IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Deep Deconstructed Learning

Author:
Max Wickham
CID:
01717673

Project Supervisor:
Prof. Jeremy Pitt
Secondary Project Supervisor:
Dr. Dan Goodman
Secondary Marker:
Dr Adam Spiers

Date: June 21, 2023

1 Plagurism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

I have used ChatGPT V4 to aid in generating latex math equations and tables in my report.

2 Acknowledgements

Firstly I would like to thank Professor Jeremy Pitt for his support, advice and lastly patience throughout this project, especially since, due to the broad starting scope, the project's direction was often worryingly unclear.

Secondly, I would like to thank Dr Dan Goodman for his advice and additional insight into this field.

3 Abstract

The concept of Multiple Agents controlling a single mind has been present in literature for some time, both in the fields of machine learning as well as the study of human cognition. This project delves into the adaptation of this concept to the field of RL in an attempt to investigate existing and discover novel techniques to deconstruct an RL agent into multiple specialised sub-agents. Throughout this project many different problems that occur when trying to attempt this challenge are considered, as well as the different ways these problems can be alleviated. Some encouraging results were obtained when deconstructing a Policy Based RL agent, with minor increases in training times obtained as well as an indication of problems that may arise when training the entirety of a mono-agent at once.

Contents

Part I
Introduction

4 Problem Specification and Motivation

The focus of this project is to investigate the deconstruction of RL Models into internal sub-agents working in unison to control a single "body". Conceptually this idea bears a resemblance to and is inspired by the concepts presented in Minsky's "Society of Minds" [20], in which Minsky describes the human brain as an assembly of sub-agents, each specialised at a specific task. Within the assembly, each agent is vying for control of the various actuators available to the system through which to interact with the external state.

The project aims to investigate how agent policies naturally separate through exploration using a variety of training methods, unlike previous work that encourages sub-agent specialisation by forcing differing policies through regularisation. Also, unlike many previous investigations into multi-agent control of a single system, this project does not provide manual deconstruction of tasks and rewards to agents, requiring all policy deconstructing to be done by the training process itself.

The appropriate manner in which to go about implementing the ideas presented was not clear at the start of the project, which enforced a large degree of research into the field of RL and the various algorithms commonly used. Developing a good understanding of the internal working of these base algorithms necessitated developing implementations of these algorithms from scratch in order to then introduce modifications.

There are many reasons for wanting to achieve this decomposition, the first of which is potentially decreasing the communication bandwidth between different components of the overall model via the subdivision, allowing greater parallelisation of computation. Secondly, allowing for the tuning of different time periods between inferences for each sub-agent, enabling faster inference times and finally by deconstructing the reward signal giving some insight into the manner in which a model learns different strategies.

The decomposition of the models described in this project is done entirely by the developed algorithm and does not require human reasoning to construct sub-reward functions. The exact mechanism through which this decomposition occurs is dependent on the RL algorithm used, and developing this mechanism is one of the main focuses of the project. The project should also use the theories presented in multi-agent style problems, such as game theory, to enable and stabilise the training of sub-agents.

The models investigated in this project will all be trained to play Atari Style games that will be developed to construct differing scenario types for the agents to solve.

5 Report Structure

The rest of this report is composed of the following structure.

- Part II presents an overview of the background research needed in this project, including both basic RL techniques as well as more in-depth research into Multi-Agent RL.
- Part III describes the general infrastructure implemented to test different models as well as the implementations of the base RL algorithms used.
- Part IV goes into detail on the different agent deconstruction methods investigated.
- Part V outlines the testing and experimentation performed to evaluate the performance of each algorithm used.
- Part VI finally gives an analysis of the experimental results as well as comments on the general outcomes and findings of the project.

Part II
Background

Before being able to start building multi-agent models, a comprehensive review of single-agent reinforcement learning methods was required to both design the models used within the project itself as well as build single-agent models to compare the performance of different designs given different tasks.

Reinforcement learning models are generally split into two broad categories, Model-Free and Model-Based, both of which will be discussed here [15]. A common requirement in many algorithms in both of these categories is the ability to describe a problem as a Markov decision process, a necessity that is difficult to maintain when extending to multi-agent problems. The form of a Markov decision process as described by Bellman [30] is given in equation 1

$$S: \text{ Set of states} \tag{1a}$$
$$A: \text{ Set of actions} \tag{1b}$$
$$P(s_{t+1}|s_t,a_t): \text{ Probability of moving from state } s_{t+1} \text{ given the current state and action} \tag{1c}$$
$$R(s): \text{ The reward for moving to a state} \tag{1d}$$

The MDP can be described with a tuple of these four values, $(S,A,P(s_{t+1}|s_t,a_t),R(s))$, and has the requirement that the probability of moving from one state to another given a specific action is only dependent on the current state and not the history of previous states. It will be shown later that this requirement is far easier to model in single-agent RL than in a multi-agent system, becoming a significant limitation in multi-agent RL. The general goal of solving an MDP is to discover the optimum policy which maps a state to a distribution of action probabilities to maximise the cumulative discount reward as shown in equation 2.

$$\text{Discount factor: } \gamma \tag{2a}$$
$$\text{Policy:} \tag{2b}$$
$$\pi : \mathscr{S} \rightarrow \Delta(\mathscr{A}) \tag{2c}$$
$$\text{Cumulative Discount Reward:} \tag{2d}$$
$$E\left[\sum_{t\geq 0} \gamma^t R(s_t,a_t,s_{t+1}|a_t\ \pi(|_t),s_0\right] \tag{2e}$$

6 Model-Free

Model-Free RL makes decisions during the training process without any prediction of what the following or future states will be; instead, simply gives an action recommendation based on the current state and continuous feedback. Model-Free RL is itself split into two major categories, Off Policy (e.g. Q-Learning) and On Policy (e.g. Policy-Optimisation) [15] that will both be discussed here.

6.1 Q-Learning

Q-learning is a method of updating the reward given a particular action with a given state in a table in order to be able to compute the cumulative reward of taking any action at any state. This table is updated using the Bellman Equation described by Richard Bellman [31] with the update form shown in equation 3.

S : **State Space** (3a)

A : **Action Space** (3b)

R : **Reward Function** (3c)

γ : **Reward discount rate** (3d)

$\pi : S \rightarrow A$: **The policy we are trying to learn** (3e)

We want to give a value to each state given an action (3f)

$$Q(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right] \tag{3g}$$

This q-value can be updated with the Bellman equation (3h)

$$Q^*(s,a) = E[r_{t+1} + \gamma \max_{a'} Q^*(s',a')] \tag{3i}$$

$$Q^{n+1}(s,a) = (1-\alpha)Q^n(s,a) + \alpha(r_{t+1} + \gamma \max_{a'} Q^n(s',a')) \tag{3j}$$

Using this formulation of the Q-function, we can describe the Q-function and V-function (state value function) for a given policy within an MDP as shown in equation 4

$$Q_\pi(s,a) = E\left[\sum_{t\geq 0} \gamma^t R(s_t, a_t, s_{t+1} | a_t \ \pi(\dot{|}_t), a_0 = a, s_0 = sZ\right] \tag{4a}$$

$$V_\pi(s) = E\left[\sum_{t\geq 0} \gamma^t R(s_t, a_t, s_{t+1} | a_t \ \pi(\dot{|}_t), s_0 = sZ\right] \tag{4b}$$

It has been shown that updating $Q(s,a)$ using this method will ensure convergence to a $Q^*(s,a)$ that implements the optimum policy [41]. This idea is fundamental in many RL techniques.

Since the size of the state and action space in most problems is extremely large, storing every state-action pair in a table with the associated values is infeasible and therefore, many methods try and find a function that approximates the Q-function, $Q^*(s,a) \approx Q(s,a)$, often using neural networks or other machine learning techniques. In this 2013 paper from DeepMind [22], it is shown that this method of Q-function approximation can be used to play emulations of Atari games with good performance. The function approximator used in the paper, $Q(s,a;\theta)$, uses the value θ as the parameters for the neural network used to approximate the function, with the requirement that this parameter must be updated at each training step in a manner that tends towards the optimum policy described using the Bellman equation. This set of Atari 2600 games used in the paper appears to be a standard metric of the performance of RL algorithms.

Sutton et al. [34] described a modification to standard Q-Learning that significantly improves performance by using a method called "experience replay" that stores the values $(s_t, a_t, r_t, s_{t+1}, a_{t+a})$ in a memory set. Then during training, random subsets of this memory are taken to form mini-batches used to train the model. The state values represent the entire history of inputs to the model up to time t, along with the actions taken at each step, with the inputs being the values of each pixel in the given game. The state is described in this way to ensure the system maintains the properties of an MDP, which is a requirement for Q-Learning, and the reward value is the change in in-game score and is specific to each game type.

If only the current frame were used as the state, the probability of moving to another state would not always be consistent as the frame itself does not contain time-dependent information such as velocities; however, by combining the history of frames seen as the current state all the necessary information to form a system with consistent state change probabilities is maintained. In the paper, instead of using the full set of pixels, some preprocessing was done to reduce dimensionality in addition to only using the last four frames of history as input to the model as opposed to the entire history. By only using the last few frames,

the input to the model is maintained at a fixed size, making the neural network easier to implement. The complete algorithm, "Deep Q-learning with Experience Replay" as described by this paper, is shown in Algorithm 1. The complete algorithm from the paper has been included here, as many references will be made to it throughout the report.

Algorithm 1 Deep Q-learning with Experience Replay [22]

Initialise replay memory $\mathscr{D}$ to capacity $\mathscr{N}$
Initialise action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With random probability η select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathscr{D}$
    Sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathscr{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non terminal } \phi_{j+1} \end{cases}$
    Perform a gradient step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Some adaptations to the "Deep Q-learning with Experience Replay" algorithm were suggested in a 2015 paper [12]. It is proposed that using the same function approximate to both evaluate actions and to select actions results in overestimated values being more frequently chosen, a common problem in Q-Learning [38]. The 2015 paper utilises "Double Q-Learning", first described by van Hasselt in 2010 [39] applied to Deep Q-Learning to reduce the overestimation errors. This method uses two models trained in parallel for each of the action selections used in algorithm 1. One model for choosing the action to take and one for adjusting the value of y_j using the discount reward. The secondary model is updated to match the original model periodically. The change described resulted in a significant increase in performance across all Atari 2600 games, often with scores several times better than those achieved using the original Deep Q-learning approach.

In 2017, a new approach was described by DeepMind, and the University of Cambridge researchers named "Distributional Reinforcement Learning with Quantile Regression" [6] that provided further performance gains on Atari 2600 games. This paper proposes methods for learning the "Value Distribution" instead of the "Value Function" used in traditional RL. The agent environment is described using the Markov decision process $(\mathscr{X}, \mathscr{A}, \mathscr{R}, \mathscr{P}, \gamma)$, where $\mathscr{X}$ is the state space $\mathscr{A}$ the action space, $\mathscr{R}$ the random variable reward function, $P(x'|x;a)$ the probability of transitioning to state x' given action a and $\gamma \in [0,1]$ is the discount factor. An approximator $Z^\pi(x,a)$ is found that gives the probability distribution of the cumulative reward of a given action, where the standard $Q^\pi(x,a)$ would be the mean of this distribution. This method was able to improve on the performance of Double-DQN.

6.2 Policy Optimisation

In addition to Q-learning, policy optimisation is also used in Model-Free RL. Policy optimisation algorithms use an On-Policy instead of an Off-Policy like Q-Learning, meaning that updates are only made using samples collected with the current model version. Policy optimisations methods are formulated using $\pi_\theta(a|s)$ [15] with θ optimised using gradient ascent over a performance index $J(\pi_\theta)$. This formulation allows for learning over the policy space instead of the reward space, meaning that if the reward function is too complex, the policy optimisation method may outperform DQN. Policy optimisation also has the additional benefit that it can be applied to a continuous action space, whereas DQN must discretise the action space, which may be expensive.

Policy gradient methods iteratively improve on the current policy by creating an estimate for the policy gradient to increase the expected performance for a policy, $J(\pi_\theta)$. A common form of this gradient estimator, according to a paper from OpenAi [32], is shown in equation 5.

$$E_{\tau\ \pi_\theta}[\nabla_\theta \log(\pi_\theta(a_t, |s_t)R(\tau)] \tag{5}$$

Here π_θ is a stochastic policy which maps a state and an action to the probability of taking that action, unlike the deterministic policies mentioned earlier that map a state directly to an action, and the $E_{\tau\ \pi_\theta}$ represents an average over a set of training samples. A complete derivation of this log gradient can be found in this paper [25].

An algorithm proposed in a 2017 paper [32] improved on existing policy optimisation techniques whilst being simpler to implement than existing state-of-the-art policy optimisation methods. This algorithm is called "Proximal Policy Optimisation", and variations of it are the current state of the art in on-policy learning. The process described in this paper is outlined in algorithm 2.

Algorithm 2 PPO, Actor-Critic Style [32]

for iteration $= 1, 2, ...$ **do**
  for actor $= 1, 2, ...N$ **do**
    Run policy $\pi_{\theta\text{old}}$ in environment for T time-steps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  Optimise surrogate L using wrt θ, with K epochs and mini-batch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

The advantage estimates used in the algorithm are described by equation 12 taken from OpenAi [25] and give an estimate for the increase in expected reward of taking a specific action as opposed to taking a random action according to the distribution given by the policy π. These advantage estimates computed at a set of time steps by several actors can be used to compute a loss function for each sample allowing for the training of the policy model [32], with the paper providing a lot of information on how best to compute these advantages and the resultant loss. The most common method of advantage estimation is the use of a critic network [32].

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{6}$$

6.3 Actor-Critic

The actor-critic model is a general category of policy optimisation. Two networks (actor and critic) form the model, with the actor making policy decisions and the critic used during training to inform the actor on how good their decision was through advantage estimates. The actor part of the model is simply an on-policy model that can be implemented in several ways, such as with the PPO algorithm. Some parameters in the actor and critic models are often shared as many of the features useful in calculating an action will also be present in calculating the value function.

One of the most popular actor-critic-based algorithms is A3C [21] (Asynchronous advantage actor-critic), which approximates both a policy function $\pi(a_t|s_t;\theta)$ and a value function $\mathscr{V}(s_t;\theta_v)$, with both functions updated periodically during training. The update to the policy function is made using an estimate of the advantage function, which is calculated using the value function. The update to the value function is made using the reward. All modifications to the two models are implemented asynchronously so that multiple simulations can be run in parallel, all training the same model.

The paper describing the A3C algorithm found that it could outperform DQN-based algorithms significantly in Atari 2600 games [21].

6.4 Comparison of Policy Optimisation vs Q-Learning

Generally, policy optimisation methods have many advantages over Q-Learning-based approaches. These include faster learning, greater applicability, and the ability to be used with a stochastic or continuous action space [23]. Despite these advantages, policy optimisation methods can suffer from poor sample efficiency [23], in addition to being more likely to remain stuck at local minima compared to Off-Policy methods. This poor sample efficiency comes from Q-Learning methods being trained from any trajectory sampled from the environment, whilst Policy-Optimisation must collect samples using the current policy. Current comparisons appear to find that PPO-related algorithms are often able to beat DQN-related algorithms in many games, such as the Atari 2600 set. This paper [42] demonstrates DQN marginally outperforming PPO, especially in navigation, however, research done by OpenAi demonstrates improved results for PPO compared to state-of-the-art DQN algorithms such as Rainbow [24]. In fact, this paper found that the Joint-PPO method could marginally outperform the DQN-based Rainbow algorithm as the Joint-PPO pre-trains the model on a set of test levels and uses this pre-trained model as the initialisation for the model used on the test levels. No noticeable improvement was found when using the same pretraining method for the Rainbow algorithm, demonstrating the increased ability of PPO to transfer learning from other environments to a test environment compared to DQN-based approaches.

Since both model types have advantages in different situations and have been shown to achieve similar performance in Atari 2600 games, both will be tested in the project to investigate their respective applicability. It may even be the case that different components of the multi-agent models may implement either Policy-Optimisation or Q-Learning-based algorithms in conjunction.

7 Model Based RL

Unlike Model-Free RL, model-based RL allows for the agent to make predictions about the future of the state space using a model that approximates the environment [15]. Examples of where this technique is highly applicable exist in games such as chess, where the exact rules of the environment can easily be programmed. Model-Based RL can also be employed in situations where the environment can't be so easily described, in which case a model must be trained to approximate the environment. However, model-based RL often has many disadvantages due to inaccuracies in the model and in the case of Atari 2600-style games, it is challenging to learn a model that is accurate enough to be used in decision-making [17]. Due to the apparent lack of applicability in the literature for this model type with Atari 2600-style games, the project will not focus on this form of model.

8 Multi Agent RL

The fundamental problem facing multi-agent RL as opposed to single-agent systems is that as agents learn in parallel, the environment around them changes, resulting in a loss of the Markov property [43]. Additional challenges also exist in handling the multi-dimensional objectives of agents who may not be co-operative, resulting in the need to discover equilibria on top of the actual task of deciding agent objectives. The design of agent objectives results in three forms of a multi-agent system:

1. Fully Cooperative
2. Fully Adversarial
3. Combination of Cooperative and Adversarial

Although fundamentally similar, these three categories have specific challenges to overcome, some of which will be outlined here.

8.1 Markov Games and Extensive Form Games

The framework for describing an environment as an MDP can be adapted to an environment with multiple players within the environment to provide a mathematical basis to describe the optimum policy of each player. Markov Games extend the concept of an MDP to a multi-agent system, with their use in RL

described in 1994 [19]. Equation 7 provides the general formulation for a Markov Game as described in this paper [43].

$\mathscr{N}$ Set of agents (7a)
$\mathscr{S}$ Global state observed by all agents (7b)
$\mathscr{A}$ Set of action spaces for each agent (7c)
$\mathscr{A}^i$ Action space for agent i (7d)
Probability of moving from one state to another given a set of actions (7e)
$\mathscr{P} : \mathscr{S} \times \mathscr{A} \to \Delta(\mathscr{S})$ (7f)
Reward for an agent moving from one state to another with a given action (7g)
$\mathscr{R}^i : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \to R$ (7h)

Each agent described in this system is trying to discover a policy that maximises their long-term reward; however, the reward given by a policy is, of course, dependent on the policies of other agents, as the state change transitions are dependent on all agents' policies. If the policies of other agents are stable, these policies can be thought of as part of the state space resulting in an MDP from the point of view of the given agent. The solution to this overarching problem is, of course, dependent on what the goal of the system is; however, a common way of describing a solution to such games is through a Nash equilibrium, which is a joint policy such that for each agent, given every other agent's policy their policy is optimum, meaning no agent is incentivised to deviate from the joint policy. Nash equilibria are not necessarily unique but are guaranteed to exist for finite-space, infinite-horizon Markov Game [19].

If all agents within a Markov Game are fully cooperative and have full knowledge of the state space, the system can be described as an MDP, with the joint Policy being the solution given through methods such as Q-learning. This fact demonstrates the need to ensure the rewards of cooperative agents are decoupled, as the agents within this project will be working towards a single reward function.

An issue with Markov Games, as shown in the formulation, is that each agent requires full knowledge of the global state, which will likely not always be the case for models within this project. Extensive form games are an alternative to Markov Games that do not have this constraint [43] as described in this 1994 book [26]. Equation 8 defines an extensive form game [26].

