference framework and then use the inference network to approximate the posterior that is generated by the model. Another direction of research that is worth investigating is performing variational inference in one step. That is, using the variational autoencoder framework, to directly approximate the unknown posterior with an inference distribution that is conditioned only on locally available information. This method can be trained end-to-end along with the policy, removing the assumption that trajectories of the opponents

are provided for training the model.

- Even though our proposed opponent modelling method can model more than one opponent, practically, a large number of opponents could be proven problematic. An exciting direction would be to investigate attention mechanisms so that the network will only be able to focus on opponent models that are more important at each timestep. A step in this direction has been proposed by Iqbal and Sha [2018], which proposes the use of attention along with the MADDPG [Lowe et al., 2017].

- Another fruitful direction could be the evaluation of the opponent model for dealing with non-stationarity. Our model assumes that pretrained opponents are provided. We would also like to evaluate scenarios, where multiple agents are learning in the same environment, and each agent tries to model its opponents in order to achieve optimal performance.

Bibliography

M. Al-Shedivat, T. Bansal, Y. Burda, I. Sutskever, I. Mordatch, and P. Abbeel. Continuous adaptation via meta-learning in nonstationary and competitive environments. *International Conference on Learning Representations*, 2018.

S. V. Albrecht and P. Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95, 2018.

D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 2005.

M. Bowling and M. Veloso. Rational and convergent learning in stochastic games. 17 (1):1021–1026, 2001.

M. Bowling and M. Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.

S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio. Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*, 2015.

V. Conitzer and T. Sandholm. Awesome: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents. *Machine Learning*, 67(1-2):23–43, 2007.

Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl ^{2}: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

J. N. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. pages 1587–1596, 2018.

A. Grover, M. Al-Shedivat, J. K. Gupta, Y. Burda, and H. Edwards. Learning policy representations in multiagent systems. *International Conference on Machine learning*, 2018.

A. Gupta, R. Mendonca, Y. Liu, P. Abbeel, and S. Levine. Meta-reinforcement learning of structured exploration strategies. In *Advances in Neural Information Processing Systems*, pages 5302–5311, 2018.

D. Ha and J. Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

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

M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

K. Hausman, J. T. Springenberg, Z. Wang, N. Heess, and M. Riedmiller. Learning an embedding space for transferable robot skills. 2018.

H. He, J. Boyd-Graber, K. Kwok, and H. Daumé III. Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pages 1804–1813, 2016.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. 2018.

I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. *International Conference on Learning Representations*, 2(5):6, 2017.

S. Iqbal and F. Sha. Actor-attention-critic for multi-agent reinforcement learning. *arXiv preprint arXiv:1810.02912*, 2018.

M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

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

D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, pages 157–163. Elsevier, 1994.

R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

C. J. Maddison, A. Mnih, and Y. W. Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, page 529, 2015.

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

I. Mordatch and P. Abbeel. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*, 2017.

G. Papoudakis, F. Christianos, A. Rahman, and S. V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning. *arXiv preprint arXiv:1906.04737*, 2019.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

N. C. Rabinowitz, F. Perbet, H. F. Song, C. Zhang, S. Eslami, and M. Botvinick. Machine theory of mind. *International Conference on Machine Learning*, pages 4218–4227, 2018.

R. Raileanu, E. Denton, A. Szlam, and R. Fergus. Modeling others using oneself in multi-agent reinforcement learning. *arXiv preprint arXiv:1802.09640*, 2018.

K. Rakelly, A. Zhou, D. Quillen, C. Finn, and S. Levine. Efficient off-policy meta-reinforcement learning via probabilistic context variables. pages 5331–5340, 2019.

S. Ruder. An overview of gradient descent optimization algorithms. `http://ruder.io/optimizing-gradient-descent/index.html`, 2016.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

J. Schmidhuber and S. Hochreiter. Long short-term memory. *Neural Comput*, 9(8): 1735–1780, 1997.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning*, 2014.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

R. Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

J. Walker, C. Doersch, A. Gupta, and M. Hebert. An uncertain future: Forecasting from static images using variational autoencoders. In *European Conference on Computer Vision*, pages 835–851. Springer, 2016.

J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2017.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

L. Zintgraf, M. Igl, K. Shiarlis, A. Mahajan, K. Hofmann, and S. Whiteson. Variational task embeddings for fast adaptation in deep reinforcement learning. *International Conference on Learning Representations*, 2019.