$\mathscr{N}$ Set of agents (8a)
$\mathscr{H}$ Set of all possible sequences of actions (histories) (8b)
$\mathscr{Z}$ Set of terminal histories (8c)
$\mathscr{A}(h)$ The set of possible actions after a non terminal history h (8d)
Function that maps a history to an agent to decide the next action, (8e)
c represents chance deciding the next action (8f)
$\mathscr{P} : \mathscr{H} \to N \cup \{c\}$ (8g)
Function that gives the probability distribution of taking each action if left to chance (8h)
$f_c(.|h)$ (8i)
$\mathscr{I}^i$ For each agent, a set of sets of histories that will result in the same action by that agent (8j)
$\mathscr{S}$ A set of sets of histories that will result in the same player chosen and same action taken (8k)
The games is described with the tuple: (8l)
$< \mathscr{N}, \mathscr{H}, \mathscr{P}, f_c, (\mathscr{I}_i)_{i \in N} >$ (8m)

The fact that agents cannot distinguish between histories in the same set $s \in S$ demonstrates that agents don't have complete knowledge of the global state.

8.2 Multi Agent Algorithms

Many algorithms have been developed to extend those used in single-agent RL to multi-agent systems. Some ignore the multi-agent dynamics entirely and train each agent independently of the others using standard RL techniques. Despite the limitation of this method, it has been found to achieve good results in some scenarios [**Hernandez-LealA**]. Another branch of algorithms within MARL allows agents to model other agents, especially in competitive situations; however, since this results in repetition of the computation done by agents, this report focuses on algorithms that don't use this method or actively avoid it to minimise resource use.

Some of the standard algorithms used in MARL, which are used to inform the methods used in the implementation of this project.

1. MD-MADDPG [28] (Using shared memory for communication)
2. Lenient-DQN [27]
3. RIAL / DIAL [9] (Parameter sharing for communication)
4. VDN [33] (Decompose the value function between agents)
5. LAPG [5] (Reduces communication between agents through the use of communication trigger rules)
6. CCEAs [29] (Decompose Problems into Sub-Problems)

In some situations, agents may need to cooperate to maximise the global reward. Some algorithms ignore this problem by assuming that the optimum local action for agents will result in coordination which is rarely the case [3]. Others make use of a variety of methods to enable the coordination required. An example of this is the decomposition of a Q-function to local Q-functions dependent on the global state, meaning that the Q-functions can be designed to ensure that if each agent maximises their Q-function, the coordination required will be inherent [3]. Another method of achieving this result is to allow agents to form models of other agents to predict their actions. As previously discussed, this project will avoid such algorithms to reduce model complexity.

8.3 Learning Communication

One desired characteristic of the agents within this project may be a mechanism to allow for inter-agent communication. Such a system needs to minimise the amount of information flow between agents to ensure they are completely decoupled and to prevent agents from modelling other agents. This can be done through several mechanisms, including coupling the neural networks to some extent, providing some form of shared memory or providing outputs that will be fed to other agents on the next timestep [**Hernandez-LealA**].

An example of an algorithm that passes and inputs to other agents on the subsequent timestep is Reinforced Inter Agent Learning [9]. This algorithm is based on DQN and models a Q-function using a neural network to make decisions upon an action. Each agent Q-function depends on the observed state, hidden state and messages received from other agents. To prevent the need for messages and actions to be outputted by a single function, a separate Q-function is used to output actions and messages for each agent. The paper also describes modifications made to the DQN algorithm due to the presence of multiple agents, the most important of which is the removal of experience replay due to the non-stationary nature of the environment. Messages outputted by agents are only provided to other agents on the next timestep, as well as the previous action each agent took.

In addition to RIAL, the paper describes a second communication mechanism called Differentiable Inter-Agent-Learning. This algorithm reduces constraints on the bandwidths of messages during training. It allows gradients of messages to be passed back during training permitting agents to give feedback on communication during training, resulting in a single end-to-end model to be trained with support for the backpropagation of errors over message channels. During execution, the information sent through messaging is discretised, reducing the bandwidth of information sent once again.

The paper's experimental results found that DIAL significantly increased performance on RIAL, learning the optimal solution to given problems much faster. Although generally, RIAL outperformed a no-

communication baseline in some scenarios; it wasn't able to in many problems, giving similar results to the no-comm version.

The problems used to test these two algorithms in the original paper are all structured to encourage homogeneous agent model parameters and networks. Therefore, although this is not a requirement of the algorithms, the experimental results may not apply to the problems researched in this project.

Another approach to inter-agent messaging is Memory-Driven Communication. The Memory-Driven Multi-Agent Deep Deterministic Policy Gradient algorithm [28] is one proposed approach. This method provides a shared memory pool that each agent reads from along with the observed state when making decisions and then writes a response to the memory. The algorithm described by the paper process each agent sequentially, such that each agent reads from memory and then calculates the changes to the memory and their action. This means the memory is updated n times for an n agent system each timestep, with some randomly generated noise added to the memory on each update. The models themselves are trained using an actor-critic-based approach, as described earlier.

The paper found that the more complex the task the algorithm was tested on, the more it outperformed other communication systems and no-comms models. This is indicative of the fact that communication becomes more important the higher the complexity of the problem, however, the paper does not provide much information on performance as the number of agents increased.

8.4 Cooperative MARL

As previously stated, one of the significant problems facing MARL frameworks is the changing environment dynamics resulting in specific Markov properties being void. This paper by Foerster et al. [8] investigates how this problem affects the use of experience replay memory, as is used in DQN algorithms. As stated in the paper, experience replay is an essential component of DQN as it improves the sample efficiency by reusing previous samples as well as helping stabilise the training of the network. However, since the policy of other agents is constantly changing, it may not be feasible to use previously stored state changes from replay memory due to the changing dynamics of the system in a MARL environment. To overcome this problem, the paper discusses the use of IQL, where other agents' policies are treated as part of the state space, meaning that replay memory can be used as the changing dynamics are now incorporated into the state. The issue with this method is that the increase in state space size can make the learning infeasible [8], introducing the requirement to find a low dimensionality "fingerprint" representing the policy state of other agents. The paper found that the current rate of exploration of an agent was a sufficient descriptor of their current policy to stabilise replay memory effectively. The reasoning provided is that it is not necessary to know the policy of other agents but rather the position along the trajectory of their policy that a sample is taken at, which is indicated by the rate of exploration.

As previously discussed, cooperative MARL can use homogeneous agents that implement the same policies but differ in actions due to a different partial observation of the state space. Gupta et al. [11] outline how parameter sharing between such agents can allow for the training of one model from the experiences of many agents in parallel and how the homogeneity of the model reduces some of the problems in MARL, such as the changing dynamics of the environment. The use of homogeneous agents may apply to some tasks researched by this project, although further research will be required if this is implemented.

It may not be evident in some MARL problems whether specific agents are working towards a similar goal and, thus, if they should cooperate. Zhang et al. demonstrate [44] the use of correlation coefficients to measure the correlation between agent rewards, thus indicating whether they should cooperate. To implement coordination between agents whose reward functions have a high correlation, a level of communication is added between them to allow for information transfer. This method ensures that the overhead of communication is not added between agents who are not likely to need to work cooperatively and thus need to exchange information. The three correlation coefficients used include Pearson, Spearman and Kendall correlation, with Pearson giving the best results. The experiments carried out by the paper found that their algorithm (ALC-MADDPG) performed better than other state-of-the-art algorithms, with an increased performance benefit found in more complex tasks, again highlighting the importance of cooperation in complex tasks.

9 Modular Reinforcement Learning

Modular Reinforcement Learning is a field in RL in which a single monolithic task is decomposed into many simpler problems that can be trained independently. An example of this is the Hybrid Reward Architecture [40] which decomposes a reward function into several sub-reward functions in a manner very similar to the aim of this project. In this paper, different sub-agents, each implementing an off-policy model, could be trained to make decisions about a single global action, with their outputs passed to a global aggregator that combined each decision to form the final action. The HAR algorithm proposed by the paper uses the DQN techniques discussed previously to train each model on an individual reward function, adding the requirement that the individual reward functions sum to give the global reward function as shown in equation 9.

$$R_{env}(s,a,s') = \sum_{k=1}^{n} R_k(s,a,s') \tag{9a}$$

The main requirement described by HAR on the sub-reward functions is that they should be affected by only a small number of state variables in order to make the Q function easier to learn. The different agents can globally be modelled as a single model that shares many of the lower network levels between sub-agents, which then split into multiple "heads" that output the actions of each agent. The output of each "head" in the global model is summed with an additional fully connected final layer that has all weights set to one, giving the result shown in equation 9. In the original HRA paper, the manner in which the reward function is decomposed is entirely done by hand based on a human evaluation of the problem. For example, in testing the HRA algorithm on Pacman, the paper describes how each object in the game was given a separate reward function; for instance, touching a ghost may have a negative reward and touching a fruit may have a positive reward. Each reward function is used to train an RL agent that can have multiple heads, where each head could represent a possible position for an object. During the training process, heads are added to each model as objects are found at different locations within the game.

9.1 Reward Decomposition

Recently there has been a fair amount of research into the problem of reward function decomposition, with several approaches having been proposed. To learn meaningful rewards, it is important the sub-policies for each sub-reward only maximise their corresponding reward and not any other, as otherwise, the benefits of decomposition are negated. The Horde architecture [35] describes the concept of General Value Function, which considers not only the policy as an input to a Q function but also the reward function and termination function, i.e. $Q^{\pi,r,z}$, where the r and z are functions that represent the reward and termination functions as a pseudo reward and termination. The Horde architecture also discusses the potential of describing the terminal function γ as a pseudo function, where the termination function describes when to reset the state of an agent. The final description of the GVF given these functions is of the form shown in equation 10

$$q : \mathscr{S} \times \mathscr{A} \to \mathscr{R} \tag{10a}$$
$$\text{Auxiliary functions: } \gamma, r, z, \pi \tag{10b}$$
$$q(s,a;\gamma,r,z,\pi) \tag{10c}$$

This definition of a GVF allows for the possibility of defining GVFs that don't relate to the base problem being solved, an idea used by the Hore architecture to describe sub-agents, also called demons, that can each be ascribed to a unique GVF. Since the GVFs describe an off-policy q-learning approach, many GVFs can be trained in parallel, as it is not a requirement that the overall agent implements the policy of a specific GVF to train it, allowing for models to be composed of many thousands of GVFs. An example of how the Horde architecture could be implemented on a robot is by creating a demon for each combination of sensor and action, in which case each demon would be trying to learn the discounted sum of the input when repeating the given action; the combination of these sums can be combined to make a final description of the action.

An implementation of Modular RL named Distributed Reward Decomposition related to the Horde Architecture was described by Lin et al. [18], in which a distribution of the expected return of different reward functions is learned by each sub-agent. This is similar to the Rainbow algorithm mentioned earlier, and in fact, the agents in this algorithm were implemented using the Rainbow algorithm. Within the algorithm, each agent learns an approximate of a function that outputs the discrete probability distribution of the expected sub-return, with the learnt function denoted by $\mathscr{F}_i(x,a)$. To find the distribution of the sum of expected returns, the individual distributions can be convoluted to combine into this total distribution, a result taken from probability theory. As mentioned earlier, it is vital that the sub-reward functions are relatively independent, and so to ensure this fact the algorithm tries to maximise the cross entropy between total distributions outputted when trying the actions that maximise each sub-reward function in turn. In testing, the paper states that improved performance was gained compared to the standard Rainbow algorithm on Atari-2600 games, specifically Subnautica. An interesting observation from this paper found in testing is that the algorithm performs best when the game in question has multiple mechanisms for outputting rewards, such as shooting sharks or saving divers in Subnautica. This reflects the idea that the global reward function can be described as a sum of reward functions, and in testing, it was found that each sub-reward often matched a specific kind of reward outputted by the game.

The Horde architecture and similar proposals all essentially relate to building neural networks that are composed of many highly connected heads that have sparse connections, with potentially a very high number of heads present. This may differ from the approach being investigated in this project which will likely involve completely independent agents who may be trained with entirely different algorithms depending on the problem they are trying to solve, although approaches similar to these may also be investigated.

A key concept investigated by several decomposition approaches such as CCEA [29] is the idea of separability of a problem which is defined in the CCEA paper as being able to write a function dependent on N variables as the sum of M functions that are each dependent on less than N variables. This means each decomposed function is dependent on only part of the state which is not a requirement in this project. Sub-agents may have access to the entirety of the state and if multiple agents are controlling the same action the combination of their outputs is an important factor. If the actions are linearly combined the function being approximated must be separable as a linear sum, thus using a none linear aggregator may provide more flexibility.

9.2 W-Learning

W-Learning is an extension of standard Q-Learning RL described in 2000 by Mark Humphrys [16]. The original article describes an architecture in which multiple agents compete for control of a mobile robot in order to maximise their own cumulative rewards. Within this architecture, each agent given a state x suggests an action a with a weight w according to a table it maintains similar to a Q table.

In order to output this w value each agent maintains two tables, a Q that is updated according to the standard Bellman equation used in Q-Learning whenever the agent's selected action is used and a W table which is updated when the agent is not listened to. Each agent has a separate reward signal R_i which is used to update its respective tables.

The goal of the W table is to store the difference between the predicted reward if an agent is listened to and the actual reward received if it is not listened to, $\mathscr{W} = \mathscr{P} - \mathscr{A}$.

To this end equations can be constructed for updating the Q tables and W tables of each agent given an action chosen by agent A_k, these equations are described in equation 11. It should be noted that W_k is not updated as if it is, then the leader's W value will converge to 0 for this state whilst the other agents will converge to a W value greater than zero, thus there will never be a stable leader for a given state REF.

$$Q_i(x_n,a_k) = (1-\alpha_q)Q_i(x_n,a_k) + \alpha_q(r_i + \gamma \max b \in AQ_i(x_{n+1},b) \tag{11a}$$

$$W_i(x_n) = (1-\alpha_w)W_i(x_n) + \alpha_w(Q_i(x_n,a_k) - (r_i + \gamma \max b \in AQ_i(x_{n+1},b)) \tag{11b}$$

Part III
Basic Algorithm Implementation and Infrastructure

10 General Implementation Notes

The entirety of this project was implemented in Python due to the high availability of Machine Learning libraries. The games discussed below were also all implemented in Python using numpy, and although some performance loss is to be expected from using Python for this task the inference time of the models used is so much greater than the time to calculate a game frame that game efficiency becomes irrelevant.

The low-level library used to implement the models described in this project was TensorFlow[37], with the Keras library also used on top of TensorFlow in some cases to describe the neural networks required. All networks implemented are simple fully connected neural networks as the focus of this project is on the training algorithms and not on the network design. This is discussed further in section 14.

10.0.0.1 Testing Configs A system to store model configs and test hyper-parameters was also implemented in order to ease the testing procedure used. This system provided a Python Base class that could be used to define config classes with easy load and save methods for storing settings in JSON files. This system also allowed for all training tests to be restarted at any point as all the necessary settings were saved in a standard format. The links to all the code can be found in Appendix C.

11 Game Design

A key focus of this project's experimentation is the deployment of environments that served as testing platforms for various training algorithms. A prominent benchmark in numerous Reinforcement Learning (RL) studies, the Open-AI gym set [2], features various Atari2600 games readily adaptable for Python scripting. However, this project opted for custom environments during its implementation phase due to several distinct advantages offered.

There were two primary reasons for utilising custom environments. Firstly, it granted the freedom to engineer scenarios tailored with specific optimum strategies. This customisation made it simpler to quantify the proficiency of diverse algorithms in learning strategies with different time scale projections.

The second benefit was the absolute control these custom environments provided over the test environment's implementation. This control facilitated easy data collection of any necessary metrics and statistics, proving crucial when deciphering the behaviour of different models.

11.0.0.1 Modularity and Interfaces The project's environment design mandated a highly modular structure. This modular approach decoupled game logic from user interfaces (UI) and user input, while also separating the environment from the algorithm and model implementations. The game's interaction requirements with the UI and the models were as follows:

- Processing a set of actions to facilitate the transition of the game from one state to another and provide a reward output for the state change.
- Provide a means to receive the game state in a numpy format that can be used as input to a model
- Supplying a method to ascertain the shape of data conveyed to the model, a requirement for parameterising model inputs.
- Delivering an output of the game state that can be visualised by a UI library.

In order to simplify the learning difficulty of the games a decision was made to always output positional game state relative to the player's position, this point will be made clearer when describing specific game implementations but simply removes the need for the model to calculate where points in space are relative

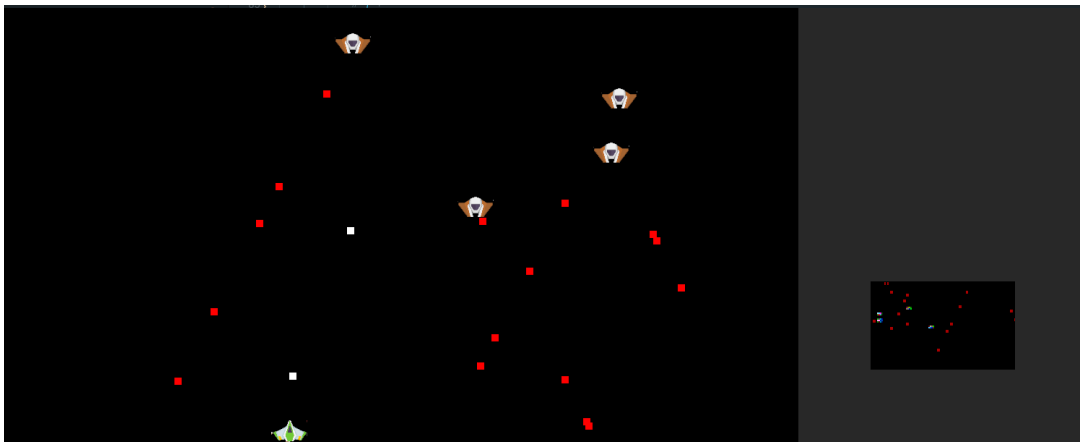

Figure 1: Space Invaders game, player is the green character at the bottom of the screen, debug information on the right

to the player. This decision, as well as a desire to lower the dimensionality of the model inputs, resulted in the need for two output forms of the game state, one for the model and one for the UI library as opposed to a single two-dimensional output corresponding to pixels.

The game provides the UI component with an output consisting of rectangles filled with colours or images. This universal format simplified the implementation of the UI component with any UI library, for example, in this project the PyGame 2D graphics library was utilised. The UI component functioned by either taking actions from user keyboard inputs or querying a model using the game's output.

11.0.0.2 Debugging Tools Debugging the behaviour of the models researched throughout this report presented a difficult challenge, and often visual indications of model outputs proved more useful than simply printing streams of numerical data to the console. To this end as well as providing the game's UI state to the UI component, a generic interface for debug information was also provided.

11.1 Space Invaders Clone

The first game developed was a Space Invaders Clone inspired by the 1978 Taito Game [36]. The goal of this game was to provide clear strategies that operated over three different time periods, these strategies are as follows.

- Short Term: Basic avoidance of nearby bullets
- Medium Term: Avoidance of enemies, and position relative to enemies to all for successful shots to be made.
- Long Term: Utilising shields to avoid enemies. Since enemies shoot more often when above the player the best strategy is to avoid shields early on to save them till later stages of the game.

11.1.0.1 Game Description Figure 1 illustrates a frame from the game UI. The player, located at the bottom, has the ability to move horizontally, both left and right. The screen is designed such that moving off the left side wraps around to the right, enabling continuous motion.

The enemy characters either move continuously left or right and fire shots at a frequency determined by their proximity to the player. While the player can also fire shots to defeat these enemies, the shooting ability of the player is subject to a cool-down period. Consequently, the timing of shots is critical, as given the horizontal motion of enemies by the time a shot reaches their height they will have moved out of the way.

11.1.0.2 Model Input The positions of each type of game object were stored in a 2D numpy matrix in order to create model inputs. A matrix was constructed for both enemies, shots and shields with all positions in the matrices relative to the player's position. The use of a wrap-around screen in the game made

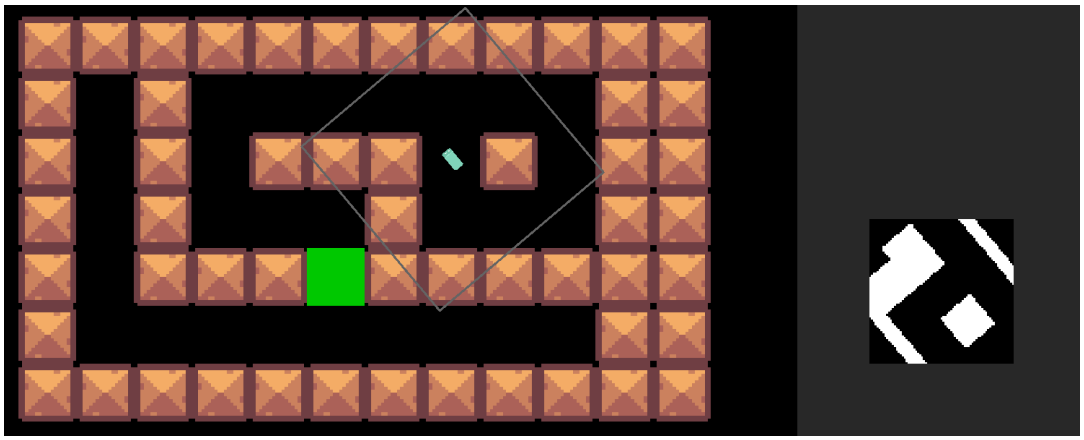

Figure 2: Driving Maze Game, the blue character represents the player with the green square representing the goal position. The model input visualisation is present on the right.

the implementation of this relative position more intuitive. These different matrices were then flattened, appended and outputted to be provided as model inputs. The previous two states were also appended to this array to give the model a sense of temporal information such as velocities.

The reward outputted by this game is simply a positive reward for each frame survived and a large negative reward on the frame the game finishes. The reasoning for this is to not encourage the learning of different strategies through reward shaping and rather allow the model to discover these strategies naturally.

11.2 Driving Maze Game

The second game designed in this project is a basic driving game, incorporating a randomly generated maze as the map. The objective of the game is for the player to reach the finish square as quickly as possible while avoiding any collisions with the maze walls. Figure 2 depicts the game UI for reference.

Constructing a reward system for this game proved to be more complex, primarily because accomplishing the game is relatively challenging. Merely providing a reward upon reaching the final square may not promote successful training. Therefore, the reward system was designed to acknowledge the distance between the player and the finish square at the end of the game, providing an incentive for progression. Additionally, if the player reaches the finish square, a bonus reward is given based on the speed of completion. Each frame of the game outputted a small positive reward if the player moved towards the goal and a small negative reward if the player moved away or stood still. Finally, if the player collided with a wall a large negative reward was given that was further decreased dependent on the time spent alive. This gives an overall fairly complex reward structure that is designed to be difficult for a model to learn in order to stress the algorithms implemented as much as possible.

11.2.0.1 Maze Algorithm The use of a maze for a map is a simple solution to ensuring the random generation of maps, preventing models from simply learning set sequences of actions. This game isn't supposed to be a maze-solving problem, however, the maze nature does allow room for testing the use of memory in model tests, as this may be a requirement for the model to successfully solve the maze.

The algorithm used is a variation of the Depth First Search algorithm with the full implementation steps presented in the appendix Appendix A.

11.2.0.2 Model Input In the case of the driving game, the model input is again provided relative to the player. The input to the model can be seen in figure 2, with the square present on the right giving a visualisation of this input. The input is constructed of a low-resolution grid that is moved and rotated with the player such that each input node provides a consistent transform relative to the player. In addition to this flattened square matrix, the distance from the player to the goal position and the angle between the player's forward direction and the vector between the player and the goal square is also given.

12 Basic Algorithms Implementation

12.1 PPO

As mentioned in the background, the Proximal Policy Optimisation (PPO) algorithm, developed by OpenAI [32], represents a state-of-the-art policy optimisation reinforcement learning (RL) method. In this report, PPO is implemented as a starting point for testing policy-based methodologies.

Typically, the implementation of PPO involves utilising the actor-critic method [32]. This approach incorporates a critic network that estimates the value function of a state, which is then used to estimate the advantage function. The OpenAI algorithm provides a detailed outline of how to construct a loss function based on the probability of an action being taken, as well as the updated probability of an action. These values, along with the advantage estimate, are used to formulate the loss function.

To optimise the Policy Networks, an optimiser such as Adam can be employed to compute the gradients. These gradients are then utilised to adjust the networks, aiming to minimise the loss function.

In order to update the policy based on a set of observations, it is essential to first estimate the advantage of taking a specific action instead of the action dictated by the current policy. As explained in the background section, this advantage can be calculated by finding the difference between the Q-value of a particular state-action pair and the value of the state, as illustrated in equation 12.

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(S) \tag{12a}$$

Intuitively, if the policy π is optimal, the advantage of deviating from the policy would be negative. However, until the optimal policy is achieved, the advantage serves as an indication of how much better or worse an action performs in comparison to the current policy.

In order to make as estimation of this advantage we need an estimation of the Q function for the state action pair as we have the value function estimate given by the critic. There are many ways to do this as described by Schulman et al [24], some of which will be investigated further in later sections. However, for now, a basic approach is employed, which involves considering the cumulative discounted reward (*CDR*) obtained from a state as the Q value for that particular state [24]. This estimation necessitates playing a complete game and summing up the rewards from a given frame, discounting the sum by a given discount factor to favour reward received closer to the current point in time, as depicted in equation 13.

$$CDR_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{13a}$$

The formula shown in equation 14 demonstrates how the advantage of an action given a state is estimated from the difference between the cumulative discount reward and the value of the observation. The PPO paper also specifies a necessity to normalise the advantages across a batch which improves learning performance.

$$A(s_t|a) = CDR_t - V(s_t) \tag{14a}$$

Equation 15 is employed in Proximal Policy Optimisation (PPO) to compute the Policy Network Loss. It incorporates the minimum of two surrogate losses for each observation, followed by the calculation of the mean across all observations.

The function $g(\varepsilon, A)$ is utilised to restrict the magnitude of changes, thereby preventing excessively large updates in either direction. The parameter ε acts as a tuning factor that regulates the amount of regulari-

sation, discouraging significant updates.

$$L(s,a,\theta_k,\theta) = \min\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\varepsilon, 1+\varepsilon\right)A^{\pi_{\theta_k}}(s,a)\right] \tag{15a}$$

$$L(s,a,\theta_k,\theta) = \min\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), g(\varepsilon,A^{\pi_{\theta_k}}(s,a))\right] \tag{15b}$$

$$g(\varepsilon,A) = \begin{cases}(1+\varepsilon)A & \text{if } A \geq 0,\\ (1-\varepsilon)A & \text{if} A < 0.\end{cases} \tag{15c}$$

In many cases, an extra regulariser is included in the Proximal Policy Optimisation (PPO) loss function to promote exploration by the agent as shown by Z. Ahmed et al [1]. This additional regulariser aims to encourage the policy's probability distributions to have a more balanced spread, and a reliable metric for assessing this balance is the entropy of the distribution [1]. Eqn 16 gives the formula for the entropy of a discrete probability distribution. By scaling and incorporating this entropy value into the overall loss, the degree of exploration exhibited by the agent during training can be adjusted accordingly.

$$X = -\sum_{i=1}^{n} p_i \log(p_i) \tag{16a}$$

In addition to calculating the loss of the policy, it is also necessary to calculate the loss of the critic network which is a significantly easier task. The value of an observation given the current policy is known as it is simply the cumulative discount rewards received from that observation by definition, leading to the loss described in equation 17, which is the MSE of the difference between the output and the expected output. This use of MSE is described in the PPO paper[32].

$$L_{\text{critic}} = E_t\left[(V_\theta(s_t) - G_t)^2\right] \tag{17a}$$

12.1.1 PPO Implementation

The equations given above provide the bases for implementing PPO in code using the TensorFlow Python library which allows for a low-level description of operation on large amounts of data. A key requirement throughout this implementation is to make heavy use of TensorFlow's operation vectorisation, whereby operations can be carried out element-wise on a vector of values. If this vectorisation is not strictly adhered to then large performance penalties can be introduced when iterating through vectors in Python. The flow diagram given in figure 3 gives a high-level overview of how the algorithm has been implemented.

12.2 Basic Model Design

As described earlier, the initial implementation of the games that the models are trained on output a flat vector of data as input to the models, negating the use of constructs such as convolutions. This implementation decision maintains greater simplicity of game and model design while maintaining the focus on the training algorithms and not the model implementation. More complex model designs are, however, investigated in later sections. For the PPO algorithm, two networks are required, one for the policy and one for the critic with both implemented here using standard fully connected dense neural networks. The policy network outputs a probability distribution using "softmax" activation in the final layer that gives the probability of taking each action combination, meaning four possible combinations are required for two binary action choices. The critic layer outputs a single value with no final activation.

12.3 Parallelisation of Data Collection

The TensorFlow backend has been extremely well optimised to take advantage of both parallelisations of operation across multiple CPUs or on a GPU and allows for the speed-up of training through added

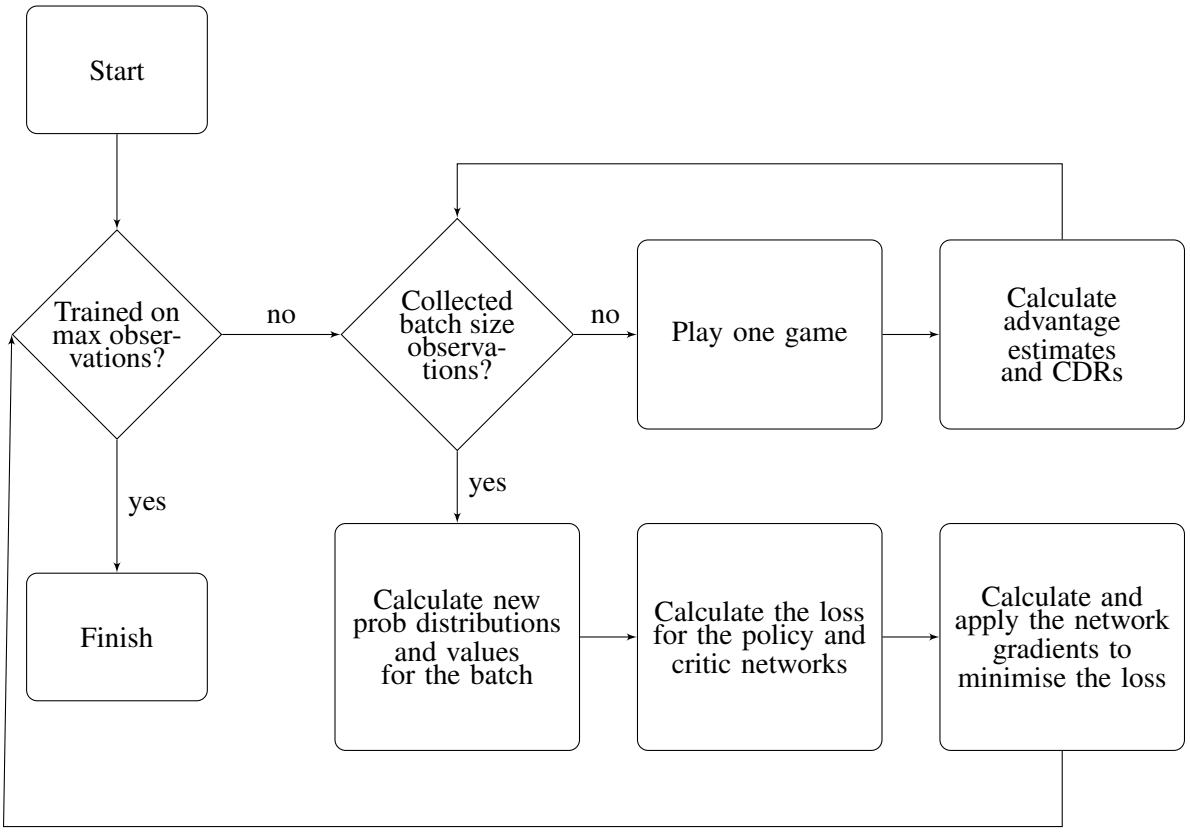

Figure 3: Flow diagram illustrating PPO implementation

hardware up to a certain point. However, eventually, operations can't be parallelised any further; for instance, playing a game requires the computation of the previous frame, and so to increase training times, it was necessary to collect training data from multiple games played simultaneously across multiple CPU cores. Unfortunately, due to the infamous Python GIL splitting workloads across cores is not trivial. Python does provide a multiprocessing library which allows for a function to be run by another thread but is relatively strict on what kinds of data can be transferred between these threads.

To overcome this issue, a worker queue system was devised that uses several workers instantiated at startup that receives messages to start collecting training data on a task queue, load the current saved model and create a batch of training data. This batch is then put onto an output queue where it can be received by an aggregator which combines the batches from each agent into a single batch which can be trained over. At this stage, TensorFlow can be relied on to parallelise training across many samples using multiple cores. Once training is complete, a message is sent to the task queue for each worker to repeat the process.

12.4 DDQN

As described in the background Double Deep Q-Learning (DDQN) is an algorithm that implements the Deep Q-Learning technique, providing performance close to the best Q-Learning based approached whilst not being overly complex to implement. Similar to using PPO as a policy-based starting point DDQN is used here as a starting point for Q-Learning and more generally off-policy-based algorithms.

Algorithm 3 gives an interpretation of the steps described by Hasselt et al 2015[12] to implement DDQN.

The use of this second network was shown by Hasselt et al 2015 [12] to decrease the overestimation of the Q function of a state action value by a significant proportion. Hasselt et al give an example demonstrating the reason for this bias, whereby if the true value of all actions for a state is 0 and the current network randomly distributes the values it outputs about 0 for different actions, the maximum of these random values is used to update the Q value according to the bellman equation[12], meaning a systematic overestimation is introduced. The reason adding a second network reduces this overestimation is due to

Algorithm 3 Double Deep Q-Learning

Initialise primary network Q_θ and target network $Q_{\theta'}$
for iteration $= 1, 2, \ldots.$ **do**
  for environment step $t = 1, 2, \ldots.$ **do**
    Select $a_t = \max_a Q_\theta(a, s_t)$
    Execute a_t and receive reward r_t and observation s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathscr{D}$
    If game complete restart
  end for
  for update $= 1, 2, \ldots.$ **do**
    Sample observation from $\mathscr{D}$
    Compute the target Q value:
    $Q^*_\theta(s_t, a_t) = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_a Q_\theta(s_{t+1}, a))$
    Calculate MSE, $Q^*_\theta(s_t, a_t) - Q_\theta(s_t, a_t)$ and perform gradient descent
    If performed refresh iterations update target network to match current network
    $\theta' \leftarrow \tau\theta' + (1-\tau)\theta$
  end for
end for

the disentanglement of action selection from action evaluation, with the target values more stable during training due to less frequent updates. An additional benefit is that the network also over-fits less to recent experiences due to the infrequent update of the target network [12].

Unlike in PPO DDQN does not provide a means to encourage exploration through regularisation of the model so instead an epsilon greedy strategy is often used [12] whereby during training an action is chosen either randomly or using the model according to a parameter ε. The parameter ε dictates the proportion of actions that are randomly chosen to be selected by the model or generated themselves randomly. This parameter is decreased linearly as a specified rate across training steps till it reaches a minimum.

Figure 30 in the appendix gives an overview of the DDQN implementation used, although the training process is parallelised in a similar manner to that described for PPO so is slightly more complex than the description given. The major difference with the DDQN training is that any previous observation can be used for training due to the fact that the update algorithm only depends on the reward between two observations and not on the future reward according to the current policy. This difference also means that the final frame must be treated slightly differently due to there being no next state. In the case of the final frame of a game, the Q-Value is just the reward received for the frame, this fact also necessitates storing whether a frame is a final frame when adding to the pool $\mathscr{D}$.

12.4.0.1 N-Step Returns N-Step returns is a common approach to balancing the trade-off between variance and bias of the value estimates made in RL-Algorithms [13]. In this case, it means instead of updating based on the reward between state S_t and S_{t+1} we consider the change between S_t and S_{t+n}. This results in the following change to the update method described in equation 18. Hernandez-Garcia and Sutton [13] demonstrated that for small n an off policy corrections are not necessary to account for the fact that some actions are chosen randomly and so do not follow the current policy when using this method.

$$R_t + \gamma R_{t+1} + \gamma^2 \gamma R_{t+2} \ldots + \gamma^n \gamma R_{t+n} + \max_a Q(S_{t+n+1}, a) \tag{18a}$$

12.4.0.2 Value Learning When considering the Space Invaders Game, here the survival of the agent in each frame is rewarded with a minor positive value, while its demise results in a significant negative value. At the outset of the model's training, when the estimates are arbitrary, all frames, excluding the final one, will produce a random Q-Value increased by a consistent value. Hence, in the initial stages, the model's training is primarily based on frames that are adjacent to the agent's death. This is because the substantial negative value gradually propagates through the estimates during the training. This introduces a need to both ensure the negative reward received is appropriately tuned and potentially optimise the frames used for training early on, both of which are investigated in the analysis and testing chapter.

Part IV
Multi Agent Algorithms

13 Multi Agent System Design

As discussed in the background, many different approaches are available when attempting to deconstruct an RL agent to a sub-agent system, with this problem comprising three main challenges.

The breakdown of a Reinforcement Learning (RL) agent into a network of sub-agents presents a complicated problem, inherently encompassing three core challenges:

- Partitioning the reward signal
- Constructing policies to optimise each reward signal
- Merging sub-agent policies to maximise the collective reward

The choice of strategy for each of these sub-tasks inherently impacts the feasible solutions for the others. For instance, the nature of reward decomposition significantly determines the suitability of on-policy or off-policy methods, and the policy structure, in turn, prescribes the possible methods of policy integration. This project considers a variety of approaches to this trio of challenges, drawing from established research as well as novel methodologies.

The principal methods examined are outlined below:

- Off-Policy
  - Multi-Agent Q-Learning
  - W-Learning
- On-Policy
  - Multi-Agent PPO
    - Arithmetic Policy Integration
    - Smart Policy Integration
  - Voting-based Multi-Agent PPO

13.1 Distributed Reward Decomposition

The core methodology used throughout this report to decompose reward signals is based on the work presented in the Distributed Reward Decomposition [18] system previously described. This system is necessary to overcome the challenge of removing the need to deconstruct reward signals by hand, which most architectures such as W-Learning [**“nobreakspace –˝“cite –HumphrysW-learning:Mind˝**] rely on. The core assumption made by the paper is that the global reward signal is composed of the sum of all sub-reward signals, as shown in equation 19, resulting in the main limitation of the kind of rewards that can be described through this mechanism.

$$r = \sum_{i=1}^{N} r_i \tag{19a}$$

The system given in this paper is a modification of a principal described by Grimm and Singh 2019 [10] which demonstrates how to ensure Independence between each reward signal. Given sub-value functions V_i^{π} according to a policy π the paper describes two additional values shown in equation 20.

$$J_{\text{independent}}(r_1, ...r_n) = E_s\ \mu \left[\sum_{i \neq j} V_i(s) \right] \tag{20a}$$

$$J_{\text{nontrivial}}(r_1, ...r_n) = E_s\ \mu \left[\sum_{i=1}^{n} V_i(s) \right] \tag{20b}$$

Grimm and Singh proposed that the value $J_{\text{nontrivial}} - J_{\text{independent}}$ should be maximised to achieve the reward decomposition, as this ensures that the amount of r_i obtained by policy π_j is minimised.

In this project, methods that don't rely on distributional Q-Learning are investigated, which negates the ability to decouple policies using the work done by Grimm and Singh. This project uses the concept that the value function and the policy are coupled, and if policies differ through exploration, the value functions should naturally diverge through this coupling.

13.2 Critic Based Reward Decomposition

As an alternative approach to decompose rewards that are more decoupled from the policy implementations proposed by this project is the use of a critic network as an estimator of the current value function. The core principle behind this is based on the same assumption as the Distributed Reward Decomposition paper [18]. Whereby sub-reward signals are assumed to sum to give the global reward signal, as well as the fact that $R = V - \gamma V'$ by the definition of the value function as the expected discount return from a state. Multiple different methods for encouraging the decoupling of these reward signals are investigated that do not influence the value function itself but rather the policies trained on it.

To split the value function into M sub-value functions, a neural network is used with a final layer that contains M outputs. This layer is then trained such that the value estimate is taken as the sum of the outputs so the same MSE method as in PPO can be used as a loss function. It is hoped that in order to give a good estimate of a complex value function, these signals will diverge naturally, but by adding different regularisers to the agents training to maximise them, it is hoped that their divergence can be further maximised, unless otherwise specified all references to a "sub-critic value" made in this report refer to the corresponding output in this neural network layer. Equation 21 gives the formula for the sub-critic values.

$$\sum_{j=t}^{\infty} \gamma^{j-t} r_j = \sum_{i=1}^{M} V_i \tag{21a}$$

$$V_i = \text{ith Sub-Critic Value} \tag{21b}$$

13.3 Multi Agent Q-Learning

In order to test Q-Learning and W-Learning-based architectures, it is necessary to have a reward signal and not just the value function, and here the $R_i = V_i - \gamma V_i'$ property is used as a naive approach. Although this identity describing the reward is fundamentally true, the problem is both that the value function used is not the true value function but an inaccurate estimate and that the agent isn't always following the policy the value function is true for due to exploration. These two facts mean that if an agent takes an action that causes the value function to rise, the reward received will be negative, whereas in this case, we would want to reward the agent. The negative change would never be possible for a true value function that describes a game that always outputs positive rewards but is a consequence of the problems described. For this reason, the behaviour expected in trying this method is not obvious as the reward and policy are so entangled, leading to potential feedback loops.

13.3.1 Q-Aggregator

A Q-Learning Algorithm based on sub-rewards that sum to global rewards was described by Chen et al [4], and is fundamentally the same as the standard Q-Learning algorithm, except with multiple Q-Functions whose outputs are summed to give the final return estimates. In this case, a similar implementation is used as the DDQN algorithm except with modifications to the code to parameterise the number of agents present. The other modification is the addition of the critic described previously, which is trained in the same manner as the PPO critic.

The first method is to use the sub-value estimates as described above (that form the final layer of the critic network) as the reward estimates; however, as expected, this resulted in training that was extremely unstable and was not able to learn any meaningful strategy in the Space Invaders game.

If we consider briefly what the cause of this instability is, we can take the example where the model tries an action that differs from the policy that ends up being beneficial. This will cause the value function to increase (if it is trained well enough to recognise this change). The model will then be updated to be less likely to take this action, and over time the value function will give the original state a lower value due to the less likely hood of taking the better action. Thus, this leads to an unstable situation, although much further analysis is needed to rigorously prove this.

A natural alternative to this method may be to invert the operation and take the difference between the next state value and the current one. In this case, we are no longer considering the true reward but rather, in a way, improvements in the policy as reward. In the case of convergence of training and no exploration, assuming a true value function, the output would be $V_{i+1} - V_i = V_{i+1} - r_i - \gamma V_{i+1} = -r_i - (1-\gamma)V_{i+1}$.

For the Space Invaders game, the reward for surviving a frame is 0.2, and for dying is -10; for most game lengths of around 250 frames, this gives value estimates of around 40. Taking then a relatively high gamma of 0.995, we can see that the $(1-\gamma)V_{i+1}$ term is pretty similar in magnitude to the reward signal (if we assume a V_i of 20 we get $[1-\gamma]V_{i+1} = 0.1$). The problem then is that we may have an overall negative value as the reward, and so to maximise the future discount reward, the agent is encouraged to end the game as soon as possible, demonstrating why this method could be predicted to be unstable.

These two examples outline the fundamental problem with this approach: either we discourage improvements to the policy or destabilise the final policy we want to converge to.

A second approach that removes these issues is to take the actual reward received and scale it proportionally according to the critic values, ensuring a total reward that is always positive. There are two ways this could be done: scaling by the proportional current sub-critic value or by the proportional change in sub-critic value. It was found that scaling with the current critic values resulted in more stable performance providing the final implementation of this algorithm. The reward used for each agent is described in equation 22, and the critic is trained in an identical manner to that of the PPO implementation.

$$R_i(x) = \frac{V_i(x)}{\sum_{j=1}^{A} V_j(x)} R(x) \tag{22a}$$

$$A = \text{Number of agents} \tag{22b}$$

$$V_i(x) = \text{ith output of the critic} \tag{22c}$$

A more in-depth investigation of the performance of this method is presented in the testing and analysis chapter.

13.3.2 W-Learning

To confirm the assumptions made about the instability of the previous method and attempt an improved approach to Q-Aggregation, the W-Learning algorithm explored in the background was implemented using both methods described of reward splitting.

13.3.2.1 Critic Based W-Learning In order to stabilise the W-Learning training, the N-Step learning returns is used as described in the DDQN implementation. Giving a modified Bellman update as in

equation 23.

$$Q(a_t,x_t) := (1-\alpha)Q(a_t,x_t) + \alpha(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} ... \gamma^n r_{t+n} + \gamma^{n+1} \max_{a \in A} Q(a, x_{t+n+1})) \tag{23a}$$

In this case, in a similar vein to how DDQN uses a second network for evaluation, it may be possible to use the critic value here as the $\max_{a \in A} Q(a, x_{t+n+1})$, since this is an estimate of the value of the state. Again we can weigh the reward received as proportional to the critic value. Since we are taking multiple rewards in the future into consideration, this proportional weighting could be expected to be more accurate than when done with a single time step; this also removes the need for having a second target network in addition to a critic network.

The W -Value can be updated in a similar manner to the Q-Values using the future state value, with the W update only applied to the agents that were not used in that time step. The W target value is described in equation 24, along with the modified Q-Update using the sub-critic value. The two equations described are used as the target outputs for the neural networks implementing the models, such that the update does not include a weighting of the previous value as in the background equation 11 as the learning rate essentially implements this weighting.

$$W_i(x_t) := Q_i(a_t,x_t) - (r_{t_i} + \gamma r_{t+1_i} ... + \gamma^n V_i(x_{t+n})) \tag{24a}$$

$$Q_i(a_t,x_t) := (1-\alpha)Q_i(a_t,x_t) + \alpha(r_{t_i} + \gamma r_{t+1_i} + \gamma^2 r_{t+2_i} ... \gamma^n r_{t+n_i} + \gamma^{n+1} \max_{a \in A} V_i(x)) \tag{24b}$$

This algorithm resulted in training that was of similar stability to that found with DDQN, and it appears that the critic estimates are coupled enough to agent policies to be used in the evaluation step of the updates. However, the performance was less than policy-based algorithms. Analysis of multi-agent performance and reward signal decoupling is given in the analysis and testing chapter. The algorithm was confirmed to be stable with a single agent, with the use of the critic instead of a second target network giving performance similar to that of DDQN (which is outlined in Part V).

13.4 Multi Agent PPO

The latest advancements in Reinforcement Learning (RL) algorithms, particularly those centred around policy optimisation, have surpassed the performance of DQN-based algorithms in the majority of standard benchmarks. As a result, the focus of this project has been heavily tilted towards the exploration of policy-based deconstruction. The algorithms that have been experimented with are all variations of the PPO actor-critic model, which are based on the same sub-reward summation assumption utilised in DQN methods.

13.4.1 Architecture

In order to make estimates of the advantage function, the same sub-critic values described earlier are used such that the sum of critic outputs is trained to estimate the future discount cumulative reward received in a game. The big problem here is how to weigh the cumulative discount reward received, with the method decided being to just weigh the CDR (cumulative discount reward) according to the critic values. This assumption might not be entirely fair as it may not accurately describe which sub-agent is actually responsible for the change in advantage; unfortunately, there is no other obvious manner to go about this available. This gives a sub-agent advantage function described by equation 25.

$$A_i(x) = \frac{CDR(x)V_i(x)}{\sum_j V_j(x)} - V_i(x) \tag{25a}$$

13.4.2 Summation vs Product

Given the sub-agent advantage functions, we can train multiple policy models that are identical to those used in the PPO implementation; however, this leaves a problem of how to combine these different distributions. The most basic implementation of this is the combine the outputted probability distributions using some fixed function, specifically, in this case, summation and product.

13.4.2.1 Summation When we perform an element-wise summation of the sub-policy distributions, we can anticipate various resulting behaviours. This approach allows each sub-agent to contribute to the decision-making process of the main agent. Particularly when the reward signals have not significantly diverged, we would expect the system to learn to play the game in a manner similar to PPO. However, as the game progresses, there might be instances where one sub-agent, having learned to handle the current situation, needs to take over the system.

Nevertheless, suppose the other sub-agents are uncertain about their next move and produce evenly distributed probability values. In that case, a single sub-agent can only influence the probability distribution to a certain degree. For instance, consider two sub-agents deciding whether to move left or right. If the first sub-agent is certain about moving left and outputs a probability distribution of $[1,0]$, and the second sub-agent is uncertain, outputting $[0.5,0.5]$, the resulting normalised distribution is $[0.666,0.333]$. Thus, the first sub-agent cannot ensure moving left, leading to unpredictable behaviour.

Figure 4 shows the training outcome for a system with three agents, and as can be the results are consistent with the outcomes described. At first, the model is able to learn to play the game reasonably well, but eventually, as the reward signals diverge, the behaviour becomes erratic, resulting in a collapse of scores.

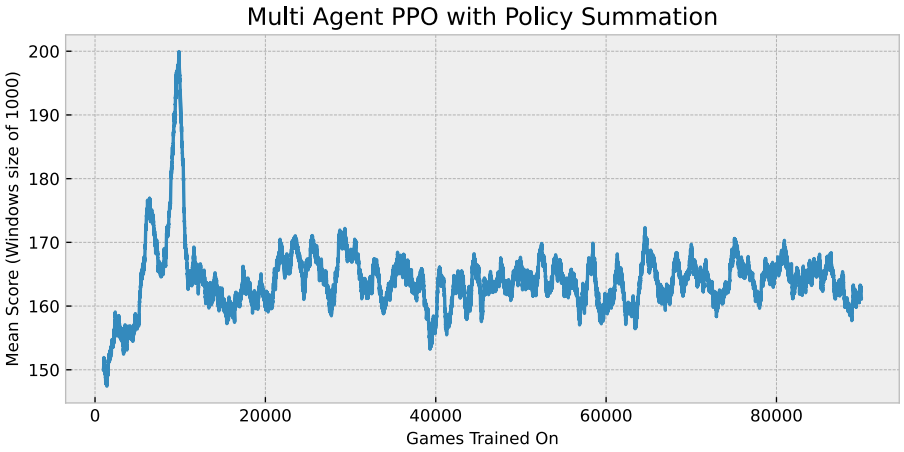

Figure 4: Average game score during training with three agent PPO using policy summation

13.4.2.2 Product An alternative implementation that overcomes the issues described with summation is simply to take the element-wise product of the sub-action probability distributions. In this way, if one agent outputs $[1,0]$ and another $[0.5,0.5]$ as in the previous example, the resultant normalised distribution is $[1,0]$, illustrating how a single agent can still take control. The problem with this method is if the second agent in the above example, who should not be in control, outputs $[0,1]$, the resultant distribution is $[0.5,0.5]$. This demonstrates how as the agents become more confident in their outputs and output less evenly distributed probabilities, the system can still become unstable.

Figure 5 shows the outcome of training three agents using this method, and as can be seen, the results are far more successful than using summation; however, eventually, the average scores do decrease. More in-depth analysis of the performance of this method and the divergence of reward signals is given in Part V.

13.4.2.3 Cyclic Training Initially, when testing this method, unstable behaviour more similar to the summation method was observed; however, this was eventually shown to be due to the effect of training

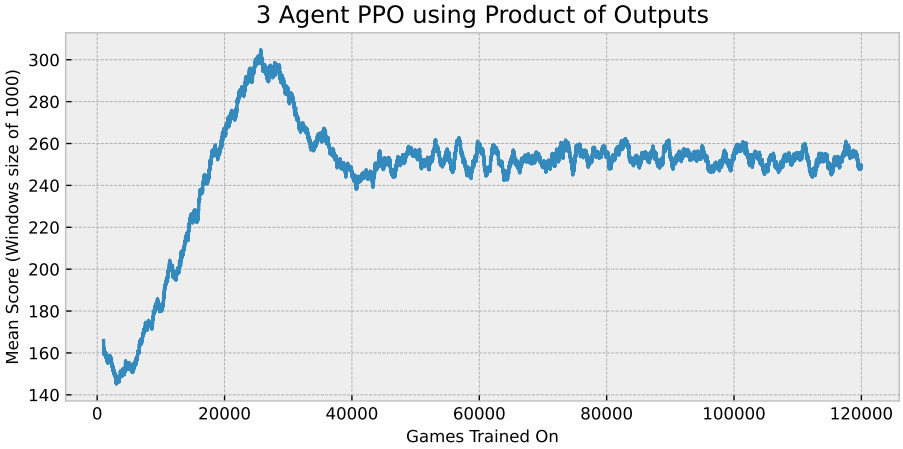

Figure 5: Average game score during training with three agent PPO using policy product

all agents every time step. As described in the background, when modifying multiple agents' policies, the Markov Property of a system is no longer present as the environment itself is changing with respect to each agent. It was found that when cyclically training agents, such that each agent was trained for N games before rotating to the next, performance was improved significantly to give that presented here.

13.4.3 Intelligent Switch

Based on work done by Chen Chunlin et al. [4], another agent was integrated to serve as a switch mechanism, determining the policy to apply among multiple agents. In this system, the outputs of each agent policy are concatenated and passed as the input to the switch agent, which has a custom first layer that performs multiplications of inputs, similar to taking the element-wise product as before. This custom layer ensures the switch can make use of the product of distributions and not just linear combinations. This agent is then trained in the same cyclic manner as the others using the PPO algorithm.

Unfortunately, this design results in the following dilemma. If the switch solely takes in the outputs from the sub-agents, it's probable that it will not possess sufficient information to make accurate decisions. For example, if two agents generate the same distribution, either could be the appropriate agent to heed, contingent on the current state. However, should we provide the current state to the switch, it is likely to merely duplicate the computations performed by the sub-agents in determining the correct sub-agent. When training the sub-agents, the action probability distribution outputted by the switch is used to train the sub-agents, as this is the distribution that was used to

This method results in erratic unsuccessful training due to the switch being unable to learn the correct agent output.

13.5 Multi Agent PPO With Voting

In order to overcome the challenges faced with integrating sub-agents, a system similar to the concept behind W-Learning was devised, whereby each agent outputs a vote along with its action probability distribution, indicating how much it wants to be listened to. The key requirement here is that agents should be encouraged to only give large votes if necessary for survival.

Unlike the models discussed thus far, the sub-agents in this algorithm were constructed to output a separate action distribution for each action instead of a single distribution giving each combination of actions. Since all actions in the games built for this project are binary (i.e. left/right or shoot/not shoot), the distributions for each agent are a two-value layer outputting log probabilities. This means that the total action output is $2A$, where A is the number of actions (although since the outputs give binary probabilities, we can reduce this to A outputs by the condition that the exponent of each pair must sum to 1).

In addition to a binary output for each action, the agents also needed to output a continuous value for the vote. As discussed earlier, this is one of the advantages of PPO vs DDQN, in that continuous actions

are possible to output as the PPO algorithm simply requires the probability of an action being outputted in order to create model gradients. Therefore the output of the agent relating to the vote needs to parameterise a probability density function that can be sampled to generate votes. For simplicity, the PDF used is that of a normal distribution; therefore, the vote requires two outputs describing the mean and standard deviation of the vote distribution. Thus each agent requires $4A$ outputs in total.

In order to incentivise each sub-agent to take control for as short a time as possible, the reward function needs to be manipulated to disadvantage higher votes. One solution to this problem is to subtract the vote of each agent from the reward received, where the reward system is the same as that used in the previous PPO-based algorithms described. Since the CDR is scaled by the sub-critic values when calculating advantages, as in the previous examples, if we scale the CDR after adding votes, we will scale the votes unintentionally. For this reason, we calculated the cumulative discount sum of votes and the scaled CDR separately and then summed those to give the final CDR as described in equation 26.

$$CDR_i = \frac{CDR * V_i}{\sum_i V_i} - \sum_{n}^{\text{num frames}} \left(\gamma^n \sum_{a}^{\text{num actions}} \text{vote}_{i,a} \right) \tag{26a}$$

If the system were to be implemented with the above equation and the same method for updating the critic, unstable behaviour would be observed for several reasons. The first is that the critic is trained to predict the CDR, not the vote-adjusted CDR. So if the system could achieve convergence, the advantage calculated for an action with a non-zero vote would still be negative, not zero, as it would include the subtracted votes, demonstrating that a stable convergence with none zero votes may not be possible.

Training was unsuccessful when the system was tested using the CDR with votes subtracted as the critic estimate. This is a consequence of the fact that the policy outputs should not be trained on advantages that are modified with votes, and only the agent outputs related to voting should use the vote-adjusted CDR.

This problem shows that we need a critic value function that predicts the negative effect of higher votes, as this is a true description of the reward function used, as well as the standard critic value function. Equation 27 shows the new loss function for the critic for a single observation with this adjustment implemented. V_i^* indicates the sub-value function with agent votes subtracted.

$$\mathscr{L}_t = (CDR - \sum_{i}^{M} V_i(x_t)) + ((CDR - \sum_{j=t}^{\infty} \sum_{i}^{M} \sum_{a}^{A} \gamma^{j-t} \frac{\text{vote}_{j,i,a}}{A}) - \sum_{i}^{M} V_i^*(x_t)) \tag{27a}$$

$$V(x)_i = \text{Standard critic output} \tag{27b}$$

$$V^*(x)_i = \text{Critic output with vote adjustment} \tag{27c}$$

$$A = \text{Num Actions} \tag{27d}$$

$$M = \text{Num Agents} \tag{27e}$$

$$\text{vote}_{j,i,a} = \text{The vote at time } j \text{ for agent } i \text{ for action } a \tag{27f}$$

When training the action distributions, we are concerned with whether or not a particular action resulted in an advantage to the given agent unrelated to the voting system. Therefore, we do not need this advantage to include the negative result of voting higher. Due to this fact, the critic network must have two sets of outputs, one for predicting the vote-adjusted CDR and one for the standard CDR. The standard critic output can then be used to train the agent action distributions using the standard PPO algorithm.

13.5.0.1 Layer Freezing Unfortunately, when adding the voting head to the critic and the sub-agents, it was found that with a single sub-agent, the model was unable to learn, even though the voting system should have had no effect on the system. This was attributed to the idea that the high uncertainty in voting earlier on meant the labels used to train the voting heads were meaningless early on. This may have resulted in the gradients computed for the voting loss damaging the standard gradients updates in too unpredictable a manner.

To counteract this effect, the neural networks used were designed such that outputs related to voting had separate heads from the standard outputs. Then when training on loss related to voting, the rest of the model could be frozen, and only these heads trained. This method assumes that the information passed to the heads was sufficient for authentic learning. It also introduces a new parameter in the size of the heads and how deep in the network they should be connected.

This change did result in stable training for a single agent system, equivalent to the results found in standard PPO. However, stable performance was not achieved with multi-agent systems, with the training results presented in figure 6. In this figure, it is clear that the system is able to learn at first before the training then collapses.

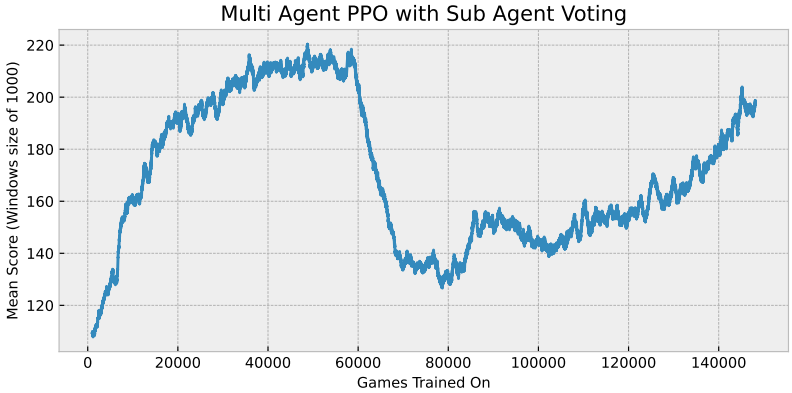

Figure 6: Average game score during training with three agent PPO with voting

13.5.0.2 Recurrent Networks As discussed in the background, the use of communication can prevent agents from attempting to learn each other's strategies in order to preempt their outputs. This was tested on both the voting system and intelligent switch systems by providing the previous output of each agent distribution as an additional input to the system along with the current state. The idea behind this is that agents can respond to the outputs of other agents and adjust their outputs if necessary. For instance, in the case of voting, if agents can see the votes of other agents, they should be able to adjust their vote to be higher if they can see their action won't be chosen. This system, in theory, should not be necessary as this kind of communication should, in a sense, happen naturally through the training process and the effect each agent has on the advantages used in training. However, it is possible to conceive a scenario including two agents where in two different states, agent one gives the same output and agent two gives two different outputs. Therefore, if agent one wants to be chosen, it must learn to use different votes in each scenario, thus requiring it to learn some state feature extraction that is only really required by agent two. However, if agent one is given access to the outputs of agent two, it can adjust its own vote without having to predict the vote of agent two.

Unfortunately, it was observed that this addition had no effect on increasing the stability of either system, suggesting that the problems causing the stability observed are not related to the improvements possible through this modification.

14 Model Design

Throughout this project, all agents described use a similar neural network structure unless otherwise specified. A simple model design has been used since the focus of this project has not been on trying to maximise the amount of performance possible by tuning the neural networks themselves but rather on the training algorithms. The input and output layers of the networks are parameterised by the number of agent outputs and the state inputs, but the hidden layers are a consistent size of $(10, 5, 5)$ across all agents. RELU activation is used for the hidden layers and activation specific to the type of action output (value, probability etc).

Part V
Testing and Experimentation

In order to achieve notable performance, the various algorithms previously described required significant experimentation and tuning in order to both tune the hyper-parameters and debug the implementations themselves. Unless otherwise specified, all the training experiments presented in this chapter were run on the Space Invaders game and use moving average windows of size 1000 to present the data. The moving average is necessary due to the extremely high variance in scores achieved during gameplay.

Only the experimentation that yielded note-worthy results is presented here, as that done for algorithm debugging and implementation generally followed similar patterns and is not relevant to understanding the implementations and results presented.

15 Tuning PPO

As described in the implementation chapter, the PPO implementation is dependent on several hyper-parameters that must be tuned in order to evaluate its performance and use these parameters as baselines in the other PPO-based multi-agent systems investigated. The most important parameter tested here is the discount factor γ, which determines how far into the future the agent "considers" reward when training. Hence, naturally, it is anticipated that a longer-term strategy would necessitate a higher γ value. The second major parameter PPO introduces is the clip value, with restricts the size of changes to the policy that can be made in order to stabilise training.

15.0.0.1 Varying Discount Factor Using a constant clip of 0.2, figure 7 shows the training curves for three different discount factors, $0.995, 0.95, 0.9$, when training on the Space Invaders Game. These curves display a moving average of the score achieved in each game played during training, with a window size of 1000 used in order to make the trends visible. As can be seen, the higher discount factor of 0.995 does slightly impact performance as the agent struggles to learn to avoid bullets as successfully, so any improvements in learning to shoot enemies quicker were negated by this disadvantage. This outcome indicated that learning to avoid bullets in the Space Invaders game is significantly more complex than learning to destroy enemies and demonstrates how dependent on specific strategies the optimum discount factor is.

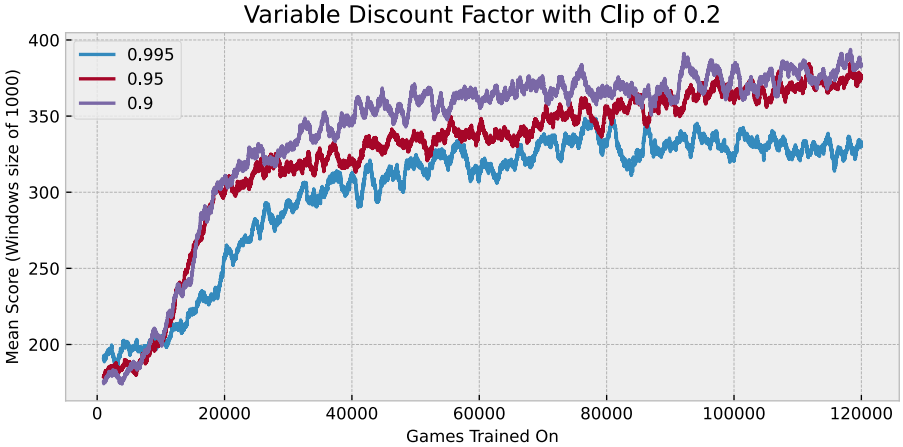

Figure 7: Average game score during training with variable discount factors

15.0.0.2 Varying Clip In addition to γ, the clip hyper-parameter can also have a large impact on the learning performance of a model. A larger clip can allow for more exploration as it allows for larger steps that can have a more random effect on the policy. The optimum clip is tied to the γ value, as a higher γ can

mean a policy that is more difficult to learn due to less immediate change in cumulative discount reward due to an action. This suggests that a larger γ may require a smaller clip to encourage smaller changes in policy each step. Figure 8 shows training results for three different clip values in the Space Invaders game with a γ of 0.2. As can be seen, introducing a lower clip of 0.05 or 0.01 decreased the performance of the models, as smaller policy steps were possible.

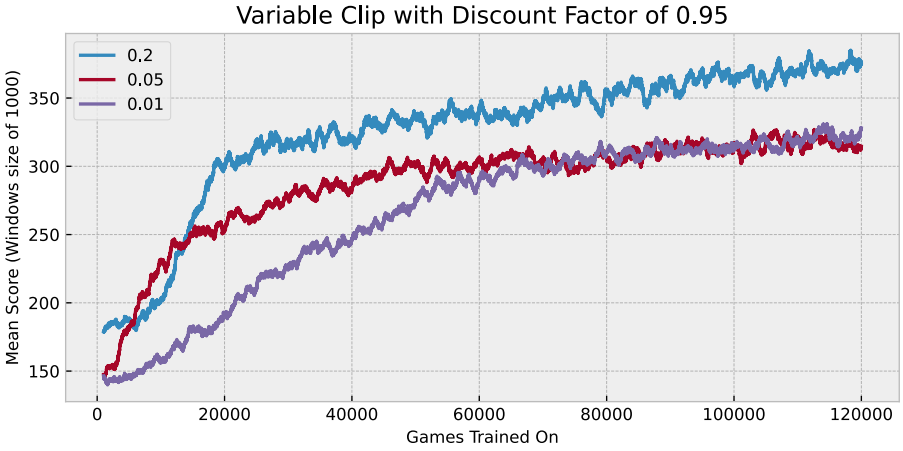

Figure 8: Average game score during training with variable clip

Figure 9 again demonstrates varying the clip value but with a higher γ of 0.995. Here we can see that the lower clip value of 0.01 gave the best performance, with clip scores too low preventing proper policy updates and clip values too high, allowing for unstable policy updates.

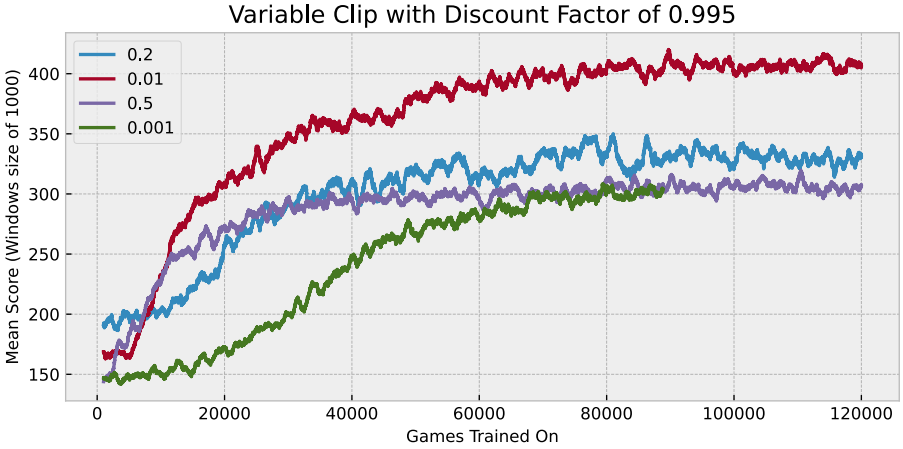

Figure 9: Average game score during training with variable clip

It should be noted that the relationship described here between the clip value and γ is extremely game and strategy specific, and the differences found in performance are not large enough to be indicative of more general relationships. The scores collected are the result of the mean of three trials per test, which is too small a sample size to ignore the randomness in the route a model takes during policy learning. Unfortunately, larger sample sizes were not feasible in this project due to the large time and resources required to train the models. The testing presented her for PPO does allow for basic tuning of these hyper-parameters and demonstrates that the algorithm is implemented in a stable manner which can be relied on in other tests.

The final PPO test given here in figure 10 demonstrates the negligible effect that batch size had on training, with batch sizes greater than 5000 preferable due to slight speed gains in playing a higher number of games between training due to the scaling architecture implemented.

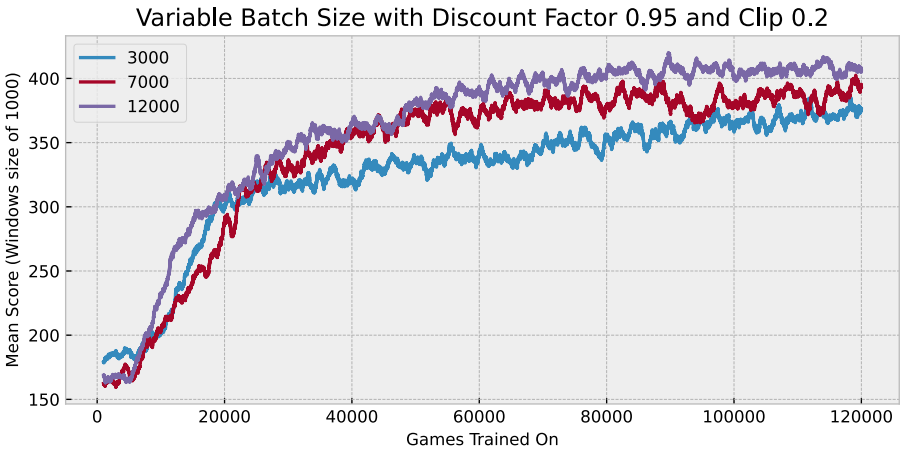

Figure 10: Average game score during training with batch size

16 Tuning DDQN

A common theme found throughout the research of existing RL algorithms was that DQN algorithms are generally harder to tune than PPO-based methods, which has been reflected in the testing of the DDQN implementation used here. The main hyper-parameters tested were the discount factor as in PPO, the rate of decay of epsilon and the number of reward steps to update over. In addition to the experiments given here, increasing the memory size to a sufficient size and using a low learning rate was also found to be critical to successful learning.

16.0.0.1 Discount Factor The first test uses a var ε decay rate of $1e-6$, which decreases ε linearly from 1 to 0.05 over the course of training, controlling the proportion of random actions using an epsilon greedy strategy. In this test, the discount factor γ varies with the training curves given in figure 11. As can be seen, a lower γ resulted in higher performance; however, this means the algorithm is less able to learn longer-term strategies. It is also evident that the performance is significantly less than that of PPO, although this is to be expected from previous comparisons of this algorithms [14]. The highly reduced γ of 0.9 results in somewhat erratic training, suggesting the agent cannot converge to a consistent policy due to the higher variance in value function resulting from this low γ.

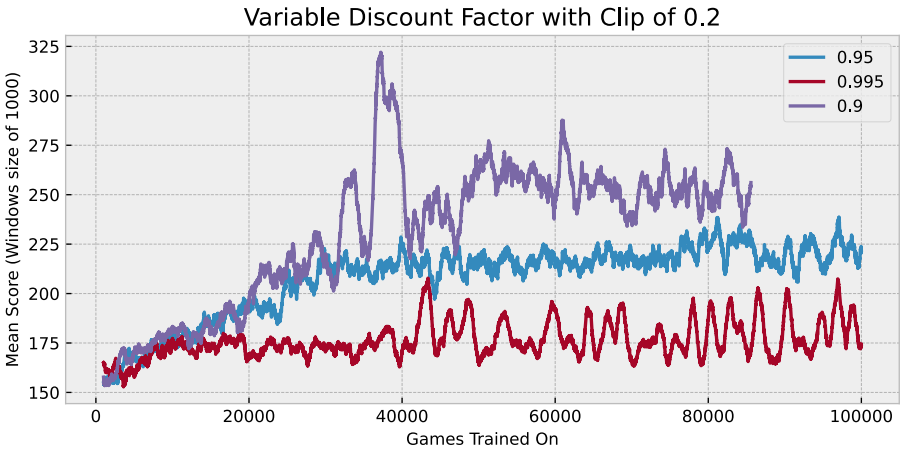

Figure 11: Average game score during training with variable discount factor

16.0.0.2 Epsilon Decay As discussed previously, the ε decay rate used in DDQN controls the amount of exploration, but since a lower decay rate will result in more random actions over a longer period of time,

it would be expected that this will decrease the average score during this period of training. As can be seen, using an extremely low ε of $1e-7$ results in significantly harmed training as the agent is unable to learn from states that occur as a result of following the policy until extremely late. The value of $5e-7$ appears to be low enough to allow for greater exploration but not prevent sufficient policy discovery further on in training.

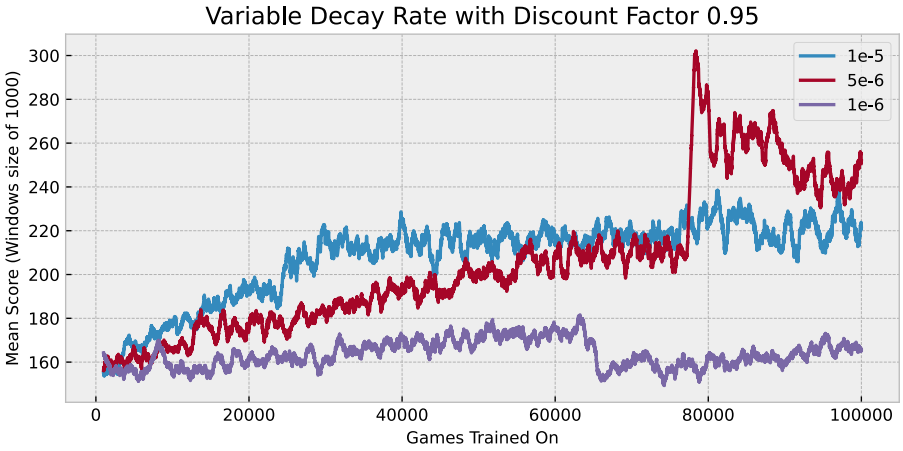

Figure 12: Average game score during training with varying ε decay rate

16.0.0.3 N-Step Reward The N-Step reward modification discussed in the implementation chapter was also used to tune DDQN. Here a greater number of reward steps should result in less bias in value estimates but greater variance [7]. Figure 13 shows the training curves for three different N values. The test demonstrates that the N values had no significant effect on the training performance and so a step of 1 is used in further implementation.

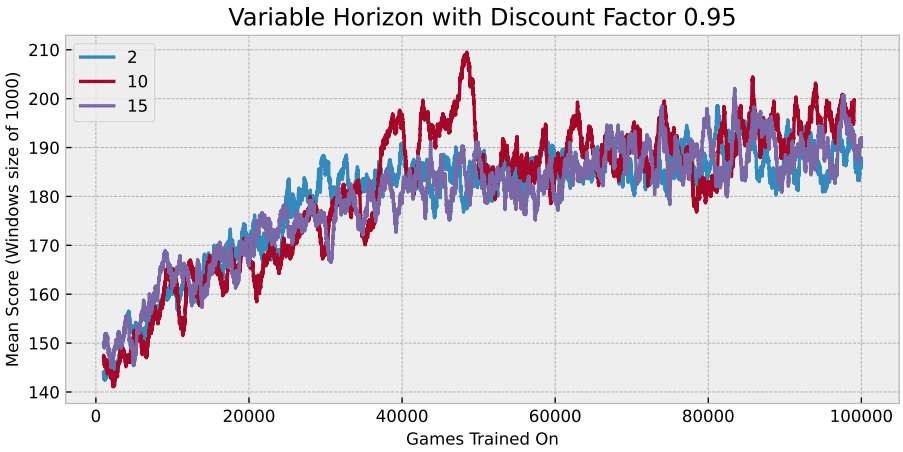

Figure 13: Average game score during training with varying ε decay rate

16.0.0.4 Performance of DDQN vs PPO Within the training graphs presented, the performance of PPO is significantly higher than that of DDQN. This may be due to the specific game type used, as the two algorithms are generally found to perform better in different scenarios. In addition to worse performance, greater instability of the DDQN algorithm can also be observed, which may indicate too great a learning rate or too small a memory size; however, decreasing and increasing these respectively didn't improve performance.

17 Evaluation of Deconstruction Methods

Throughout this section, the different multi-agent systems described are tested in a similar manner to that of the base algorithms. In addition to mean score training curves, the cross-correlation between sub-critic values is also heavily used as an indicator of the separation of reward signals. The measurement is taken by playing games 10 times after each batch and for each game, recording the sub-critic values at each frame. From these sub-critic values, a correlation matrix can be taken, giving the difference between combinations of signals, and finally, the mean across this matrix is used as the cross-correlation score.

17.1 Divergence of Rewards with Multi Model PPO Policy Product

The policy product mechanism for combining agent outputs was the only policy-based deconstruction algorithm tested that yielded relatively stable training. Despite this, the training curves were still less stable than standard PPO and did present several issues. The greatest of these issues is found in trying to balance the maximisation of agent specialisation whilst maintaining performance stability, a trade that was difficult to achieve.

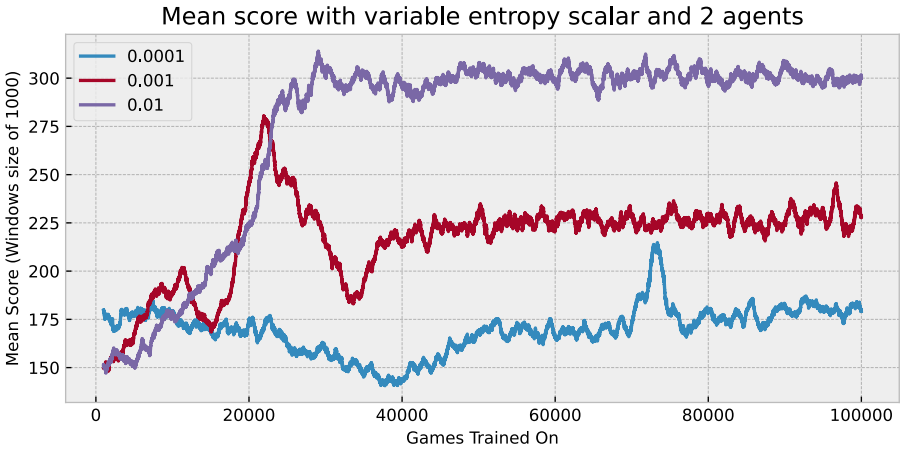

Figure 14: Average game score during training with varying entropy scalars and two agents

Figure 14 gives the training curves for a two-agent system using the Space Invaders game with varying entropy scalars. This scalar is multiplied by the entropy term used in the loss function during agent training, and increasing it leads to greater exploration. As can be seen, the greater the entropy scalar used, the higher the performance achieved (although entropy scalars greater than around 0.01 were found to give a worse performance. The use of a higher entropy scalar may alleviate the problem described earlier whereby when one agent outputs probabilities that are two concentrated around a single outcome this can disable the effects of other agent distributions. The increase in entropy alleviates this by discouraging the concentration of probabilities within the given distribution.

However, figure 15 shows how the mean cross correlation between the sub-value outputs given by the critic. These values demonstrate that a higher entropy scalar results in worse divergence performance of the critic values, suggesting lower sub-agent specialisation. These results indicate that the greater entropy scalar and so greater randomness in agent actions has some effect on preventing the specialisation of agents, as perhaps it is harder to learn longer strategies due to the randomness in the policy used.

In addition to the entropy scalar, the number of agents used was found to have a significant effect on the divergence of the critic sub-values. Figure 16 shows the training performance using 2, 3 an 4 agents. It is evident that a greater number of agents results in higher performance as well as greater stability in training. The greater performance is to be expected, as this is somewhat equivalent to using a larger single model, which would also be expected to have higher performance; however, the reason for greater stability is not so clear.

Figure 17 shows the critic sub-value divergence for this same training set and demonstrates that a higher agent number also yields a lower average divergence of reward signals. This correlates with the fact

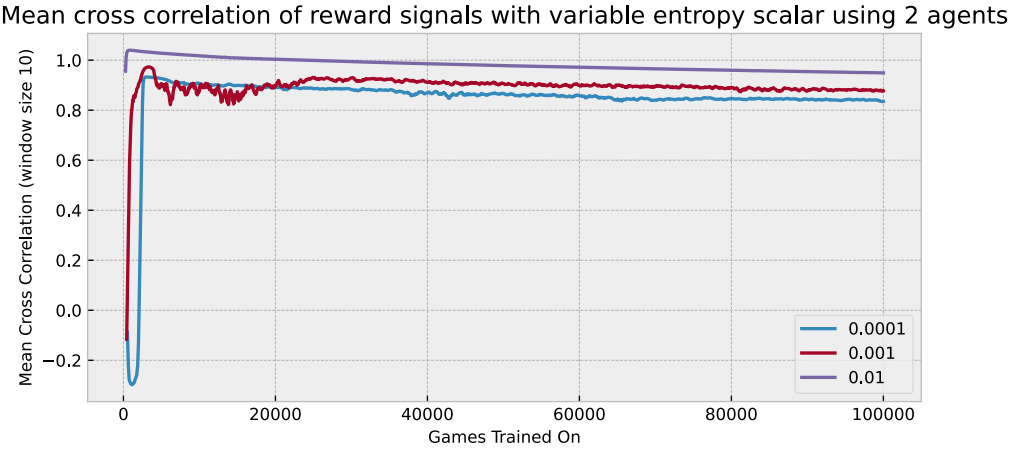

Figure 15: Average sub-critic value cross-correlation during training with varying entropy scalars and two agents

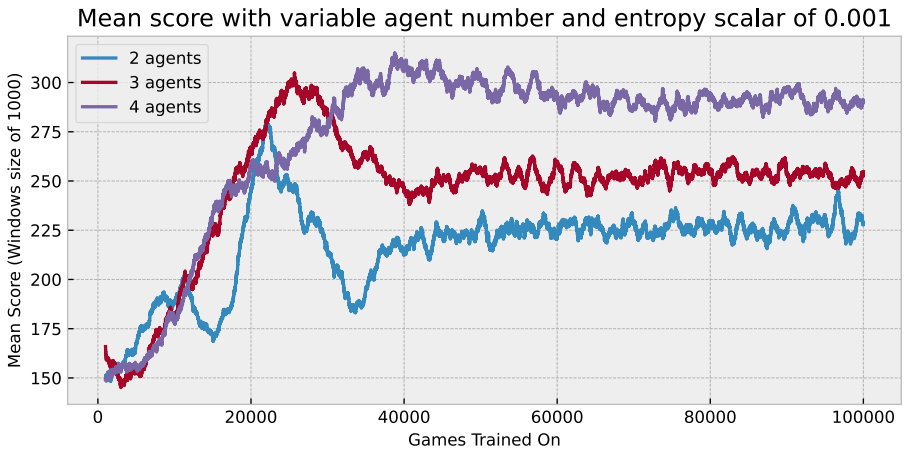

Figure 16: Average game score during training with varying agent number

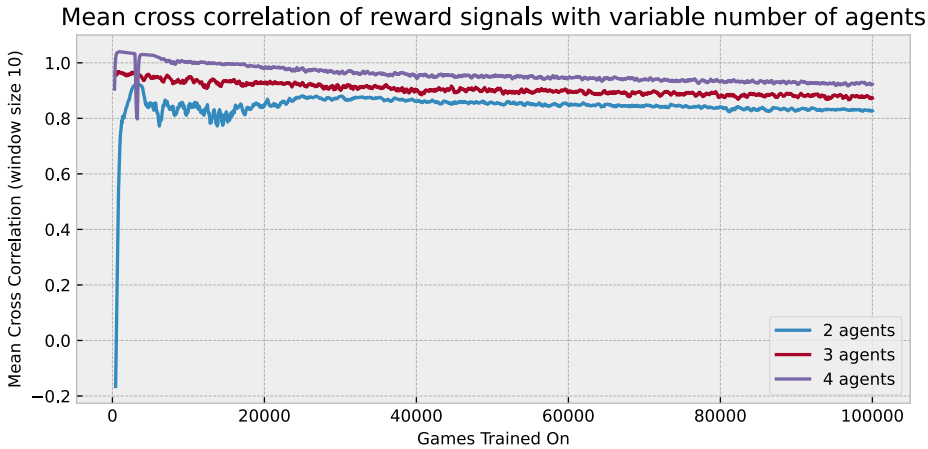

Figure 17: Average sub-critic value cross-correlation during training with varying agent number

that a higher number of agents should lead to more redundant and so similar sub-critic values with high correlation, and as could be seen in previous experiments varying entropy, lower divergence is correlated with higher stability of training.

17.1.1 Performance of Multi-Model PPO with Manually Deconstructed Rewards

The two-agent system with parameters that achieved the highest score (entropy scalar of 0.01) was re-trained with a modified version of Space Invaders that also gave reward for killing enemies. This resulted in identical maximum score convergence but did yield a more significant divergence of sub-critic values.

The original two-agent system was able to achieve a minimum value cross-correlation of 0.82, whilst after adding the additional reward output, the correlation reduced to 0.75. This perhaps indicates that reduced cross-correlation does indeed suggest a separation of reward signals.

Figure 18 gives an example of the two critic sub-value functions throughout a single game, with the red dashed lines marking frames where an enemy was killed. As can be seen, these points correspond more often with dips in the blue value function than that of the second orange value function, indicating a potential relationship between the first blue value function and the killing of enemies.

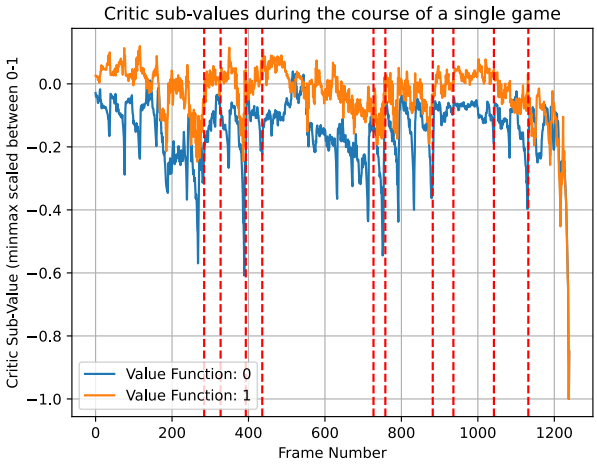

Figure 18: Sub critic values over the course of a single game using Space Invaders with additional rewards for killing enemies, dashed lines indicate enemy deaths

Figure 19 shows a game using a two-agent system trained with the standard Space Invaders reward signal. Here there is no clear connection between the value functions and the killing of enemies, indicating that a simpler reward signal does make divergence harder.

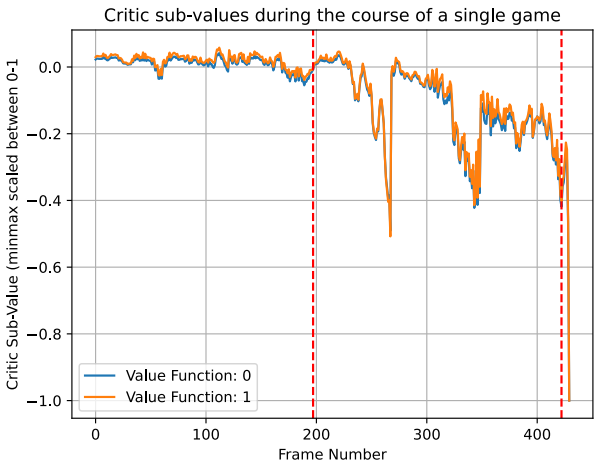

Figure 19: Sub critic values over the course of a single game using Space Invaders with standard rewards, dashed lines indicate enemy deaths

17.2 W-Learning and Q-Aggregator Performance

17.2.1 W-Learning

Unlike using the product of outputs with PPO, W-Learning uses the output of only a single agent at any one time. This form of algorithm appears to generally give less stable results, as can be observed here with W-Learning and from the poor performance of Multi-Agent PPO with Voting. The scores achieved in the following tests are fairly inline with those found with single-agent DDQN but are far less stable.

17.2.1.1 Varying Epsilon Decay The first experiment, as in DDQN, varies the epsilon decay rate, thus modifying the amount of exploration permitted. Interestingly, the optimum decay rate seems to be lower than that found with single-agent DDQN. Figure 20 gives the training curve for three different decay rates, and as can be seen lowest value of $1e-6$ gives both the best and most stable performance. This might indicate that this kind of agent is more susceptible to a higher correlation between training samples, with the effects of this negated by the higher randomness in actions. All the models used here are two agent systems with a discount factor of 0.95.

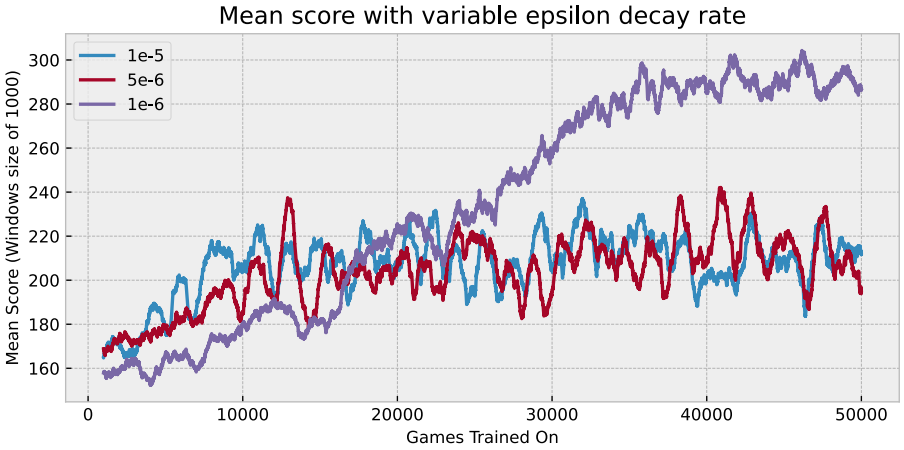

Figure 20: Average game score during training with varying ε decay rate

Figure 21 gives the mean divergence during training of the W-Learning models with varying epsilon decay rates, and as can be seen, the lower decay rates gives significantly higher divergence, with the amount of divergence also far higher than that achieved with the policy based algorithms.

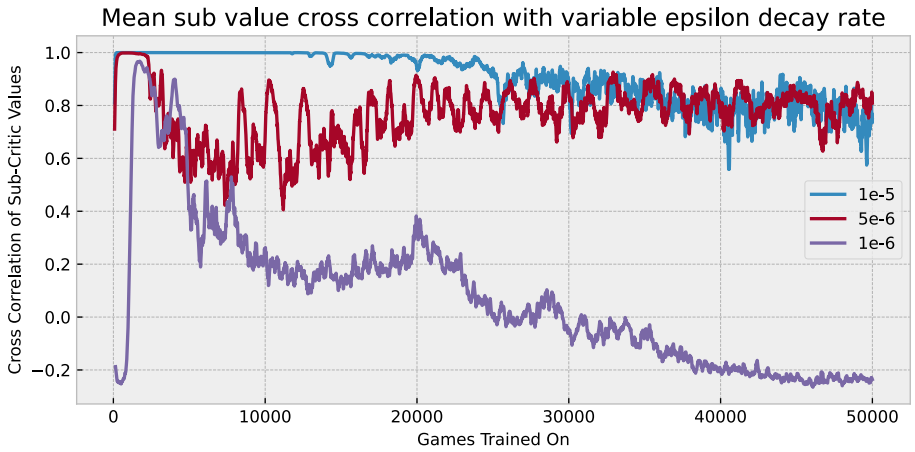

Figure 21: Average game score during training with varying ε decay rate

17.2.1.2 Varying Agent Number Figure 22 compares the performance of a dual-agent system against that of a three-agent system. Higher performance is achieved with the three-agent system as expected due to the larger overall model size, and both systems give similar divergence performance.

Figure 23 gives the divergence rates for a two vs three-agent system, with both systems showing similar levels of divergence. This suggests that in this case, the third agent isn't adding too much redundancy, as both performance is increased and divergence is maintained with this greater number.

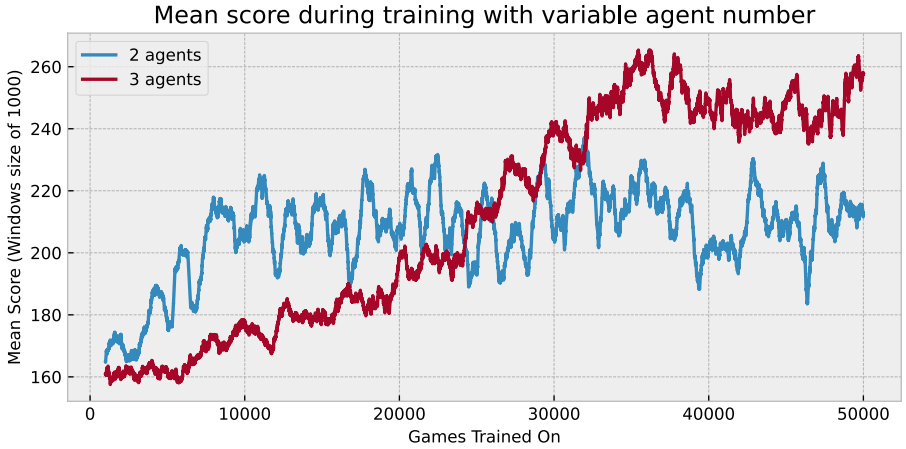

Figure 22: Average game score during training with varying ε decay rate

17.2.1.3 Experience Replay As discussed in the background, the work done by Foerster et al.[9] showed that the use of experience replay could give a poor performance with multi-agent systems due to the agent's policies being different at previous states, and so the environment not being consistent. The use of a fingerprint in the form of the current ε value was not found in this case to make a difference in training performance in this case, unlike the results found by Foerster et al.[9]. Removing the experience replay was also found to give worse performance across all tests discussed, suggesting the benefits of decreased sample correlation from cross-correlation outweighs the effects of environment changes.

17.2.2 Q-Aggregator

The second DDQN-based deconstruction method that achieved successful training was the use of a Q-Aggregator, where the sum of sub-agent Q-Values was summed to give the final output. As shown in

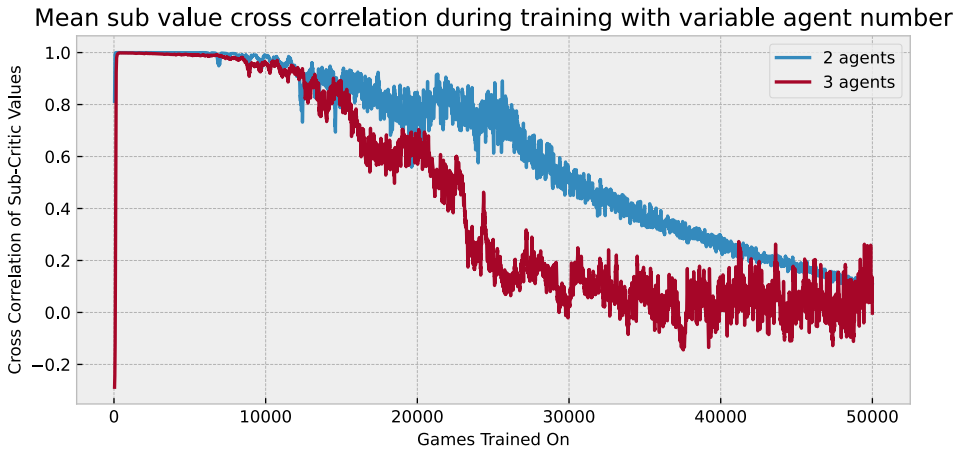

Figure 23: Average game score during training with varying ε decay rate

figure 24, the performance achieved is higher than that of standard DDQN and W-Learning, with increased stability. Figure 25 gives the mean cross-correlation of the critic values and demonstrates that in this case, three agents may be too many as the mean divergence is much higher and the performance is not significantly different, suggesting redundancy in the number of agents. These tests were performed with a discount factor of 0.95 and an epsilon decay rate of $1e-6$. A similar relationship was found to these values as with W-Learning, and this combination achieved optimum performance.

Interestingly the Q-Aggregator method consistently outperforms basic DDQN, with results similar to policy optimisation methods. This is the only deconstruction method tested that outperformed the equivalent base algorithm with a single agent.

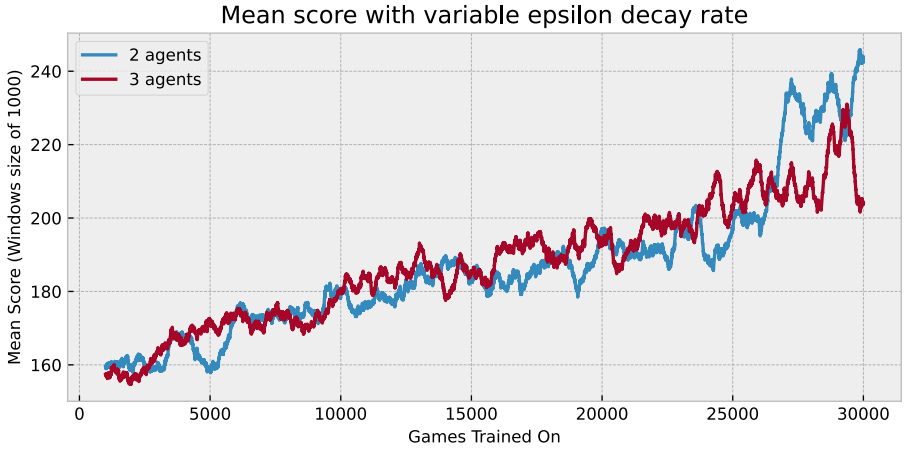

Figure 24: Average game score during training with varying ε decay rate

17.2.3 Performance of W-Learning and Q-Aggregator with Manually Deconstructed Rewards

Unlike the case of PPO, using the modified reward signal whereby the game outputs an additional reward for destroying enemies resulted in increased W-Learning performance, from about 250 frames to 300 frames. However, there was no significant change in divergence as the W-Learning already presented very high divergence. Figure 26 shows the lack of effect of an enemy death on the sub-value signals using the modified reward within a single game.

The reason for the increase in score achieved is more apparent when observing the strategy that the models use when given the modified reward signal. Agents are more likely to try and shoot enemies, which results

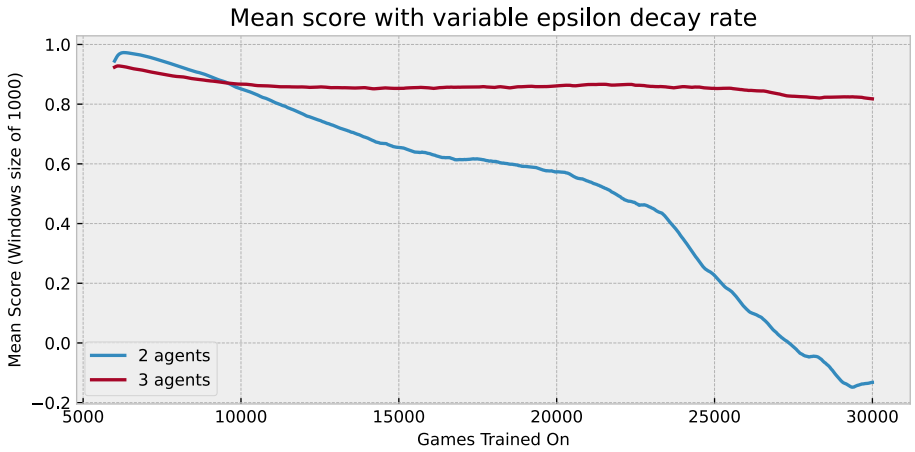

Figure 25: Average game score during training with varying ϵ decay rate

in higher performance. In contrast, in the policy-based algorithms, the agents were already learning to shoot enemies to a similar extent.

In the case of the Q-Aggregator algorithm, no noticeable increase in performance or divergence was achieved by adding the second reward signal.

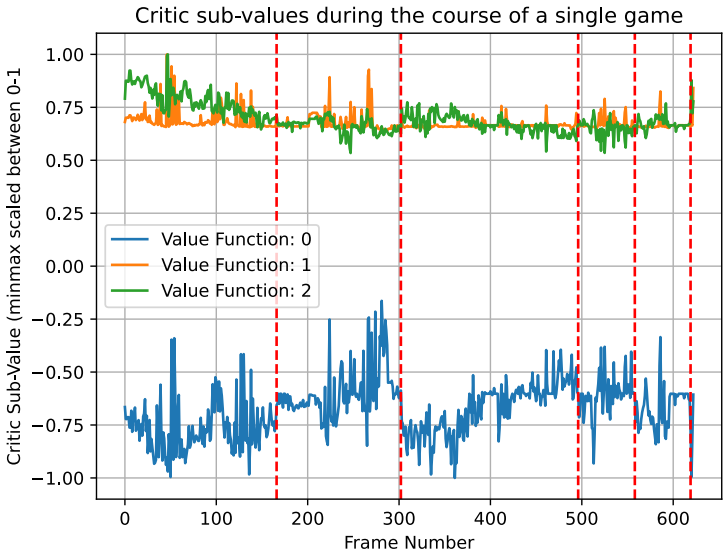

Figure 26: Sub critic values over the course of a single game using Space Invaders with additional rewards for killing enemies, dashed lines indicate enemy deaths.

18 Agent Specialisation

In order to demonstrate the amount of specialisation achieved for each deconstruction method, the individual agents were trained from the point achieved during multi-agent training, and the time taken to achieve the convergence score was recorded. It is assumed that a more specialised agent should take longer to learn to play individually, such that the higher training time would indicate greater specialisation. This experiment was carried out on dual and triple agent systems for the Q-Aggregator, W-Learning and Multi-Agent PPO Product algorithms, using the highest performing parameters for each. Table 1 gives the results achieved.

It should be noted that when interpreting the results due to the design of the Space Invaders game, the standard increase in performance over the course of training is in the region of 100% (i.e. doubling the number of frames survived. Therefore a 30% increase in start performance for a retrained model corresponds to starting roughly 30% of the way through the standard training process.

Multi-Agent PPO with Product			
	Agent 1	Agent 2	Agent 3
Δ% start performance	+14	+30	+20
Δ% peak performance	-10	-5	-7
Δ% time to peak	-5	-15	-20

Table 1: Retrain performance of three agent system relative to training performance of a single agent system, using Multi-Agent PPO with Product of distributions

Q-Aggregator			
	Agent 1	Agent 2	Agent 3
Δ% start performance	+15	+17	+25
Δ% peak performance	-4	-1	+1
Δ% time to peak	-22	-20	-37

Table 2: Retrain performance of three agent system relative to training performance of a single agent system, using Q-Aggregator

W-Learning			
	Agent 1	Agent 2	Agent 3
Δ% start performance	+12	+20	+12
Δ% peak performance	0	+2	0
Δ% time to peak	-29	-35	-20

Table 3: Retrain performance of three agent system relative to training performance of a single agent system, using Deep W-Learning

The results presented demonstrate that in each case, the sub-agent that started with the greatest increase in start performance also had the greatest decrease in time needed to achieve peak performance as expected. It is also clear from the range of different start performances for each agent in every algorithm type that there are agents with different levels of specialisation. This suggests that in each case, one agent is most responsible for the standard control of the player and is most able to play successfully immediately. However, the fact that this increase is relatively small in each case also suggests that one agent isn't in full control of the system and removing any agent severely decreases performance.

18.1 Decreasing Agent Operation Frequency

In order to test the premise of decreasing agent sub-agent frequency, the rate at each agent processed game frames was decreased gradually until a difference in average game score was measured. This test was done using trials of 10000 games, using the best-performing three-agent system for each algorithm type. Since the standard deviation of 10000 game scores was measured to be 158 frames, the standard deviation

of the mean of 10000 game scores should be 158 $\sqrt{10000} \approx 5$. Therefore a decrease of 1 frame average score should correspond to a change of 0.2 standard deviations, so it is not particularly significant; based on this, each agent frequency was reduced till a change in average game score of 1 was measured.

Table 4 gives the result of this test using the PPO Product method. As can be seen, there is only a difference of one frame between the maximum and minimum frame difference, which, although it does demonstrate how it may be possible to run each agent at different frequencies, is not sufficient evidence to prove this idea. The same test is done for W-Learning in table 5 and Q-Aggregator in table6 gives very similar results with again no changes large enough to be significant.

PPO Product, Max frames between dist update		
Agent 1	Agent 2	Agent 3
2	3	2

Table 4: Maximum number of frames between distribution calculations for each sub-agent using PPO Product

W-Learning, Max frames between dist update		
Agent 1	Agent 2	Agent 3
3	3	2

Table 5: Maximum number of frames between distribution calculations for each sub-agent using W-Learning

Q-Aggregator, Max frames between dist update		
Agent 1	Agent 2	Agent 3
4	2	2

Table 6: Maximum number of frames between distribution calculations for each sub-agent using Q-Aggregator

18.2 Comparison with Large Single Agent Networks

As described previously, the same neural network structure is used for each agent type in all the training examples. However, it is important to note the difference in the performance of the multi-agent systems against a single-agent system with a model size equivalent to the combination of the three agents.

Figure 27 compares the three-agent PPO product method against a single-agent PPO implementation with a model that uses layers with three times the nodes. This is, in fact, a larger amount of weights in the single-agent system due to the greater number of connections between nodes not present in the multi-agent system due to the separation between agents. As can be seen, the multi-agent system actually reaches peak performance faster than the single-agent system, which is surprising in this case as the actual training time is far less due to the cyclic training method used in the multi-agent system. As in all other graphs, the x-axis displays the number of games training on and not the training time, even though the training time per game is not always equivalent. The reason for this is the training time per game is highly dependent on the implementation of the algorithms. Since it is unclear that the most efficient manner of implementing the respective algorithms has been achieved in this project, comparisons of training times would not be fair and could be misleading. In order to make such comparisons, more time would be needed to ensure all optimisations have been applied to each implementation.

Figure 28 shows the same comparison for the Q-Aggregator method. Once again, the multi-agent system appears to learn faster; however, the increase is not great enough to be notable. As found in the previous DDQN-based tests, the training is also fairly unstable, with a peak performance reached before converging to a lower stable performance in the case of the multi-agent system. It should be noted that over 5 runs of each of these tests, there was a 10% improvement in the number of enemies killed using the multi-agent system before reaching the peak performance. This may suggest some increase in the ability of the system to learn longer-term strategies but is too small an increase to be significant or give an increase in mean frame survival time. Similar results were found for W-Learning as for Q-Aggregator in this test.

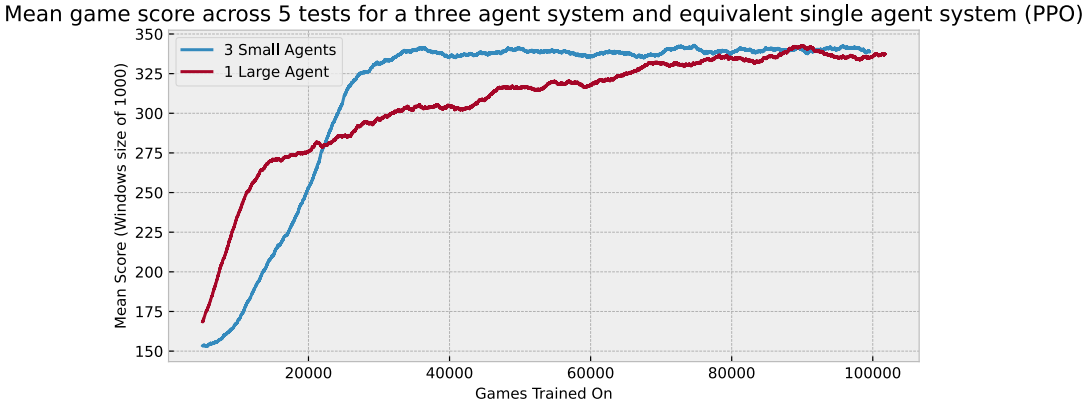

Figure 27: Comparison of large single agent training curve against equivalent size three agent system, (average across 5 tests), using PPO product method

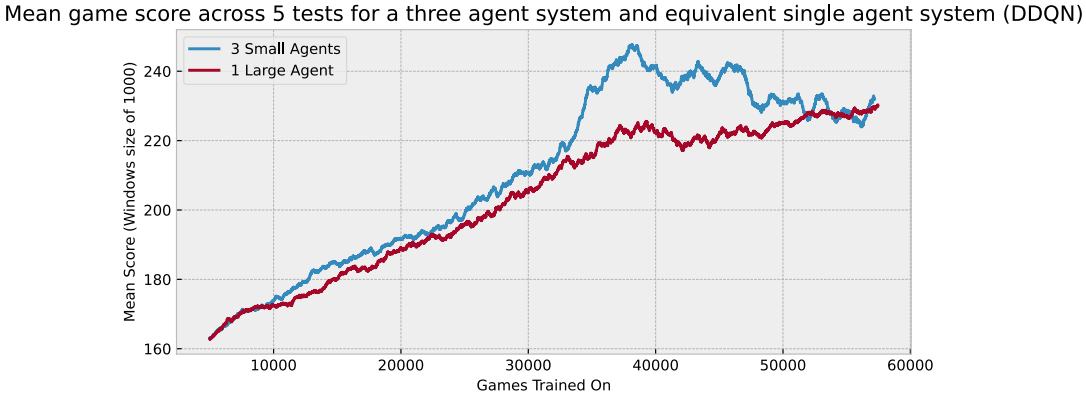

Figure 28: Comparison of large single agent training curve against equivalent size three agent system, (average across 5 tests), using Q-Aggregator method

Part VI
Analysis and Conclusion

19 Analysis of Results

Throughout the implementation of the project and the experimentation, a large number of different algorithms and variations on those algorithms were investigated, resulting in a wide variety of results. Unfortunately, the policy merging methods that had the most time spent invested, specifically the PPO with Voting and Intelligent Switch methods, yielded no stable results.

This outcome, however, is suggestive of the fact previously mentioned that systems that take the output of only a single agent resulted in less stable outputs than those that used combinations of agent actions. Both the Intelligent Switch and PPO Voting methods suffer from this problem as well as W-Learning. Although W-Learning did provide some stable training, the testing demonstrates that it achieved lower performance in Space Invaders than the Q-Aggregator implementation when fully tuned.

The reason for this outcome is not entirely clear but is likely to do with the problems described in the background regarding the loss of the Markov property when using multi-agent systems. However, the fact that removing experience replay did not stabilise the W-Learning may disprove this, as the use of experience replay should exasperate this problem. This is due to the fact that samples stored when using experience replay give a state change based on an agent action; however, with the new sub-agent policies, the same sub-agent action may no longer give the same state space change when combined with the outputs of other agents. This problem can be negated by the fact that the action used is the global action and not the sub-agent action; however, this does not entirely remove the problem, as the true value of taking an action is still dependent on the other sub-agent policies at the time.

Unfortunately, the effect of removing experience replay has a larger set of consequences than just removing the problems with changing environments, as it introduces problems in the learning of Q-Function algorithms. So the loss of Markov property may still be the factor causing this instability.

It may seem at first that the systems that combine agent outputs should be just as affected by this problem, but this is not necessarily true. Due to the use of randomness in agent actions necessary for exploration, the proportion of steps directly following a state where a sub-agent is in control may be subject to high variance in methods that select a single agent. This variance may introduce many training samples where the agent's policy isn't followed to any extent for several subsequent state changes introducing errors in training. In the case of policy combination, the agent's policy always has some effect on the global policy. So the proportion of the time it is listened to may have a lower variance, giving fewer samples that aren't related to the sub-agents policy.

19.0.0.1 PPO Product Performance The performance of the PPO-based multi-agent system that used element-wise policy products to combine sub-agent policies is suggestive of several interesting results. As described during the testing in subsection 18.2, the training speed of the multi-agent system was greater than that of the single-agent system, both in the number of games required and the total time required to train. The use of cyclic training means only one agent is trained during each batch of games, whereas in the large single-agent system, the entire agent is trained for each batch. The exact effect on speed, as mentioned earlier, is difficult to specify due to implementation details that might not maximise speed, but in the implementation used here, a roughly 20% speed increase was gained during training between the multi and single-agent systems.

The fact that cyclic training was necessary may also suggest that the type of strategy learnt using the multi-agent method might not be possible using a single agent that trains the entire model simultaneously. The fact that stable performance was not necessary without cyclic training in the multi-agent system suggests that it might not be possible for the single-agent model to form the same sub-agents internally. However, this might not be true as the single-agent system might simply learn the sub-agent strategies one by one or through some other method.

19.0.0.2 Game Performance All the tests discussed thus far have used the Space Invaders game, which was the focus of the project development. Interestingly, despite relatively similar scores across algorithms, the manner in which the different agents played the game was quite algorithm-dependent. For instance, in the case of PPO-based methods, the output of the agent is the probability of each action, thus if an agent gives a $1/3$ chance of moving left and a $2/3$ chance of moving right, this averages out to just moving right at $2/3$ of the top speed. This ability to choose a speed of movement just through a static output means that the PPO-based algorithms could set their speed such that when they moved past an enemy, two bullets would hit the said enemy when constantly shooting. If the agent moved faster, one or zero, shots would be guaranteed to land, and if the agent moved slower, it would not keep up with the enemy and would have the same issue. In the case of DDQN-based methods, if the agent outputs a static set of Q-Vals, then the same action will always be taken; therefore, to move at a specific speed, the Q-Values would have to oscillate using a feedback loop from the time-dependent information it receives in its input. This system is inherently more complex to learn, and the DDQN-based agents were not observed to learn to move at this specific speed very frequently.

Interestingly the DDQN-based models were also much more likely to not always choose the shoot action but rather learn to time their shots. This might be related to the fact that they were less able to learn to move at a speed that naturally landed more enemy shots. In fact, the strategy given here was more similar to that of human players, as a human player could not generally oscillate moving left and right fast enough to move at a constant speed in the manner done by PPO.

The performance achieved by the Multi-Agent PPO-based algorithms of 350 frames is pretty similar to that found with a human after some practice and generally outperforms most people on their first few attempts.

19.0.0.3 Driving Game Performance In addition to the experimentation outlined, the driving game described in subsection 11.2 was also used to test agent performance. As mentioned, the reward signal for this game is significantly more complex than that of the Space Invaders game. In addition to this, to complete the game, the agent would often have to endure a period of long negative reward as it moved away from the target due to the maze design, necessitating an extremely high discount factor. The testing with Space Invaders demonstrated already that a higher discount factor could give a lower performance with the implementation and model sizes present in this game, and this was reflected in the testing with the maze game.

Unfortunately, none of the algorithms investigated in this project could learn meaningful strategy in this game, and agents simply learned basic wall avoidance and took at most one turn towards the goal state. The game design may be inherently too complex for the size of the models used, and in the future, testing with longer training periods and larger models may yield better results.

19.0.0.4 Reward Separation Many of the experiments shown gave significant evidence that the specialisation of agents and the separation of reward signals is possible using the critic method suggested in this project. Especially the low starting performance of sub-agents retrained to play individually shows that each agent is not simply learning the same strategy and that the entire global system was necessary. Figure 29 shows the sub-critic values across a single game for the PPO-based method with three agents that achieved the lowest cross mean cross correlation between agents. In this figure, it is evident that the three signals give vastly different information. One point to note is that early on in training, the value functions often correlate quite highly with simple game statistics like the closest bullet distance or the number of enemies. However, as training progresses and the critic because more confident the player will survive a situation, the correlation of signals such as the closest bullet distance has almost no correlation with the value function as the prediction of future reward becomes more complex.

The idea presented that the use of sub-agents trained on the sub-critic values encourages divergence is also supported when testing the cross-correlation of sub-critic values using a single large agent. In the case presented earlier, testing a large agent with PPO with cross-correlation of the single agent system using three outputs for the critic value had a mean of 0.90, whereas, for the three agent system, this value was 0.82. Further testing would be required to test these results better; however, it is a good indication that coupling the sub-critic values to individual policies results in a feedback loop that can allow for divergence through the random exploration done by each agent.

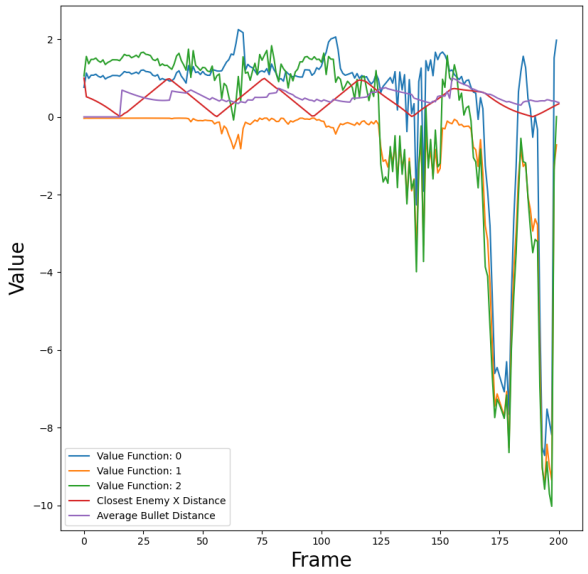

Figure 29: Sub critic values over the course of a single game, with simple game stats overlaid

20 Comments on Implementation Experience

Over the course of this project, many technical difficulties were faced with regard to algorithm implementation and data collection. This section includes some notes and suggestions on how these problems might be avoided.

Implementing algorithms using TensorFlow[37] presented many issues in terms of debugging mistakes, as these mistakes are often not caught at compile time but are more subtle. Often it can be difficult to keep track of what dimensions of a tensor store what information, and if two dimensions are the same size, this can lead to issues. For instance, if we have a tensor of dimensions ”Num Observations” by ”Num Agents” by ”Num Actions” by ”Distribution Size”, the total number of dimensions is 4, which can be hard to keep track of. If we imagine a second tensor with the ”Num Agents” and ”Num Actions” dimensions swapped and have the same number of actions as agents present, we could add two such tensors without any errors. In this case, the error would not become apparent until hours of training had passed, and even then, the source of the error would not be clear or even if it is a problem with implementation and not the algorithm itself.

To alleviate this issue, heavy usage of the Python typing system was used, which does catch some errors of this kind. For instance, if we have a function that returns an agent index, it is better to create a type alias for “int” named “AgentIndex” and then set the return type of this function as “AgentIndex”.

Unfortunately, the Python type hinting system cannot handle the dimension issue described above. To fix this, a new system would be needed that allows for the naming of dimensions and prevents values from being combined that don’t belong to the same sequence of dimensions. In theory, it could be possible to build such a dimension typing system on top of Python, which could have a lot of use outside of just machine learning within the wider scientific community. Such a system may be investigated as part of future work inspired by this project (if not directly related).

21 Concluding Remarks

The project undertook an extensive examination of various deconstruction algorithms. This investigation involved a combination of methods that were already in existence in the field and the exploration of novel methods. A plethora of different variations on these algorithms were tested, with experimentation carried out on the variations that gave successful training performance.

Several of the algorithms that underwent testing demonstrated fairly stable performance during training. This was encouraging; however, it was somewhat marred by the fact that each method also presented its unique set of issues. These problems were not trivial; in fact, they were quite fundamental to the designs used and emanated from various core design flaws in the algorithms.

The largest issue present was the problem of how best to combine sub-agent policies or values, with the results found demonstrating that certain methods, which utilise some form of a combination of the outputs from different sub-agents, tend to provide a better stability level than those methods that rely on the selection of a single agent.

This issue, in turn, is intrinsically related to the difficulty in maintaining the Markov property of the training environment, which is an issue that plagues much Multi-Agent research. Some of the methods researched for alleviating this issue, such as cyclic training, did improve the performance of select algorithms, although other results, such as using a fingerprint in experience replay of sub-agent states, were not replicable.

The most promising results found over the course of testing were those relating to the product system for combing PPO-trained sub-agents. There is some evidence to suggest that faster training may be possible through this method, as well as the potential to train model types that may not be possible using standard single-model training. In addition to this, multiple different algorithms resulted in the successful divergence of reward signals.

References

[1] Zafarali Ahmed et al. "Understanding the Impact of Entropy on Policy Optimization". In: (). URL: https://proceedings.mlr.press/v97/ahmed19a/ahmed19a.pdf.

[2] Greg Brockman et al. "OpenAI Gym". In: (June 2016). URL: https://github.com/openai/gym.

[3] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 38.2 (Mar. 2008), pp. 156–172. ISSN: 10946977. DOI: 10.1109/TSMCC.2007.913919. URL: https://ieeexplore.ieee.org/document/4445757.

[4] Chun Lin Chen et al. "Hybrid MDP based integrated hierarchical Q-learning". In: *Science China Information Sciences* 54.11 (Nov. 2011), pp. 2279–2294. ISSN: 1674733X. DOI: 10.1007/s11432-011-4332-6. URL: https://www.researchgate.net/publication/220362932_Hybrid_MDP_based_integrated_hierarchical_Q-learning.

[5] Tianyi Chen et al. "Communication-Efficient Policy Gradient Methods for Distributed Reinforcement Learning". In: (). URL: http://airc.rpi.edu.

[6] Will Dabney et al. "Distributional Reinforcement Learning with Quantile Regression". In: (Oct. 2017). URL: http://arxiv.org/abs/1710.10044.

[7] Matteo Hessel Deepmind et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: (). URL: https://arxiv.org/abs/1710.02298v1.

[8] Jakob Foerster et al. *Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning*. Tech. rep. 2017. URL: https://arxiv.org/abs/1702.08887.

[9] Jakob N Foerster et al. "Learning to Communicate with Deep Multi-Agent Reinforcement Learning". In: (). URL: https://arxiv.org/abs/1605.06676.

[10] Christopher Grimm and Satinder Singh. "Learning Independently-Obtainable Reward Functions". In: (). URL: https://arxiv.org/abs/1901.08649.

[11] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. "Cooperative Multi-agent Control Using Deep Reinforcement Learning". In: (). DOI: 10.1007/978-3-319-71682-4{_}5. URL: https://doi.org/10.1007/978-3-319-71682-4_5.

[12] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: (Sept. 2015). URL: http://arxiv.org/abs/1509.06461.

[13] J. Fernando Hernandez-Garcia and Richard S. Sutton. "Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target". In: (Jan. 2019). URL: http://arxiv.org/abs/1901.07510.

[14] Jianghang Huang et al. "A Comparison Study of DQN and PPO based Reinforcement Learning for Scheduling Workflows in the Cloud". In: (). URL: https://iwaciii2021.bit.edu.cn/docs/2021-12/f82acbb613964d32b461de42928867c6.pdf.

[15] Qingyan Huang. "Model-based or model-free, a review of approaches in reinforcement learning". In: *Proceedings - 2020 International Conference on Computing and Data Science, CDS 2020*. Institute of Electrical and Electronics Engineers Inc., Aug. 2020, pp. 219–221. ISBN: 9781728171067. DOI: 10.1109/CDS49703.2020.00051. URL: https://ieeexplore.ieee.org/document/9275964.

[16] Mark Humphrys. *W-learning: A simple RL-based Society of Mind*. Tech. rep. URL: http://www.cl.cam.ac.uk/users/mh10006/publications.html.

[17] Michael Janner et al. *When to Trust Your Model: Model-Based Policy Optimization*. Tech. rep. URL: When%20to%20Trust%20Your%20Model:Model-Based%20Policy%20Optimization.

[18] Zichuan Lin et al. "Distributional Reward Decomposition for Reinforcement Learning". In: (). URL: https://arxiv.org/abs/1911.02166.

[19] Michael L. Littman. "Markov games as a framework for multi-agent reinforcement learning". In: *Machine Learning Proceedings 1994* (Jan. 1994), pp. 157–163. DOI: 10.1016/B978-1-55860-335-6.50027-1. URL: https://www.sciencedirect.com/science/article/abs/pii/B9781558603356500271?via%3Dihub.

[20] Marvin Minsky. *The Society of Minds*. Simon and Schuster, 1986.

[21] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: (2016). URL: https://arxiv.org/abs/1602.01783.

[22] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Tech. rep. URL: https://arxiv.org/abs/1312.5602.

[23] Ofir Nachum et al. *Bridging the Gap Between Value and Policy Based Reinforcement Learning*. Tech. rep. URL: https://arxiv.org/abs/1702.08892.

[24] Alex Nichol et al. *Gotta Learn Fast: A New Benchmark for Generalization in RL*. Tech. rep. URL: https://www.libretro.com/index.php/api.

[25] OpenAI. *OpenAI Intro To Policy Optimisation*.

[26] Martin J Osborne and Ariel Rubinstein. "A Course in Game Theory". In: (). URL: https://arielrubinstein.tau.ac.il/books/GT.pdf.

[27] Gregory Palmer et al. "Lenient Multi-Agent Deep Reinforcement Learning". In: (2018). URL: www.ifaamas.org.

[28] Emanuele Pesce et al. "Improving coordination in small-scale multi-agent deep reinforcement learning through memory-driven communication". In: *Machine Learning* 109 (2020), pp. 1727–1747. DOI: 10.1007/s10994-019-05864-5. URL: https://doi.org/10.1007/s10994-019-05864-5.

[29] Tapabrata Ray and Xin Yao. "A cooperative coevolutionary algorithm with correlation based adaptive variable partitioning". In: *2009 IEEE Congress on Evolutionary Computation, CEC 2009* (2009), pp. 983–989. DOI: 10.1109/CEC.2009.4983052. URL: https://ieeexplore.ieee.org/document/4983052.

[30] Richard Bellman. "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6.4 (1957).

[31] Richard Bellman. "The Theory of Dynamic Programming". In: (1954). URL: https://www.ams.org/journals/bull/1954-60-06/S0002-9904-1954-09848-8/S0002-9904-1954-09848-8.pdf.

[32] John Schulman et al. "Proximal Policy Optimization Algorithms". In: (July 2017). URL: http://arxiv.org/abs/1707.06347.

[33] Peter Sunehag DeepMind et al. "Value-Decomposition Networks For Cooperative Multi-Agent Learning". In: (). URL: https://arxiv.org/abs/1706.05296.

[34] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction Second edition, in progress*. Tech. rep.

[35] Richard S Sutton et al. "Horde: A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction". In: (). URL: https://www.cs.swarthmore.edu/~meeden/DevelopmentalRobotics/horde1.pdf.

[36] Taito. *Space Invaders*. Video game. 1978.

[37] *TensorFlow*. URL: https://www.tensorflow.org/.

[38] Sebastian Thrun and Anton Schwartz. *Issues in Using Function Approximation for Reinforcement Learning*. Tech. rep. 1993. URL: https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf.

[39] Hado Van Hasselt. *Double Q-learning*. Tech. rep. 2010.

[40] Harm Van Seijen et al. "Hybrid Reward Architecture for Reinforcement Learning". In: (). URL: https://arxiv.org/abs/1706.04208.

[41] Christopher J C H Watkins and Peter Dayan. *Q-Learning*. Tech. rep. 1992, pp. 279–292.

[42] Anton Zakharenkov and Ilya Makarov. "Deep Reinforcement Learning with DQN vs. PPO in VizDoom". In: *21st IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2021 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 131–136. ISBN: 9781665426848. DOI: 10.1109/CINTI53070.2021.9668479.

[43] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. “Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms”. In: (Nov. 2019). URL: http://arxiv.org/abs/1911.10635.

[44] Yang Zhang et al. *Coordination Between Individual Agents in Multi-Agent Reinforcement Learning*. Tech. rep. 2021. URL: www.aaai.org.

Part VII
Appendix

A Depth First Search Implementation

1. **Initialisation**: Create an empty maze as a 2D grid (‘rows‘ × ‘cols‘), filling all cells with ‘False‘ representing walls.
2. **Choose Start and Goal**: Randomly select starting and goal positions within the maze, ensuring they are not on the borders. If the Manhattan distance between the start and goal position is less than 5 units, re-run the maze generation process to get a new pair of start and goal points.
3. **Mark Start and Goal**: Mark the start and goal positions on the maze by setting their corresponding positions in the 2D grid to ‘True‘, indicating open passages.
4. **Initialise Stack and Visited Set**: Create an empty stack and add the start position to it. Also create a ‘visited‘ set to keep track of all visited cells.
5. **DFS Loop**: Begin the loop which continues as long as there are positions in the stack:
    - Extract the current position (topmost position in the stack).
    - Mark the current position as visited in the maze by setting the corresponding position in the 2D grid to ‘True‘.
    - Generate the list of neighbouring positions and shuffle them to ensure randomness.
    - Iterate over the neighbours. For each neighbours:
        - Check if the neighbour is within the maze boundaries and has not been visited before.
        - If valid, mark the passage to the neighbour and the neighbour itself as ‘True‘ in the maze.
        - Add the neighbour to the stack and the ‘visited‘ set, and set ‘found‘ as ‘True‘.
        - If a valid neighbour is found, break the neighbour iteration loop.
    - If no valid neighbour was found, remove the current position from the stack (backtrack).
6. **Output**: Once all reachable positions have been visited (i.e., the stack is empty), return the start position, goal position, and the generated maze.

B DDQN Implementation Visualisation

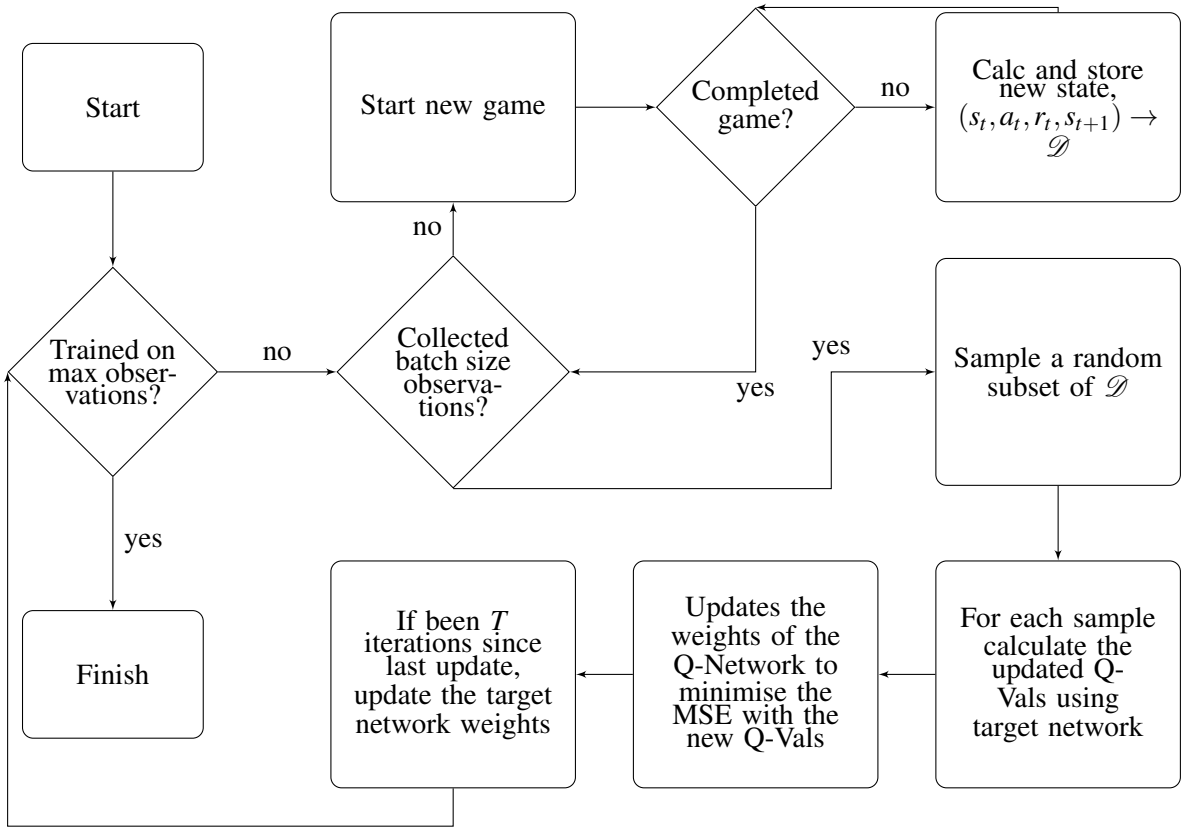

Figure 30: Flow diagram illustrating DDQN implementation

C Implementation Code

The code for the entirety of this project can be found at:
https://github.com/max-wickham/FYPDeepDeconstructedLearning